

Solutions to Kleinberg & Tardos's Algorithm Design

Aayush Bajaj

2026-10-21T12:00:00+11:00

Contents

1	Introduction: Some Representative Problems	3
1.1	Exercises 1.1–1.7	4
1.2	Exercise 1.8	8
2	Basics of Algorithm Analysis	10
2.1	Exercises 2.1–2.7	11
2.2	Exercise 2.8	15
3	Graphs	17
3.1	Exercises 3.1–3.7	18
3.2	Exercises 3.8–3.12	21
4	Greedy Algorithms	27
4.1	Exercises 4.1–4.7	28
4.2	Exercises 4.8–4.14	32
4.3	Exercises 4.15–4.21	37
4.4	Exercises 4.22–4.28	41
4.5	Exercises 4.29–4.33	46
5	Divide and Conquer	51
5.1	Exercises 5.1–5.7	52
6	Dynamic Programming	58
6.1	Exercises 6.1–6.7	59
6.2	Exercises 6.8–6.14	65
6.3	Exercises 6.15–6.21	72
6.4	Exercises 6.22–6.28	78
6.5	Exercise 6.29	84
7	Network Flow	86
7.1	Exercises 7.1–7.7	87
7.2	Exercises 7.8–7.14	90
7.3	Exercises 7.15–7.21	95
7.4	Exercises 7.22–7.28	102
7.5	Exercises 7.29–7.35	108
7.6	Exercises 7.36–7.42	113
7.7	Exercises 7.43–7.49	120
7.8	Exercises 7.50–7.51	127
8	NP and Computational Intractability	130
8.1	Exercises 8.1–8.7	131

8.2	Exercises 8.8–8.14	135
8.3	Exercises 8.15–8.21	140
8.4	Exercises 8.22–8.28	146
8.5	Exercises 8.29–8.35	150
8.6	Exercises 8.36–8.42	156
9	PSPACE: A Class of Problems beyond NP	162
9.1	Exercises 9.1–9.3	163
10	Extending the Limits of Tractability	166
10.1	Exercises 10.1–10.7	167
10.2	Exercises 10.8–10.9	173
11	Approximation Algorithms	175
11.1	Exercises 11.1–11.7	176
11.2	Exercises 11.8–11.12	180
12	Local Search	185
12.1	Exercises 12.1–12.4	186
13	Randomized Algorithms	189
13.1	Exercises 13.1–13.7	190
13.2	Exercises 13.8–13.14	194
13.3	Exercises 13.15–13.18	198

Solutions to every end-of-chapter exercise in Jon Kleinberg and Éva Tardos’s *Algorithm Design*, published by Addison-Wesley in 2005. Each design exercise gets an algorithm, a proof that it is correct, and a running-time bound; the chapters’ Solved Exercises are worked in the book itself and are not repeated here. The book itself: Algorithm Design.

1 Introduction: Some Representative Problems

Contents

1.1 Exercises 1.1–1.7	4
1.2 Exercise 1.8	8

1.1 Exercises 1.1–1.7

Problem 1.1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? In every instance of the Stable Matching Problem, there is a stable matching containing a pair (m, w) such that m is ranked first on the preference list of w and w is ranked first on the preference list of m .

Solution. False. Take $n = 2$ with

$$\begin{aligned} m_1 : w_1 \succ w_2, & & w_1 : m_2 \succ m_1, \\ m_2 : w_2 \succ w_1, & & w_2 : m_1 \succ m_2. \end{aligned}$$

The first choices of the men are $m_1 \rightarrow w_1$, $m_2 \rightarrow w_2$, and the first choices of the women are $w_1 \rightarrow m_2$, $w_2 \rightarrow m_1$; no man and woman rank each other first, so *no* matching, stable or not, contains such a pair.

Problem 1.2. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? Consider an instance of the Stable Matching Problem in which there exists a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m . Then in every stable matching S for this instance, the pair (m, w) belongs to S .

Solution. True. Let S be a perfect matching with $(m, w) \notin S$, say $(m, w'), (m', w) \in S$ with $w' \neq w$ and $m' \neq m$. Since w is first on m 's list and m is first on w 's list,

$$w \succ_m w' \quad \text{and} \quad m \succ_w m',$$

so (m, w) is an instability with respect to S and S is not stable. Hence every stable matching contains (m, w) .

Problem 1.3. There are many other settings in which we can ask questions related to some type of “stability” principle. Here’s one, involving competition between two enterprises.

Suppose we have two television networks, whom we’ll call $\mathcal{A}$ and $\mathcal{B}$. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a *schedule* — an assignment of each show to a distinct slot — so as to attract as much market share as possible.

Each show has a fixed *rating*, based on the number of people who watched it last year; assume that no two shows have exactly the same rating. A network *wins* a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network $\mathcal{A}$ reveals a schedule S and Network $\mathcal{B}$ reveals a schedule T . On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. We say that the pair of schedules (S, T) is *stable* if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' such that Network $\mathcal{A}$ wins more slots with the pair (S', T) than it did with the pair (S, T) ; and symmetrically, there is no schedule T' such that Network $\mathcal{B}$ wins more slots with the pair (S, T') than it did with the pair (S, T) .

The analogue of Gale and Shapley’s question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

(a) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or

(b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

Solution. (b): take $n = 2$, network $\mathcal{A}$ with shows of ratings 1 and 3, network $\mathcal{B}$ with shows of ratings 2 and 4.

A pair (S, T) only matters through which show of $\mathcal{A}$ faces which show of $\mathcal{B}$, and with $n = 2$ there are two such pairings; either network, by swapping its own two shows, switches to the other pairing.

(i) 3 vs 2 and 1 vs 4: each network wins one slot. Network $\mathcal{B}$ swaps, giving 3 vs 4 and 1 vs 2, and wins $2 > 1$ slots. Not stable.

(ii) 3 vs 4 and 1 vs 2: $\mathcal{A}$ wins 0 slots. Network $\mathcal{A}$ swaps, giving 3 vs 2 and 1 vs 4, and wins $1 > 0$ slots. Not stable.

Problem 1.4. Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is *stable* if neither of the following situations arises.

- First type of instability: There are students s and s' , and a hospital h , so that
 - s is assigned to h , and
 - s' is assigned to no hospital, and
 - h prefers s' to s .
- Second type of instability: There are students s and s' , and hospitals h and h' , so that
 - s is assigned to h , and
 - s' is assigned to h' , and
 - h prefers s' to s , and
 - s' prefers h to h' .

So we basically have the Stable Matching Problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

Solution. Run Gale-Shapley with the hospitals proposing: a hospital with an open position offers it to the best student it has not yet offered to, and a student holds the best offer received so far. Let hospital h have $c_h \geq 1$ positions, with $\sum_h c_h < n$.

```

all students unassigned; every hospital has all positions open
while some hospital h has an open position
  and has not yet offered to every student:
    s = highest-ranked student on h's list not yet offered by h
    h offers to s
    if s is unassigned: assign s to h
    else if s prefers h to current hospital h2:
      assign s to h    (h2 regains one open position)
    else: s rejects h
return the assignment

```

Termination. Each iteration is an offer of some h to some s , and no pair repeats, so there are at most mn iterations.

As in (1.1), once a student receives an offer she stays assigned and her hospital only improves; each

hospital offers in decreasing order of its list.

All positions filled. Suppose h has an open position at the end. The loop stopped, so h has offered to all n students, and then all n students are assigned. That needs $n \leq \sum_h c_h$ positions, contradicting $\sum_h c_h < n$.

No instability of the first type. Suppose s is at h , s' is unassigned, and h prefers s' to s . Then h offered to s , so it offered to s' earlier. After that s' is assigned forever, a contradiction.

No instability of the second type. Suppose s is at h , s' is at $h' \neq h$, h prefers s' to s , and s' prefers h to h' . As before, h offered to s' . From then on, the hospital s' holds is h or one she prefers to h . So her final hospital h' is not worse than h , a contradiction.

Running time. Store the students' rankings as inverse arrays $\text{rank}_s[h]$ and keep a list of hospitals with open positions, as in Section 2.3. Each iteration then costs $O(1)$, so the algorithm takes $O(mn)$ time after $O(mn)$ preprocessing.

Problem 1.5. The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be *indifferent* between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with $n = 4$), a woman could say that m_1 is ranked in first place; second place is a tie between m_2 and m_3 (she has no preference between them); and m_4 is in last place. We will say that w *prefers* m to m' if m is ranked higher than m' on her preference list (they are not tied).

With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) A *strong instability* in a perfect matching S consists of a man m and a woman w , such that each of m and w prefers the other to their partner in S . Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

(b) A *weak instability* in a perfect matching S consists of a man m and a woman w , such that their partners in S are w' and m' , respectively, and one of the following holds:

- m prefers w to w' , and w either prefers m to m' or is indifferent between these two choices; or
- w prefers m to m' , and m either prefers w to w' or is indifferent between these two choices.

In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.

Solution. (a) Yes. Break every tie arbitrarily, which gives strict preference lists, and run Gale-Shapley on the resulting instance. By (1.5) and (1.6) it returns a perfect matching S that is stable for the strict lists. Suppose (m, w) were a strong instability in S , with partners w' and m' . Then m strictly prefers w to w' and w strictly prefers m to m' in the original lists. Breaking ties keeps every strict preference, so the same holds in the strict lists, and (m, w) is an instability there. This contradicts (1.6). The running time is that of Gale-Shapley, $O(n^2)$ (Section 2.3), plus $O(n^2)$ for breaking ties.

(b) No. Take $n = 2$ with

$$m_1, m_2 : w_1 \succ w_2, \quad w_1, w_2 : m_1 \sim m_2,$$

where $\sim$ means indifferent: both men strictly prefer w_1 , and both women are indifferent. There are two perfect matchings:

(i) $\{(m_1, w_1), (m_2, w_2)\}$: m_2 prefers w_1 to w_2 , and w_1 is indifferent between m_2 and m_1 . So (m_2, w_1) is a weak instability.

(ii) $\{(m_1, w_2), (m_2, w_1)\}$: m_1 prefers w_1 to w_2 , and w_1 is indifferent between m_1 and m_2 . So (m_1, w_1) is a weak instability.

Problem 1.6. Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships and provides service to n ports. Each of its ships has a *schedule* that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(†) *No two ships can be in the same port on the same day.*

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to *truncate* each ship's schedule: for each ship S_i , there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the *truncation* of S_i 's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P ; the remainder of the truncated schedule simply has it remain in port P .

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an algorithm to find them.

Example. Suppose we have two ships and two ports, and the "month" has four days. Suppose the first ship's schedule is

port P_1 ; at sea; port P_2 ; at sea

and the second ship's schedule is

at sea; port P_1 ; at sea; port P_2

Then the (only) way to choose truncations would be to have the first ship remain in port P_2 starting on day 3, and have the second ship remain in port P_1 starting on day 2.

Solution. Run Gale-Shapley with ships as the proposing side and ports as the other: each ship ranks the ports in the order it visits them (earliest first), and each port ranks the ships in **reverse** order of their visits (latest visitor first); if ship S is matched to port P , it stops at P on the day $d(S, P)$ it is scheduled there.

By (1.5) and (1.6) this produces a perfect, stable matching, so every ship gets a distinct stopping port. Suppose two ships are in the same port P on day t in the truncated schedules.

(i) Both are moving: then both are following their original schedules on day t , contradicting the original (†).

(ii) Both are stopped: then both are matched to P , impossible in a matching.

(iii) Ship S is stopped at P (so $d(S, P) \leq t$) and ship S' , matched to $P' \neq P$, is still moving and visits P on day $t = d(S', P)$. By (†), $d(S', P) \neq d(S, P)$, so

$$\begin{aligned} d(S', P) > d(S, P) &\Rightarrow P \text{ prefers } S' \text{ to } S, \\ d(S', P) < d(S', P') &\Rightarrow S' \text{ prefers } P \text{ to } P', \end{aligned}$$

the second because S' has not yet reached its stop. So (S', P) is an instability, contradicting (1.6).

Hence the truncations satisfy (†), and in particular they always exist. Building the preference lists takes one scan of the schedules, $O(mn)$ (sorting each ship's ports and each port's visitors by day is a bucket pass over the m days), and Gale-Shapley runs in $O(n^2)$ (Section 2.3), for $O(mn)$ total.

Problem 1.7. Some of your friends are working for CluNet, a builder of large communication networks, and they are looking at algorithms for switching in a particular type of input/output crossbar.

Here is the setup. There are n *input wires* and n *output wires*, each directed from a *source* to a *terminus*. Each input wire meets each output wire in exactly one distinct point, at a special piece of hardware called a *junction box*. Points on the wire are naturally ordered in the direction from source to terminus; for two distinct points x and y on the same wire, we say that x is *upstream* from y if x

is closer to the source than y , and otherwise we say x is *downstream* from y . The order in which one input wire meets the output wires is not necessarily the same as the order in which another input wire meets the output wires. (And similarly for the orders in which output wires meet input wires.) Figure 1.8 gives an example of such a collection of input and output wires.

Now, here's the switching component of this situation. Each input wire is carrying a distinct data stream, and this data stream must be *switched* onto one of the output wires. If the stream of Input i is switched onto Output j , at junction box B , then this stream passes through all junction boxes upstream from B on Input i , then through B , then through all junction boxes downstream from B on Output j . It does not matter which input data stream gets switched onto which output wire, but each input data stream must be switched onto a *different* output wire. Furthermore—and this is the tricky constraint—no two data streams can pass through the same junction box following the switching operation.

Finally, here's the problem. Show that for any specified pattern in which the input wires and output wires meet each other (each pair meeting exactly once), a valid switching of the data streams can always be found—one in which each input data stream is switched onto a different output, and no two of the resulting streams pass through the same junction box. Additionally, give an algorithm to find such a valid switching.

Figure 1.8 (an example with two input wires and two output wires): Input 1 meets Output 2 upstream from its junction with Output 1; Input 2 meets Output 1 upstream from its junction with Output 2; Output 1 and Output 2 each meet Input 2 before Input 1. A valid solution is to switch the stream of Input 1 onto Output 2 and the stream of Input 2 onto Output 1. On the other hand, if Input 1 were switched onto Output 1 and Input 2 onto Output 2, then both streams would pass through the junction box at the meeting of Input 1 and Output 2, which is not allowed.

Solution. Run Gale-Shapley with inputs proposing to outputs: each input ranks the outputs in the order it meets them (most upstream first), and each output ranks the inputs in **reverse** order of meeting them (most downstream first); switch input i onto its partner output at their common junction box B_{ij} .

By (1.5) and (1.6) the result is a perfect, stable matching, so each input goes to a different output. A box $B_{ij'}$ lies on exactly one input and one output, so two streams can meet there only if one arrives along Input i and the other along Output j' . Thus suppose i is switched onto $j \neq j'$ with $B_{ij'}$ upstream of B_{ij} on Input i , and $i' \neq i$ is switched onto j' with $B_{i'j'}$ downstream of $B_{ij'}$ on Output j' . Then

$$\begin{aligned} B_{ij'} \text{ upstream of } B_{ij} \text{ on Input } i &\Rightarrow i \text{ prefers } j' \text{ to } j, \\ B_{ij'} \text{ downstream of } B_{i'j'} \text{ on Output } j' &\Rightarrow j' \text{ prefers } i \text{ to } i', \end{aligned}$$

so (i, j') is an instability, contradicting (1.6). Hence no junction box carries two streams, and a valid switching always exists.

Reading off the $2n$ preference lists from the wiring takes $O(n^2)$, and Gale-Shapley runs in $O(n^2)$ (Section 2.3), so the algorithm runs in $O(n^2)$ time.

1.2 Exercise 1.8

Problem 1.8. For this problem, we will explore the issue of *truthfulness* in the Stable Matching Problem and specifically in the Gale-Shapley algorithm. The basic question is: Can a man or a woman end up better off by lying about his or her preferences? More concretely, we suppose each participant has a true preference order. Now consider a woman w . Suppose w prefers man m to m' , but both m and m' are low on her list of preferences. Can it be the case that by switching the order of m and m' on her list of preferences (i.e., by falsely claiming that she prefers m' to m) and running the algorithm with this false preference list, w will end up with a man m'' that she truly prefers to both m and m' ? (We can ask the same question for men, but will focus on the case of women for purposes of this question.)

Resolve this question by doing one of the following two things:

- (a) Give a proof that, for any set of preference lists, switching the order of a pair on the list cannot improve a woman's partner in the Gale-Shapley algorithm; or
- (b) Give an example of a set of preference lists for which there is a switch that would improve the partner of a woman who switched preferences.

Solution. (b) holds: a woman can do better by lying. Take three men and three women with these preference lists:

m_1	$w_1 > w_2 > w_3$	w_1 (true)	$m_3 > m_1 > m_2$
m_2	$w_1 > w_2 > w_3$	w_2	$m_1 > m_3 > m_2$
m_3	$w_2 > w_1 > w_3$	w_3	$m_1 > m_2 > m_3$

Here $w = w_1$, $m = m_1$, $m' = m_2$, $m'' = m_3$. By (1.7) every execution of Gale-Shapley gives the same matching, so any proposal order may be used.

Truthful lists. m_1 gets w_1 ; m_2 is rejected by w_1 and gets w_2 ; m_3 proposes to w_2 , who trades m_2 for m_3 ; m_2 goes to w_3 . The output is

$$\{(m_1, w_1), (m_3, w_2), (m_2, w_3)\},$$

so w_1 gets m_1 .

w_1 claims $m_3 > m_2 > m_1$. m_1 gets w_1 ; m_2 proposes to w_1 , who now trades m_1 for m_2 ; m_1 gets w_2 ; m_3 is rejected by w_2 (she holds m_1) and proposes to w_1 , who trades m_2 for m_3 ; m_2 is rejected by w_2 and gets w_3 . The output is

$$\{(m_3, w_1), (m_1, w_2), (m_2, w_3)\},$$

so w_1 gets m_3 , whom she truly prefers to both m_1 and m_2 .

2 Basics of Algorithm Analysis

Contents

2.1 Exercises 2.1–2.7	11
2.2 Exercise 2.8	15

2.1 Exercises 2.1–2.7

Problem 2.1. Suppose you have algorithms with the five running times listed below. (Assume these are the exact running times.) How much slower do each of these algorithms get when you (a) double the input size, or (b) increase the input size by one?

- (a) n^2
- (b) n^3
- (c) $100n^2$
- (d) $n \log n$
- (e) 2^n

Solution. Doubling multiplies the polynomial times by a constant and squares the exponential; adding one multiplies only the exponential by a constant. Writing $T(n)$ for the running time (log to base 2):

running time	doubling: $T(2n)$	plus one: $T(n+1)$
n^2	$4T(n)$	$T(n) + 2n + 1$
n^3	$8T(n)$	$T(n) + 3n^2 + 3n + 1$
$100n^2$	$4T(n)$	$T(n) + 200n + 100$
$n \log n$	$2T(n) + 2n$	$T(n) + (n+1) \log(n+1) - n \log n$
2^n	$T(n)^2 = 2^n T(n)$	$2T(n)$

Each entry is a direct expansion, e.g.

$$(2n) \log(2n) = 2n(\log n + 1) = 2n \log n + 2n,$$

$$2^{2n} = (2^n)^2, \quad 2^{n+1} = 2 \cdot 2^n.$$

So under doubling, (a) and (c) become 4 times slower, (b) 8 times slower, (d) slightly more than twice as slow (by the factor $2(1 + 1/\log n)$), and (e) becomes 2^n times slower, i.e. its running time is squared. Under $n \mapsto n+1$, (a)-(d) slow down by the additive amounts in the table (a factor tending to 1), and (e) becomes twice as slow.

Problem 2.2. Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size n .) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?

- (a) n^2
- (b) n^3
- (c) $100n^2$
- (d) $n \log n$
- (e) 2^n
- (f) 2^{2^n}

Solution. The budget is $N = 3600 \cdot 10^{10} = 3.6 \times 10^{13}$ operations, and each answer is the largest n with $T(n) \leq N$ (taking $\log = \log_2$).

- (a) $n^2 \leq 3.6 \times 10^{13} \iff n \leq 6 \times 10^6$, so $n = 6,000,000$.
- (b) $33019^3 \approx 3.59991 \times 10^{13} \leq N < 33020^3$, so $n = 33,019$.
- (c) $100n^2 \leq N \iff n^2 \leq 3.6 \times 10^{11} \iff n \leq 6 \times 10^5$, so $n = 600,000$.
- (d) $n \log_2 n$ is increasing, and

$$906316482853 \cdot \log_2 906316482853 \approx 3.6 \times 10^{13} - 7 \leq N,$$

$$906316482854 \cdot \log_2 906316482854 \approx 3.6 \times 10^{13} + 34 > N,$$

so $n = 906,316,482,853 \approx 9.06 \times 10^{11}$.

- (e) $2^{45} \approx 3.52 \times 10^{13} \leq N < 2^{46} \approx 7.04 \times 10^{13}$, so $n = 45$.
- (f) $2^{2^n} \leq N \iff 2^n \leq \log_2 N \approx 45.03$, and $2^5 = 32 \leq 45.03 < 64 = 2^6$, so $n = 5$.

Problem 2.3. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$\begin{aligned} f_1(n) &= n^{2.5} \\ f_2(n) &= \sqrt{2n} \\ f_3(n) &= n + 10 \\ f_4(n) &= 10^n \\ f_5(n) &= 100^n \\ f_6(n) &= n^2 \log n \end{aligned}$$

Solution. The order is $f_2, f_3, f_6, f_1, f_4, f_5$, that is,

$$\sqrt{2n}, \quad n + 10, \quad n^2 \log n, \quad n^{2.5}, \quad 10^n, \quad 100^n.$$

Each consecutive pair is certified as follows (for $n \geq 2$):

$$\begin{aligned} \sqrt{2n} &\leq 2n \leq 2(n + 10), \\ n + 10 &\leq 11n \leq 11n^2 \log n, \\ n^2 \log n &= n^2 \cdot O(n^{0.5}) = O(n^{2.5}) \quad \text{by (2.8) with } x = 0.5, \\ n^{2.5} &= O(10^n) \quad \text{by (2.9) with } r = 10, \quad d = 2.5, \\ 10^n &\leq 100^n. \end{aligned}$$

Problem 2.4. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$\begin{aligned} g_1(n) &= 2^{\sqrt{\log n}} \\ g_2(n) &= 2^n \\ g_4(n) &= n^{4/3} \\ g_3(n) &= n(\log n)^3 \\ g_5(n) &= n^{\log n} \\ g_6(n) &= 2^{2^n} \\ g_7(n) &= 2^{n^2} \end{aligned}$$

Solution. The order is $g_1, g_3, g_4, g_5, g_2, g_7, g_6$, that is,

$$2^{\sqrt{\log n}}, \quad n(\log n)^3, \quad n^{4/3}, \quad n^{\log n}, \quad 2^n, \quad 2^{n^2}, \quad 2^{2^n}.$$

Write everything with $\log = \log_2$, so $n = 2^{\log n}$ and $n^{\log n} = 2^{(\log n)^2}$. Each consecutive pair is certified as follows:

$$\begin{aligned} 2^{\sqrt{\log n}} &\leq 2^{\log n} = n \leq n(\log n)^3 & (n \geq 2, \text{ as } \log n \geq 1), \\ n(\log n)^3 &= n \cdot O(n^{1/3}) = O(n^{4/3}) & \text{by (2.8) with } x = 1/9, \text{ cubed}, \\ n^{4/3} &= 2^{\frac{4}{3} \log n} \leq 2^{(\log n)^2} = n^{\log n} & (n \geq 4, \text{ as } \log n \geq 2), \\ n^{\log n} &= 2^{(\log n)^2} \leq 2^n & (n \geq 16, \text{ as } (\log n)^2 \leq n), \\ 2^n &\leq 2^{n^2} & (n \geq 1), \\ 2^{n^2} &\leq 2^{2^n} & (n \geq 4, \text{ as } n^2 \leq 2^n). \end{aligned}$$

For $(\log n)^2 \leq n$, i.e. $\log_2 x \leq \sqrt{x}$ for $x \geq 16$: equality holds at $x = 16$, and

$$\frac{d}{dx}(\sqrt{x} - \log_2 x) = \frac{1}{2\sqrt{x}} - \frac{1}{x \ln 2} > 0 \iff \sqrt{x} > \frac{2}{\ln 2} \approx 2.89.$$

For $n^2 \leq 2^n$ ($n \geq 4$): it holds at $n = 4$, and $(n+1)^2 \leq 2n^2$ for $n \geq 3$ carries the induction.

Problem 2.5. Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

- (a) $\log_2 f(n)$ is $O(\log_2 g(n))$.
- (b) $2^{f(n)}$ is $O(2^{g(n)})$.
- (c) $f(n)^2$ is $O(g(n)^2)$.

Solution. (a) False. Take $f(n) = 2$ and $g(n) = 1$: then $f(n) \leq 2g(n)$, so $f = O(g)$, but

$$\log_2 f(n) = 1 \leq c \cdot 0 = c \log_2 g(n)$$

fails for every c and every n , so $\log_2 f$ is not $O(\log_2 g)$.

(b) False. Take $f(n) = 2n$ and $g(n) = n$: then $f = O(g)$ with $c = 2$, but

$$\frac{2^{f(n)}}{2^{g(n)}} = \frac{4^n}{2^n} = 2^n \rightarrow \infty,$$

so no constant c gives $2^{2n} \leq c 2^n$ for all large n .

(c) True. There are $c > 0$ and n_0 with $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$ (running times are nonnegative), and squaring preserves inequalities between nonnegative numbers, so

$$f(n)^2 \leq c^2 g(n)^2 \quad (n \geq n_0),$$

i.e. $f^2 = O(g^2)$ with constant c^2 .

Problem 2.6. An array A holds n integers $A[1], \dots, A[n]$. The goal is to produce an $n \times n$ array B such that, for every $i < j$, the entry $B[i, j]$ equals $A[i] + A[i+1] + \dots + A[j]$; entries with $i \geq j$ may hold anything. The obvious algorithm is: for $i = 1, \dots, n$ and, inside that, for $j = i+1, \dots, n$, add up $A[i]$ through $A[j]$ from scratch and store the total in $B[i, j]$.

- (a) Choose a function f and prove that the running time of this algorithm on inputs of size n (its number of operations) is $O(f(n))$.
- (b) For the same f , prove that the running time is also $\Omega(f(n))$, so that the bound $\Theta(f(n))$ is tight.
- (c) The algorithm, natural as it is, wastes a great deal of work. Design a different algorithm for the same problem whose running time is $O(g(n))$ for some g with $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$.

Solution. Take $f(n) = n^3$. Summing $A[i]$ through $A[j]$ touches $j - i + 1$ entries, so (up to the constant cost of each read, addition, and store) the pair (i, j) costs $j - i + 1$ steps. Writing $d = j - i$, there are $n - d$ pairs with difference d , so the total work is

$$\begin{aligned} T(n) &= \sum_{1 \leq i < j \leq n} (j - i + 1) = \sum_{d=1}^{n-1} (n-d)(d+1) \\ &= \frac{(n-1)n(n+4)}{6}. \end{aligned}$$

(The closed form is a routine sum of $n \sum d$ and $\sum d^2$. Check!)

(a) Because $(n-1)(n+4) \leq n \cdot 5n$, we get $T(n) \leq \frac{5}{6}n^3$, and therefore $T(n) = O(n^3)$.

(b) For $n \geq 2$ we have $3n \geq 4$, so $(n-1)(n+4) = n^2 + 3n - 4 \geq n^2$. Hence $T(n) \geq \frac{1}{6}n^3$, and $T(n) = \Omega(n^3)$. Together with (a), $T(n) = \Theta(n^3)$.

(c) Reuse the previous sum: $B[i, j] = B[i, j-1] + A[j]$, which costs one addition per entry.

```

For i = 1, 2, ..., n
  s = A[i]
  For j = i+1, i+2, ..., n
    s = s + A[j]
    Store s in B[i, j]
  Endfor
Endfor

```

By induction on j (starting from $s = A[i]$, each iteration appends exactly $A[j]$), $s = A[i] + \dots + A[j]$ when $B[i, j]$ is stored. For a fixed i the inner loop does $O(n-i)$ work, so the total is $\sum_i O(n-i+1) = O(n^2)$. With $g(n) = n^2$ we get $g(n)/f(n) = 1/n \rightarrow 0$.

Problem 2.7. Some folk and holiday songs are cumulative: every verse repeats the whole previous verse and adds one new line (as in “The Twelve Days of Christmas”, where verse five sings the five golden rings and then all the lines of verse four, down to the partridge in the pear tree; the Aramaic “Had gadya” from the Passover Haggadah works the same way). Such a song can be written down compactly by giving only the new line of each verse plus instructions for how to sing it, so each line is written once even though it is sung in every later verse. Assume every line has length at most a constant c (words), and that the song, sung aloud, lasts n words in total. Show how to encode such a song by a script of length $f(n)$, with $f(n)$ growing as slowly as possible.

Solution. $f(n) = O(\sqrt{n})$: write each of the k new lines once, followed by a constant-length singing instruction.

Let the song have k verses with new lines $\ell_1, \dots, \ell_k$ of lengths $1 \leq |\ell_i| \leq c$, where verse j is $\ell_j, \ell_{j-1}, \dots, \ell_1$. The script is

```

Lines: l_1, l_2, ..., l_k
For j = 1, 2, ..., k:
  sing l_j, l_(j-1), ..., l_1

```

which reproduces the song exactly, and whose length is $\sum_i |\ell_i| + O(1) \leq ck + O(1)$ (the instruction is a fixed text; even writing the number k explicitly adds only $O(\log k)$). Since every line has at least one word, verse j has at least j words, so

$$n = \sum_{j=1}^k \sum_{i=1}^j |\ell_i| \geq \sum_{j=1}^k j = \frac{k(k+1)}{2} \geq \frac{k^2}{2},$$

$$\text{hence } k \leq \sqrt{2n}.$$

Therefore the script has length at most

$$f(n) = c\sqrt{2n} + O(1) = O(\sqrt{n}).$$

This is the best possible in general. Each verse has at most cj words, so

$$n \leq c \frac{k(k+1)}{2} \leq ck^2 \quad (k \geq 1),$$

$$\text{hence } k \geq \sqrt{n/c},$$

and when the k new lines are arbitrary (distinct, unrelated words) any script that determines the song must spell out each of them at least once, so its length is at least $k \geq \sqrt{n/c} = \Omega(\sqrt{n})$. Thus the cumulative script achieves the optimal growth rate $f(n) = \Theta(\sqrt{n})$.

2.2 Exercise 2.8

Problem 2.8. A ladder has n rungs, and we want the *highest safe rung*: the highest rung from which a jar of a given type can be dropped without breaking (dropping from any higher rung breaks it). A dropped jar that survives can be reused; a broken one cannot. Binary search (drop from the middle rung, then from rung $n/4$ or $3n/4$ according to the outcome, and so on) may break many jars. At the other extreme, a single jar suffices if we drop it from rung 1, then rung 2, and so on until it breaks, but this can take n drops instead of about $\log n$. To quantify the trade-off, fix a budget of $k \geq 1$ jars: we must determine the highest safe rung while breaking at most k jars.

(a) With a budget of $k = 2$ jars, describe a strategy that always finds the highest safe rung using at most $f(n)$ drops, where f grows more slowly than linearly, i.e. $\lim_{n \rightarrow \infty} f(n)/n = 0$.

(b) With a budget of $k > 2$ jars, for a given k , describe a strategy that finds the highest safe rung using at most k jars. Writing $f_k(n)$ for the number of drops your strategy needs, the functions $f_1, f_2, f_3, \dots$ must each grow asymptotically more slowly than the one before: $\lim_{n \rightarrow \infty} f_k(n)/f_{k-1}(n) = 0$ for every k .

Solution. Split the ladder into blocks of size about $n^{(k-1)/k}$, use the first jar to find the block by dropping from the top rung of each block in turn, and then solve that block recursively with the remaining $k - 1$ jars; this gives $f_k(n) = \Theta(n^{1/k})$, and in particular $f_2(n) \leq 2\lceil \sqrt{n} \rceil$.

Label the possible answers $0, 1, \dots, n$, where 0 means that rung 1 already breaks the jar. Fix $r = \lceil n^{1/k} \rceil$, so $r^k \geq n$. The strategy S_j uses j jars on a stretch of $m \leq r^j$ consecutive unknown rungs $\ell+1, \dots, \ell+m$, where rung ℓ is already known to be safe:

```

S_j(l, m):                                     # rungs l+1..l+m unknown, rung l safe
  if j = 1: drop at l+1, l+2, ... until a break; answer = last safe rung
  s = r^(j-1); x = l + s
  while x <= l + m:
    drop at x
    if it breaks: return S_(j-1)(x - s, s - 1)
    x = x + s
  return S_(j-1)(x - s, l + m - (x - s))

```

The full strategy is $S_k(0, n)$. For $k = 2$ the first jar is dropped at rungs $r, 2r, 3r, \dots$ with $r = \lceil \sqrt{n} \rceil$, and the second jar scans upward one rung at a time inside the single block of fewer than r rungs that is left.

Correctness and the jar budget. We use induction on j . For $j = 1$ this is the linear scan from the statement: it is correct, takes at most $m \leq r$ drops and breaks at most one jar. For $j \geq 2$, suppose the first break is at rung x . Then rung $x - s$ is safe (it is either ℓ or a rung that survived a drop), so the answer lies in $[x - s, x - 1]$, and the $s - 1 < r^{j-1}$ rungs $x - s + 1, \dots, x - 1$ are handed to S_{j-1} . If there is no break, the answer lies at or above the last rung tested, $x - s$, and fewer than $s = r^{j-1}$ rungs remain above it. In both cases the hypothesis of S_{j-1} holds. The loop breaks at most one jar, because it stops at the first break, and S_{j-1} breaks at most $j - 1$ by induction. So S_k is correct and breaks at most k jars.

Upper bound. The loop tests multiples of $s = r^{j-1}$ inside a stretch of $m \leq r^j$ rungs, so it makes at most $m/s \leq r$ drops. By induction, S_j therefore makes at most jr drops. Using $\lceil y \rceil \leq 2y$ for $y \geq 1$,

$$f_k(n) \leq k \lceil n^{1/k} \rceil \leq 2k n^{1/k}.$$

Lower bound. If the answer is n , no jar ever breaks, and the top-level loop drops at every multiple of $s = r^{k-1} \leq (2n^{1/k})^{k-1}$ up to n . Hence

$$f_k(n) \geq \left\lfloor \frac{n}{s} \right\rfloor \geq \frac{n^{1/k}}{2^{k-1}} - 1.$$

For $k = 1$ we have $r = n$ and S_1 is the plain scan, so $f_1(n) = n$.

(a) For $k = 2$ the upper bound gives $f_2(n) \leq 2\lceil \sqrt{n} \rceil$, so $f_2(n)/n \leq 2(\sqrt{n} + 1)/n \rightarrow 0$.

(b) For $k \geq 2$, combining the upper bound on f_k with the lower bound on f_{k-1} (for $k = 2$, use $f_1(n) = n$)

directly) gives

$$\begin{aligned}\frac{f_k(n)}{f_{k-1}(n)} &\leq \frac{2k n^{1/k}}{n^{1/(k-1)}/2^{k-2} - 1} \\ &= \frac{2^{k-1}k n^{1/k-1/(k-1)}}{1 - 2^{k-2}n^{-1/(k-1)}} \longrightarrow 0,\end{aligned}$$

because $1/k - 1/(k-1) < 0$ and the denominator tends to 1. So each f_k grows strictly more slowly than f_{k-1} .

3 Graphs

Contents

3.1 Exercises 3.1–3.7	18
3.2 Exercises 3.8–3.12	21

3.1 Exercises 3.1–3.7

Problem 3.1. Consider the directed acyclic graph G in Figure 3.10, which has nodes a, b, c, d, e, f and the six edges

$$(a, b), (b, c), (c, f), (a, d), (d, e), (e, f).$$

How many topological orderings does it have?

Solution. Six.

The only node with no incoming edge is a and the only node with no outgoing edge is f , so every topological ordering starts with a and ends with f . The four middle positions hold b, c, d, e , subject only to b before c and d before e (there are no edges between $\{b, c\}$ and $\{d, e\}$). Such an ordering is determined by choosing which two of the four middle positions hold b and c , so the count is

$$\binom{4}{2} = 6 :$$

$$\begin{array}{lll} abcdef, & abdcef, & abdecf, \\ adbcef, & adbecf, & adebcf. \end{array}$$

Problem 3.2. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m+n)$ for a graph with n nodes and m edges.

Solution. Run BFS from every undiscovered node, recording parents and layer numbers; the first time an edge (u, v) is scanned with v already discovered and $v \neq \text{parent}(u)$, that edge is not a tree edge, and it closes the cycle formed with the two tree paths up to the lowest common ancestor of u and v .

```

for each node s not yet discovered:
    discover s; layer(s) = 0; parent(s) = none; Q = [s]
    while Q nonempty:
        u = dequeue(Q)
        for each edge (u, v):
            if v not discovered:
                discover v; parent(v) = u; layer(v) = layer(u) + 1
                enqueue(Q, v)
            else if v != parent(u):
                x = u; y = v; P = [u]; R = [v]
                while layer(x) > layer(y): x = parent(x); append x to P
                while layer(y) > layer(x): y = parent(y); append y to R
                while x != y:
                    x = parent(x); append x to P
                    y = parent(y); append y to R
                output P followed by R reversed (without its last node w = x = y)
                stop
    output "no cycle"
```

Correctness. (We take G simple, as in the book.) Every node is dequeued once, so every edge is scanned. Suppose the algorithm outputs “no cycle”. Then each scan of an edge (u, v) found either v undiscovered, making (u, v) a tree edge, or $v = \text{parent}(u)$, so (u, v) is again a tree edge. Hence every edge of G lies in the BFS forest, so G is a forest and has no cycle.

Suppose the algorithm stops at edge (u, v) . This edge is not in the BFS forest: $v \neq \text{parent}(u)$ by the test, and $u \neq \text{parent}(v)$ because a child of u is discovered during the scan of its unique edge to u , whereas v was already discovered when this edge was scanned. The parent pointers from u and v climb to the same BFS root, so the climb reaches their lowest common ancestor w (the first two loops equalize layers, the third climbs in lockstep until the pointers meet). The tree paths $u \rightarrow w$ and $v \rightarrow w$ share only w , so

$$u \rightarrow \cdots \rightarrow w \rightarrow \cdots \rightarrow v \rightarrow u$$

is a simple cycle; it has at least three nodes, since a cycle on two nodes would need (u, v) to be a tree edge.

Running time. The BFS over all components takes $O(m + n)$ by (3.11), and the cycle extraction runs once and climbs each tree path at most once, costing $O(n)$. The total is $O(m + n)$.

Problem 3.3. The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G , it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G , thus establishing that G is not a DAG. The running time of your algorithm should be $O(m + n)$ for a directed graph with n nodes and m edges.

Solution. Run the $O(m + n)$ implementation of Section 3.6 until the set S of active nodes with no active in-neighbours is empty; if all n nodes were deleted, output the deletion order, and otherwise walk backward along edges among the still-active nodes, exactly as in the proof of (3.19), until a node repeats, and output the cycle between the two visits.

```

build in-lists and out-lists; count(w) = indegree(w); S = {w : count(w) = 0}
while S nonempty:
    remove v from S; mark v deleted; append v to ORDER
    for each edge (v, w):
        count(w) = count(w) - 1; if count(w) = 0: add w to S
if |ORDER| = n: output ORDER
else:
    v = any active node; i = 0
    while v has no index:
        index(v) = i; WALK[i] = v; i = i + 1
        v = an active u with (u, v) an edge      (scan in-list of v)
    output WALK[index(v)], ..., WALK[i-1] in reverse order

```

Correctness. If $|ORDER| = n$, every node v was deleted while it had no edge from an active node, so every edge into v comes from a node deleted earlier; thus every edge points forward in ORDER and it is a topological ordering, certifying by (3.18) that G is a DAG.

Otherwise the set A of active nodes is nonempty and $S = \emptyset$, so every $v \in A$ has $\text{count}(v) \geq 1$, i.e. an incoming edge from a node of A . Hence the backward walk never gets stuck and stays inside A ; since A is finite, some node v is reached a second time. If $v = \text{WALK}[j]$, the walk gives edges

$$\text{WALK}[k+1] \rightarrow \text{WALK}[k] \quad (j \leq k < i-1), \quad \text{WALK}[j] = v \rightarrow \text{WALK}[i-1],$$

so

$$\text{WALK}[i-1] \rightarrow \text{WALK}[i-2] \rightarrow \cdots \rightarrow \text{WALK}[j] \rightarrow \text{WALK}[i-1]$$

is a directed cycle on the distinct nodes $\text{WALK}[j], \dots, \text{WALK}[i-1]$ (a self-loop when $j = i-1$).

Running time. The first phase is the book's $O(m + n)$ implementation from Section 3.6. In the walk each node receives an index at most once, so each in-list is scanned at most once, costing $O(m + n)$ in total; building the in-lists and outputting the cycle are also $O(m + n)$.

Problem 3.4. Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught, thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups (those that belong to A and those that belong to B), but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j , they study them carefully side by side. If they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B . So more concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have different labels.

Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.

Solution. Build the graph G on the n specimens with one edge per judgment, propagate labels by BFS from an arbitrary root of each component (a "same" edge copies the label, a "different" edge flips it), and then check every judgment against the labels: they are consistent if and only if every check passes.

```

for each specimen  $s$  not yet labeled:
    label( $s$ ) =  $A$ ;  $Q = [s]$ 
    while  $Q$  nonempty:
         $u = \text{dequeue}(Q)$ 
        for each judgment  $(u, v)$  with  $v$  not yet labeled:
            label( $v$ ) = label( $u$ )      if  $(u, v)$  is "same"
            label( $v$ ) = other(label( $u$ )) if  $(u, v)$  is "different"
            enqueue( $Q, v$ )
for each judgment  $(i, j)$ :
    if it is violated by label( $i$ ), label( $j$ ): output "inconsistent"; stop
output "consistent"

```

Correctness. If every check passes, the computed labeling satisfies all m judgments, so they are consistent. Conversely suppose some labeling L satisfies all judgments, and let C be a component of G with BFS root s . Swapping A and B on C preserves every judgment (each judgment lies inside one component and depends only on whether its two labels agree), so we may assume $L(s) = A = \text{label}(s)$. Each node $v \neq s$ of C receives its label from its BFS parent u across the judgment (u, v) , and L satisfies that judgment, so by induction on the BFS layer

$$L(u) = \text{label}(u) \Rightarrow L(v) = \text{label}(v).$$

Hence $\text{label} = L$ on every component, so every check passes. The algorithm therefore reports "consistent" exactly when the judgments are consistent.

Running time. The labeling phase is BFS over all components of a graph with n nodes and m edges, $O(m + n)$ by (3.11), and the check costs $O(1)$ per judgment, for $O(m + n)$ in total.

Problem 3.5. A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

Solution. Induct on the number of nodes $n \geq 1$: write $t(T)$ for the number of nodes of T with two children and $\ell(T)$ for its number of leaves; we show $t(T) = \ell(T) - 1$.

For $n = 1$ the root is a leaf, so $t(T) = 0 = \ell(T) - 1$. Let $n \geq 2$, assume the claim for all binary trees with fewer than n nodes, and let r be the root of T , which has one or two children. Every node other than r has the same children in T as in the subtree containing it, and r is not a leaf.

(i) r has one child, rooting a subtree T' with $n - 1$ nodes. Since r has neither zero nor two children,

$$t(T) = t(T') = \ell(T') - 1 = \ell(T) - 1.$$

(ii) r has two children, rooting subtrees T_1, T_2 with fewer than n nodes each. Then r is counted in $t(T)$

and

$$\begin{aligned} t(T) &= t(T_1) + t(T_2) + 1 = (\ell(T_1) - 1) + (\ell(T_2) - 1) + 1 \\ &= \ell(T_1) + \ell(T_2) - 1 = \ell(T) - 1. \end{aligned}$$

This completes the induction.

Problem 3.6. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u , and obtain a tree T that includes all nodes of G . Suppose we then compute a breadth-first search tree rooted at u , and obtain the same tree T . Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u , then G cannot contain any edges that do not belong to T .)

Solution. A non-tree edge would have to join an ancestor to a descendant at least two levels below it (DFS), yet join nodes in adjacent or equal layers (BFS), which is impossible.

Let $\text{depth}(x)$ be the depth of x in T rooted at u ; since T is a BFS tree, x lies in layer $L_{\text{depth}(x)}$. Suppose $(x, y) \in E$ is not an edge of T . Since T is a DFS tree, (3.7) says one endpoint, say x , is an ancestor of y ; as $(x, y) \notin T$, y is not a child of x , so

$$\text{depth}(y) \geq \text{depth}(x) + 2.$$

But since T is a BFS tree, (3.4) gives $|\text{depth}(x) - \text{depth}(y)| \leq 1$, a contradiction. Hence every edge of G is in T , i.e. $G = T$.

Problem 3.7. Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around (actually, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.) They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy the following property: at all times, each device i is within 500 meters of at least $n/2$ of the other devices. (We'll assume n is an even number.) What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

Here's a concrete way to formulate the question as a claim about graphs.

Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $n/2$, then G is connected.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

Solution. True.

Suppose G is not connected. Its connected components partition V (3.8) into at least two parts, so some component C has $|C| \leq n/2$. Take any $x \in C$; every neighbour of x lies in $C \setminus \{x\}$, so

$$\deg(x) \leq |C| - 1 \leq \frac{n}{2} - 1 < \frac{n}{2},$$

contradicting the hypothesis. Hence G is connected.

3.2 Exercises 3.8–3.12

Problem 3.8. A number of stories in the press about the structure of the Internet and the Web have focused on some version of the following question: How far apart are typical nodes in these networks? If you read these stories carefully, you find that many of them are confused about the difference between the *diameter* of a network and the *average distance* in a network; they often jump

back and forth between these concepts as though they're the same thing.

As in the text, we say that the *distance* between two nodes u and v in a graph $G = (V, E)$ is the minimum number of edges in a path joining them; we'll denote this by $\text{dist}(u, v)$. We say that the *diameter* of G is the maximum distance between any pair of nodes; and we'll denote this quantity by $\text{diam}(G)$.

Let's define a related quantity, which we'll call the *average pairwise distance* in G (denoted $\text{apd}(G)$). We define $\text{apd}(G)$ to be the average, over all $\binom{n}{2}$ sets of two distinct nodes u and v , of the distance between u and v . That is,

$$\text{apd}(G) = \left[\sum_{\{u,v\} \subseteq V} \text{dist}(u, v) \right] / \binom{n}{2}.$$

Here's a simple example to convince yourself that there are graphs G for which $\text{diam}(G) \neq \text{apd}(G)$. Let G be a graph with three nodes u, v, w , and with the two edges $\{u, v\}$ and $\{v, w\}$. Then $\text{diam}(G) = \text{dist}(u, w) = 2$, while

$$\text{apd}(G) = [\text{dist}(u, v) + \text{dist}(u, w) + \text{dist}(v, w)]/3 = 4/3.$$

Of course, these two numbers aren't all *that* far apart in the case of this three-node graph, and so it's natural to ask whether there's always a close relation between them. Here's a claim that tries to make this precise.

Claim: There exists a positive natural number c so that for all connected graphs G , it is the case that

$$\frac{\text{diam}(G)}{\text{apd}(G)} \leq c.$$

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

Solution. False: a huge clique with a long path hanging off it has $\text{apd} \leq 2$ but arbitrarily large diameter. For $k \geq 1$ let $N = 16k^2$ and let G_k consist of a complete graph on nodes $a = c_1, c_2, \dots, c_N$ together with a path $a, p_1, p_2, \dots, p_k$; so $n = N + k$ and G_k is connected. Then $\text{dist}(p_k, c_2) = k + 1$ and no distance exceeds this, so $\text{diam}(G_k) = k + 1$.

The $\binom{N}{2}$ clique pairs have distance 1. The remaining pairs each contain some p_i , so there are at most $kn \leq 2kN$ of them, each at distance at most $k + 1 \leq 2k$. Using $\binom{n}{2} \geq \binom{N}{2} = N(N - 1)/2 \geq N^2/4$,

$$\begin{aligned} \text{apd}(G_k) &\leq \frac{\binom{N}{2}}{\binom{n}{2}} + \frac{2kN \cdot 2k}{\binom{n}{2}} \leq 1 + \frac{4k^2N}{N^2/4} \\ &= 1 + \frac{16k^2}{N} = 2. \end{aligned}$$

Hence $\text{diam}(G_k)/\text{apd}(G_k) \geq (k + 1)/2$, which exceeds any fixed c once $k \geq 2c$.

Problem 3.9. There's a natural intuition that two nodes that are far apart in a communication network—separated by many hops—have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.

Suppose that an n -node undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there must exist some node v , not equal to either s or t , such that deleting v from G destroys all s - t paths. (In other words, the graph obtained from G by deleting v contains no path from s to t .) Give an algorithm with running time $O(m + n)$ to find such a node v .

Solution. Run BFS from s ; some layer strictly between s and t consists of a single node v , and v is the required cut node.

If t is unreachable from s , any $v \neq s, t$ works; otherwise let $d = \text{dist}(s, t) > n/2$ and let $L_0 = \{s\}, L_1, \dots$ be the BFS layers, so $t \in L_d$ by (3.3). Each of $L_1, \dots, L_{d-1}$ is nonempty (the i -th node of a shortest s - t path is at distance i , hence in L_i). If each had at least two nodes, then

$$n \geq |L_0| + |L_d| + \sum_{i=1}^{d-1} |L_i| \geq 2 + 2(d-1) = 2d > n,$$

a contradiction; so some L_i with $1 \leq i \leq d-1$ equals $\{v\}$, and $v \neq s, t$.

Every node of an s - t path $s = x_0, x_1, \dots, x_r = t$ lies in the component of s , hence in some layer (3.3), and by (3.4) consecutive layer indices differ by at most 1. The indices go from 0 to d in steps of size at most 1, so some $x_j \in L_i$, i.e. $x_j = v$. Thus every s - t path uses v , and $G - v$ has no s - t path.

Algorithm. Run BFS from s recording each node's layer (if t is unreached, return any $v \neq s, t$), count $|L_i|$ for every i , and return the unique node of any layer L_i with $1 \leq i \leq d-1$ and $|L_i| = 1$. BFS takes $O(m+n)$ by (3.11), and the counting and scan take $O(n)$.

Problem 3.10. A number of art museums around the country have been featuring work by an artist named Mark Lombardi (1951–2000), consisting of a set of intricately rendered graphs. Building on a great deal of research, these graphs encode the relationships among people involved in major political scandals over the past several decades: the nodes correspond to participants, and each edge indicates some type of relationship between a pair of participants. And so, if you peer closely enough at the drawings, you can trace out ominous-looking paths from a high-ranking U.S. government official, to a former business partner, to a bank in Switzerland, to a shadowy arms dealer.

Such pictures form striking examples of *social networks*, which, as we discussed in Section 3.1, have nodes representing people and organizations, and edges representing relationships of various kinds. And the short paths that abound in these networks have attracted considerable attention recently, as people ponder what they mean. In the case of Mark Lombardi's graphs, they hint at the short set of steps that can carry you from the reputable to the disreputable.

Of course, a single, spurious short path between nodes v and w in such a network may be more coincidental than anything else; a large number of short paths between v and w can be much more convincing. So in addition to the problem of computing a single shortest v - w path in a graph G , social networks researchers have looked at the problem of determining the *number* of shortest v - w paths.

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph $G = (V, E)$, and we identify two nodes v and w in G . Give an algorithm that computes the number of shortest v - w paths in G . (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m+n)$ for a graph with n nodes and m edges.

Solution. Run BFS from v and set $N(v) = 1$ and, for $x \in L_j$ with $j \geq 1$, $N(x) = \sum N(y)$ over the neighbours y of x in L_{j-1} ; output $N(w)$ (and 0 if w is never reached).

```

for each node x: layer[x] = undefined, N[x] = 0
layer[v] = 0; N[v] = 1; Q = queue containing v
while Q is not empty:
    x = Q.dequeue()
    for each neighbour y of x:
        if layer[y] is undefined:
            layer[y] = layer[x] + 1; Q.enqueue(y)
        if layer[y] == layer[x] + 1:
            N[y] = N[y] + N[x]
return N[w]
```

Correctness. Let $P(x)$ be the number of shortest v - x paths. If $x \in L_j$, $j \geq 1$, the node of a shortest v - x path just before x is at distance exactly $j-1$ from v , so it is a neighbour $y \in L_{j-1}$ of x (3.3), and the path minus x is a shortest v - y path; conversely a shortest v - y path followed by the edge (y, x) is a

v - x path of length j . This bijection gives

$$P(x) = \sum_{y \in L_{j-1}, (x,y) \in E} P(y), \quad P(v) = 1,$$

so $N = P$ by induction on j , provided each $N[y]$ is final when y is dequeued. It is: the queue dequeues all of L_{j-1} before any node of L_j (BFS processes layers in order), and $N[x]$ for $x \in L_j$ only receives contributions from dequeued nodes of L_{j-1} , one per edge.

Running time. This is BFS with $O(1)$ extra work per edge scan, so $O(m + n)$ by (3.11), counting each addition as one step.

Problem 3.11. You're helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are n computers in the system, labeled $C_1, C_2, \dots, C_n$, and as input you're given a collection of *trace data* indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples (C_i, C_j, t_k) ; such a triple indicates that C_i and C_j exchanged bits at time t_k . There are m triples total.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing.

The security analysts you're working with would like to be able to answer questions of the following form: If the virus was inserted into computer C_a at time x , could it possibly have infected computer C_b by time y ? The mechanics of infection are simple: if an infected computer C_i communicates with an uninfected computer C_j at time t_k (in other words, if one of the triples (C_i, C_j, t_k) or (C_j, C_i, t_k) appears in the trace data), then computer C_j becomes infected as well, starting at time t_k . Infection can thus spread from one machine to another across a *sequence* of communications, provided that no step in this sequence involves a move backward in time. Thus, for example, if C_i is infected by time t_k , and the trace data contains triples (C_i, C_j, t_k) and (C_j, C_q, t_r) , where $t_k \leq t_r$, then C_q will become infected via C_j . (Note that it is okay for t_k to be equal to t_r ; this would mean that C_j had open connections to both C_i and C_q at the same time, and so a virus could move from C_i to C_q .)

For example, suppose $n = 4$, the trace data consists of the triples

$$(C_1, C_2, 4), (C_2, C_4, 8), (C_3, C_4, 8), (C_1, C_4, 12),$$

and the virus was inserted into computer C_1 at time 2. Then C_3 would be infected at time 8 by a sequence of three steps: first C_2 becomes infected at time 4, then C_4 gets the virus from C_2 at time 8, and then C_3 gets the virus from C_4 at time 8. On the other hand, if the trace data were

$$(C_2, C_3, 8), (C_1, C_4, 12), (C_1, C_2, 14),$$

and again the virus was inserted into computer C_1 at time 2, then C_3 would not become infected during the period of observation: although C_2 becomes infected at time 14, we see that C_3 only communicates with C_2 *before* C_2 was infected. There is no sequence of communications moving forward in time by which the virus could get from C_1 to C_3 in this second example.

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y . The algorithm should run in time $O(m + n)$.

Solution. Sweep the triples in time order, one block of equal timestamps at a time, and inside each block run BFS over that block's communications starting from the machines already infected. (Equal timestamps need a whole block at once: the first example still infects C_3 even if $(C_3, C_4, 8)$ is listed before $(C_2, C_4, 8)$.)

```

if x > y: return false
infected[1..n] = false; infected[a] = true
for each maximal block B of triples sharing one time t, in order:
    if t > y: stop

```

```

    if t < x: skip B
    H = graph on the endpoints of B, one edge per triple of B
    Q = endpoints of B that are infected
    BFS in H from all of Q; mark every node reached infected
    return infected[b]

```

Correctness. Say C_c is infected by time t if there are computers $C_a = C_{c_0}, C_{c_1}, \dots, C_{c_k} = C_c$ ($k \geq 0$) and triples joining $C_{c_{l-1}}$ to C_{c_l} at times t_l with

$$x \leq t_1 \leq t_2 \leq \dots \leq t_k \leq t.$$

We show by induction over the blocks that, once all blocks with time at most t (and at least x) are processed, infected[c] holds exactly when C_c is infected by time t . Before the first block only C_a is marked, which is the case $k = 0$. Now take the block B at time t , and let $t^- < t$ be the previous processed time.

(i) Soundness. Each node newly marked in B is reached in H by a path $u = v_0, v_1, \dots, v_r$ from some u that was already infected by time t^- . Adding the r steps, all at time $t \geq t^-$, to u 's sequence gives a valid sequence for v_r .

(ii) Completeness. Take a valid sequence for C_c with $t_k \leq t$. If $t_k < t$, then C_c is infected by time t^- and was already marked. Otherwise let l be the first index with $t_l = t$. The prefix ending at $C_{c_{l-1}}$ uses only times below t , so $C_{c_{l-1}}$ was marked before B (if $l = 1$, it is C_a). The steps $l, \dots, k$ all happen at time t , so they are edges of H , and $C_{c_{l-1}}, \dots, C_{c_k}$ is a path in H from a node of Q . BFS therefore reaches C_c by (3.5).

The sweep stops before the first block with time above y , so the answer is infected[b] at time y , which is exactly the question asked (for $x > y$ the virus is not yet present at time y , so the answer is no).

Running time. The array takes $O(n)$ to initialize. A block with m_B triples has at most $2m_B$ endpoints, so its adjacency lists can be built in $O(m_B)$, using arrays of length n allocated once and reset through the list of touched endpoints. Its BFS also takes $O(m_B)$ by (3.11). Since $\sum_B m_B \leq m$, the total is

$$O\left(n + \sum_B m_B\right) = O(m + n).$$

Problem 3.12. You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who've lived there over the past two hundred years.

From these interviews, they've learned about a set of n people (all of them now deceased), whom we'll denote $P_1, P_2, \dots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- For some i and j , person P_i died before person P_j was born; or
- for some i and j , the life spans of P_i and P_j overlapped at least partially.

Naturally, they're not sure that all these facts are correct; memories are not so good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.

Solution. Build a directed graph G whose $2n$ nodes are the events b_i (birth of P_i) and d_i (death of P_i), with an edge for each forced “earlier than”. The facts are consistent if and only if G is a DAG, and in that case a topological ordering gives the dates.

The edges, with m the number of facts, are:

- (b_i, d_i) for every i ;
- (d_i, b_j) for each fact “ P_i died before P_j was born”;
- (b_i, d_j) and (b_j, d_i) for each fact “ P_i and P_j overlapped”.

So G has $2n$ nodes and at most $n + 2m$ edges. Run the $O(m + n)$ topological-ordering algorithm of Section 3.6 on G . If it deletes all $2n$ nodes and produces the ordering $v_1, \dots, v_{2n}$, give the event v_k the date k . If it gets stuck with active nodes left over, report that the facts are inconsistent.

Dates output are valid. A topological ordering sends every edge forward, so every edge (u, v) gets $\text{date}(u) < \text{date}(v)$. In particular each $b_i < d_i$. Each “died before” fact gets $d_i < b_j$. Each overlap fact gets $b_i < d_j$ and $b_j < d_i$, which is precisely the condition that the intervals $[b_i, d_i]$ and $[b_j, d_j]$ share a nonempty open interval:

$$\max(b_i, b_j) < \min(d_i, d_j).$$

Reported inconsistency is correct. If the algorithm gets stuck, then the remaining subgraph has no source, so by (3.19) it is not a DAG and G contains a cycle C . Suppose some dates satisfied all the facts. Even reading lives and overlaps weakly ($b_i \leq d_i$, and $b_i \leq d_j, b_j \leq d_i$ for an overlap), every edge (u, v) would satisfy $\text{date}(u) \leq \text{date}(v)$. Also, every edge out of a death node is a “died before” edge, which forces a strict inequality $d_i < b_j$. Edges out of birth nodes all go to death nodes, so C must enter a death node and then leave it along a strict edge. Going around C then gives

$$\text{date}(u) < \text{date}(u)$$

for any node u of C , which is impossible. So no consistent dates exist.

Running time. Building G takes $O(n + m)$. The topological ordering, done with active-in-degree counts and the set S of active sources, runs in $O(|V| + |E|) = O(n + m)$ (Section 3.6). Detecting a failure is the case where S empties before all nodes are deleted. In total the algorithm runs in

$$O(n + m).$$

4 Greedy Algorithms

Contents

4.1 Exercises 4.1–4.7	28
4.2 Exercises 4.8–4.14	32
4.3 Exercises 4.15–4.21	37
4.4 Exercises 4.22–4.28	41
4.5 Exercises 4.29–4.33	46

4.1 Exercises 4.1–4.7

Problem 4.1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary connected, undirected graph with a distinct cost $c(e)$ on every edge e . Suppose e^* is the cheapest edge in G ; that is, $c(e^*) < c(e)$ for every edge $e \neq e^*$. Then there is a minimum spanning tree T of G that contains the edge e^* .

Solution. True. Write $e^* = (v, w)$ and take $S = \{v\}$, which is neither empty nor all of V (since $w \notin S$). The edge e^* has one end in S and the other in $V - S$, and it is cheaper than every other edge of G , so in particular it is the cheapest edge with one end in S and the other in $V - S$. The costs are distinct, so the Cut Property (4.17) applies: every minimum spanning tree of G contains e^* . Since G is connected, a minimum spanning tree exists, and any one of them is the required T .

Problem 4.2. For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G , with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2 , thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

(b) Suppose we are given an instance of the Shortest s - t Path Problem on a directed graph G . We assume that all edge costs are positive and distinct. Let P be a minimum-cost s - t path for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2 , thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost s - t path for this new instance.

Solution. (a) True. Let K be the tree Kruskal's Algorithm outputs for the costs c_e . The proof of (4.18) shows that each edge Kruskal adds is the cheapest edge crossing some cut, so by the Cut Property (4.17) (costs distinct) it lies in every minimum spanning tree. Hence $K \subseteq T$, and since both have $n - 1$ edges, $T = K$.

Kruskal's Algorithm only compares costs, and $x \mapsto x^2$ is strictly increasing on positive numbers:

$$c_e < c_f \iff c_e^2 < c_f^2 \quad (c_e, c_f > 0).$$

So on the squared costs (still distinct) Kruskal considers the edges in the same order, makes the same decisions, and outputs the same tree $K = T$, which is a minimum spanning tree for the new instance by (4.18).

(b) False. Take nodes s, a, t and directed edges

$$c_{(s,t)} = 4, \quad c_{(s,a)} = 2, \quad c_{(a,t)} = 3.$$

The only s - t paths are $P = s \rightarrow t$ and $Q = s \rightarrow a \rightarrow t$. Originally P is the unique minimum, since $c(P) = 4 < 5 = 2 + 3 = c(Q)$. After squaring,

$$c^2(P) = 16 > 13 = 4 + 9 = c^2(Q),$$

so P is no longer a minimum-cost s - t path.

Problem 4.3. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by

one, and each package i has a weight w_i . The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it “stays ahead” of all other solutions.

Solution. The measure is the number of boxes shipped by the first r trucks: greedy’s first r trucks always carry at least as many boxes as the first r trucks of any other valid packing.

Number the boxes $1, \dots, n$ in arrival order (each $w_i \leq W$, else no packing exists). Because boxes ship in order, any packing with k trucks is described by indices $0 = b_0 < b_1 < \dots < b_k = n$, truck r carrying boxes $b_{r-1} + 1, \dots, b_r$, with $\sum_{i=b_{r-1}+1}^{b_r} w_i \leq W$. Let greedy’s indices be $0 = g_0 < g_1 < \dots < g_q = n$; since greedy dispatches a truck only when the next box does not fit,

$$g_r = \max \left\{ j : \sum_{i=g_{r-1}+1}^j w_i \leq W \right\}.$$

Claim: $g_r \geq b_r$ for every $r \leq \min(k, q)$ (with $g_r := n$ once greedy has finished). Induction, as in (4.2): $g_0 = b_0$. Suppose $g_{r-1} \geq b_{r-1}$. (i) If $b_r \leq g_{r-1}$, then $g_r \geq g_{r-1} \geq b_r$. (ii) If $b_r > g_{r-1}$, the boxes $g_{r-1} + 1, \dots, b_r$ are a subset of truck r ’s load $b_{r-1} + 1, \dots, b_r$, so

$$\sum_{i=g_{r-1}+1}^{b_r} w_i \leq \sum_{i=b_{r-1}+1}^{b_r} w_i \leq W,$$

and the maximality of g_r gives $g_r \geq b_r$.

Now let the packing b be optimal with k trucks and suppose $q > k$. Then $g_k < n$ (greedy needs truck $k + 1$), yet the claim gives $g_k \geq b_k = n$, a contradiction. Hence $q \leq k$: greedy uses the minimum number of trucks.

Problem 4.4. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges (what’s being bought) are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain “patterns” in them; for example, they may want to know if the four events

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

occur in this sequence S , in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S' . So, for example, the sequence of four events above is a subsequence of the sequence

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S . So this is the problem they pose to you: Give an algorithm that takes two sequences of events,

S' of length m and S of length n , each possibly containing an event more than once, and decides in time $O(m + n)$ whether S' is a subsequence of S .

Solution. Scan S once from left to right, matching each event of S' to its earliest possible occurrence in S after the previous match.

```

i = 1
for j = 1 to n:
    if i <= m and S[j] = S'[i]:
        i = i + 1      (S'[i] is matched to position j)
return (i > m)

```

If the algorithm answers yes, the matched positions $g_1 < g_2 < \dots < g_m$ satisfy $S[g_r] = S'[r]$, so deleting all other events of S leaves S' . Conversely, suppose S' is a subsequence of S via positions $k_1 < \dots < k_m$ with $S[k_r] = S'[r]$. The greedy match stays ahead: we claim it matches $S'[r]$ at some $g_r \leq k_r$. For $r = 1$, g_1 is the first occurrence of $S'[1]$ in S , so $g_1 \leq k_1$. If $g_{r-1} \leq k_{r-1}$, then position k_r satisfies

$$k_r > k_{r-1} \geq g_{r-1}, \quad S[k_r] = S'[r],$$

so the scan, which looks for $S'[r]$ from position $g_{r-1} + 1$ onward, meets it at some $g_r \leq k_r \leq n$. By induction all m events are matched and the algorithm answers yes.

Running time: the loop runs n times with $O(1)$ work per iteration (one comparison of events), and the index i only increases up to $m + 1$; reading the inputs costs $O(m + n)$, so the total is $O(m + n)$.

Problem 4.5. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Solution. Sweep from west to east; whenever a house is not yet covered, place a station four miles east of it.

Measure positions in miles from the western endpoint and sort the houses, $x_1 \leq x_2 \leq \dots \leq x_n$.

```

covered = -infinity
for i = 1 to n:
    if x[i] > covered:
        place a station at s = x[i] + 4
        covered = x[i] + 8

```

(If $x_i + 4$ lies beyond the eastern endpoint E , the station goes at E instead, still covering $[x_i, E]$.) Each house is either covered when examined, or becomes covered by the station placed for it, so the output is valid.

Let greedy place stations $s_1 < \dots < s_q$, triggered by houses $y_1 < \dots < y_q$, so $s_r = \min(y_r + 4, E)$, and let $t_1 \leq \dots \leq t_k$ be any valid placement. Claim: for every $r \leq q$ we have $k \geq r$ and $t_r \leq s_r$. Suppose it holds for $r - 1$ (vacuous for $r = 1$). By construction y_r is not covered by s_{r-1} , i.e. $y_r > s_{r-1} + 4$, and therefore, for $j \leq r - 1$,

$$y_r - t_j \geq y_r - t_{r-1} \geq y_r - s_{r-1} > 4.$$

So y_r is covered by none of $t_1, \dots, t_{r-1}$, but by some t_j with $j \geq r$; hence $k \geq r$ and

$$t_r \leq t_j \leq \min(y_r + 4, E) = s_r.$$

Taking $r = q$ gives $k \geq q$: no valid placement uses fewer stations than greedy.

Running time: $O(n \log n)$ to sort the houses, then $O(n)$ for the sweep (constant work per house).

Problem 4.6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming ... and so on. Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

Solution. Send the contestants out in decreasing order of biking time plus running time.

Write s_i, b_i, r_i for contestant i 's three times and $t_i = b_i + r_i$. Sort the contestants so that $t_{i_1} \geq t_{i_2} \geq \dots \geq t_{i_n}$ and start them in that order, each entering the pool the moment the previous swimmer leaves.

Correctness. There is never a reason to leave the pool idle, so in the order $i_1, \dots, i_n$ contestant i_k finishes at time $S_k + t_{i_k}$, where $S_k = s_{i_1} + \dots + s_{i_k}$, and the completion time is $\max_k (S_k + t_{i_k})$. Call a pair an *inversion* if i goes before j but $t_i < t_j$; as in the proof of (4.9), an order with an inversion has an adjacent inversion. Swapping an adjacent inversion i, j that starts swimming at time S leaves everyone else's finishing time unchanged, and replaces the two finishing times

$$\begin{aligned} \text{before: } & S + s_i + t_i, \quad S + s_i + s_j + t_j, \\ \text{after: } & S + s_j + t_j, \quad S + s_i + s_j + t_i, \end{aligned}$$

and both "after" values are at most $S + s_i + s_j + t_j$ (using $t_i < t_j$). So the swap does not increase the completion time and strictly decreases the number of inversions. Starting from an optimal order and swapping repeatedly therefore reaches an order with no inversions and completion time no larger. All orders with no inversions differ only in the arrangement of contestants with equal t , and these have the same completion time (swapping an adjacent pair with $t_i = t_j$ leaves the larger of the two finishing times at $S + s_i + s_j + t_i$, by the display). Hence the greedy order is optimal.

Running time. $O(n \log n)$, for sorting by t_i .

Problem 4.7. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \dots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be *preprocessed* on the supercomputer, and then it needs to be *finished* on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel: all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be

handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

Solution. Feed the jobs to the supercomputer in decreasing order of finishing time f_i .

This is Exercise 4.6 with the pool replaced by the supercomputer ($s_i = p_i$) and the bike-plus-run time replaced by the PC time ($t_i = f_i$): in the order $i_1, \dots, i_n$ job J_{i_k} is done at $P_k + f_{i_k}$, where $P_k = p_{i_1} + \dots + p_{i_k}$, and the completion time is $\max_k(P_k + f_{i_k})$. Swapping an adjacent pair i, j with $f_i < f_j$ that starts on the supercomputer at time P changes only their two completion times,

$$\begin{aligned} \max(P + p_i + f_i, P + p_i + p_j + f_j) &\geq P + p_i + p_j + f_j \\ &\geq \max(P + p_j + f_j, P + p_i + p_j + f_i), \end{aligned}$$

so it never increases the completion time, and swaps between equal f 's leave it unchanged. Removing the inversions of an optimal order one adjacent swap at a time, as in the proof of (4.9), reaches the sorted order with no larger completion time, so the sorted order is optimal.

Running time. $O(n \log n)$, for the sort.

4.2 Exercises 4.8–4.14

Problem 4.8. Suppose you are given a connected graph G , with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution. Every edge of a minimum spanning tree T lies in every minimum spanning tree, so all minimum spanning trees coincide with T .

A minimum spanning tree exists since G is connected. Fix one, T , and an edge $e = (v, w) \in T$. Deleting e splits T into two components; let S be the node set of the one containing v , so $w \in V - S$. If some edge $f \neq e$ with one end in S and the other in $V - S$ had $c_f < c_e$, then $T - \{e\} + \{f\}$ would be connected (as f joins the two components), have $n - 1$ edges, and so be a spanning tree with

$$c(T - \{e\} + \{f\}) = c(T) - c_e + c_f < c(T),$$

contradicting the minimality of T . Hence e is the cheapest edge with one end in S and the other in $V - S$ (strictly, by distinctness), and the Cut Property (4.17) says e belongs to every minimum spanning tree. Thus $T \subseteq T'$ for every minimum spanning tree T' , and since both have $n - 1$ edges, $T' = T$.

Problem 4.9. One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G ; we define the *bottleneck edge* of T to be the edge of T with the greatest cost.

A spanning tree T of G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G ? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G ? Prove or give a counterexample.

Solution. (a) No. Take nodes a, b, c, d and edges

$$c_{ab} = 1, \quad c_{bc} = 2, \quad c_{ac} = 3, \quad c_{cd} = 4.$$

Every spanning tree contains the only edge cd at d , so every spanning tree has bottleneck cost 4 and every spanning tree is a minimum-bottleneck tree. In particular $T = \{ab, ac, cd\}$ is one, but it costs 8, while $\{ab, bc, cd\}$ costs 7; so T is not a minimum spanning tree.

(b) Yes. Let T be a minimum spanning tree with bottleneck edge e , and suppose some spanning tree T' has every edge cheaper than c_e . Deleting e from T splits it into two components with node sets S and $V - S$. Since T' is connected, it has an edge f with one end in S and the other in $V - S$, and $c_f < c_e$. Then $T - \{e\} + \{f\}$ is connected with $n - 1$ edges, hence a spanning tree, and

$$c(T - \{e\} + \{f\}) = c(T) - c_e + c_f < c(T),$$

contradicting the minimality of T . So no spanning tree has a cheaper bottleneck edge than T .

Problem 4.10. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G . Now assume that a new edge is added to G , connecting two nodes $v, w \in V$ with cost c .

(a) Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree T and the graph G .

(b) Suppose T is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree T to the new minimum-cost spanning tree.

Solution. (a) T remains a minimum spanning tree if and only if $c \geq c_f$, where f is the most expensive edge on the v - w path P in T ; this is testable in $O(|V|)$ time.

Assumption. T is stored as adjacency lists, each entry carrying the edge's cost (the representation of G is never used). Run DFS (or BFS) in T from v , recording for each discovered node its parent edge; when w is found, walk the parent pointers back to v , which traverses P , and keep the most expensive edge f . Answer "yes" iff $c \geq c_f$. Since T has $|V| - 1$ edges, DFS takes $O(|V|)$ time (3.13), and the walk back $O(|V|)$; this is $O(|E|)$ as well, since G is connected.

Correctness. Write e for the new edge and $G' = G + e$. (i) If $c < c_f$: $T - \{f\} + \{e\}$ is connected (deleting f splits T into two components separating v from w , and e rejoins them) with $|V| - 1$ edges, so it is a spanning tree of G' of cost $c(T) - c_f + c < c(T)$, and T is not minimum. (ii) If $c \geq c_f$, so $c \geq c_g$ for every $g \in P$: let T^* be a minimum spanning tree of G' . If $e \notin T^*$, then T^* is a spanning tree of G and $c(T) \leq c(T^*)$. If $e \in T^*$, deleting e splits T^* into components with node sets $S \ni v$ and $V - S \ni w$; the path P runs from S to $V - S$, so some $g \in P$ has one end in each, and $T^* - \{e\} + \{g\}$ is a spanning tree of G , whence

$$c(T) \leq c(T^*) - c + c_g \leq c(T^*).$$

Either way $c(T) \leq c(T^*)$, so T is a minimum spanning tree of G' .

(b) Replace T by $T' = T - \{f\} + \{e\}$, where f is the edge found in (a).

Correctness. Here $c < c_f$. T' is a spanning tree of G' by (a)(i), of cost $c(T') = c(T) - c_f + c$. Let T'' be any spanning tree of G' . (i) If $e \notin T''$: $c(T'') \geq c(T) > c(T')$. (ii) If $e \in T''$: let $\hat{G}$ be G plus a copy $\hat{e}$ of the edge (v, w) with cost c_f instead of c . Since c_f is the largest cost on P , part (a)(ii) shows T is a minimum spanning tree of $\hat{G}$. The tree $\hat{T}''$ obtained from T'' by using $\hat{e}$ in place of e is a spanning tree of $\hat{G}$, so

$$c(T'') - c + c_f = c(\hat{T}'') \geq c(T) = c(T') - c + c_f,$$

i.e. $c(T'') \geq c(T')$. So T' is a minimum spanning tree of G' .

Running time. Finding f is the $O(|V|)$ search of (a); deleting f from the two adjacency lists of its ends and inserting e take $O(|V|)$ more. The total $O(|V|) \subseteq O(|E|)$.

Problem 4.11. Suppose you are given a connected graph $G = (V, E)$, with a cost c_e on each edge e . In an earlier problem, we saw that when all edge costs are distinct, G has a unique minimum spanning tree. However, G may have many minimum spanning trees when the edge costs are not all distinct. Here we formulate the question: Can Kruskal's Algorithm be made to find all the minimum spanning trees of G ?

Recall that Kruskal's Algorithm sorted the edges in order of increasing cost, then greedily processed edges one by one, adding an edge e as long as it did not form a cycle. When some edges have the same cost, the phrase "in order of increasing cost" has to be specified a little more carefully: we'll say that an ordering of the edges is *valid* if the corresponding sequence of edge costs is nondecreasing. We'll say that a *valid execution* of Kruskal's Algorithm is one that begins with a valid ordering of the edges of G .

For any graph G , and any minimum spanning tree T of G , is there a valid execution of Kruskal's Algorithm on G that produces T as output? Give a proof or a counterexample.

Solution. Yes: sort the edges by cost and, among edges of equal cost, put the edges of T before the edges not in T . This ordering is valid; we show Kruskal's Algorithm run on it accepts exactly the edges of T .

Suppose not, and look at the first edge on which the run deviates from T (accepting an edge outside T or rejecting an edge of T). Up to that point the accepted edges are exactly the edges of T processed so far.

(i) An edge $e \in T$ is rejected. Then e closes a cycle with the accepted edges, which all lie in T ; but T is acyclic. Contradiction.

(ii) An edge $f = (u, v) \notin T$ is accepted. Adding f to T creates a cycle C . For each $e \in C - f$, the graph $T - e + f$ is again a spanning tree, so minimality of T gives

$$c(T) \leq c(T) - c_e + c_f, \quad \text{i.e.} \quad c_e \leq c_f.$$

So every edge of the u - v path $C - f$ in T has cost at most c_f , and by the tie-breaking rule it comes before f in the ordering; by the choice of f as the first deviation, all of these edges were accepted. Hence u and v were already connected by accepted edges, and Kruskal's Algorithm rejects f . Contradiction. So the run accepts every edge of T and no other edge, and its output is T .

Problem 4.12. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a *schedule* for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^n t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

Now, because you're just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter r :

(*) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt .

Note that this constraint is only imposed for time intervals that start at 0, *not* for time intervals that start at any other value. We say that a schedule is *valid* if it satisfies the constraint (*) imposed by the link.

The Problem. Given a set of n streams, each specified by its number of bits b_i and its time duration t_i , as well as the link parameter r , determine whether there exists a valid schedule.

Example. Suppose we have $n = 3$ streams, with $(b_1, t_1) = (2000, 1)$, $(b_2, t_2) = (6000, 2)$, $(b_3, t_3) = (2000, 1)$, and suppose the link's parameter is $r = 5000$. Then the schedule that runs the streams in the order 1, 2, 3 is valid, since the constraint (*) is satisfied: at $t = 1$ the whole first stream has been sent, and $2000 < 5000 \cdot 1$; at $t = 2$ half of the second stream has also been sent, and $2000 + 3000 < 5000 \cdot 2$. Similar calculations hold for $t = 3$ and $t = 4$.

(a) Consider the following claim:

Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i , as well as the link parameter r , and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n .

Solution. (a) False. Take $n = 2$, $r = 2$, $(b_1, t_1) = (1, 1)$ and $(b_2, t_2) = (3, 1)$. The order 1, 2 sends

$$1 \leq 2 \cdot 1 \quad (t = 1), \quad 1 + 3 = 4 \leq 2 \cdot 2 \quad (t = 2),$$

and nothing more is sent for $t \geq 2$, so the schedule is valid; yet $b_2 = 3 > 2 = rt_2$.

(b) A valid schedule exists if and only if

$$\sum_{i=1}^n b_i \leq r \sum_{i=1}^n t_i,$$

and then sending the streams in increasing order of rate b_i/t_i is valid. So the algorithm computes the two sums and compares them.

Necessity: put $T = \sum_i t_i$, a natural number > 0 . Every schedule has sent all $\sum_i b_i$ bits by time T , so (*) at $t = T$ is exactly the inequality.

Sufficiency: let $B(t)$ be the number of bits sent by time t in the increasing-rate schedule. It is piecewise linear with $B(0) = 0$, and its slope on successive pieces is b_i/t_i in nondecreasing order, so B is convex on $[0, T]$. For $0 < t \leq T$, convexity between the points 0 and T gives

$$B(t) \leq \left(1 - \frac{t}{T}\right)B(0) + \frac{t}{T}B(T) = \frac{t}{T} \sum_i b_i \leq \frac{t}{T} \cdot rT = rt,$$

and for $t > T$ we have $B(t) = B(T) \leq rT < rt$. So (*) holds for every $t > 0$.

The test takes $O(n)$ time (producing the schedule itself means sorting by rate, $O(n \log n)$).

Problem 4.13. A small business — say, a photocopying service with a single large machine — faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i 's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i . For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i , we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i 's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^n w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^n w_i C_i$.

Example. Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

Solution. Do the jobs in increasing order of the ratio t_i/w_i (weights positive), breaking ties arbitrarily. Call a pair i, j an *inversion* of a schedule if i is done before j but $t_i/w_i > t_j/w_j$. Swapping two adjacent jobs i, j (with i first, starting at time S) changes only C_i and C_j , and

$$\begin{aligned} & [w_i(S + t_i) + w_j(S + t_i + t_j)] \\ & - [w_j(S + t_j) + w_i(S + t_j + t_i)] \\ & = w_j t_i - w_i t_j = w_i w_j \left(\frac{t_i}{w_i} - \frac{t_j}{w_j} \right). \end{aligned}$$

So swapping an adjacent inversion strictly lowers the cost, and swapping two adjacent jobs of equal ratio leaves it unchanged.

Take an optimal schedule O . If it has an inversion, then (exactly as in the proof of (4.9)) it has an adjacent one, and swapping it would strictly lower the cost, contradicting optimality; so O has no inversions. Any two schedules without inversions differ only in the order of jobs with equal ratio, which form a contiguous block in each (as in the proof of (4.8)), and one is turned into the other by adjacent swaps within these blocks, each of cost change 0. Hence the greedy schedule has the same cost as O and is optimal.

Sorting takes $O(n \log n)$ time and computing the C_i takes $O(n)$.

Problem 4.14. You're working with a group of security consultants who are helping to monitor a large computer system. There's particular interest in keeping track of processes that are labeled "sensitive." Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a program called `status_check` that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of `status_check` as lasting for only this single point in time.) What they'd like to do is to run `status_check` as few times as possible during the day, but enough that for each sensitive process P , `status_check` is invoked at least once during the execution of process P .

(a) Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possible at which to invoke `status_check`, subject to the requirement that `status_check` is invoked at least once during each sensitive process P .

(b) While you were designing your algorithm, the security consultants were engaging in a little back-of-the-envelope reasoning. "Suppose we can find a set of k sensitive processes with the property that no two are ever running at the same time. Then clearly your algorithm will need to invoke `status_check` at least k times: no one invocation of `status_check` can handle more than one of these processes."

This is true, of course, and after some further discussion, you all begin wondering whether something stronger is true as well, a kind of converse to the above argument. Suppose that k^* is the largest value of k such that one can find a set of k sensitive processes with no two ever running at the same time. Is it the case that there must be a set of k^* times at which you can run `status_check` so that some invocation occurs during the execution of each sensitive process? (In other words, the kind of argument in the previous paragraph is really the only thing forcing you to need a lot of invocations of `status_check`.) Decide whether you think this claim is true or false, and give a proof or a counterexample.

Solution. (a) Scan the processes in order of finish time, and whenever a process is not yet covered, invoke `status_check` at its finish time. Process P runs over the closed interval $[s_P, f_P]$.

```
sort processes so that f(1) <= f(2) <= ... <= f(n)
p = -infinity; times = empty
for j = 1 to n:
    if s(j) > p:                (process j not yet covered)
        add f(j) to times; p = f(j)
```

return times

The output covers every process: when j is reached either $s_j > p$ and f_j is added, or $s_j \leq p \leq f_j$ (as p is the finish time of an earlier process in the sorted order), so p lies in $[s_j, f_j]$; and since every invocation time is at most the current p , the test $s_j > p$ correctly detects that j is uncovered.

Let $j_1, \dots, j_k$ be the processes whose finish times were chosen, so $|\text{times}| = k$. When j_{m+1} was chosen, the current p was f_{j_m} , so

$$s_{j_{m+1}} > f_{j_m} \quad (1 \leq m < k),$$

i.e. $j_1, \dots, j_k$ are pairwise never running at the same time. One invocation lies in at most one of them, so every valid set of times has size at least k , and the output is optimal.

The sort costs $O(n \log n)$ and the scan $O(n)$.

(b) True. With k and $j_1, \dots, j_k$ as in (a), let OPT be the minimum number of invocations. By the consultants' argument and by (a),

$$k \leq k^* \leq \text{OPT} = k,$$

so $k^* = \text{OPT}$: the k^* times returned by the algorithm in (a) cover every process.

4.3 Exercises 4.15–4.21

Problem 4.15. The manager of a large student union on campus comes to you with the following problem. She's in charge of a group of n students, each of whom is scheduled to work one *shift* during the week. There are different jobs associated with these shifts (tending the main desk, helping with package delivery, rebooting cranky information kiosks, etc.), but we can view each shift as a single contiguous interval of time. There can be multiple shifts going on at once.

She's trying to choose a subset of these n students to form a *supervising committee* that she can meet with once a week. She considers such a committee to be *complete* if, for every student not on the committee, that student's shift overlaps (at least partially) the shift of some student who is on the committee. In this way, each student's performance can be observed by at least one person who's serving on the committee.

Give an efficient algorithm that takes the schedule of n shifts and produces a complete supervising committee containing as few students as possible.

Example. Suppose $n = 3$, and the shifts are Monday 4 P.M.–Monday 8 P.M., Monday 6 P.M.–Monday 10 P.M., and Monday 9 P.M.–Monday 11 P.M. Then the smallest complete supervising committee would consist of just the second student, since the second shift overlaps both the first and the third.

Solution. Repeatedly take the uncovered shift I that finishes earliest, and put on the committee the student whose shift, among all shifts overlapping I (including I itself), finishes latest. Shift $J = [s_J, f_J]$ covers L if $L = J$ or they overlap, i.e. $s_L \leq f_J$ and $s_J \leq f_L$; a shift is *uncovered* if no committee member covers it.

```
C = empty
while some shift is uncovered by C:
    I = the uncovered shift with smallest finish time
    K = the shift covering I with largest finish time
    add K to C
return C
```

Key fact: at a step with I, K as above, if J covers I and L is uncovered and covered by J , then K covers L . Indeed $f_L \geq f_I$ (choice of I) and $f_K \geq f_J$ (choice of K , since J covers I), so

$$s_K \leq f_I \leq f_L, \quad s_L \leq f_J \leq f_K.$$

Let the greedy committee be $K_1, \dots, K_m$. We show by induction on j that some minimum complete committee contains $K_1, \dots, K_j$; the case $j = 0$ is trivial. Let O be a minimum complete committee containing $K_1, \dots, K_{j-1}$, and let I be the shift chosen at step j . As O is complete, some $J \in O$ covers I , and $J \notin \{K_1, \dots, K_{j-1}\}$ since I was uncovered. Put $O' = O - J + K_j$, so $|O'| \leq |O|$. A shift covered by $K_1, \dots, K_{j-1}$, or by a member of O other than J , is still covered in O' ; any other shift L (including $L = J$) is uncovered at step j and covered by J , hence covered by K_j by the key fact. So O' is a minimum complete committee containing $K_1, \dots, K_j$.

For $j = m$ this gives a minimum complete committee $O \supseteq \{K_1, \dots, K_m\}$, so $m \leq |O|$, and the greedy committee (complete when the loop stops) is optimal. Each round takes $O(n)$ time and covers at least one new shift, so there are at most n rounds and the total is $O(n^2)$.

Problem 4.16. Some security consultants are investigating a potential money-laundering scheme. The evidence indicates that n suspicious transactions took place in recent days, each involving money transferred into a single account; they do not know the identity of the account, the amounts, or the exact times. What they have is an *approximate time-stamp* for each transaction: transaction i took place at time $t_i \pm e_i$, for some “margin of error” e_i (that is, sometime between $t_i - e_i$ and $t_i + e_i$). Different transactions may have different margins of error. They have come across a bank account that they suspect might be the one involved. There are n recent *events* involving the account, which took place at times $x_1, x_2, \dots, x_n$. They want to know whether it is possible to associate each of the account’s n events with a distinct one of the n suspicious transactions in such a way that, if the account event at time x_i is associated with the suspicious transaction that occurred approximately at time t_j , then $|t_j - x_i| \leq e_j$. (The tricky part is that they don’t know which account event to associate with which suspicious transaction.) Give an efficient algorithm that takes the given data and decides whether such an association exists. If possible, the running time should be at most $O(n^2)$.

Solution. Process the events in increasing order of time, and give each event the still-unused transaction window that contains it and has the earliest right endpoint. Write $I_j = [\ell_j, r_j] = [t_j - e_j, t_j + e_j]$; event x may be associated with transaction j exactly when $x \in I_j$.

```

sort the events so that  $x_1 \leq x_2 \leq \dots \leq x_n$ 
mark every window  $I_j$  as unused
for  $i = 1$  to  $n$ :
    among unused  $j$  with  $\ell_j \leq x_i \leq r_j$ , pick  $j$  with the smallest  $r_j$ 
    if there is no such  $j$ : return "no association"
    associate  $x_i$  with  $j$ ; mark  $j$  as used
return "association exists"
```

If the algorithm answers “exists”, it has built an association, so it is right. For the converse, we show by induction on k : if some valid association exists, then some valid association M agrees with the algorithm’s first k choices. The case $k = 0$ is immediate. Now let M agree with the first $k - 1$ choices. Since M gives x_k an unused window containing it, the algorithm finds some window at step k . Call it I_j , and say M gives x_k the window $I_{j'} \neq I_j$ and gives I_j to the event x_m . Window I_j is unused before step k , so $m > k$ and $x_m \geq x_k$. The greedy choice gives $r_j \leq r_{j'}$, so

$$\ell_{j'} \leq x_k \leq x_m \leq r_j \leq r_{j'},$$

which means $x_m \in I_{j'}$, and of course $x_k \in I_j$. Swapping the two windows between x_k and x_m therefore gives a valid association that agrees with the first k choices. So when an association exists the algorithm never gets stuck, and it answers “exists”.

Running time: sorting takes $O(n \log n)$, and each of the n iterations scans the n windows in $O(n)$, for $O(n^2)$ in total.

Problem 4.17. Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*. If the job is accepted, it must run continuously, every day, for the period between its start and end times. (Some jobs begin before midnight and end after midnight, which makes this different from the Interval Scheduling Problem.) Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Give an algorithm that does this in time polynomial in n . You may assume that no two jobs have the same start or end times.

Example. Consider the four jobs, given as (start-time, end-time) pairs, (6 P.M., 6 A.M.), (9 P.M., 4 A.M.), (3 A.M., 2 P.M.), (1 P.M., 7 P.M.). The optimal solution picks the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

Solution. Try each job J in turn as a job that is definitely accepted. Cutting the 24-hour circle open at J turns the remaining compatible jobs into an ordinary interval scheduling instance, which the earliest-finish-time greedy algorithm of Section 4.1 solves. Return the best of these n schedules.

```

best = empty set                                (the answer when  $n = 0$ )
for each job  $J = (s_J, f_J)$ :
     $R$  = jobs that do not overlap  $J$ 
    lay out the circle as the line from  $f_J$  forward to  $s_J$ ;
        each job in  $R$  becomes an ordinary interval on this line
     $A$  = earliest-finish-time greedy schedule on  $R$ 
    if  $|A| + 1 > |\text{best}|$ : best =  $A$  plus  $J$ 
return best

```

The jobs that do not overlap J occupy arcs inside the arc of the day running from f_J forward to s_J . Measuring time forward from f_J makes this arc a line segment, and each such job becomes an interval on it. So R is an ordinary interval scheduling instance, and every candidate $A \cup \{J\}$ is a set of pairwise compatible daily jobs. Now let O be an optimal set, with $n \geq 1$, and pick any $J \in O$. Then $O \setminus \{J\} \subseteq R$ is compatible, so by (4.3) the greedy set has $|A| \geq |O| - 1$. Hence the candidate for this J already has size $|O|$, and the returned set is optimal.

Running time: each of the n rounds builds R in $O(n)$ and runs the greedy algorithm in $O(n \log n)$, for $O(n^2 \log n)$ in total. Sorting all jobs once by end time and scanning that circular order starting from f_J makes each round $O(n)$, for $O(n^2)$.

Problem 4.18. Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They have drawn up a directed graph whose nodes represent intermediate destinations and whose edges represent the roads between them. Extreme weather makes roads slow in winter, and the speed of travel depends on the time of year. A travel Web site answers queries of the following form: given an edge $e = (v, w)$ connecting two sites v and w , and a proposed starting time t from location v , the site returns a value $f_e(t)$, the predicted arrival time at w . The site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (you do not arrive earlier by starting later). Other than that, the functions f_e may be arbitrary. For example, where the travel time does not vary with the season, $f_e(t) = t + \ell_e$, where ℓ_e is the time needed to traverse e .

Your friends want to use the site to find the fastest way to travel through the directed graph from their starting point s to their intended destination. Assume that they start at time 0 and that all predictions are completely correct. Give a polynomial-time algorithm for this, counting a single query to the Web site (for a specific edge e and time t) as one computational step.

Solution. Run Dijkstra's Algorithm with the label update $d(u) + \ell_e$ replaced by $f_e(d(u))$. Here $d(u)$ is the earliest arrival time at u , and each edge is queried once, at the moment its tail u is finalized.

```

 $S = \{s\}$ ;  $d(s) = 0$ 
while some node outside  $S$  has an edge into it from  $S$ :
    for each  $v$  not in  $S$ :  $d'(v) = \min$  of  $f_e(d(u))$  over  $e = (u, v)$ ,  $u$  in  $S$ 
    pick  $v$  not in  $S$  minimizing  $d'(v)$ 
    add  $v$  to  $S$ ;  $d(v) = d'(v)$ ;  $\text{pred}(v)$  = the minimizing edge
return  $d(\text{dest})$ , with the route read off the  $\text{pred}$  pointers

```

We claim, as in (4.14), that for every $u \in S$ the pred-path P_u reaches u at time $d(u)$, and that every s - u path reaches u no earlier than $d(u)$. Argue by induction on $|S|$. The case $|S| = 1$ holds because times never decrease. Suppose v is added through $e = (u, v)$. Then P_u followed by e reaches v at $f_e(d(u)) = d(v)$. Now take any s - v path P , and let (x, y) be its first edge leaving S , so $x \in S$ and $y \notin S$. By induction, P reaches x at some time $\tau \geq d(x)$. Monotonicity of $f_{(x,y)}$ and the choice of v give

$$\text{arrival}_P(y) = f_{(x,y)}(\tau) \geq f_{(x,y)}(d(x)) \geq d'(y) \geq d'(v) = d(v),$$

and the rest of P , from y to v , can only increase the time because $f_e(t) \geq t$. So P reaches v no earlier than $d(v)$. Nodes never added to S are unreachable from s . In particular, P_{dest} is a fastest route. Running time: implement the algorithm exactly as Dijkstra's Algorithm with a priority queue (4.15). When u joins S , query $f_e(d(u))$ once for each edge e out of u and do a ChangeKey. That is m queries and $O(m \log n)$ time in total.

Problem 4.19. A group of network designers at the communications company CluNet have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge e is a communication link with a given available bandwidth b_e .

For each pair of nodes $u, v \in V$, they want to select a single u - v path P on which this pair will communicate. The *bottleneck rate* $b(P)$ of this path P is the minimum bandwidth of any edge it contains, that is, $b(P) = \min_{e \in P} b_e$. The *best achievable bottleneck rate* for the pair u, v in G is the maximum, over all u - v paths P in G , of the value $b(P)$.

Keeping track of a path for each pair of nodes is getting complicated, so one of the designers suggests the following. Maybe one can find a spanning tree T of G so that for *every* pair of nodes u, v , the unique u - v path in the tree attains the best achievable bottleneck rate for u, v in G . (In other words, even if you could choose any u - v path in the whole graph, you couldn't do better than the u - v path in T .)

Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the u - v path in T is equal to the best achievable bottleneck rate for the pair u, v in G .

Solution. Take T to be a *maximum*-bandwidth spanning tree, that is, a minimum spanning tree for the costs $c_e = -b_e$. Kruskal's Algorithm finds one: by (4.18), together with the tie-breaking perturbation argument at the end of Section 4.5, it returns a minimum spanning tree even when costs repeat.

Fix u, v , let P be the u - v path in T , and let $e \in P$ have $b_e = b(P)$. Since P is itself a u - v path in G , $b(P)$ is at most the best achievable rate. Suppose some u - v path Q in G had $b(Q) > b(P)$. Deleting e splits T into two components $S \ni u$ and $V - S \ni v$, and e is the only edge of T between them. The path Q must cross between S and $V - S$ along some edge f , so $f \notin T$ and

$$b_f \geq b(Q) > b(P) = b_e.$$

Then $T - e + f$ is a spanning tree whose total bandwidth exceeds that of T , contradicting the choice of T . Hence $b(P)$ equals the best achievable bottleneck rate, for every pair u, v simultaneously.

Running time: $O(m \log n)$ by (4.25).

Problem 4.20. There are n towns in a mountainous county, and the road system is a connected graph $G = (V, E)$ on the towns, each edge a road joining two of them. Each edge e is annotated with a number a_e , the altitude of the highest point on the road, and no two edges have the same altitude. The *height* of a path P is the maximum of a_e over all edges e on P . A path between towns i and j is *winter-optimal* if it achieves the minimum possible height over all paths from i to j .

The highway crews will select a set $E' \subseteq E$ of roads to keep clear through the winter. They require that (V, E') be connected and, more strongly, that for every pair of towns i and j the height of the winter-optimal path in (V, E') be no greater than it is in the full graph $G = (V, E)$. Call (V, E') a *minimum-altitude connected subgraph* if it has this property. Subject to this property, they want to keep as few roads clear as possible. They consider two conjectures.

(1) The minimum spanning tree of G , with respect to the edge weights a_e , is a minimum-altitude connected subgraph. (Because the a_e are distinct, the minimum spanning tree is unique.) (2) A subgraph (V, E') is a minimum-altitude connected subgraph if and only if it contains the edges of the minimum spanning tree.

(a) Is the first conjecture true, for all choices of G and distinct altitudes a_e ? Give a proof or a counterexample with explanation.

(b) Is the second conjecture true, for all choices of G and distinct altitudes a_e ? Give a proof or a counterexample with explanation.

Solution. (a) Yes. Let T be the minimum spanning tree, take towns i, j , let P be the i - j path in T , and let $e \in P$ be its highest edge. Suppose some i - j path Q in G had height less than a_e . Deleting e splits T into components $S \ni i$ and $V - S \ni j$, and Q crosses between them along some edge $f \neq e$, with $f \notin T$ and

$$a_f \leq \text{height}(Q) < a_e.$$

Then $T - e + f$ is a spanning tree of smaller total altitude than T , a contradiction. So the T -path between any two towns is winter-optimal in G , and (V, T) is a minimum-altitude connected subgraph.

(b) Yes. (i) If $E' \supseteq T$, then for every pair i, j the subgraph (V, E') contains the T -path, whose height is optimal in G by (a). Also (V, E') is connected. So (V, E') is a minimum-altitude connected subgraph.

(ii) Conversely, suppose E' misses some tree edge $e = (i, j) \in T$. The T -path from i to j is the single edge e , so by (a) the winter-optimal height between i and j in G is a_e . Deleting e splits T into components $S \ni i$ and $V - S \ni j$. Every edge $f \neq e$ crossing this cut has $a_f > a_e$, since otherwise $T - e + f$ would be a cheaper spanning tree (the altitudes are distinct). Any i - j path Q in (V, E') avoids e but must cross the cut along some such f , so

$$\text{height}(Q) \geq a_f > a_e.$$

If there is no such path, (V, E') is not even connected. Either way, (V, E') is not a minimum-altitude connected subgraph, and the second conjecture holds.

Problem 4.21. Let us say that a graph $G = (V, E)$ is a *near-tree* if it is connected and has at most $n + 8$ edges, where $n = |V|$. Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges, and returns a minimum spanning tree of G . You may assume that all the edge costs are distinct.

Solution. Repeatedly find a cycle and delete its most expensive edge, until $n - 1$ edges remain.

```

while G has more than n - 1 edges:
    run DFS on G to find a cycle C
    delete from G the most expensive edge of C
return G

```

Correctness. Let e be the most expensive edge on a cycle C of the current graph H . Since the costs are distinct, the Cycle Property (4.20) says e lies in no minimum spanning tree of H ; so the (unique) minimum spanning tree T of H is a spanning tree of $H - e$, and since every spanning tree of $H - e$ is a spanning tree of H , T is also the minimum spanning tree of $H - e$. Deleting an edge of a cycle keeps the graph connected, so the loop is well defined: a connected graph with more than $n - 1$ edges has a cycle. Hence the MST is invariant throughout, and at termination G is connected with $n - 1$ edges, i.e. a tree, so it is its own minimum spanning tree, which is therefore the MST of the input.

Running time. Each iteration deletes one edge, so there are at most $(n + 8) - (n - 1) = 9$ iterations. Each iteration is a DFS (which finds a cycle via a non-tree edge, or reports none) plus a scan of the cycle, costing $O(n + m) = O(n + n + 8) = O(n)$. Total $9 \cdot O(n) = O(n)$.

4.4 Exercises 4.22–4.28

Problem 4.22. Consider the Minimum Spanning Tree Problem on an undirected graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge, where the costs may not all be different. If the costs are not all distinct, there can in general be many distinct minimum-cost solutions. Suppose we are given a spanning tree $T \subseteq E$ with the guarantee that for every $e \in T$, e belongs to *some* minimum-cost spanning tree in G . Can we conclude that T itself must be a minimum-cost spanning tree in G ? Give a proof or a counterexample with explanation.

Solution. No. Take the triangle on nodes a, b, c with

$$c_{(a,b)} = 1, \quad c_{(b,c)} = 2, \quad c_{(a,c)} = 2,$$

and $T = \{(b, c), (a, c)\}$. The three spanning trees are the three pairs of edges, with costs

$$c(\{(a, b), (b, c)\}) = 3, \quad c(\{(a, b), (a, c)\}) = 3, \quad c(T) = 4.$$

So the minimum spanning trees are exactly $\{(a, b), (b, c)\}$ and $\{(a, b), (a, c)\}$. The edge $(b, c) \in T$ lies in the first and $(a, c) \in T$ lies in the second, so T satisfies the guarantee; yet $c(T) = 4 > 3$, so T is not a minimum-cost spanning tree.

Problem 4.23. Recall the problem of computing a minimum-cost arborescence in a directed graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge. Here we will consider the case in which G is a directed acyclic graph — that is, it contains no directed cycles.

As in general directed graphs, there can be many distinct minimum-cost solutions. Suppose we are given a directed acyclic graph $G = (V, E)$, and an arborescence $A \subseteq E$ with the guarantee that for every $e \in A$, e belongs to *some* minimum-cost arborescence in G . Can we conclude that A itself must be a minimum-cost arborescence in G ? Give a proof or a counterexample with explanation.

Solution. Yes. Let r be the root and, for $v \neq r$, let y_v be the minimum cost of an edge entering v . In a DAG, every set F consisting of exactly one edge entering each $v \neq r$ is an arborescence: (V, F) is a subgraph of an acyclic graph, so it has no cycles, and (4.34) applies. In particular, choosing a cheapest entering edge for every $v \neq r$ gives an arborescence of cost $\sum_{v \neq r} y_v$, and by (4.34) every arborescence has one entering edge per $v \neq r$, hence cost at least $\sum_{v \neq r} y_v$. So the optimum is exactly $\sum_{v \neq r} y_v$. Now let $e = (u, v) \in A$ and let A^* be a minimum-cost arborescence containing e . If $c_e > y_v$, replace e in A^* by a cheapest edge entering v : the result still has exactly one edge entering each node other than r , so it is an arborescence by the observation above, and it is cheaper than A^* , a contradiction. Hence $c_e = y_v$ for every $e = (u, v) \in A$, and since A has exactly one edge entering each $v \neq r$ (4.34),

$$c(A) = \sum_{v \neq r} y_v,$$

the minimum cost. So A is a minimum-cost arborescence.

Problem 4.24. Timing circuits are a crucial component of VLSI chips. Here's a simple model of such a timing circuit. Consider a complete balanced binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length ℓ_e , which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

Example. Consider the tree in Figure 4.20: root v has children v' (edge length 2) and v'' (edge length

1); v' has leaf children a (length 2) and b (length 1); v'' has leaf children c (length 2) and d (length 1). The unique optimal solution for this instance would be to take the three length-1 edges and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

Solution. Work bottom-up: at each internal node, lengthen the edge to the shallower child just enough to match the deeper one, and never touch the deeper side.

For a node v let $d(v)$ be the (common) distance from v to the leaves below it after its subtree has been processed; $d(v) = 0$ at a leaf. For an internal node v with child edges $a = (v, u)$ and $b = (v, w)$, put

$$p = \ell_a + d(u), \quad q = \ell_b + d(w), \quad d(v) = \max(p, q),$$

and increase ℓ_a by $d(v) - p$ and ℓ_b by $d(v) - q$ (one of these is 0). Processing nodes in post-order, the subtree of every node has zero skew with height $d(v)$, so the whole tree has zero skew. Let $g(v)$ be the total increase made inside the subtree T_v ; then $g(\text{leaf}) = 0$ and

$$g(v) = g(u) + g(w) + |p - q|.$$

Optimality. Claim: any choice of increases $x_e \geq 0$ that makes T_v zero-skew with common height H has $H \geq d(v)$ and total increase inside T_v at least $g(v) + 2(H - d(v))$. At a leaf this reads $0 \geq 0$. At internal v , the subtrees T_u, T_w are zero-skew with heights H_u, H_w , where by induction $H_u \geq d(u)$, $H_w \geq d(w)$ and

$$H = \ell_a + x_a + H_u = \ell_b + x_b + H_w,$$

so $H \geq \max(p, q) = d(v)$, and the cost is at least

$$\begin{aligned} & g(u) + g(w) + (H_u - d(u)) + (H_w - d(w)) + x_a + x_b \\ &= g(u) + g(w) + (H - p) + (H - q) \\ &= g(u) + g(w) + (2 \max(p, q) - p - q) + 2(H - d(v)) \\ &= g(v) + 2(H - d(v)). \end{aligned}$$

At the root this gives total increase $\geq g(\text{root})$, which the algorithm attains; since the final total length is the original total plus the total increase, the algorithm minimizes it.

Running time. The tree has $2n - 1$ nodes and each is processed in $O(1)$ time in a post-order traversal, so $O(n)$.

Problem 4.25. Suppose we are given a set of points $P = \{p_1, p_2, \dots, p_n\}$, together with a distance function d on the set P ; d is simply a function on pairs of points in P with the properties that $d(p_i, p_j) = d(p_j, p_i) > 0$ if $i \neq j$, and that $d(p_i, p_i) = 0$ for each i .

We define a *hierarchical metric* on P to be any distance function τ that can be constructed as follows. We build a rooted tree T with n leaves, and we associate with each node v of T (both leaves and internal nodes) a *height* h_v . These heights must satisfy the properties that $h(v) = 0$ for each leaf v , and if u is the parent of v in T , then $h(u) \geq h(v)$. We place each point in P at a distinct leaf in T . Now, for any pair of points p_i and p_j , their distance $\tau(p_i, p_j)$ is defined as follows. We determine the least common ancestor v in T of the leaves containing p_i and p_j , and define $\tau(p_i, p_j) = h_v$.

We say that a hierarchical metric τ is *consistent* with our distance function d if, for all pairs i, j , we have $\tau(p_i, p_j) \leq d(p_i, p_j)$.

Give a polynomial-time algorithm that takes the distance function d and produces a hierarchical metric τ with the following properties.

- (i) τ is consistent with d , and
- (ii) if τ' is any other hierarchical metric consistent with d , then $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ for each pair of points p_i and p_j .

Solution. Run Kruskal's Algorithm on the complete graph on P with edge costs d , and record each merge as a tree node: when an edge of cost w joins the clusters whose current tree roots are x and y , create a new node with height w and children x, y .

Start with each p_i as a leaf of height 0. Kruskal considers edges in nondecreasing order of cost, so a node created at cost w has children of height at most w (they are leaves or were created earlier); thus $h(\text{parent}) \geq h(\text{child})$, and after $n - 1$ merges we have a rooted tree T with the points at distinct leaves, i.e. a hierarchical metric τ . By construction, $\tau(p_i, p_j)$ is the cost w of the edge at which the clusters of p_i and p_j were merged.

(i) When the edge (p_i, p_j) is considered, either p_i, p_j are already in one cluster, merged at some cost $\leq d(p_i, p_j)$, or they are merged now at cost $d(p_i, p_j)$. Either way $\tau(p_i, p_j) \leq d(p_i, p_j)$.

(ii) Every hierarchical metric τ' (with tree T') satisfies the ultrametric inequality

$$\tau'(x, z) \leq \max(\tau'(x, y), \tau'(y, z)).$$

Indeed $a = \text{lca}(x, y)$ and $b = \text{lca}(y, z)$ are both ancestors of y , so one, say a , is an ancestor of the other; then a is an ancestor of x and z , so $\text{lca}(x, z)$ lies on the path from a down, and monotonicity of heights gives $h(\text{lca}(x, z)) \leq h_a$.

Now fix p_i, p_j merged at cost $w = \tau(p_i, p_j)$ via an edge (q, q') with q in the cluster of p_i and q' in that of p_j . Each cluster is connected by the Kruskal edges already added, all of cost $\leq w$, so there is a path $p_i = s_0, s_1, \dots, s_k = p_j$ with $d(s_{t-1}, s_t) \leq w$ for all t . Iterating the ultrametric inequality and using consistency of τ' ,

$$\tau'(p_i, p_j) \leq \max_t \tau'(s_{t-1}, s_t) \leq \max_t d(s_{t-1}, s_t) \leq w = \tau(p_i, p_j).$$

Running time. Kruskal's Algorithm on the $\binom{n}{2}$ edges takes $O(n^2 \log n)$ time by (4.25), and creating the $n - 1$ tree nodes is $O(n)$ extra; reading off all $\tau(p_i, p_j)$ by LCA queries costs $O(n)$ each, $O(n^3)$ in all, which is polynomial.

Problem 4.26. One of the first things you learn in calculus is how to minimize a differentiable function such as $y = ax^2 + bx + c$, where $a > 0$. The Minimum Spanning Tree Problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved (rather than a continuum of possibilities) and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge e now has a *time-varying edge cost* given by a function $f_e : \mathbb{R} \rightarrow \mathbb{R}$. Thus, at time t , it has cost $f_e(t)$. We'll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t ; we'll denote this function $c_G(t)$. A natural problem then becomes: find a value of t at which $c_G(t)$ is minimized.

Suppose each function f_e is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $\{(a_e, b_e, c_e) : e \in E\}$ and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G . You may assume that arithmetic operations on the numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

Solution. Cut the time line at the $O(m^2)$ times where two cost parabolas cross, take the Kruskal tree of each piece, minimize its quadratic cost over the piece, and return the best time found.

```

P = empty
for each pair e != e' with f_e - f_e' not identically 0:
    add to P the (at most 2) real roots of f_e(t) - f_e'(t)
sort P as t_1 < ... < t_r; set t_0 = -inf, t_(r+1) = +inf
for each piece I_i = [t_i, t_(i+1)], i = 0..r:
    s = an interior point of I_i    (midpoint, or t_1 - 1, t_r + 1, or 0 if r = 0)

```

```

T_i = Kruskal's tree for costs f_e(s), ties broken by edge index
A, B', C = sums of a_e, b_e, c_e over e in T_i
tau_i = -B'/(2A) clamped into I_i; val_i = A tau_i^2 + B' tau_i + C
return the tau_i with the smallest val_i

```

Correctness. On the interior of a piece I_i , any two edges e, e' either have $f_e \equiv f_{e'}$ or have $f_e - f_{e'}$ continuous with no root, hence of constant sign. So the order of the edges by $(f_e(t), \text{index})$ is the same for every interior t ; Kruskal's Algorithm only compares costs, so it outputs the same tree T_i for each such t , and T_i is a minimum spanning tree at time t by (4.18) (ties allowed, by the perturbation argument of Section 4.5). Writing $F_T(t) = \sum_{e \in T} f_e(t)$,

$$c_G(t) = \min_T F_T(t) = F_{T_i}(t) \quad (t \in \text{int } I_i).$$

Both sides are continuous (c_G is a minimum of finitely many polynomials), so equality holds on all of I_i . The pieces cover $\mathbb{R}$, so

$$\min_{t \in \mathbb{R}} c_G(t) = \min_i \min_{t \in I_i} F_{T_i}(t).$$

Each $F_{T_i}(t) = At^2 + B't + C$ with $A = \sum_{e \in T_i} a_e > 0$ is a convex parabola, so its minimum over the interval I_i is attained at the vertex $-B'/(2A)$ clamped into I_i ; this is τ_i . Hence the returned τ_i minimizes c_G (and the minimum exists).

Running time. Each pair contributes at most 2 roots (a quadratic, a linear, or a nonzero constant difference), so $r \leq 2\binom{m}{2} < m^2$; sorting costs $O(m^2 \log m)$, and the $r + 1 = O(m^2)$ pieces each need one $O(m \log n)$ run of Kruskal's Algorithm (Section 4.6), for $O(m^3 \log n)$ in total (counting a square root as one arithmetic operation).

Problem 4.27. In trying to understand the combinatorial structure of spanning trees, we can consider the space of *all* possible spanning trees of a given graph and study the properties of this space. This is a strategy that has been applied to many similar problems as well.

Here is one way to do this. Let G be a connected graph, and T and T' two different spanning trees of G . We say that T and T' are *neighbors* if T contains exactly one edge that is not in T' , and T' contains exactly one edge that is not in T .

Now, from any graph G , we can build a (large) graph $\mathcal{H}$ as follows. The nodes of $\mathcal{H}$ are the spanning trees of G , and there is an edge between two nodes of $\mathcal{H}$ if the corresponding spanning trees are neighbors.

Is it true that, for any connected graph G , the resulting graph $\mathcal{H}$ is connected? Give a proof that $\mathcal{H}$ is always connected, or provide an example (with explanation) of a connected graph G for which $\mathcal{H}$ is not connected.

Solution. Yes, $\mathcal{H}$ is always connected. We show by induction on $d(T, T') = |T - T'|$ that any two spanning trees T, T' are joined by a path in $\mathcal{H}$ of length $d(T, T')$; the case $d = 0$ is $T = T'$.

Let $d(T, T') \geq 1$ and pick $e \in T' - T$. Adding e to the tree T creates a unique cycle $C \subseteq T + e$. Not every edge of C lies in T' , since T' is acyclic; and $e \in T'$, so there is an edge $f \in C - T'$, $f \neq e$, hence $f \in T$. Put

$$T_1 = T + e - f.$$

Deleting an edge of the cycle C from the connected graph $T + e$ leaves it connected, and T_1 has $n - 1$ edges, so T_1 is a spanning tree. It differs from T exactly in $e \in T_1 - T$ and $f \in T - T_1$, so T and T_1 are neighbors. Since $e \in T'$ and $f \notin T'$,

$$d(T_1, T') = |T - T'| - 1 = d(T, T') - 1,$$

so by induction T_1 is joined to T' by a path of length $d(T, T') - 1$, and prefixing the edge TT_1 of $\mathcal{H}$ gives the required path from T .

Problem 4.28. Suppose you're a consultant for the networking company CluNet, and they have the following problem. The network that they're currently working on is modeled by a connected graph $G = (V, E)$ with n nodes. Each edge e is a fiber-optic cable that is owned by one of two companies (creatively named X and Y) and leased to CluNet.

Their plan is to choose a spanning tree T of G and upgrade the links corresponding to the edges of T . Their business relations people have already concluded an agreement with companies X and Y stipulating a number k so that in the tree T that is chosen, k of the edges will be owned by X and $n - k - 1$ of the edges will be owned by Y .

CluNet management now faces the following problem. It is not at all clear to them whether there even *exists* a spanning tree T meeting these conditions, or how to find one if it exists. So this is the problem they put to you: Give a polynomial-time algorithm that takes G , with each edge labeled X or Y , and either (i) returns a spanning tree with exactly k edges labeled X , or (ii) reports correctly that no such tree exists.

Solution. Compute a spanning tree $T_{\min}$ with the fewest X -edges and one $T_{\max}$ with the most, then walk from $T_{\min}$ to $T_{\max}$ by single-edge exchanges; each exchange changes the X -count by at most one, so every value in between is hit.

For a spanning tree T let $x(T)$ be its number of X -edges.

```

T_min = Kruskal with cost 1 on X-edges, 0 on Y-edges
T_max = Kruskal with cost 0 on X-edges, 1 on Y-edges
if k < x(T_min) or k > x(T_max): report "no such tree"
T = T_min
while x(T) != k:
    pick e in T_max - T; find the cycle C of T + e
    pick f in C - T_max; T = T + e - f
return T

```

Correctness. With costs 1 on X -edges and 0 on Y -edges the cost of a tree is exactly $x(T)$, so by (4.18) (ties handled as in Section 4.5) $x(T_{\min}) = \min_T x(T)$; symmetrically the cost $n - 1 - x(T)$ is minimized by $T_{\max}$, so $x(T_{\max}) = \max_T x(T)$. Hence if k lies outside $[x(T_{\min}), x(T_{\max})]$ no tree has k edges labeled X , and report (ii) is correct.

Otherwise, the loop body is exactly the exchange step in the solution of Exercise 4.27: f exists, $T + e - f$ is a spanning tree, and $|T - T_{\max}|$ drops by one, so after at most $n - 1$ iterations $T = T_{\max}$. One exchange changes the count by

$$x(T + e - f) - x(T) = [e \in X] - [f \in X] \in \{-1, 0, 1\}.$$

The integer sequence $x(T)$ thus starts at $x(T_{\min}) \leq k$, ends at $x(T_{\max}) \geq k$, and moves in steps of size at most 1, so it equals k at some iteration, where the loop stops and returns a spanning tree with exactly k edges labeled X .

Running time. The two runs of Kruskal's Algorithm take $O(m \log n)$; each of at most $n - 1$ exchanges finds the cycle (the tree path between the ends of e) by a BFS of T in $O(n)$ time, so the total is $O(m \log n + n^2)$.

4.5 Exercises 4.29–4.33

Problem 4.29. Given a list of n natural numbers $d_1, d_2, \dots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \dots, d_n$. (That is, if $V = \{v_1, v_2, \dots, v_n\}$, then the degree of v_i should be exactly d_i .) G should not contain multiple edges between the same pair of nodes, or “loop” edges with both endpoints equal to the same node.

Solution. Use the Havel-Hakimi rule: join a node of largest required degree d to the d other nodes of largest required degree, delete it, and recurse on the reduced list.

```

loop:
    sort the list so that d_1 >= d_2 >= ... >= d_n
    if d_1 = 0: answer YES

```

if $d_1 > (\text{current length of the list}) - 1$: answer NO
 replace the list by $(d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n)$
 if some entry is negative: answer NO

Call a list *graphical* if some simple graph realizes it. Correctness rests on one claim.

Claim. Let $d_1 \geq d_2 \geq \dots \geq d_n$ be graphical and $S = \{v_2, \dots, v_{d_1+1}\}$. Then some realization has $N(v_1) = S$.

Take a realization maximizing $|N(v_1) \cap S|$, and suppose $N(v_1) \neq S$. Both sets have size d_1 , so there are $w \in S - N(v_1)$ and $u \in N(v_1) - S$, with $d_w \geq d_u$ (every node of S precedes every node outside $S \cup \{v_1\}$). Let $A = N(w) - \{u\}$ and $B = N(u) - \{w\}$; then

$$|A| = d_w - [uw \in E] \geq d_u - [uw \in E] = |B|.$$

Since $v_1 \in B - A$, there is some $x \in A - B$; so $x \neq u, w, v_1$, $wx \in E$ and $ux \notin E$. Replace the edges v_1u, wx by v_1w, ux : both new pairs were non-edges and are not loops, and every degree is unchanged. This raises $|N(v_1) \cap S|$ by one, a contradiction. $\square$

Now one round is exact: if the list is graphical, then $d_1 \leq n - 1$, and deleting v_1 from the realization of the Claim realizes the reduced list (so no entry is negative); conversely, if the reduced list is realized by a graph on $v_2, \dots, v_n$, adding v_1 adjacent to S realizes the original list. Hence each NO is correct, each round preserves graphicality, and the all-zero list (reached after at most n rounds, since the list shortens by one each round) is realized by the empty graph, so YES is correct.

Running time. At most n rounds, each a sort and $O(n)$ updates, for $O(n^2 \log n)$.

Problem 4.30. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j) ; and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V' .

We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a spanning subtree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T .

Show that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(n^{O(k)})$.

Solution. Try every set $Y \subseteq V - X$ with $|Y| \leq k - 2$, take a minimum spanning tree of $G[X \cup Y]$, and return the lightest one found; this suffices because some optimal Steiner tree uses at most $k - 2$ non-terminals. (For $k \leq 1$ the answer is $Z = X$ with the empty tree, of weight 0.)

Claim. For $k \geq 2$ some minimum-weight Steiner tree (Z, T) has $|Z - X| \leq k - 2$.

Among minimum-weight Steiner trees take one with $|Z|$ smallest. Every $v \in Z - X$ has $\deg_T(v) \geq 3$:

(i) if $\deg_T(v) = 1$, then $(Z - \{v\}, T - v)$ is a Steiner tree of smaller weight (weights are positive), impossible; (ii) if $\deg_T(v) = 2$ with neighbors a, b , then $T - va - vb + ab$ spans $Z - \{v\}$, is a tree, and has weight changed by $w_{ab} - w_{av} - w_{vb} \leq 0$ by the triangle inequality, contradicting minimality of $|Z|$.

So every leaf of T is a terminal. Let T have $L \leq k$ leaves, r nodes of degree 2 and s nodes of degree ≥ 3 ; as $|Z| \geq k \geq 2$, no node has degree 0, and

$$2(L + r + s - 1) = \sum_{v \in Z} \deg_T(v) \geq L + 2r + 3s \implies s \leq L - 2 \leq k - 2.$$

Since $Z - X$ consists of nodes of degree ≥ 3 , $|Z - X| \leq s \leq k - 2$. $\square$

Correctness. Every candidate (a set $X \cup Y$ with a spanning tree of $G[X \cup Y]$, which is complete hence connected) is a Steiner tree, so the output weighs at least the optimum OPT. Conversely, for the tree (Z, T) of the Claim, $Y = Z - X$ is enumerated, and T is a spanning tree of $G[Z]$, so the minimum spanning tree found for Y (optimal by (4.19)) weighs at most $w(T) = \text{OPT}$.

Running time. The number of sets Y is at most

$$\sum_{j=0}^{k-2} \binom{n}{j} \leq \sum_{j=0}^{k-2} n^j \leq n^{k-1} \quad (n \geq 2),$$

and Prim's Algorithm on the complete graph $G[X \cup Y]$ with at most $2k - 2$ nodes takes $O(k^2)$ time with an array in place of the heap. The total is $O(k^2 n^{k-1}) = O(n^{k+1}) = O(n^{O(k)})$.

Problem 4.31. Let's go back to the original motivation for the Minimum Spanning Tree Problem. We are given a connected, undirected graph $G = (V, E)$ with positive edge lengths $\{\ell_e\}$, and we want to find a spanning subgraph of it. Now suppose we are willing to settle for a subgraph $H = (V, F)$ that is “denser” than a tree, and we are interested in guaranteeing that, for each pair of vertices $u, v \in V$, the length of the shortest u - v path in H is not much longer than the length of the shortest u - v path in G . By the *length* of a path P here, we mean the sum of ℓ_e over all edges e in P . Here's a variant of Kruskal's Algorithm designed to produce such a subgraph.

- First we sort all the edges in order of increasing length. (You may assume all edge lengths are distinct.)
- We then construct a subgraph $H = (V, F)$ by considering each edge in order.
- When we come to edge $e = (u, v)$, we add e to the subgraph H if there is currently no u - v path in H . (This is what Kruskal's Algorithm would do as well.) On the other hand, if there is a u - v path in H , we let d_{uv} denote the length of the shortest such path; again, length is with respect to the values $\{\ell_e\}$. We add e to H if $3\ell_e < d_{uv}$.

In other words, we add an edge even when u and v are already in the same connected component, provided that the addition of the edge reduces their shortest-path distance by a sufficient amount.

Let $H = (V, F)$ be the subgraph of G returned by the algorithm.

(a) Prove that for every pair of nodes $u, v \in V$, the length of the shortest u - v path in H is at most three times the length of the shortest u - v path in G .

(b) Despite its ability to approximately preserve shortest-path distances, the subgraph H produced by the algorithm cannot be too dense. Let $f(n)$ denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all n -node input graphs with edge lengths. Prove that

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^2} = 0.$$

Solution. (a) Every edge $e = (u, v) \in E$ satisfies $d_H(u, v) \leq 3\ell_e$: either $e \in F$, or when e was considered u and v were already joined in H with $d_{uv} \leq 3\ell_e$, and later additions to H only shorten distances. Summing over the edges of a shortest u - v path $P = (u = x_0, x_1, \dots, x_k = v)$ in G and using the triangle inequality for d_H ,

$$\begin{aligned} d_H(u, v) &\leq \sum_{i=1}^k d_H(x_{i-1}, x_i) \\ &\leq \sum_{i=1}^k 3\ell_{(x_{i-1}, x_i)} = 3d_G(u, v). \end{aligned}$$

(b) H has no cycle of length at most 4, and a graph with no 4-cycle has at most $\frac{n}{2}(1 + \sqrt{n})$ edges.

No short cycles. Let C be a cycle of H and $e = (u, v)$ its longest edge. Edges are scanned in increasing order of (distinct) lengths, so every other edge of C was already in H when e was considered; hence $C - e$ was a u - v path in H , and since e was nevertheless added,

$$3\ell_e < d_{uv} \leq \ell(C - e) < (|C| - 1)\ell_e,$$

so $|C| \geq 5$.

Counting. If two distinct vertices u, w had two common neighbours $x \neq x'$, then u, x, w, x' would be a 4-cycle of H ; so each pair $\{u, w\}$ is the pair of ends of at most one path of two edges. Counting such paths by their middle vertex, with $m = |F|$ and average degree $\bar{d} = 2m/n$, convexity of $x \mapsto \binom{x}{2}$ gives

$$n \binom{\bar{d}}{2} \leq \sum_{x \in V} \binom{\deg x}{2} \leq \binom{n}{2}, \quad \text{so} \quad \bar{d}(\bar{d} - 1) \leq n - 1.$$

Hence $\bar{d} \leq 1 + \sqrt{n}$, i.e. $m \leq \frac{n}{2}(1 + \sqrt{n})$. As this holds for every output,

$$0 \leq \frac{f(n)}{n^2} \leq \frac{1 + \sqrt{n}}{2n} \rightarrow 0.$$

Problem 4.32. Consider a directed graph $G = (V, E)$ with a root $r \in V$ and nonnegative costs on the edges. In this problem we consider variants of the minimum-cost arborescence algorithm.

(a) The algorithm discussed in Section 4.9 works as follows. We modify the costs, consider the subgraph of zero-cost edges, look for a directed cycle in this subgraph, and contract it (if one exists). Argue briefly that instead of looking for cycles, we can instead identify and contract strong components of this subgraph.

(b) In the course of the algorithm, we defined y_v to be the minimum cost of an edge entering v , and we modified the costs of all edges e entering node v to be $c'_e = c_e - y_v$. Suppose we instead use the following modified cost: $c''_e = \max(0, c_e - 2y_v)$. This new change is likely to turn more edges to 0 cost. Suppose now we find an arborescence T of 0 cost. Prove that this T has cost at most twice the cost of the minimum-cost arborescence in the original graph.

(c) Assume you do not find an arborescence of 0 cost. Contract all 0-cost strong components and recursively apply the same procedure on the resulting graph until an arborescence is found. Prove that this T has cost at most twice the cost of the minimum-cost arborescence in the original graph.

Solution. Throughout, edges entering r are deleted (no arborescence uses them), so $\{r\}$ is a strong component of every subgraph and every nontrivial strong component S avoids r ; write $Y = \sum_{v \neq r} y_v$ and $\text{OPT}(G, c)$ for the minimum cost of an arborescence.

(a) Contracting a nontrivial 0-cost strong component S is the same as contracting 0-cost cycles one at a time, each a legal step of the Section 4.9 algorithm. Indeed S contains a 0-cost cycle C ; if $C \neq S$, some 0-cost edge of S enters the contracted node C , so its new y -value is 0 and no cost changes, and S/C is again strongly connected by 0-cost edges; repeat until S is one node. By (4.38) and (4.39) every such run returns an optimal arborescence. Moreover a 0-cost subgraph has a cycle if and only if it has a nontrivial strong component, so the algorithm stops at the same moment as before.

(b) Every edge $e = (u, v) \in T$ has $c''_e = 0$, i.e. $c_e \leq 2y_v$; and every arborescence T^* uses one edge into each $v \neq r$, of cost at least y_v . Hence

$$c(T) = \sum_{(u,v) \in T} c_{(u,v)} \leq 2 \sum_{v \neq r} y_v = 2Y \leq 2c(T^*).$$

(c) The key inequalities are, for every edge e entering v (using $c_e \geq y_v$ in the case $c_e < 2y_v$),

$$c_e - 2y_v \leq c''_e \leq c_e - y_v.$$

The procedure terminates: every $v \neq r$ has a c'' -zero entering edge (its cheapest one), so if there were no nontrivial 0-cost strong component, one such edge per node would be acyclic and hence a 0-cost arborescence by (4.34). We prove $c(T) \leq 2\text{OPT}(G, c)$ by induction on $|V|$; the base case (a 0-cost arborescence is found) is (b). Otherwise let G' be the contracted graph with costs c'' , T' the arborescence returned on it, and T its expansion to G by 0-cost arborescences inside each contracted component, as in (a), so $c''(T) = c''(T')$. Contracting any arborescence of G leaves a subgraph of G' in which r still reaches every node, which contains an arborescence of no larger c'' -cost, so $\text{OPT}(G', c'') \leq \text{OPT}(G, c)$. With the inductive hypothesis on G' ,

$$\begin{aligned} c(T) &\leq c''(T) + 2Y = c''(T') + 2Y \leq 2\text{OPT}(G', c'') + 2Y \\ &\leq 2\text{OPT}(G, c'') + 2Y \\ &\leq 2(\text{OPT}(G, c) - Y) + 2Y = 2\text{OPT}(G, c), \end{aligned}$$

where the first and last inequalities sum the two bounds on c''_e over an arborescence (one edge into each $v \neq r$).

Problem 4.33. Suppose you are given a directed graph $G = (V, E)$ in which each edge has a cost of either 0 or 1. Also suppose that G has a node r such that there is a path from r to every other node in G . You are also given an integer k . Give a polynomial-time algorithm that either constructs an arborescence rooted at r of cost *exactly* k , or reports (correctly) that no such arborescence exists.

Solution. Compute a cheapest arborescence T_1 (cost a) and a most expensive one T_2 (cost b); if $k \notin [a, b]$ report that none exists, and otherwise walk from T_1 to T_2 by changing one parent at a time, stopping when the cost equals k .

Write $p_T(v)$ for the tail of the edge of T entering v .

```

T1 := min-cost arborescence for costs c_e      (Section 4.9)
T2 := min-cost arborescence for costs 1 - c_e  (a max-cost one for c)
a := c(T1);  b := c(T2)
if k < a or k > b: report "no such arborescence"
T := T1
while c(T) != k:
    v := a node with p_T(v) != p_T2(v) of least depth in T2
    replace the edge (p_T(v), v) in T by (p_T2(v), v)
return T

```

Correctness. The algorithm of Section 4.9 is optimal by (4.39) (all costs are nonnegative, and an arborescence exists by (4.35)); since every arborescence has exactly $n - 1$ edges, its cost under $1 - c$ is $n - 1 - c(T)$, so T_2 maximizes c . Hence $a \leq c(T) \leq b$ for every arborescence T , and the report for $k \notin [a, b]$ is correct.

Each swap keeps T an arborescence. Every node of T_2 -depth less than that of v has the same parent in T as in T_2 , so the T_2 -path from r to $p_{T_2}(v)$ lies in T and is the T -path to $p_{T_2}(v)$; it avoids v , which has larger T_2 -depth. So v is not a T -ancestor of $p_{T_2}(v)$, the swap creates no cycle, and each node $\neq r$ still has one entering edge, so the result is an arborescence by (4.34).

Each swap lowers the number of nodes whose parents in T and T_2 differ by one, so after at most $n - 1$ swaps $T = T_2$. One edge leaves and one enters, both of cost 0 or 1, so consecutive costs satisfy

$$|c(T_{\text{after}}) - c(T_{\text{before}})| \leq 1.$$

An integer sequence from a to b with steps of size at most 1 takes every integer value in $[a, b]$, so the loop stops at an arborescence of cost exactly k .

Running time. Two runs of the Section 4.9 algorithm take $O(mn)$ time (at most n contractions, each $O(m)$), and the walk does at most $n - 1$ swaps of $O(n)$ each (recompute T_2 -depths once; maintain $c(T)$ incrementally), for $O(mn)$ in total.

5 Divide and Conquer

Contents

5.1 Exercises 5.1–5.7	52
---------------------------------	----

5.1 Exercises 5.1–5.7

Problem 5.1. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values (so there are $2n$ values total), and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Solution. Compare the two databases' "middle" elements and throw away the lower half of the database whose middle element is smaller together with the upper half of the other one; the median of what remains is still the median.

Write $A[i]$, $B[i]$ for the i^{th} smallest values of the two databases (each $A[i]$ or $B[i]$ costs one query). We maintain offsets a, b and a size m such that the median sought is the m^{th} smallest element of the $2m$ values $A[a+1..a+m] \cup B[b+1..b+m]$; initially $a = b = 0$, $m = n$.

```

a = 0; b = 0; m = n
while m > 1:
    k = ceil(m/2); d = floor(m/2)
    if A[a+k] < B[b+k]: a = a + d      (drop A[a+1..a+d] and B[b+k+1..b+m])
    else:               b = b + d      (drop B[b+1..b+d] and A[a+k+1..a+m])
    m = m - d
return min(A[a+1], B[b+1])

```

Correctness. It suffices to show one iteration preserves the invariant. Say $A[a+k] < B[b+k]$ (the other case is symmetric), and rank elements within the current $2m$ values. For $j \leq d = m - k < k$, the values below $A[a+j]$ are at most $j-1$ elements of A and the elements of B below $A[a+j] < A[a+k] < B[b+k]$, at most $k-1$ of them; for $j \geq k+1$, the value $B[b+j]$ exceeds $B[b+1..b+k]$ and $A[a+1..a+k]$. Hence

$$\begin{aligned} \text{rank } A[a+j] &\leq (j-1) + (k-1) + 1 \leq m-1 & (j \leq d), \\ \text{rank } B[b+j] &\geq 2k+1 \geq m+1 & (j \geq k+1). \end{aligned}$$

So we discard d elements below the median and $m-k = d$ elements above it: what remains is $A[a+d+1..a+m] \cup B[b+1..b+k]$, two blocks of size $m-d = k$, and the median is its $(m-d)^{\text{th}} = k^{\text{th}}$ smallest element, which is the invariant with $(a+d, b, k)$. When $m = 1$ the median is the smaller of the two remaining values.

Queries. Each iteration makes 2 queries and replaces m by $\lceil m/2 \rceil$, so the query count $Q(m)$ satisfies $Q(m) \leq Q(\lceil m/2 \rceil) + 2$, $Q(1) = 2$; this is recurrence (5.16) of Solved Exercise 1, giving

$$Q(n) \leq 2\lceil \log_2 n \rceil + 2 = O(\log n).$$

Problem 5.2. Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, \dots, a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

Solution. Run Sort-and-Count (Section 5.3), but before each merge of the sorted halves A (first half) and B (second half) count the cross pairs $x \in A$, $y \in B$ with $x > 2y$ by a separate two-pointer pass.

```

Sort-and-Count-Significant(L):
    if |L| <= 1: return (0, L)
    split L into first half and second half
    (rA, A) = Sort-and-Count-Significant(first half)

```

```

(rB, B) = Sort-and-Count-Significant(second half)
c = 0; p = 1
for each y in B, in increasing order:
    while p <= |A| and A[p] <= 2y: p = p + 1
    c = c + (|A| - p + 1)
L' = Merge(A, B)          (ordinary merge, as in Mergesort)
return (rA + rB + c, L')

```

Correctness. By induction on $|L|$. A significant inversion (i, j) either has both indices in one half, and is counted in r_A or r_B by induction, or has i in the first half and j in the second; whether $a_i > 2a_j$ depends only on the two values, so the cross pairs are exactly the pairs $(x, y) \in A \times B$ with $x > 2y$, regardless of the order inside each half. Since A is sorted, $\{x \in A : x \leq 2y\}$ is a prefix of A , and since $2y$ increases as y runs through B in increasing order, this prefix only grows; so the pointer p never moves back, and when y is processed $A[p..|A|]$ are exactly the $x \in A$ with $x > 2y$. Hence c is the number of cross significant inversions. The returned list is L sorted, as in Mergesort, which the parent call needs.

Running time. The counting pass moves p at most $|A|$ times and visits each y once, and the merge is linear, so the combining work is $O(n)$ and the running time satisfies recurrence (5.1), $T(n) \leq 2T(n/2) + cn$; by (5.2), $T(n) = O(n \log n)$. (To compare two orderings, relabel as in Section 5.3 so that one ordering is $1, 2, \dots, n$ and read off the other as $a_1, \dots, a_n$.)

Problem 5.3. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are *equivalent* if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Solution. Split the cards into two halves, recursively find a "majority card" of each half (a card whose class contains more than half of that half), and test each of the at most two candidates against all n cards.

```

Majority(S):          returns a card of a class of size > |S|/2, or none
    if |S| = 1: return the card in S
    split S into S1, S2 of sizes floor(|S|/2), ceil(|S|/2)
    for c in {Majority(S1), Majority(S2)} (skip "none"):
        test c against every other card of S
        if more than |S|/2 cards of S are equivalent to c: return c
    return none

```

The answer to the bank's question is "yes" if and only if Majority(all n cards) returns a card.

Correctness. By induction on $|S|$; the base case is clear. A class of size more than $|S|/2$ is unique, so it suffices to show that if a class K has $|K \cap S| > |S|/2$, then a card of K is among the candidates. Indeed

$$|K \cap S_1| + |K \cap S_2| = |K \cap S| > \frac{|S_1|}{2} + \frac{|S_2|}{2},$$

so $|K \cap S_i| > |S_i|/2$ for some i , and by induction Majority(S_i) returns a card of K , which then passes the count. Conversely, a returned card passed the count, so its class is a majority of S .

Tests. Each call makes at most $2(|S| - 1)$ tests besides its recursive calls, so the number of tests satisfies $T(n) \leq 2T(n/2) + 2n$, recurrence (5.1); by (5.2), $T(n) = O(n \log n)$.

Problem 5.4. You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, \dots, n\}$ on the real line; and at each of these points j , they have a particle with charge q_j . (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j , by Coulomb's Law, is equal to

$$F_j = \sum_{i < j} \frac{C q_i q_j}{(j - i)^2} - \sum_{i > j} \frac{C q_i q_j}{(j - i)^2}.$$

They've written the following simple program to compute F_j for all j :

```

For j = 1, 2, ..., n
  Initialize F_j to 0
  For i = 1, 2, ..., n
    If i < j then
      Add C q_i q_j / (j - i)^2 to F_j
    Else if i > j then
      Add - C q_i q_j / (j - i)^2 to F_j
    Endif
  Endfor
  Output F_j
Endfor

```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i , takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Solution. Each F_j is Cq_j times one coordinate of the convolution of the charge vector with a fixed vector of signed inverse squares, so one FFT convolution computes them all.

Let

$$g(t) = \begin{cases} 1/t^2, & t > 0, \\ 0, & t = 0, \\ -1/t^2, & t < 0, \end{cases}$$

and define vectors indexed from 0:

$$\begin{aligned} a &= (a_0, \dots, a_{n-1}), & a_p &= q_{p+1}, \\ b &= (b_0, \dots, b_{2n-2}), & b_r &= g(r - (n - 1)). \end{aligned}$$

Then the two sums in F_j combine into one, $F_j = Cq_j \sum_{i=1}^n q_i g(j - i)$. In the convolution $a * b$ (Section 5.6), $(a * b)_s = \sum_{p+r=s} a_p b_r$; putting $p = i - 1$ forces $r = s - i + 1$, and $b_{s-i+1} = g(j - i)$ exactly when $s = j + n - 2$, where every $i \in \{1, \dots, n\}$ gives a valid index $r = j - i + n - 1 \in [0, 2n - 2]$. Hence

$$F_j = Cq_j (a * b)_{j+n-2}, \quad j = 1, \dots, n.$$

Algorithm. Build b in $O(n)$ time, pad a with zeros to length $2n - 1$, compute $a * b$ with the Fast Fourier Transform, and output $F_j = Cq_j (a * b)_{j+n-2}$ for each j .

Running time. Building the vectors and the final n multiplications take $O(n)$; by (5.15) the convolution of two vectors of length $2n - 1$ takes $O(n \log n)$. Total: $O(n \log n)$.

Problem 5.5. *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, ... well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, \dots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x -coordinate x_0 if its y -coordinate at x_0 is greater than the y -coordinates of all the other lines at x_0 : $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x -coordinate at which it is uppermost; intuitively, some portion of it can be seen if you look down from " $y = \infty$."

Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all of the ones that are visible. (Figure 5.10 shows an instance with five lines $L_1, \dots, L_5$ in which every line except L_2 is visible.)

Solution. Sort the lines by slope and compute the upper envelope by divide and conquer: the envelopes of the low-slope half and the high-slope half cross exactly once, so they merge in linear time.

Preprocessing. Sort the lines by slope. Among lines of equal slope (parallel lines) all but the one with the largest intercept lie strictly below it everywhere and are invisible; delete them in one $O(n)$ pass (lines are assumed distinct). Now $a_1 < a_2 < \dots < a_n$.

Recursion. For a set of lines with consecutive slopes, return its visible lines $\ell_1, \dots, \ell_p$ in increasing slope order together with breakpoints $\beta_1 < \dots < \beta_{p-1}$ such that ℓ_i is uppermost exactly on the open interval (β_{i-1}, β_i) (with $\beta_0 = -\infty, \beta_p = +\infty$). A single line returns itself with no breakpoints. Otherwise solve the lower-slope half $\mathcal{L}$, giving $(\ell_i; \beta_i)$ and envelope $U_L(x) = \max_{L \in \mathcal{L}}(ax + b)$, and the higher-slope half $\mathcal{R}$, giving $(r_i; \gamma_i)$ and U_R . Merge:

```
walk the merged sorted list of breakpoints beta and gamma, keeping track
  of the current envelope line of each half (two pointers);
find the elementary interval on which  $U_R - U_L$  changes sign;
there  $\ell_s$  and  $r_t$  are the current lines;  $x^* = \text{intersection of } \ell_s \text{ and } r_t$ ;
return lines  $\ell_1.. \ell_s, r_t.. r_q$ 
  with breakpoints  $\beta_1.. \beta_{s-1}, x^*, \gamma_t.. \gamma_{q-1}$ 
```

Correctness. U_L and U_R are convex, piecewise linear, with every slope of U_L at most $\max_{\mathcal{L}} a$ and every slope of U_R at least $\min_{\mathcal{R}} a > \max_{\mathcal{L}} a$. Hence $D = U_R - U_L$ is continuous with all one-sided derivatives at least $\delta = \min_{\mathcal{R}} a - \max_{\mathcal{L}} a > 0$, so

$$D(x) \rightarrow \mp \infty \quad (x \rightarrow \mp \infty), \quad D(x) < 0 \iff x < x^*$$

for a unique x^* , which the walk finds (on the elementary interval containing it both envelopes are single lines ℓ_s, r_t , and x^* is their intersection). A line of $\mathcal{L}$ is uppermost among all lines at x iff it is uppermost in $\mathcal{L}$ at x and $U_L(x) > U_R(x)$, i.e. $x \in (\beta_{i-1}, \beta_i) \cap (-\infty, x^*)$; this is nonempty iff $\beta_{i-1} < x^*$, i.e. $i \leq s$, and ℓ_s is then uppermost on (β_{s-1}, x^*) . Symmetrically r_j is visible iff $j \geq t$, with r_t uppermost on (x^*, γ_t) . (x^* is not a breakpoint of either half, since that would make three lines concurrent.) So the returned lines and breakpoints satisfy the specification, and the top-level call returns exactly the visible lines.

Running time. Sorting is $O(n \log n)$. The merge walks $O(n)$ breakpoints with two pointers and does $O(1)$ work at each, so $T(n) \leq 2T(n/2) + cn$, recurrence (5.1), and $T(n) = O(n \log n)$ by (5.2).

Problem 5.6. Consider an n -node complete binary tree T , where $n = 2^d - 1$ for some d . Each node v of T is labeled with a real number x_v . You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T , but the labeling is only specified in the following *implicit* way: for each node v , you can determine the value x_v by *probing* the node v . Show how to

find a local minimum of T using only $O(\log n)$ probes to the nodes of T .

Solution. Walk down from the root, always stepping to the smaller child, and stop at the first node smaller than both its children (or at a leaf).

```

v = root; probe v
while v is not a leaf:
    probe the children c1, c2 of v
    if x_v < x_c1 and x_v < x_c2: return v
    v = the child with the smaller label
return v

```

Correctness. Every node v the walk visits satisfies the invariant

$$v = \text{root} \quad \text{or} \quad x_v < x_{\text{parent}(v)}.$$

It holds at the root vacuously, and when we step from v to its smaller child c we have $x_c = \min(x_{c_1}, x_{c_2}) < x_v$ (we did not stop at v , and labels are distinct). The neighbours of v are its parent (if any) and its children (if any). If we return an internal v , it beats both children by the stopping test and its parent by the invariant; if we return a leaf, its only neighbour is its parent, beaten by the invariant. Either way v is a local minimum.

Probes. The walk visits at most one node per level, and there are d levels; it probes the root once and two children per non-leaf visited, so

$$\# \text{probes} \leq 1 + 2(d - 1) = 2 \log_2(n + 1) - 1 = O(\log n).$$

Problem 5.7. Suppose now that you're given an $n \times n$ grid graph G . (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j) , where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i - k| + |j - \ell| = 1$.)

We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v ; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G . (Note that G has n^2 nodes.)

Solution. Recurse on a shrinking rectangle S : probe its middle row and middle column (the cross C), and move into the quadrant that contains a node smaller than everything on the quadrant's border. For a rectangle S of nodes write ∂S for the set of nodes outside S adjacent to some node of S . Call (S, u) good if either $S = G$ (so $\partial S = \emptyset$), or $u \in S$ is a probed node with $x_u < x_y$ for all $y \in \partial S$.

```

S = G, u = none
loop:
    C = middle row and middle column of S; probe all of C
    w = node of minimum label in C + {u}
    if w == u and u is not in C: z = u
    else:
        probe the (at most 4) neighbours of w in G
        if w is smaller than all of them: return w
        z = the smallest neighbour of w
    S = the quadrant of S - C containing z; u = z

```

Correctness. The four quadrants are the connected pieces of $S \setminus C$, and for each quadrant Q every neighbour of Q lies in C (if inside S) or in ∂S (if outside), i.e. $\partial Q \subseteq C \cup \partial S$. Assume (S, u) is good. By the choice of w and goodness,

$$x_w \leq x_c \ (c \in C), \quad x_w \leq x_u < x_y \ (y \in \partial S).$$

(i) $w = u \notin C$: then u lies in a quadrant Q , and $x_u < x_y$ for every $y \in C \cup \partial S \supseteq \partial Q$, so (Q, u) is good.
(ii) $w \in C$ and w beats all its neighbours: w is a local minimum of G , and we return it.

(iii) $w \in C$ with a smaller neighbour z : then $x_z < x_w$, so $z \notin C$ and $z \notin \partial S$ by the display; hence $z \in S \setminus C$ lies in a quadrant Q , and $x_z < x_w \leq x_y$ for all $y \in C \cup \partial S \supseteq \partial Q$, so (Q, z) is good.

Goodness is therefore preserved, and S strictly shrinks each round while staying nonempty (it contains u), so the loop ends in case (ii) with a local minimum.

Probes. If S has both sides at most m , the round probes at most $2m$ nodes of C plus 4 neighbours (u was already probed), and each quadrant has both sides at most $\lfloor m/2 \rfloor$. Starting from $m = n$,

$$\begin{aligned} P(n) &\leq \sum_{k \geq 0, n/2^k \geq 1} \left(2 \cdot \frac{n}{2^k} + 4 \right) \\ &\leq 4n + 4(\log_2 n + 1) = O(n). \end{aligned}$$

6 Dynamic Programming

Contents

6.1 Exercises 6.1–6.7	59
6.2 Exercises 6.8–6.14	65
6.3 Exercises 6.15–6.21	72
6.4 Exercises 6.22–6.28	78
6.5 Exercise 6.29	84

6.1 Exercises 6.1–6.7

Problem 6.1. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \dots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i , we associate a positive integer *weight* w_i .

Consider, for example, the five-node path drawn in Figure 6.28 (weights 1, 8, 6, 3, 6 in order; the maximum weight of an independent set is 14). The *weights* are the numbers drawn inside the nodes. The goal in this question is to solve the following problem: *Find an independent set in a path G whose total weight is as large as possible.*

(a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
Start with S equal to the empty set
While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
Endwhile
Return S
```

(b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let S_1 be the set of all v_i where i is an odd number
Let S_2 be the set of all v_i where i is an even number
(Note that S_1 and S_2 are both independent sets)
Determine which of S_1 or S_2 has greater total weight,
and return this one
```

(c) Give an algorithm that takes an n -node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n , independent of the values of the weights.

Solution. (a) Take the path with weights 2, 3, 2. Heaviest-first picks v_2 (weight 3), deletes v_1, v_3 , and returns $\{v_2\}$ of weight 3, while $\{v_1, v_3\}$ is independent of weight 4.

(b) Take the path with weights 3, 1, 1, 3. Then $w(S_1) = 3 + 1 = 4 = w(S_2)$, so the algorithm returns weight 4, while $\{v_1, v_4\}$ is independent of weight 6.

(c) Let $\text{OPT}(j)$ be the maximum weight of an independent set in the subpath $v_1, \dots, v_j$; this is weighted interval scheduling with $p(j) = j - 2$, so (6.1) gives

$$\text{OPT}(j) = \max(w_j + \text{OPT}(j - 2), \text{OPT}(j - 1)), \quad \text{OPT}(0) = \text{OPT}(-1) = 0.$$

Directly: an optimal set S for $v_1, \dots, v_j$ either omits v_j , and is then an optimal set for $v_1, \dots, v_{j-1}$, or contains v_j , hence omits its only earlier neighbour v_{j-1} , and $S - \{v_j\}$ is an optimal set for $v_1, \dots, v_{j-2}$ (any independent set there plus v_j is independent).

```
M[-1] = M[0] = 0
for j = 1 to n: M[j] = max(w_j + M[j-2], M[j-1])
trace back from j = n: if w_j + M[j-2] >= M[j-1]
    then put v_j in S and go to j-2, else go to j-1
```

By induction on j (exactly as in (6.3)) $M[j] = \text{OPT}(j)$, and the traceback returns a set achieving $M[n]$ as in (6.5). Each of the n table entries takes $O(1)$ arithmetic operations and the traceback takes $O(n)$ steps, so the running time is $O(n)$, independent of the weights.

Problem 6.2. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of

possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i , then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i , it's required that they do no job (of either type) in week $i - 1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i - 1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i - 1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i , you add ℓ_i to the value if you choose "low-stress" in week i , and you add h_i to the value if you choose "high-stress" in week i . (You add 0 if you choose "none" in week i .)

The problem. Given sets of values $\ell_1, \ell_2, \dots, \ell_n$ and $h_1, h_2, \dots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	Week 2	Week 3	Week 4
ℓ_i	10	1	10	10
h_i	5	50	5	1

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```

For iterations i = 1 to n
  If  $h_{i+1} > \ell_i + \ell_{i+1}$  then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i+1"
    Continue with iteration i+2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i+1
Endif
End

```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$. In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an efficient algorithm that takes values for $\ell_1, \ell_2, \dots, \ell_n$ and $h_1, h_2, \dots, h_n$ and returns the *value* of an optimal plan.

Solution. (a) Take $n = 3$ with $(\ell_1, \ell_2, \ell_3) = (2, 2, 2)$ and $(h_1, h_2, h_3) = (1, 5, 10)$. At $i = 1$ the test $h_2 = 5 > \ell_1 + \ell_2 = 4$ succeeds, so the algorithm outputs none, high-stress for weeks 1, 2; at $i = 3$ the test $h_4 = 0 > \ell_3 + \ell_4 = 2$ fails, so it outputs low-stress for week 3. Its plan has value $0 + 5 + 2 = 7$. The correct answer is the plan low-stress, none, high-stress, of value $2 + 0 + 10 = 12$ (optimal by checking all 3^3 plans).

(b) Let $\text{OPT}(i)$ be the maximum value of a plan for weeks $1, \dots, i$, with $\text{OPT}(0) = \text{OPT}(-1) = 0$. Then for $i \geq 1$

$$\text{OPT}(i) = \max(\ell_i + \text{OPT}(i-1), h_i + \text{OPT}(i-2)).$$

Compute $M[i]$ by this recurrence for $i = 1, \dots, n$ and return $M[n]$.

Correctness, by induction on i : consider an optimal plan P for weeks $1, \dots, i$ and its choice in week i .

(i) Low-stress: no constraint links week i to earlier weeks, so P restricted to weeks $1, \dots, i-1$ is an arbitrary plan, and $\text{OPT}(i) = \ell_i + \text{OPT}(i-1)$. (ii) High-stress: week $i-1$ (if it exists) is "none", and weeks $1, \dots, i-2$ form an arbitrary plan, so $\text{OPT}(i) = h_i + \text{OPT}(i-2)$. (iii) None: replacing "none"

by “low-stress” in week i keeps the plan valid (a low-stress week constrains nothing) and gains $\ell_i > 0$, so this case never beats (i). Conversely each term in the max is the value of a valid plan (in case (ii), append “none”, “high-stress” to an optimal plan for weeks $1, \dots, i-2$; for $i=1$, just “high-stress”), so $\text{OPT}(i)$ equals the max.

Each of the n entries costs $O(1)$, so the running time is $O(n)$.

Problem 6.3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \dots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i , $i = 1, 2, \dots, n-1$, there is at least one edge of the form (v_i, v_j) .

The length of a path is the number of edges in it. The goal in this question is to solve the following problem: *Given an ordered graph G , find the length of the longest path that begins at v_1 and ends at v_n .* (Figure 6.29 shows an example: the ordered graph on $v_1, \dots, v_5$ with edges (v_1, v_2) , (v_1, v_4) , (v_2, v_4) , (v_2, v_5) , (v_3, v_4) , (v_4, v_5) . The correct answer for it is 3: the longest path from v_1 to v_n uses the three edges (v_1, v_2) , (v_2, v_4) , and (v_4, v_5) .)

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

```

Set w = v_1
Set L = 0
While there is an edge out of the node w
    Choose the edge (w, v_j)
        for which j is as small as possible
    Set w = v_j
    Increase L by 1
end while
Return L as the length of the longest path

```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n . (Again, the *length* of a path is the number of edges in the path.)

Solution. (a) Take $n = 5$ with edges (v_1, v_2) , (v_1, v_3) , (v_2, v_5) , (v_3, v_4) , (v_4, v_5) (ordered: every edge increases the index and $v_1, \dots, v_4$ each have an out-edge). The algorithm walks $v_1 \rightarrow v_2 \rightarrow v_5$ and returns 2; the correct answer is 3, via $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5$.

(b) Let $\text{OPT}(j)$ be the length of the longest path from v_1 to v_j , with $\text{OPT}(j) = -\infty$ if there is none. Since every edge into v_j comes from some v_i with $i < j$ by (i), the last edge of such a path is some $(v_i, v_j) \in E$ and the rest is a path from v_1 to v_i , which may be taken longest; conversely any v_1 - v_i path extended by (v_i, v_j) is a v_1 - v_j path (indices strictly increase, so no node repeats). Hence

$$\text{OPT}(1) = 0, \quad \text{OPT}(j) = \max_{i < j, (v_i, v_j) \in E} (\text{OPT}(i) + 1) \quad (j \geq 2),$$

with the max of an empty set (or of $-\infty$ terms) equal to $-\infty$.

```

M[1] = 0; M[j] = -infinity for j = 2..n
for i = 1 to n-1:
    if M[i] > -infinity:
        for each edge (v_i, v_j): M[j] = max(M[j], M[i] + 1)
return M[n]

```

When node v_i is processed, all its in-edges come from $v_1, \dots, v_{i-1}$, which were processed earlier, so by induction on i we have $M[i] = \text{OPT}(i)$ at that moment, and the loop returns $\text{OPT}(n)$. This is finite: by (ii) the walk from v_1 that always follows some out-edge strictly increases the index, so it ends at the only node without out-edges, v_n . Each node and each edge is handled once, so the running time is $O(n + m)$ for $m = |E|$, with G given as adjacency lists.

Problem 6.4. Suppose you're running a lightweight consulting business—just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i , you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i , and then out of the other city in month $i + 1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations—each one equal to either NY or SF—such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem. Given a value for the moving cost M , and sequences of operating costs $N_1, \dots, N_n$ and $S_1, \dots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations [NY, NY, SF, SF], with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```

For i = 1 to n
  If N_i < S_i then
    Output "NY in Month i"
  Else
    Output "SF in Month i"
End

```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an example of an instance in which every optimal plan must move (i.e., change locations) at least three times. Provide a brief explanation, saying why your example has this property.

(c) Give an efficient algorithm that takes values for n , M , and sequences of operating costs $N_1, \dots, N_n$ and $S_1, \dots, S_n$, and returns the *cost* of an optimal plan.

Solution. (a) Take $n = 2$, $M = 10$, $(N_1, N_2) = (1, 2)$, $(S_1, S_2) = (2, 1)$. The algorithm outputs [NY, SF], of cost $1 + 1 + 10 = 12$. The correct answer is [NY, NY] (or [SF, SF]), of cost 3; the four plans NN, NS, SN, SS cost 3, 12, 14, 3.

(b) Take $n = 4$, $M = 1$, $(N_1, \dots, N_4) = (1, 10, 1, 10)$, $(S_1, \dots, S_4) = (10, 1, 10, 1)$. The plan [NY, SF, NY, SF] moves three times and costs $4 \cdot 1 + 3 \cdot 1 = 7$. Every other plan is in the wrong city in some month, since the cheap city alternates, so it costs at least

$$10 + 3 \cdot 1 = 13 > 7.$$

Hence this plan is the unique optimum, and it moves three times.

(c) Let $\text{OPT}_N(i)$ and $\text{OPT}_S(i)$ be the minimum cost of a plan for months $1, \dots, i$ that ends in NY, respectively SF. Then $\text{OPT}_N(1) = N_1$, $\text{OPT}_S(1) = S_1$, and for $i \geq 2$

$$\begin{aligned} \text{OPT}_N(i) &= N_i + \min(\text{OPT}_N(i-1), M + \text{OPT}_S(i-1)), \\ \text{OPT}_S(i) &= S_i + \min(\text{OPT}_S(i-1), M + \text{OPT}_N(i-1)). \end{aligned}$$

Compute both tables for $i = 1, \dots, n$ and return $\min(\text{OPT}_N(n), \text{OPT}_S(n))$.

Correctness, by induction on i : the cost of a plan for months $1, \dots, i$ ending in NY is N_i , plus M if month $i - 1$ was SF, plus the cost of its prefix on months $1, \dots, i - 1$, which is an arbitrary plan ending in the city of month $i - 1$; minimizing the prefix separately for each of the two possible cities of month

$i - 1$ gives the first line, and symmetrically the second. Every optimal plan ends in one of the two cities, giving the returned value. There are $2n$ entries, each computed in $O(1)$, so the running time is $O(n)$.

Problem 6.5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the *word segmentation problem*—inferring likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like “meetateight” and deciding that the best segmentation is “meet at eight” (and not “me et at eight,” or “meet ate ight,” or any of a huge number of even less plausible alternatives). How could we automate this process? A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative “quality” of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2 \cdots x_k$, will return a number $\text{quality}(x)$. This number can be either positive or negative; larger numbers correspond to more plausible English words. (So $\text{quality}(\text{“me”})$ would be positive, while $\text{quality}(\text{“ght”})$ would be negative.)

Given a long string of letters $y = y_1y_2 \cdots y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The *total quality* of a segmentation is determined by adding up the qualities of each of its blocks. (So we’d get the right answer above provided that $\text{quality}(\text{“meet”}) + \text{quality}(\text{“at”}) + \text{quality}(\text{“eight”})$ was greater than the total quality of any other segmentation of the string.)

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing $\text{quality}(x)$ as a single computational step.)

(A final note, not necessary for solving the problem: To achieve better performance, word segmentation software in practice works with a more complex formulation of the problem—for example, incorporating the notion that solutions should not only be reasonable at the word level, but also form coherent phrases and sentences. If we consider the example “theyouthevent,” there are at least three valid ways to segment this into common English words, but one constitutes a much more coherent phrase than the other two. If we think of this in the terminology of formal languages, this broader problem is like searching for a segmentation that also can be parsed well according to a grammar for the underlying language. But even with these additional criteria and constraints, dynamic programming approaches lie at the heart of a number of successful segmentation systems.)

Solution. With $q(i, j) = \text{quality}(y_iy_{i+1} \cdots y_j)$ and $\text{OPT}(j)$ the maximum total quality of a segmentation of the prefix $y_1 \cdots y_j$, conditioning on the last block as in (6.6) gives the recurrence below. If the last block of an optimal segmentation of $y_1 \cdots y_j$ is $y_i \cdots y_j$, the remaining blocks form a segmentation of $y_1 \cdots y_{i-1}$, which may be replaced by an optimal one; conversely every such combination is a segmentation of $y_1 \cdots y_j$. Hence

$$\text{OPT}(0) = 0, \quad \text{OPT}(j) = \max_{1 \leq i \leq j} (q(i, j) + \text{OPT}(i - 1)) \quad (1 \leq j \leq n).$$

```

M[0] = 0
for j = 1 to n:
    M[j] = max over i = 1..j of quality(y_i..y_j) + M[i-1]
    B[j] = an index i attaining that max
j = n
while j > 0: output block y_{B[j]}..y_j; j = B[j] - 1

```

By induction on j , $M[j] = \text{OPT}(j)$ (each $M[i - 1]$ used has $i - 1 < j$), and the traceback outputs, in reverse order, blocks whose qualities sum to $M[n] = \text{OPT}(n)$, as in (6.5); quality values may be negative, which does not affect this argument.

Computing $M[j]$ takes j black-box calls and $O(j)$ arithmetic, so the total is $\sum_{j=1}^n O(j) = O(n^2)$ steps, and the traceback is $O(n)$.

Problem 6.6. In a word processor, the goal of “pretty-printing” is to take text with a ragged right margin, like this,

Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.

and turn it into text whose right margin is as “even” as possible, like this.

Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be “even.” So suppose our text consists of a sequence of *words*, $W = \{w_1, w_2, \dots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L . We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of a partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, \dots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1} (c_i + 1) \right] + c_k \leq L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line – that is, the number of spaces left at the right margin.

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the *squares* of the slacks of all lines (including the last line) is minimized.

Solution. Let $\text{OPT}(j)$ be the minimum sum of squared slacks over valid formattings of $w_1, \dots, w_j$; the last line of such a formatting is some $w_i, \dots, w_j$, so

$$\text{OPT}(0) = 0, \quad \text{OPT}(j) = \min_{\substack{1 \leq i \leq j \\ S(i,j) \geq 0}} [\text{OPT}(i-1) + S(i,j)^2],$$

where $S(i,j) = L - (\sum_{t=i}^j c_t) - (j-i)$ is the slack of the line $w_i, \dots, w_j$ (the line is valid iff $S(i,j) \geq 0$).

We assume every $c_i \leq L$; otherwise no valid formatting exists at all.

$P[0] = 0$; for $t = 1..n$: $P[t] = P[t-1] + c_t$

$\text{OPT}[0] = 0$

for $j = 1..n$:

$\text{OPT}[j] = \text{infinity}$

 for $i = j$ down to 1:

$s = L - (P[j] - P[i-1]) - (j - i)$

 if $s < 0$: break

 if $\text{OPT}[i-1] + s*s < \text{OPT}[j]$: $\text{OPT}[j] = \text{OPT}[i-1] + s*s$; $\text{last}[j] = i$

 trace back: the final line is $w_{\text{last}[n]} .. w_n$, then recurse on $n' = \text{last}[n] - 1$

Correctness, by induction on j . A valid formatting of $w_1, \dots, w_j$ is exactly a valid last line $w_i, \dots, w_j$ together with a valid formatting of $w_1, \dots, w_{i-1}$ (empty when $i = 1$), and its cost is $S(i,j)^2$ plus the cost of that prefix formatting. Minimizing over the prefix first gives $\text{OPT}(i-1) + S(i,j)^2$ by the induction hypothesis, and minimizing over i gives the recurrence. The set of i is nonempty since $S(j,j) = L - c_j \geq 0$. The inner loop may stop at the first i with $S(i,j) < 0$, because $S(i,j)$ strictly decreases as i decreases. The answer is $\text{OPT}(n)$, and the stored $\text{last}[\cdot]$ pointers recover the lines.

Running time. The prefix sums make each $S(i,j)$ an $O(1)$ computation, and there are at most n choices

of i for each of the n values of j , so the time is $O(n^2)$.

Problem 6.7. As a solved exercise in Chapter 5, we gave an algorithm with $O(n \log n)$ running time for the following problem. We're looking at the price of a given stock over n consecutive days, numbered $i = 1, 2, \dots, n$. For each day i , we have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) We'd like to know: How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it, if we want to maximize the profit per share, $p(j) - p(i)$? (If there is no way to make money during the n days, we should conclude this instead.)

In the solved exercise, we showed how to find the optimal pair of days i and j in time $O(n \log n)$. But, in fact, it's possible to do better than this. Show how to find the optimal numbers i and j in time $O(n)$.

Solution. Scan the days left to right, maintaining the cheapest day m seen so far; the best sale on day j buys on day m , so the best pair is the best of these $n - 1$ candidates.

```
m = 1; best = 0; (bi, bj) = none
for j = 2..n:
    if p(j) - p(m) > best: best = p(j) - p(m); (bi, bj) = (m, j)
    if p(j) < p(m): m = j
if best = 0: report that no profit is possible
else: return (bi, bj)
```

Correctness. Write $m(j)$ for an index minimizing p over $\{1, \dots, j\}$; it satisfies the one-step recurrence $m(1) = 1$ and $m(j) = j$ if $p(j) < p(m(j-1))$, else $m(j) = m(j-1)$, which is exactly the update of m , so at the top of iteration j the variable m equals $m(j-1)$. Hence

$$\max_{1 \leq i < j \leq n} [p(j) - p(i)] = \max_{2 \leq j \leq n} \left[p(j) - \min_{i < j} p(i) \right] = \max_{2 \leq j \leq n} [p(j) - p(m(j-1))],$$

and the loop computes the maximum of the right side over j , recording a pair $(m(j-1), j)$ attaining it whenever it is positive. If every candidate is ≤ 0 , no pair makes money and the algorithm says so.

Running time. Each of the $n - 1$ iterations does $O(1)$ work, so the time is $O(n)$.

6.2 Exercises 6.8–6.14

Problem 6.8. The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots.

Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the i^{th} second, x_i robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, \dots, x_n$ in advance.
- You have at your disposal an *electromagnetic pulse* (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f(\cdot)$ so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the k^{th} second, and it has been j seconds since it was previously used, then it will destroy $\min(x_k, f(j))$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the j^{th} second, then it is capable of destroying up to $f(j)$ robots.

The problem. Given the data on robot arrivals $x_1, x_2, \dots, x_n$, and given the recharging function $f(\cdot)$, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

Example. Suppose $n = 4$, and the values of x_i and $f(i)$ are given by the following table.

i	1	2	3	4
x_i	1	10	10	1
$f(i)$	1	2	4	8

The best solution would be to activate the EMP in the 3rd and the 4th seconds. In the 3rd second, the EMP has gotten to charge for 3 seconds, and so it destroys $\min(10, 4) = 4$ robots; in the 4th second, the EMP has only gotten to charge for 1 second since its last use, and it destroys $\min(1, 1) = 1$ robot. This is a total of 5.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
Schedule-EMP( $x_1, \dots, x_n$ )
  Let  $j$  be the smallest number for which  $f(j) \geq x_n$ 
  (If no such  $j$  exists then set  $j = n$ )
  Activate the EMP in the  $n$ -th second
  If  $n - j \geq 1$  then
    Continue recursively on the input  $x_1, \dots, x_{n-j}$ 
    (i.e., invoke Schedule-EMP( $x_1, \dots, x_{n-j}$ ))
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes the data on robot arrivals $x_1, x_2, \dots, x_n$, and the recharging function $f(\cdot)$, and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

Solution. (a) Take $n = 2$ with

i	1	2
x_i	2	10
$f(i)$	2	3

No $j \leq 2$ has $f(j) \geq x_2 = 10$, so Schedule-EMP sets $j = 2$, fires only in second 2 and stops ($n - j = 0$), destroying $\min(10, f(2)) = 3$ robots. The correct answer is 4: firing in seconds 1 and 2 destroys $\min(2, f(1)) + \min(10, f(1)) = 2 + 2 = 4$ (the other schedules give 2, 3 and 0).

(b) Let $\text{OPT}(j)$ be the maximum number of robots destroyed in seconds $1, \dots, j$ by a schedule whose last activation is in second j ; with the convention that “second 0” is the drained start,

$$\text{OPT}(0) = 0, \quad \text{OPT}(j) = \max_{0 \leq i < j} [\text{OPT}(i) + \min(x_j, f(j - i))],$$

and the answer is $\text{OPT}(n)$.

```
OPT[0] = 0
for  $j = 1..n$ :
  OPT[j] = max over  $i = 0..j-1$  of OPT[i] + min( $x_j$ ,  $f(j - i)$ )
return OPT[n]
```

Correctness. A schedule ending in second j has a previous activation in some second $i < j$ (or $i = 0$ if second j is the first use), the EMP has charged for $j - i$ seconds, so second j contributes $\min(x_j, f(j - i))$, and the activations up to i form a schedule ending in second i whose kills are unaffected by what happens after i . Maximizing over that prefix gives $\text{OPT}(i)$ by induction on j , and maximizing over i gives the recurrence. Finally some optimal schedule activates in second n : if a schedule's last activation is at $k < n$ (or there is none, $k = 0$), adding an activation at n leaves every earlier kill unchanged and adds $\min(x_n, f(n - k)) \geq 0$. So the optimum over all schedules is $\text{OPT}(n)$.

Running time. Entry j takes $O(j)$ evaluations of f , so the total is $\sum_{j=1}^n O(j) = O(n^2)$.

Problem 6.9. You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i , you're presented with x_i terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's

running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n ; we assume $s_1 > s_2 > s_3 > \dots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data $x_1, x_2, \dots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, \dots, s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

Example. Suppose $n = 4$, and the values of x_i and s_i are given by the following table.

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.)

(a) Give an example of an instance with the following properties.

- There is a "surplus" of data in the sense that $x_i > s_1$ for every i .
- The optimal solution reboots the system at least twice.

In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

(b) Give an efficient algorithm that takes values for $x_1, x_2, \dots, x_n$ and $s_1, s_2, \dots, s_n$ and returns the total *number* of terabytes processed by an optimal solution.

Solution. (a) Take $n = 5$, $x_i = 11$ for every i , and $(s_1, \dots, s_5) = (10, 4, 3, 2, 1)$. The optimal solution reboots on days 2 and 4, processing

$$10 + 0 + 10 + 0 + 10 = 30,$$

whereas never rebooting gives $10 + 4 + 3 + 2 + 1 = 20$, and the best single reboot (on day 3) gives $10 + 4 + 0 + 10 + 4 = 28$.

(b) Let $\text{OPT}(i, j)$ be the maximum amount processed on days $i, \dots, n$ given that day i is the j^{th} day since the last reboot ($1 \leq j \leq i$). On day i we either process, or reboot and start day $i + 1$ fresh, so

$$\begin{aligned} \text{OPT}(n+1, j) &= 0, \\ \text{OPT}(i, j) &= \max\{\min(x_i, s_j) + \text{OPT}(i+1, j+1), \text{OPT}(i+1, 1)\}, \end{aligned}$$

and the answer is $\text{OPT}(1, 1)$.

```
for j = 1..n+1: OPT[n+1][j] = 0
for i = n down to 1:
  for j = 1..i:
    OPT[i][j] = max( min(x_i, s_j) + OPT[i+1][j+1], OPT[i+1][1] )
return OPT[1][1]
```

Correctness, by downward induction on i . The only state that matters for days $i, \dots, n$ is the number j of days since the last reboot, since it alone fixes the capacity s_j on day i if we process. If we process on day i we gain $\min(x_i, s_j)$ and day $i + 1$ is day $j + 1$ since the reboot; if we reboot we gain 0 and day $i + 1$ is day 1. In either case the rest of the schedule is an arbitrary schedule for days $i + 1, \dots, n$ from the resulting state, whose best value is $\text{OPT}(i+1, j+1)$ or $\text{OPT}(i+1, 1)$ by induction; the better of the two options is $\text{OPT}(i, j)$. Day 1 starts freshly rebooted, so the optimum is $\text{OPT}(1, 1)$. (Indices stay in range: $j \leq i$ forces $j+1 \leq n+1$.)

Running time. There are $\sum_{i=1}^n i = O(n^2)$ entries, each filled in $O(1)$ time, so the time is $O(n^2)$.

Problem 6.10. You're trying to run a large computing job in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call *A* and *B*) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i , you would be able to run $a_i > 0$ steps of the simulation if your job is on machine *A*, and $b_i > 0$ steps of the simulation if your job is on machine *B*. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is specified by a choice of *A*, *B*, or "move" for each minute, with the property that choices *A* and *B* cannot appear in consecutive minutes. For example, if your job is on machine *A* in minute i , and you want to switch to machine *B*, then your choice for minute $i + 1$ must be *move*, and then your choice for minute $i + 2$ can be *B*. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on *A*, plus the sum of b_i over all minutes in which the job is on *B*.

The problem. Given values $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines *A* or *B* in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
<i>A</i>	10	1	1	10
<i>B</i>	5	1	20	20

Then the plan of maximum value would be to choose *A* for minute 1, then *move* for minute 2, and then *B* for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```

In minute 1, choose the machine achieving the larger of a_1, b_1
Set i = 2
While i <= n
  What was the choice in minute i - 1?
  If A:
    If b_(i+1) > a_i + a_(i+1) then
      Choose move in minute i and B in minute i + 1
      Proceed to iteration i + 2
    Else
      Choose A in minute i
      Proceed to iteration i + 1
    Endif
  If B: behave as above with roles of A and B reversed
EndWhile

```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_n$ and returns the value of an optimal plan.

Solution. (a) Take $n = 3$ with

	Minute 1	Minute 2	Minute 3
<i>A</i>	2	1	1
<i>B</i>	1	10	10

The algorithm picks *A* in minute 1 (since $2 > 1$); at $i = 2$ it sees $b_3 = 10 > a_2 + a_3 = 2$, so it chooses move in minute 2 and *B* in minute 3, and stops. Its plan *A*, move, *B* has value $2 + 0 + 10 = 12$. The correct answer is 21, from the plan *B*, *B*, *B* with value $1 + 10 + 10$ (no plan can do better, since $\sum_i \max(a_i, b_i) = 22$ would require *A* then *B* in consecutive minutes).

(b) Let $A(i)$, $B(i)$, $M(i)$ be the maximum value of a plan for minutes $1, \dots, i$ whose minute i is *A*, *B*, move respectively, with $A(0) = B(0) = M(0) = 0$ standing for the empty plan. Minute i can be *A*

only if minute $i - 1$ is A or move (or absent), likewise for B , and a move may follow anything, so

$$\begin{aligned} A(i) &= a_i + \max\{A(i-1), M(i-1)\}, \\ B(i) &= b_i + \max\{B(i-1), M(i-1)\}, \\ M(i) &= \max\{A(i-1), B(i-1), M(i-1)\}, \end{aligned}$$

and the optimal value is $\max\{A(n), B(n), M(n)\}$.

```
A = B = M = 0
for i = 1..n:
    (A, B, M) = (a_i + max(A, M), b_i + max(B, M), max(A, B, M))
return max(A, B, M)
```

Correctness, by induction on i . The only constraint on a plan is that A and B are never in consecutive minutes, so a sequence ending in A at minute i is a plan iff its first $i - 1$ minutes form a plan whose last choice is not B (or $i = 1$). Its value is a_i plus the value of that prefix, and the best such prefix has value $\max\{A(i-1), M(i-1)\}$ by induction (for $i = 1$ this is 0, the empty plan). The same argument gives $B(i)$, and for $M(i)$ every plan for minutes $1, \dots, i - 1$ may be extended by move, contributing 0. Every plan for minutes $1, \dots, n$ ends in one of the three choices, so the optimum is the largest of the three final values.

Running time. Each of the n iterations does $O(1)$ work, so the time is $O(n)$.

Problem 6.11. Suppose you're consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected *supply* s_i of equipment (measured in pounds), which has to be shipped by an air freight carrier. Each week's supply can be carried by one of two air freight companies, A or B.

- Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a week's supply s_i).
- Company B makes contracts for a fixed amount c per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A *schedule*, for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The *cost* of the schedule is the total amount paid to company A and B, according to the description above.

Give a polynomial-time algorithm that takes a sequence of supply values $s_1, s_2, \dots, s_n$ and returns a *schedule* of minimum cost.

Example. Suppose $r = 1$, $c = 10$, and the sequence of values is

$$11, 9, 9, 12, 12, 12, 12, 9, 9, 11.$$

Then the optimal schedule would be to choose company A for the first three weeks, then company B for a block of four consecutive weeks, and then company A for the final three weeks.

Solution. Let $\text{OPT}(j)$ be the minimum cost of a schedule for weeks $1, \dots, j$; week j is either shipped by A, or it ends a B-block covering weeks $j - 3, \dots, j$, so

$$\text{OPT}(j) = \min(\text{OPT}(j-1) + rs_j, \text{OPT}(j-4) + 4c), \quad \text{OPT}(0) = 0,$$

where the second option is present only for $j \geq 4$.

Algorithm. Fill $M[0..n]$ by this recurrence for $j = 1, \dots, n$, recording which option attained the minimum; then trace back from $j = n$: if A was chosen, assign A to week j and go to $j - 1$; otherwise assign B to weeks $j - 3..j$ and go to $j - 4$.

Correctness. By induction on j . In any valid schedule for weeks $1..j$ the B-weeks split into consecutive blocks of four, so either week j is an A-week and the rest is a valid schedule for weeks $1..j - 1$, costing at least $\text{OPT}(j - 1)$, or week j is the last week of a B-block $\{j - 3, \dots, j\}$ and the rest is a valid schedule for weeks $1..j - 4$, costing at least $\text{OPT}(j - 4)$. Conversely, appending an A-week or a B-block to an optimal shorter schedule gives a valid schedule of the stated cost. Hence the recurrence holds, and the traceback produces a schedule achieving $\text{OPT}(n)$.

Running time. Each of the n entries takes $O(1)$ time, and the traceback is $O(n)$: total $O(n)$.

Problem 6.12. Suppose we want to replicate a file over a collection of n servers, labeled $S_1, S_2, \dots, S_n$. To place a copy of the file at server S_i results in a *placement cost* of c_i , for an integer $c_i > 0$.

Now, if a user requests the file from server S_i , and no copy of the file is present at S_i , then the servers $S_{i+1}, S_{i+2}, S_{i+3}, \dots$ are searched in order until a copy of the file is finally found, say at server S_j , where $j > i$. This results in an *access cost* of $j - i$. (Note that the lower-indexed servers $S_{i-1}, S_{i-2}, \dots$ are not consulted in this search.) The access cost is 0 if S_i holds a copy of the file. We will require that a copy of the file be placed at server S_n , so that all such searches will terminate, at the latest, at S_n .

We'd like to place copies of the files at the servers so as to minimize the sum of placement and access costs. Formally, we say that a *configuration* is a choice, for each server S_i with $i = 1, 2, \dots, n - 1$, of whether to place a copy of the file at S_i or not. (Recall that a copy is always placed at S_n .) The *total cost* of a configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all access costs associated with all n servers.

Give a polynomial-time algorithm to find a configuration of minimum total cost.

Solution. Let $\text{OPT}(j)$ be the minimum total cost (placement plus access) of servers $S_1, \dots, S_j$ over configurations that place a copy at S_j ; conditioning on the previous copy S_i ($i = 0$ meaning none),

$$\text{OPT}(j) = c_j + \min_{0 \leq i < j} \left(\text{OPT}(i) + \binom{j-i}{2} \right), \quad \text{OPT}(0) = 0,$$

and the answer is $\text{OPT}(n)$.

Algorithm. Compute $\text{OPT}(1), \dots, \text{OPT}(n)$ in order by the recurrence, storing the minimizing i for each j ; trace back from n through the stored indices to list the servers that receive a copy.

Correctness. Since searches only move to higher indices, the access cost of S_k depends only on the first copy at or after S_k . Fix a configuration of $S_1..S_j$ with a copy at S_j , and let S_i be the last copy before S_j ($i = 0$ if there is none). Each server S_k with $i < k < j$ finds its copy at S_j , at access cost $j - k$, while servers $S_1..S_i$ are served within the prefix ending at the copy S_i . So the cost splits as

$$\begin{aligned} \text{cost} &= [\text{cost of } S_1..S_i] + c_j + \sum_{k=i+1}^{j-1} (j - k) \\ &= [\text{cost of } S_1..S_i] + c_j + \binom{j-i}{2}. \end{aligned}$$

The bracket ranges over all configurations of $S_1..S_i$ with a copy at S_i , independently of the rest, so its minimum is $\text{OPT}(i)$. Minimizing over i gives the recurrence, and by induction on j the traceback yields a configuration of cost $\text{OPT}(n)$, which is optimal because every configuration has a copy at S_n .

Running time. Entry j takes a minimum over j terms, each computed in $O(1)$, so the total is $\sum_j O(j) = O(n^2)$.

Problem 6.13. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in n different companies. For each pair $i \neq j$, they maintain a trade ratio r_{ij} , meaning that one share of i trades for r_{ij} shares of j . Here we allow the rate r to be fractional; that is, $r_{ij} = \frac{2}{3}$ means that you can trade three shares of i to get two shares of j .

A *trading cycle* for a sequence of shares $i_1, i_2, \dots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2 , then shares in company i_2 for shares i_3 , and so on, finally trading shares in i_k back to shares in company i_1 . After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an *opportunity cycle*, if trading along

the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

Give a polynomial-time algorithm that finds such an opportunity cycle, if one exists.

Solution. Give the edge (i, j) of the complete directed graph on the n companies the cost $c_{ij} = -\log r_{ij}$; opportunity cycles are then exactly negative cycles, which the negative-cycle algorithm of Section 6.10 finds.

Algorithm. Build the directed graph G with nodes $1, \dots, n$ and an edge (i, j) of cost $c_{ij} = -\log r_{ij}$ for every $i \neq j$. Add a sink t with a cost-0 edge from every node to t , as in (6.29), and run the Bellman-Ford-based negative-cycle detection of Section 6.10 on it. If it reports no negative cycle, report that no opportunity cycle exists; otherwise output the negative cycle C it returns, which lies in G because no edge leaves t .

Correctness. For any cycle $C = i_1 i_2 \dots i_k i_1$ (write $i_{k+1} = i_1$),

$$\sum_{a=1}^k c_{i_a i_{a+1}} = -\log \prod_{a=1}^k r_{i_a i_{a+1}},$$

so C has negative cost if and only if the product of its ratios exceeds 1, i.e. if and only if C is an opportunity cycle. By (6.29) the augmented graph has a negative cycle with a path to t if and only if G has a negative cycle, and by (6.34) the algorithm finds such a cycle whenever one exists. So it outputs an opportunity cycle if and only if one exists.

Running time. The graph has $n + 1$ nodes and $m = n(n - 1) + n = n^2$ edges, so by (6.34) the running time is $O(mn) = O(n^3)$.

Problem 6.14. A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes V , and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1 , then to E_2 , then to E_3 , and so on, to an edge set E_b . For $i = 0, 1, 2, \dots, b$, let G_i denote the graph (V, E_i) . So if we were to watch the structure of the network on the nodes V as a “time lapse,” it would look precisely like the sequence of graphs $G_0, G_1, G_2, \dots, G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

Now consider two particular nodes $s, t \in V$. For an s - t path P in one of the graphs G_i , we define the *length* of P to be simply the number of edges in P , and we denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, \dots, P_b$ so that for each i , P_i is an s - t path in G_i . We want the paths to be relatively short. We also do not want there to be too many *changes* — points at which the identity of the path switches. Formally, we define $\text{changes}(P_0, P_1, \dots, P_b)$ to be the number of indices i ($0 \leq i \leq b - 1$) for which $P_i \neq P_{i+1}$.

Fix a constant $K > 0$. We define the *cost* of the sequence of paths $P_0, P_1, \dots, P_b$ to be

$$\text{cost}(P_0, P_1, \dots, P_b) = \sum_{i=0}^b \ell(P_i) + K \cdot \text{changes}(P_0, P_1, \dots, P_b).$$

- (a) Suppose it is possible to choose a single path P that is an s - t path in each of the graphs $G_0, G_1, \dots, G_b$. Give a polynomial-time algorithm to find the shortest such path.
- (b) Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \dots, P_b$ of minimum cost, where P_i is an s - t path in G_i for $i = 0, 1, \dots, b$.

Solution. (a) Run breadth-first search from s in the intersection graph $G^* = (V, E_0 \cap E_1 \cap \dots \cap E_b)$ and return the BFS-tree path to t . A path P is an s - t path in every G_i if and only if every edge of P lies in every E_i , i.e. if and only if P is an s - t path in G^* ; so the paths in question are exactly the s - t paths of G^* , and BFS finds one with the fewest edges (Section 3.2). With $m = \max_i |E_i|$, the intersection is built in $O(bm)$ time by marking edges (e.g. in an $n \times n$ array or a hash table), and BFS takes $O(m+n)$, for a total of $O(b(m+n))$.

(b) For $0 \leq i \leq j \leq b$ let $\ell(i, j)$ be the length of a shortest s - t path in $G_{i,j} = (V, E_i \cap \dots \cap E_j)$ (with $\ell(i, j) = \infty$ if there is none), and let $\text{OPT}(j)$ be the minimum cost of a sequence $P_0, \dots, P_j$. Splitting at the start i of the last stretch on which the path is constant,

$$\text{OPT}(j) = \min_{0 \leq i \leq j} (\text{OPT}(i-1) + K + (j-i+1)\ell(i, j)), \quad \text{OPT}(-1) = -K,$$

and the answer is $\text{OPT}(b)$; recording the minimizing i and a shortest path of $G_{i,j}$ lets us trace back the sequence.

Algorithm. For $j = 0, 1, \dots, b$: let $I = E_j$; for $i = j, j-1, \dots, 0$, replace I by $I \cap E_i$, compute $\ell(i, j)$ by BFS in (V, I) (stop the inner loop once it is ∞ , since intersections only shrink), and update the minimum above.

Correctness. Write $D(j)$ for the right-hand side. First, $D(j) \leq \text{OPT}(j)$: in an optimal sequence for $0..j$, let i be the first index with $P_i = P_{i+1} = \dots = P_j$. This common path P is an s - t path in each of $G_i, \dots, G_j$, so in $G_{i,j}$ by part (a), and $\ell(P) \geq \ell(i, j)$. If $i > 0$ then $P_{i-1} \neq P_i$ contributes one change, and $P_0..P_{i-1}$ costs at least $\text{OPT}(i-1)$; if $i = 0$ there is no change and the convention $\text{OPT}(-1) + K = 0$ accounts for it. Hence

$$\text{OPT}(j) \geq \text{OPT}(i-1) + K + (j-i+1)\ell(i, j) \geq D(j).$$

Second, $D(j) \geq \text{OPT}(j)$: take an optimal sequence for $0..i-1$ and append $j-i+1$ copies of a shortest path of $G_{i,j}$. This adds at most one change, so the new sequence costs at most $\text{OPT}(i-1) + K + (j-i+1)\ell(i, j)$, which is therefore $\geq \text{OPT}(j)$. By induction on j , $\text{OPT}(j) = D(j)$, and the traceback produces a sequence of cost exactly $\text{OPT}(b)$ (its cost is at most the computed value by the second argument, and at least $\text{OPT}(b)$ by definition).

Running time. There are $O(b^2)$ pairs (i, j) ; for each, one intersection step costs $O(m)$ with marked edges and one BFS costs $O(m+n)$. The total is $O(b^2(m+n))$, polynomial.

6.3 Exercises 6.15–6.21

Problem 6.15. On most clear days, a group of your friends in the Astronomy Department gets together to plan out the astronomical events they're going to try observing that night. We'll make the following assumptions about the events.

- There are n events, which for simplicity we'll assume occur in sequence separated by exactly one minute each. Thus event j occurs at minute j ; if they don't observe this event at exactly minute j , then they miss out on it.
- The sky is mapped according to a one-dimensional coordinate system (measured in degrees from some central baseline); event j will be taking place at coordinate d_j , for some integer value d_j . The telescope starts at coordinate 0 at minute 0.
- The last event, n , is much more important than the others; so it is required that they observe event n .

The Astronomy Department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one degree per minute. Thus

they do not expect to be able to observe all n events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event n must be observed.

We say that a subset S of the events is *viewable* if it is possible to observe each event $j \in S$ at its appointed time j , and the telescope has adequate time (moving at its maximum of one degree per minute) to move between consecutive events in S .

The problem. Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n . Such a solution will be called *optimal*.

Example. Suppose the one-dimensional coordinates of the events are as shown here.

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	-4	-1	4	5	-4	6	7	-2

Then the optimal solution is to observe events 1, 3, 6, 9. Note that the telescope has time to move from one event in this set to the next, even moving at one degree per minute.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```

Mark all events  $j$  with  $|d_n - d_j| > n - j$  as illegal (as
observing them would prevent you from observing event  $n$ )
Mark all other events as legal
Initialize current position to coordinate 0 at minute 0
While not at end of event sequence
    Find the earliest legal event  $j$  that can be reached without
    exceeding the maximum movement rate of the telescope
    Add  $j$  to the set  $S$ 
    Update current position to be coord.  $d_j$  at minute  $j$ 
Endwhile
Output the set  $S$ 

```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for the coordinates $d_1, d_2, \dots, d_n$ of the events and returns the *size* of an optimal solution.

Solution. (a) Take $n = 4$ with coordinates

Event	1	2	3	4
Coordinate	1	-1	-2	-1

Every event is legal ($|d_4 - d_j| = 2, 0, 1, 0 \leq 4 - j = 3, 2, 1, 0$). The algorithm takes event 1 (reachable since $|1 - 0| \leq 1$); from coordinate 1 at minute 1 it cannot reach event 2 ($|-1 - 1| = 2 > 1$) or event 3 ($|-2 - 1| = 3 > 2$), and it then takes event 4 ($|-1 - 1| = 2 \leq 3$). So it outputs $S = \{1, 4\}$, of size 2. The correct answer is $\{2, 3, 4\}$, of size 3: the moves $0 \rightarrow -1 \rightarrow -2 \rightarrow -1$ cover distances 1, 1, 1 in times 2, 1, 1. (All four events are not viewable, since $1 \rightarrow -1$ in one minute is impossible, so 3 is optimal.)

(b) Let $\text{OPT}(j)$ be the maximum size of a viewable set whose last event is j ($-\infty$ if none exists); then

$$\text{OPT}(j) = 1 + \max\left(\{0 : |d_j| \leq j\} \cup \{\text{OPT}(i) : i < j, |d_j - d_i| \leq j - i\}\right),$$

with $\max \emptyset = -\infty$, and the algorithm computes $\text{OPT}(1), \dots, \text{OPT}(n)$ in order and returns $\text{OPT}(n)$ (if this is $-\infty$, i.e. $|d_n| > n$, no viewable set contains event n).

Correctness. A set $S = \{j_1 < \dots < j_k\}$ is viewable exactly when every hop is fast enough,

$$\begin{aligned}
|d_{j_1} - 0| &\leq j_1 - 0, \\
|d_{j_{a+1}} - d_{j_a}| &\leq j_{a+1} - j_a \quad (1 \leq a < k),
\end{aligned}$$

since moving at full speed toward the next target and then waiting is always possible. Hence if S ends at j , either $S = \{j\}$ with $|d_j| \leq j$, or $S \setminus \{j\}$ is viewable, ends at some $i < j$ with $|d_j - d_i| \leq j - i$, and has size at most $\text{OPT}(i)$; conversely, appending j to an optimal viewable set ending at such an i (or taking $\{j\}$ alone) gives a viewable set of size $\text{OPT}(i) + 1$ (or 1). By induction on j the recurrence holds, and $\text{OPT}(n)$ is the size of an optimal solution, since an optimal solution must end at event n .

Running time. Entry j scans $j - 1$ earlier entries in $O(1)$ each, so the total is $O(n^2)$.

Problem 6.16. There are many sunny days in Ithaca, New York; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on a rainy day. The ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism she uses to do this.

Each ROTC person on campus except the ranking officer reports to a unique *superior officer*. Thus the reporting hierarchy can be described by a tree T , rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u . See Figure 6.30, in which A is the ranking officer, B and D are the direct subordinates of A , and C is the direct subordinate of B . (Figure 6.30: a tree with root A , children B and D , and C the only child of B . The fastest broadcast scheme is for A to call B in the first round; in the second round A calls D and B calls C . If A were to call D first, then C could not learn the news until the third round.)

To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone; for example, in Figure 6.30, A would not be allowed to call C .

We can picture this process as being divided into *rounds*. In one round, each person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates. For example, in Figure 6.30, it will take only two rounds if A starts by calling B , but it will take three rounds if A starts by calling D .

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

Solution. Compute bottom-up $r(v)$, the minimum number of rounds to notify the subtree T_v once v knows, by having every node call its children in decreasing order of r :

$$r(v) = \max_{1 \leq j \leq d} (j + r(c_j)), \quad r(c_1) \geq r(c_2) \geq \dots \geq r(c_d),$$

where $c_1, \dots, c_d$ are the children of v ($r(v) = 0$ for a leaf). The answer is $r(\text{root})$, and the calls are: each v , from the round after it is informed, calls $c_1, c_2, \dots, c_d$ in this order.

Correctness. Induction up the tree. Only nodes of T_c ever call into T_c , so in any scheme, once c is informed, T_c needs at least $r(c)$ further rounds (induction hypothesis), and our scheme inside T_c achieves exactly $r(c)$. Waiting idly never helps, so a scheme for T_v is an order $c_{\sigma(1)}, \dots, c_{\sigma(d)}$ of calls by v in rounds $1, \dots, d$, and the best it can do is $\max_j (j + r(c_{\sigma(j)}))$. If some adjacent pair has $a = c_{\sigma(j)}$, $b = c_{\sigma(j+1)}$ with $r(a) < r(b)$, swapping them replaces the two terms $j + r(a)$, $j + 1 + r(b)$ by

$$j + r(b) \leq j + 1 + r(b), \quad j + 1 + r(a) < j + 1 + r(b),$$

so the maximum does not increase. Repeated swaps reach the decreasing order without increasing the cost, so that order is optimal and gives $r(v)$.

Running time. Each node sorts its children once, so the total is $\sum_v O(d_v \log d_v) = O(n \log n)$.

Problem 6.17. Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a *rising trend*, as follows.

They have the closing price for a given stock recorded for n days in succession; let these prices be denoted $P[1], P[2], \dots, P[n]$. A *rising trend* in these prices is a subsequence of the prices $P[i_1], P[i_2], \dots, P[i_k]$, for days $i_1 < i_2 < \dots < i_k$, so that

- $i_1 = 1$, and
- $P[i_j] < P[i_{j+1}]$ for each $j = 1, 2, \dots, k - 1$.

Thus a rising trend is a subsequence of the days, beginning on the first day and not necessarily contiguous, so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose $n = 7$, and the sequence of prices is 10, 1, 2, 11, 3, 4, 12. Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

(a) Show that the following algorithm does not correctly return the *length* of the longest rising trend, by giving an instance on which it fails to return the correct answer.

```

Define i = 1
    L = 1
For j = 2 to n
    If P[j] > P[i] then
        Set i = j.
        Add 1 to L
    Endif
Endfor

```

In your example, give the actual length of the longest rising trend, and say what the algorithm above returns.

(b) Give an efficient algorithm that takes a sequence of prices $P[1], P[2], \dots, P[n]$ and returns the *length* of the longest rising trend.

Solution. (a) Take $n = 4$ and prices 1, 5, 2, 3. The algorithm jumps to day 2 (price 5), after which no later price exceeds 5, so it returns $L = 2$; but days 1, 3, 4 give the rising trend $1 < 2 < 3$, and the true length is 3 (all four days do not rise, as $5 > 2$).

(b) Let $\text{OPT}(j)$ be the length of the longest rising trend whose last day is j , with $\text{OPT}(j) = -\infty$ if none exists. Then

$$\text{OPT}(1) = 1, \quad \text{OPT}(j) = 1 + \max\{\text{OPT}(i) : 1 \leq i < j, P[i] < P[j]\} \quad (j \geq 2),$$

with $\max \emptyset = -\infty$, and the answer is $\max_{1 \leq j \leq n} \text{OPT}(j)$ (which is at least $\text{OPT}(1) = 1$). Fill the array for $j = 1, 2, \dots, n$.

Correctness. A rising trend ending at day $j \geq 2$ has at least two days (it starts at day 1); deleting day j leaves a rising trend ending at some $i < j$ with $P[i] < P[j]$, and conversely appending j to any rising trend ending at such an i gives a rising trend ending at j . So the recurrence holds, by induction on j .

Running time. Entry j scans $j - 1$ earlier entries, so the total is $\sum_j O(j) = O(n^2)$.

Problem 6.18. Consider the sequence alignment problem over a four-letter alphabet $\{z_1, z_2, z_3, z_4\}$, with a given gap cost and given mismatch costs. Assume that each of these parameters is a positive integer.

Suppose you are given two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$ and a proposed alignment between them. Give an $O(mn)$ algorithm to decide whether this alignment is the *unique* minimum-cost alignment between A and B .

Solution. Run the alignment recurrence (6.16) while also counting optimal alignments, with counts capped at 2; the proposed alignment M is the unique optimum iff its cost equals $\text{OPT}(m, n)$ and the capped count is 1.

Counting paths in G_{XY} would be wrong, since an alignment with consecutive gaps on both sides corresponds to several paths. So split the alignments of $A_i = a_1 \dots a_i$ and $B_j = b_1 \dots b_j$ into three disjoint classes, exhaustive by (6.14): (i) $(i, j) \in M$; (ii) a_i unmatched; (iii) a_i matched but $(i, j) \notin M$, so b_j unmatched. Keep two tables of pairs (minimum cost, number of alignments attaining it, capped at 2): $Q(i, j)$ over all alignments of A_i, B_j , and $P(i, j)$ over those in which a_i is matched, i.e. classes (i) and (iii). Removing (i, j) , resp. a_i , resp. b_j is a bijection from class (i), (ii), (iii) onto all alignments of (A_{i-1}, B_{j-1}) , all of (A_{i-1}, B_j) , and those of (A_i, B_{j-1}) with a_i matched, so

$$\begin{aligned}
 P(i, j) &= \text{comb}(\alpha_{a_i b_j} + Q(i-1, j-1), \delta + P(i, j-1)), \\
 Q(i, j) &= \text{comb}(P(i, j), \delta + Q(i-1, j)),
 \end{aligned}$$

where adding a constant shifts the cost and keeps the count, and comb takes the smaller cost with the sum (capped at 2) of the counts of the terms attaining it. The base cases are $Q(i, 0) = (i\delta, 1)$, $Q(0, j) = (j\delta, 1)$, $P(i, 0) = P(0, j) = (\infty, 0)$. By induction on $i + j$, the first coordinate of $Q(i, j)$ is $\text{OPT}(i, j)$ and the second is $\min(2, \#\{\text{optimal alignments of } A_i, B_j\})$; capping commutes with addition, since $\min(2, x + y) = \min(2, \min(2, x) + \min(2, y))$.

Finally, compute $\text{cost}(M) = \delta(m+n-2|M|) + \sum_{(i,j) \in M} \alpha_{a_i b_j}$ and answer yes iff $Q(m, n) = (\text{cost}(M), 1)$: the first coordinate says M is optimal, the second that no other alignment is.

Running time. There are $2(m+1)(n+1)$ table entries, each computed in $O(1)$ from earlier ones, and $\text{cost}(M)$ takes $O(m+n)$; the total is $O(mn)$.

Problem 6.19. You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k . So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'' , so that s' is a repetition of x and s'' is a repetition of y . (So each symbol in s must belong to exactly one of s' or s'' .) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y , since characters 1, 2, 5, 7, 8, 9 form 101101, a repetition of x , and the remaining characters 3, 4, 6 form 000, a repetition of y .

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: we want to make sure s "unravels" into simple repetitions of x and y . Give an efficient algorithm that takes strings s , x , and y and decides if s is an interleaving of x and y .

Solution. Let X_i and Y_j be the length- i prefix of x^∞ and the length- j prefix of y^∞ (so the repetitions of x are exactly the X_i), and fill a Boolean table $M[i, j] = "s[1..i+j] \text{ is an interleaving of } X_i \text{ and } Y_j"$, i.e. can be split into subsequences equal to X_i and Y_j . With $|s| = n$, $p = |x|$, $q = |y|$:

$$\begin{aligned} M[0, 0] &= \text{true}, \\ M[i, j] &= (i \geq 1 \wedge M[i-1, j] \wedge s[i+j] = x[((i-1) \bmod p) + 1]) \\ &\quad \vee (j \geq 1 \wedge M[i, j-1] \wedge s[i+j] = y[((j-1) \bmod q) + 1]), \end{aligned}$$

computed in increasing order of $i + j \leq n$. Then s is an interleaving of x and y iff $M[i, n-i]$ holds for some $0 \leq i \leq n$.

Correctness. Induction on $i + j$. In any split of $s[1..i+j]$ into X_i and Y_j , the last symbol $s[i+j]$ is either the last symbol of X_i , namely $x[((i-1) \bmod p) + 1]$, and the rest is a split of $s[1..i+j-1]$ into X_{i-1} and Y_j ; or symmetrically the last symbol of Y_j . Conversely each such split of $s[1..i+j-1]$ extends by putting $s[i+j]$ on the matching side. So $M[i, j]$ satisfies the recurrence. Finally s is an interleaving iff it splits into X_i and Y_{n-i} for some i , since the two parts have lengths summing to n .

Running time. There are $O(n^2)$ entries with $i + j \leq n$, each $O(1)$, so the total is $O(n^2)$.

Problem 6.20. Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: it will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to

decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project. To figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, \dots, n\}$ (rough estimates, of course) for each of your n courses; if you spend $h \leq H$ hours on the project for course i , you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are *nondecreasing*: if $h < h'$, then $f_i(h) \leq f_i(h')$.)

So the problem is: given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i , is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n , g , and H ; none of these quantities should appear as an exponent in your running time.

Solution. Maximize the total $\sum_i f_i(h_i)$ (equivalently the average) by a knapsack-style recurrence: let $\text{OPT}(i, h)$ be the largest total grade on courses $1, \dots, i$ using at most h hours in all. Then

$$\text{OPT}(0, h) = 0, \quad \text{OPT}(i, h) = \max_{0 \leq k \leq h} (f_i(k) + \text{OPT}(i-1, h-k)),$$

for $1 \leq i \leq n$, $0 \leq h \leq H$; the best average is $\text{OPT}(n, H)/n$. Recording the maximizing k in each entry and tracing back from (n, H) gives the hours $h_n, h_{n-1}, \dots, h_1$; adding the unused hours to any project does not lower its grade (the f_i are nondecreasing), so the allocation can be taken to use exactly H hours.

Correctness. Induction on i : an allocation for courses $1, \dots, i$ within h hours gives some $k \in \{0, \dots, h\}$ hours to course i and is an allocation for courses $1, \dots, i-1$ within $h-k$ hours on the rest, and every such pair combines into one; so the maximum splits as in the recurrence.

Running time. There are $(n+1)(H+1)$ entries, each a maximum over at most $H+1$ terms, so the total is $O(nH^2)$, polynomial in n and H (and independent of g).

Problem 6.21. Some time back, you helped a group of friends who were doing simulations for a computation-intensive investment company, and they've come back to you with a new problem. They're looking at n consecutive days of a given stock, at some point in the past. The days are numbered $i = 1, 2, \dots, n$; for each day i , they have a price $p(i)$ per share for the stock on that day. For certain (possibly large) values of k , they want to study what they call *k-shot strategies*. A *k-shot strategy* is a collection of m pairs of days $(b_1, s_1), \dots, (b_m, s_m)$, where $0 \leq m \leq k$ and

$$1 \leq b_1 < s_1 < b_2 < s_2 < \dots < b_m < s_m \leq n.$$

We view these as a set of up to k nonoverlapping intervals, during each of which the investors buy 1,000 shares of the stock (on day b_i) and then sell it (on day s_i). The *return* of a given *k-shot strategy* is simply the profit obtained from the m buy-sell transactions, namely,

$$1,000 \sum_{i=1}^m (p(s_i) - p(b_i)).$$

The investors want to assess the value of *k-shot strategies* by running simulations on their n -day trace of the stock price. Your goal is to design an efficient algorithm that determines, given the sequence of prices, the *k-shot strategy* with the maximum possible return. Since k may be relatively large in these simulations, your running time should be polynomial in both n and k ; it should not contain k in the exponent.

Solution. Let $\text{OPT}(j, i)$ be the largest value of $\sum (p(s_\ell) - p(b_\ell))$ over strategies with at most j pairs that use only days $1, \dots, i$; condition on whether day i is the last sell day. The answer is $1,000 \cdot \text{OPT}(k, n)$, with

$$\text{OPT}(0, i) = \text{OPT}(j, 0) = \text{OPT}(j, 1) = 0,$$

$$\text{OPT}(j, i) = \max \left(\text{OPT}(j, i-1), p(i) + \max_{1 \leq b < i} (\text{OPT}(j-1, b-1) - p(b)) \right).$$

```

for j = 1..k:
    M = -infinity                                     # max over b < i of OPT(j-1,b-1) - p(b)
    for i = 2..n:
        M = max(M, OPT[j-1][i-2] - p(i-1))
        OPT[j][i] = max(OPT[j][i-1], M + p(i))
    return 1000 * OPT[k][n]

```

Correctness. By induction on (j, i) . Take an optimal strategy $\mathcal{S}$ for (j, i) . (i) If $\mathcal{S}$ does not sell on day i , it uses only days $1, \dots, i-1$, so its value is $\text{OPT}(j, i-1)$. (ii) If $s_m = i$ with buy day $b = b_m < i$, the other $m-1 \leq j-1$ pairs lie in days $1, \dots, b-1$ (since $s_{m-1} < b_m$), so they are worth at most $\text{OPT}(j-1, b-1)$, and conversely any such strategy on days $1, \dots, b-1$ extends by (b, i) . Hence the recurrence takes the max over exactly the feasible strategies. The variable M equals $\max_{1 \leq b < i} (\text{OPT}(j-1, b-1) - p(b))$ because each i adds exactly the term $b = i-1$. The strategy itself is recovered by the usual traceback, recording for each (j, i) which branch and which b attained the max.

Running time. Each of the kn entries costs $O(1)$ thanks to the running maximum M , so the algorithm runs in $O(nk)$ time.

6.4 Exercises 6.22–6.28

Problem 6.22. To assess how “well-connected” two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the *number* of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$. Give an efficient algorithm that computes the number of shortest v - w paths in G . (The algorithm should not list all the paths; just the number suffices.)

Solution. Compute the distances $d(u)$ from every node to w with Bellman-Ford, keep only the *tight* edges (u, x) with $d(u) = c_{ux} + d(x)$, and count u - w paths in this tight subgraph, which is a DAG.

```

run Bellman-Ford with sink w to get d(u) for all u
if d(v) = infinity: return 0
H = edges (u,x) with d(x) finite and d(u) = c_ux + d(x)
N(w) = 1
for u in reverse topological order of H, u != w:
    N(u) = sum of N(x) over edges (u,x) in H
return N(v)

```

Correctness. There are no negative cycles, so by (6.24)–(6.25) Bellman-Ford returns the true distances $d(u)$ to w , and by (6.22) these are attained by simple paths. Every edge satisfies $d(u) \leq c_{ux} + d(x)$, so for any u - w path $P = u_0 u_1 \dots u_r$

$$\begin{aligned}
 c(P) &= \sum_{i=0}^{r-1} c_{u_i u_{i+1}} \geq \sum_{i=0}^{r-1} (d(u_i) - d(u_{i+1})) \\
 &= d(u) - d(w) = d(u),
 \end{aligned}$$

with equality iff every edge of P is tight. Hence the shortest u - w paths are exactly the u - w paths of H . The graph H is acyclic: a cycle C of tight edges would telescope to $c(C) = \sum (d(u_i) - d(u_{i+1})) = 0$, contradicting that every cycle has strictly positive cost. So a topological order of H exists (Section 3.6). Splitting the u - w paths of H ($u \neq w$) by their first edge $(u, x) \in H$ gives $N(u) = \sum_{(u,x) \in H} N(x)$, with $N(w) = 1$ (the trivial path; no tight path leaves w and returns, as that would be a tight cycle). By induction along the reverse topological order, $N(v)$ is the number of shortest v - w paths.

Running time. Bellman-Ford takes $O(mn)$ by (6.25); building H , topologically sorting it (Section 3.6) and the counting pass take $O(m+n)$ additions. The total is $O(mn)$ arithmetic operations, on counts of at most 2^{n-2} (a path in the DAG H is determined by its set of interior nodes), i.e. $O(n)$ -bit integers.

Problem 6.23. Suppose you are given a directed graph $G = (V, E)$ with costs on the edges c_e for $e \in E$ and a sink t (costs may be negative). Assume that you also have finite values $d(v)$ for $v \in V$. Someone claims that, for each node $v \in V$, the quantity $d(v)$ is the cost of the minimum-cost path from node v to the sink t .

(a) Give a linear-time algorithm (time $O(m)$ if the graph has m edges) that verifies whether this claim is correct.

(b) Assume that the distances are correct, and $d(v)$ is finite for all $v \in V$. Now you need to compute distances to a different sink t' . Give an $O(m \log n)$ algorithm for computing distances $d'(v)$ for all nodes $v \in V$ to the sink node t' . (*Hint:* It is useful to consider a new cost function defined as follows: for edge $e = (v, w)$, let $c'_e = c_e - d(v) + d(w)$. Is there a relation between costs of paths for the two different costs c and c' ?)

Solution. (a) Accept iff (i) $d(t) = 0$, (ii) $d(v) \leq c_{vw} + d(w)$ for every edge (v, w) , and (iii) every node reaches t using only *tight* edges, those with $d(v) = c_{vw} + d(w)$. (As in Section 6.8, “minimum-cost path” means the minimum over all v - t walks, which is finite and attained by a simple path when no negative cycle is reachable, by (6.22).)

```

if d(t) != 0: reject
for each edge (v,w): if d(v) > c_vw + d(w): reject
R = reverse of the tight edges
search (BFS) from t in R
accept iff every node was reached

```

Sufficiency. By (ii), for any v - t walk $P = v_0 v_1 \cdots v_r$,

$$c(P) \geq \sum_{i=0}^{r-1} (d(v_i) - d(v_{i+1})) = d(v) - d(t) = d(v),$$

and the same telescoping gives $c(C) \geq 0$ for every cycle, so the minimum is well defined; by (iii) some tight v - t path exists, and its cost is exactly $d(v)$. Hence $d(v)$ is the minimum cost.

Necessity. If the claim holds, (i) holds because the minimum over t - t walks is the empty walk (no negative cycles, as all d are finite); (ii) holds since c_{vw} plus a cheapest w - t path is a v - t path; and for (iii), every node has a v - t path (all $d(v)$ are finite), and along a shortest one $v_0 \cdots v_r$ each suffix is shortest, so $d(v_i) = c_{v_i v_{i+1}} + d(v_{i+1})$ and every edge is tight.

The test reads each edge a constant number of times and does one BFS (Section 3.2), so it runs in $O(m + n)$ time; since every node must reach t , an accepted graph has $m \geq n - 1$, and we may reject at once if $m < n - 1$, giving $O(m)$.

(b) Run Dijkstra's Algorithm with the reduced costs c' , which are nonnegative, toward t' (on the reversed graph), and correct the answers by $d'(v) = d_{c'}(v) + d(v) - d(t')$.

By (ii) of part (a), $c'_{vw} = c_{vw} - d(v) + d(w) \geq 0$ for every edge. For any u - t' path $P = u_0 \cdots u_r$ the potentials telescope:

$$\begin{aligned} c'(P) &= \sum_{i=0}^{r-1} (c_{u_i u_{i+1}} - d(u_i) + d(u_{i+1})) \\ &= c(P) - d(u) + d(t'). \end{aligned}$$

The correction term depends only on the endpoints, so P minimizes c' among u - t' paths iff it minimizes c , and $d'(u) = d_{c'}(u) + d(u) - d(t')$ (both sides ∞ when u cannot reach t'). Dijkstra's Algorithm from t' on the reversed graph computes $d_{c'}$ correctly since all costs are nonnegative (4.14), and with a heap-based priority queue it runs in $O(m \log n)$ by (4.15); computing c' and the corrections is $O(m + n)$. Total: $O(m \log n)$.

Problem 6.24. *Gerrymandering* is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. Recent court challenges to the practice have argued that through this calculated redistricting, large numbers of voters are being effectively (and intentionally) disenfranchised.

Computers, it turns out, have been implicated as the source of some of the “villainy” in the news coverage on this topic: Thanks to powerful software, gerrymandering has changed from an activity carried out by a bunch of people with maps, pencil, and paper into the industrial-strength process that it is today. Why is gerrymandering a computational problem? There are database issues involved in tracking voter demographics down to the level of individual streets and houses; and there are algorithmic issues involved in grouping voters into districts. Let’s think a bit about what these latter issues look like.

Suppose we have a set of n precincts $P_1, P_2, \dots, P_n$, each containing m registered voters. We’re supposed to divide these precincts into two districts, each consisting of $n/2$ of the precincts. Now, for each precinct, we have information on how many voters are registered to each of two political parties. (Suppose, for simplicity, that every voter is registered to one of these two.) We’ll say that the set of precincts is *susceptible* to gerrymandering if it is possible to perform the division into two districts in such a way that the same party holds a majority in both districts.

Give an algorithm to determine whether a given set of precincts is susceptible to gerrymandering; the running time of your algorithm should be polynomial in n and m .

Example. Suppose we have $n = 4$ precincts, and the following information on registered voters.

Precinct	1	2	3	4
Number registered for party A	55	43	60	47
Number registered for party B	45	57	40	53

This set of precincts is susceptible since, if we grouped precincts 1 and 4 into one district, and precincts 2 and 3 into the other, then party A would have a majority in both districts. (Presumably, the “we” who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: Although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

Solution. Compute, by a subset-sum style dynamic program, every value s that the party-A count of some set of exactly $n/2$ precincts can take; the precincts are susceptible iff some such s lies strictly between $nm/4$ and $T - nm/4$ (in either order), where $T = \sum_i a_i$ and a_i is the party-A count of P_i .

The criterion. Each district has $nm/2$ voters, so a party has a majority in it iff it has more than $nm/4$ of them. Let S be a district with $A(S) = \sum_{i \in S} a_i$; its complement has $T - A(S)$ A-voters, and party B holds a majority in S iff $A(S) < nm/4$. Hence

$$\begin{aligned} \text{A wins both} &\iff nm/4 < A(S) < T - nm/4, \\ \text{B wins both} &\iff T - nm/4 < A(S) < nm/4. \end{aligned}$$

The DP. Let $R(j, k, s)$ be true iff some k of the precincts $P_1, \dots, P_j$ have A-total exactly s , for $0 \leq j \leq n$, $0 \leq k \leq n/2$, $0 \leq s \leq nm/2$. Then $R(0, 0, 0)$ is true, $R(0, k, s)$ is false otherwise, and

$$\begin{aligned} R(j, k, s) = R(j-1, k, s) \vee & (k \geq 1 \wedge s \geq a_j \\ & \wedge R(j-1, k-1, s-a_j)), \end{aligned}$$

by conditioning on whether P_j is chosen (an induction on j shows the table is exactly the set of realizable (k, s)). Answer “susceptible” iff $R(n, n/2, s)$ holds for some s satisfying one of the two displayed conditions. (A sum over $n/2$ precincts is at most $nm/2$, so the range of s suffices.)

Running time. The table has $(n+1)(n/2+1)(nm/2+1) = O(n^3m)$ entries, each filled in $O(1)$, and the final scan is $O(nm)$: total $O(n^3m)$.

Problem 6.25. Consider the problem faced by a stockbroker trying to sell a large number of shares of stock in a company whose stock price has been steadily falling in value. It is always hard to predict the right moment to sell stock, but owning a lot of shares in a single company adds an extra complication: the mere act of selling many shares in a single day will have an adverse effect on the price.

Since future market prices, and the effect of large sales on these prices, are very hard to predict, brokerage firms use models of the market to help them make such decisions. In this problem, we will consider the following simple model. Suppose we need to sell x shares of stock in a company, and suppose that we have an accurate model of the market: it predicts that the stock price will take the values $p_1, p_2, \dots, p_n$ over the next n days. Moreover, there is a function $f(\cdot)$ that predicts the effect of large sales: if we sell y shares on a single day, it will permanently decrease the price by $f(y)$ from that day onward. So, if we sell y_1 shares on day 1, we obtain a price per share of $p_1 - f(y_1)$, for a total income of $y_1 \cdot (p_1 - f(y_1))$. Having sold y_1 shares on day 1, we can then sell y_2 shares on day 2 for a price per share of $p_2 - f(y_1) - f(y_2)$; this yields an additional income of $y_2 \cdot (p_2 - f(y_1) - f(y_2))$. This process continues over all n days. (Note, as in our calculation for day 2, that the decreases from earlier days are absorbed into the prices for all later days.)

Design an efficient algorithm that takes the prices $p_1, \dots, p_n$ and the function $f(\cdot)$ (written as a list of values $f(1), f(2), \dots, f(x)$) and determines the best way to sell x shares by day n . In other words, find natural numbers $y_1, y_2, \dots, y_n$ so that $x = y_1 + \dots + y_n$, and selling y_i shares on day i for $i = 1, 2, \dots, n$ maximizes the total income achievable. You should assume that the share value p_i is monotone decreasing, and $f(\cdot)$ is monotone increasing; that is, selling a larger number of shares causes a larger drop in the price. Your algorithm's running time can have a polynomial dependence on n (the number of days), x (the number of shares), and p_1 (the peak price of the stock).

Example. Consider the case when $n = 3$; the prices for the three days are 90, 80, 40; and $f(y) = 1$ for $y \leq 40,000$ and $f(y) = 20$ for $y > 40,000$. Assume you start with $x = 100,000$ shares. Selling all of them on day 1 would yield a price of 70 per share, for a total income of 7,000,000. On the other hand, selling 40,000 shares on day 1 yields a price of 89 per share, and selling the remaining 60,000 shares on day 2 results in a price of 59 per share, for a total income of 7,100,000.

Solution. Charge each day's price drop to all the shares still unsold on that day; then the income splits into per-day terms that depend only on y_j and the number $r_j = y_j + \dots + y_n$ of shares left, and a DP over (day, shares left) solves it. Set $f(0) = 0$ (selling nothing moves no price).

Rewriting the income. Swapping the order of summation,

$$\begin{aligned} \sum_{i=1}^n y_i \left(p_i - \sum_{j \leq i} f(y_j) \right) &= \sum_{i=1}^n y_i p_i - \sum_{j=1}^n f(y_j) \sum_{i \geq j} y_i \\ &= \sum_{j=1}^n (y_j p_j - f(y_j) r_j). \end{aligned}$$

The DP. Let $\text{OPT}(j, r)$ be the maximum of $\sum_{i \geq j} (y_i p_i - f(y_i) r_i)$ over natural numbers $y_j, \dots, y_n$ summing to r (so $r_j = r$). Choosing $y = y_j$ leaves $r - y$ shares for days $j + 1, \dots, n$, and the remaining terms depend only on $y_{j+1}, \dots, y_n$, so

$$\begin{aligned} \text{OPT}(n+1, 0) &= 0, & \text{OPT}(n+1, r) &= -\infty \quad (r > 0), \\ \text{OPT}(j, r) &= \max_{0 \leq y \leq r} (y p_j - f(y) r + \text{OPT}(j+1, r-y)), \end{aligned}$$

and the answer is $\text{OPT}(1, x)$, with the y_j recovered by recording the maximizing y for each (j, r) and tracing forward from $(1, x)$. Correctness is induction on j downward from $n+1$, the identity above showing that $\text{OPT}(1, x)$ is exactly the maximum total income.

Running time. There are $n(x+1)$ entries, each a max over at most $x+1$ choices: $O(nx^2)$ time.

Problem 6.26. Consider the following inventory problem. You are running a company that sells some large product (let's assume you sell trucks), and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i . We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are *stored* until the beginning of the next month. You can store at most S trucks, and it costs C to store a

single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

- There are two parts to the cost: (1) storage: it costs C for every truck on hand that is not needed that month; (2) ordering fees: it costs K for every order placed.
- In each month you need enough trucks to satisfy the demand d_i , but the number left over after satisfying the demand for the month should not exceed the inventory limit S .

Give an algorithm that solves this problem in time that is polynomial in n and S .

Solution. Run a dynamic program over (month, trucks carried into that month): there are only $S + 1$ possible stock levels, and fixing the stock left after month i fixes how much month i 's order must be. Let $\text{OPT}(i, s)$ be the minimum cost of months $i, \dots, n$ when $s \in \{0, \dots, S\}$ trucks are carried into month i , and let $\text{OPT}(n+1, s) = 0$. In month i we either place no order, which is allowed only if $s \geq d_i$ and leaves $s - d_i$ trucks, or we place one order and choose any leftover $s' \in \{\max(0, s - d_i), \dots, S\}$, which means ordering $s' + d_i - s \geq 0$ trucks. In both cases the leftover s' is stored at cost Cs' . So

$$\text{OPT}(i, s) = \min \left\{ C(s - d_i) + \text{OPT}(i+1, s - d_i) \quad [\text{only if } s \geq d_i], \right. \\ \left. \min_{\max(0, s-d_i) \leq s' \leq S} (K + Cs' + \text{OPT}(i+1, s')) \right\},$$

and the answer is $\text{OPT}(1, 0)$.

```

for s = 0..S: M[n+1][s] = 0
for i = n down to 1:
  for s = 0..S:
    M[i][s] = min over s' in [max(0, s-d_i), S] of K + C*s' + M[i+1][s']
    if s >= d_i: M[i][s] = min(M[i][s], C*(s-d_i) + M[i+1][s-d_i])
return M[1][0]    (trace back the argmins to read off the orders)

```

Correctness. We use induction on i going downward. The case $i = n+1$ holds because nothing remains to be paid. A feasible plan for months $i..n$ that starts with s trucks makes one decision in month i : order or not, and if it orders, how many trucks. That decision fixes the leftover s' , which has to satisfy $0 \leq s' \leq S$, and $s' = s - d_i$ when there is no order. The plan's cost for month i is Cs' , plus K if it ordered. Its remaining decisions form a feasible plan for months $i+1..n$ that starts with s' trucks, so by induction they cost at least $\text{OPT}(i+1, s')$. Every such pair (month- i decision, optimal rest) is feasible, so the minimum in the recurrence is attained. A single order per month is enough, since splitting it only adds fees.

Running time. The table has $n(S+1)$ entries, and each one takes a minimum over at most $S+2$ terms, so the algorithm runs in $O(nS^2)$ time.

Problem 6.27. The owners of an independently operated gas station are faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to L gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order, they need to pay a fixed price P for delivery in addition to the cost of the gas ordered. However, it costs c to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost.

They are planning to close for a week in the winter, and they want their tank to be empty by the time they close. Luckily, based on years of experience, they have accurate projections for how much gas they will need each day until this point in time. Assume that there are n days left until they close, and they need g_i gallons of gas for each of the days $i = 1, \dots, n$. Assume that the tank is empty at the end of day 0. Give an algorithm to decide on which days they should place orders, and how much to order so as to minimize their total cost.

Solution. Some optimal plan orders only on days when the tank is empty, and each such order covers exactly the next run of days, so a segmentation dynamic program like the one for segmented least

squares (Section 6.3) finds it.

Model. An order placed on day i arrives at the start of day i . The tank level after the delivery may not exceed L . Day i 's g_i gallons are used during day i . Let I_t be the level at the end of day t , with $I_0 = 0$ and $I_n = 0$. The storage cost is $c \sum_t I_t$, so a gallon delivered on day i and used on day t costs $c(t - i)$. Every plan buys the same $\sum_t g_t$ gallons, so the price of the gas itself is a constant. Write

$$G(i, j) = \sum_{t=i}^j g_t, \quad W(i, j) = \sum_{t=i}^j (t - i) g_t.$$

Zero-inventory property: every optimal plan orders only when $I_{j-1} = 0$. Suppose not. Let j be the first order day with $r = I_{j-1} > 0$, and let $i < j$ be the last order day before j . By the minimality of j we have $I_{i-1} = 0$, so the order size is $g_i \geq I_i \geq r$. Between days i and $j - 1$ nothing is delivered, so $I_t \geq I_{j-1} = r$ for $i \leq t \leq j - 1$. Now move r gallons from order i to order j . Each I_t with $i \leq t < j$ drops by r and stays ≥ 0 . The level after delivery on day j does not change, and on day i it decreases. No new order is created. The storage cost falls by $cr(j - i) > 0$ (as $c > 0$), which contradicts optimality. So an optimal plan splits days $1..n$ into consecutive blocks. Each block $[i, j]$ is served by one order of exactly $G(i, j)$ gallons on day i , which is feasible iff $G(i, j) \leq L$ and costs $P + cW(i, j)$. A leading block with $G(1, j) = 0$ needs no order. Let $\text{OPT}(j)$ be the minimum cost of days $1..j$ with $I_j = 0$. Then $\text{OPT}(0) = 0$, and

$$\text{OPT}(j) = \begin{cases} 0, & G(1, j) = 0, \\ \min_{1 \leq i \leq j, G(i, j) \leq L} (\text{OPT}(i - 1) + P + cW(i, j)), & \text{otherwise.} \end{cases}$$

To see why this holds, look at an optimal plan for days $1..j$. Let i be its last order day. By the zero-inventory property we have $I_{i-1} = 0$, so that order is exactly $G(i, j)$, and what the plan does on days $1..i - 1$ costs at least $\text{OPT}(i - 1)$. In the other direction, every term of the recurrence is achieved by a feasible plan. The answer is $\text{OPT}(n)$, and it is infeasible exactly when some $g_t > L$. To output the plan, trace back the minimizing i 's; each one is an order day, and the amount ordered is $G(i, j)$.

Running time. With the prefix sums $A_j = \sum_{t \leq j} g_t$ and $B_j = \sum_{t \leq j} t g_t$, which take $O(n)$ to compute, we get

$$G(i, j) = A_j - A_{i-1}, \quad W(i, j) = (B_j - B_{i-1}) - i(A_j - A_{i-1}),$$

each in $O(1)$. That leaves $O(n)$ choices of i for each of the n values of j , so the total time is $O(n^2)$.

Problem 6.28. Recall the scheduling problem from Section 4.2 in which we sought to minimize the maximum lateness. There are n jobs, each with a deadline d_i and a required processing time t_i , and all jobs are available to be scheduled starting at time s . For a job i to be done, it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and different jobs should be assigned nonoverlapping intervals. As usual, such an assignment of times will be called a *schedule*.

In this problem, we consider the same setup, but want to optimize a different objective. In particular, we consider the case in which each job must either be done by its deadline or not at all. We'll say that a subset J of the jobs is *schedulable* if there is a schedule for the jobs in J so that each of them finishes by its deadline. Your problem is to select a schedulable subset of maximum possible size and give a schedule for this subset that allows each job to finish by its deadline.

(a) Prove that there is an optimal solution J (i.e., a schedulable set of maximum size) in which the jobs in J are scheduled in increasing order of their deadlines.

(b) Assume that all deadlines d_i and required times t_i are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of jobs n , and the maximum deadline $D = \max_i d_i$.

Solution. (a) Take any schedulable set J of maximum size and schedule it with the greedy Earliest Deadline First rule of Section 4.2. Because J is schedulable, some schedule of J has maximum lateness 0. By (4.10), applied to the instance made of the jobs in J , the greedy schedule A , which runs J back

to back from s in increasing order of deadline, has the optimal maximum lateness. Hence

$$\max_{j \in J} (f_j - d_j)^+ \text{ in } A \leq 0,$$

so every job of J meets its deadline in A , and A is an optimal solution that runs its jobs in increasing order of deadline.

(b) Sort the jobs by deadline and run a knapsack-style dynamic program over (jobs considered, total processing time). We shift time so that $s = 0$, taking s to be an integer like the d_i and t_i . Number the jobs so that $d_1 \leq \dots \leq d_n$, and call J *EDF-feasible* if running J back to back from time 0 in index order meets every deadline, that is, $\sum_{l \in J, l \leq j} t_l \leq d_j$ for all $j \in J$. By (a), a set is schedulable iff it is EDF-feasible. (Index order is an Earliest Deadline First order, and (4.10) holds with ties broken arbitrarily.) For $0 \leq u \leq D$, let $\text{OPT}(i, u)$ be the maximum $|J|$ over EDF-feasible $J \subseteq \{1, \dots, i\}$ with $\sum_{l \in J} t_l \leq u$. Then $\text{OPT}(0, u) = 0$ and

$$\begin{aligned} \text{OPT}(i, u) = \max \{ & \text{OPT}(i-1, u), \\ & 1 + \text{OPT}(i-1, \min(u, d_i) - t_i) \text{ [only if } \min(u, d_i) \geq t_i] \}. \end{aligned}$$

The answer is $\text{OPT}(n, D)$. A traceback recovers J , which we then run back to back in deadline order. *Correctness.* Let J be EDF-feasible with $J \subseteq \{1..i\}$ and total time at most u .

- (i) If $i \notin J$, then J is counted by $\text{OPT}(i-1, u)$.
- (ii) If $i \in J$, then i runs last, finishing at $\sum_{l \in J} t_l \leq \min(u, d_i)$. The jobs of $J' = J \setminus \{i\}$ finish at the same times as before, so J' is EDF-feasible with total time at most $\min(u, d_i) - t_i$, and $|J| \leq 1 + \text{OPT}(i-1, \min(u, d_i) - t_i)$.

In the other direction, adding job i to an EDF-feasible $J' \subseteq \{1..i-1\}$ of total time at most $\min(u, d_i) - t_i$ gives an EDF-feasible set whose new last job finishes by $\min(u, d_i)$. So both branches are achievable, and induction on i proves the recurrence. Every EDF-feasible set has total time at most D , so $\text{OPT}(n, D)$ is the optimum.

Running time. Sorting takes $O(n \log n)$. The table has $(n+1)(D+1)$ entries, each filled in $O(1)$, so the total is $O(n \log n + nD)$.

6.5 Exercise 6.29

Problem 6.29. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j) ; and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V' .

We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a spanning subtree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T .

Show that there is function $f(\cdot)$ and a *polynomial function* $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(f(k) \cdot p(n))$.

Solution. We can take $f(k) = 3^k$ and $p(n) = n^2$, using the Dreyfus-Wagner dynamic program over pairs (set of terminals, one extra node).

For $D \subseteq X$ and $v \in V$, let $S(D, v)$ be the minimum weight of a Steiner tree on $D \cup \{v\}$, and set $w_{vv} = 0$. Then $S(\emptyset, v) = 0$, $S(\{x\}, v) = w_{xv}$, and $S(D, v) = S(D \setminus \{v\}, v)$ when $v \in D$. For $v \notin D$ with $|D| \geq 2$,

$$S(D, v) = \min_{u \in V} \left(w_{vu} + \min_{\emptyset \neq D_1 \subsetneq D} [S(D_1, u) + S(D \setminus D_1, u)] \right).$$

The answer is $S(X \setminus \{x\}, x)$ for any $x \in X$ (and 0 if $k = 0$). We fill the table in order of increasing $|D|$ and recover the tree by tracing back.

The recurrence is an upper bound. Take trees achieving $S(D_1, u)$ and $S(D \setminus D_1, u)$ and add the edge (v, u) . The union is a connected subgraph that contains $D \cup \{v\}$ and has weight at most the bracketed

sum. One of its spanning trees is a Steiner tree on $D \cup \{v\}$ of no greater weight.

The recurrence is a lower bound. Let T be an optimal Steiner tree on $D \cup \{v\}$, where $v \notin D$ and $|D| \geq 2$. Every leaf of T lies in $D \cup \{v\}$, since a leaf outside that set could be deleted and the weights are positive. For any node u , call a split of a tree containing $D' \cup \{u\}$ at u the following: break it at u into two edge-disjoint subtrees through u , each containing a terminal. This needs $\deg(u) \geq 2$, and each branch ends in a leaf of D' . Such a tree then weighs at least $S(D_1, u) + S(D' \setminus D_1, u)$ for some nonempty proper $D_1 \subset D'$.

- (i) $\deg_T(v) \geq 2$. Split T at $u = v$, and use $w_{vv} = 0$.
- (ii) $\deg_T(v) = 1$. Walk from v to the first node $u \neq v$ that lies in D or has $\deg_T(u) \geq 3$. The nodes passed on the way have degree 2 and are not terminals. Removing the path P from v to u leaves a tree T' that contains $D \cup \{u\}$, and $w(P) \geq w_{vu}$ by the triangle inequality. One of three things holds:
 - (ii-a) $\deg_{T'}(u) \geq 2$, and we split T' at u ;
 - (ii-b) $\deg_{T'}(u) = 1$, so $u \in D$, and we take $D_1 = \{u\}$, which gives $S(\{u\}, u) = 0$ and $w(T') \geq S(D \setminus \{u\}, u)$;
 - (ii-c) $\deg_{T'}(u) = 0$, which is impossible because $|D| \geq 2$.

In every case $w(T)$ is at least one term of the recurrence.

Running time. For each D and u , precompute $B(D, u) = \min_{D_1} [S(D_1, u) + S(D \setminus D_1, u)]$. Over all D this takes $n \sum_D 2^{|D|} = n 3^k$ steps. Then each $S(D, v) = \min_u (w_{vu} + B(D, u))$ costs $O(n)$, which adds up to $n^2 2^k$. The total is

$$O(3^k n + 2^k n^2) = O(3^k \cdot n^2).$$

7 Network Flow

Contents

7.1 Exercises 7.1–7.7	87
7.2 Exercises 7.8–7.14	90
7.3 Exercises 7.15–7.21	95
7.4 Exercises 7.22–7.28	102
7.5 Exercises 7.29–7.35	108
7.6 Exercises 7.36–7.42	113
7.7 Exercises 7.43–7.49	120
7.8 Exercises 7.50–7.51	127

7.1 Exercises 7.1–7.7

Problem 7.1. (a) List all the minimum s - t cuts in the flow network of Figure 7.24: nodes s, u, v, t and edges $s \rightarrow u$, $s \rightarrow v$, $u \rightarrow v$, $u \rightarrow t$, $v \rightarrow t$, each of capacity 1.
(b) What is the minimum capacity of an s - t cut in the flow network of Figure 7.25: nodes s, u, v, t and edges $s \rightarrow u$ (capacity 2), $s \rightarrow v$ (4), $u \rightarrow v$ (6), $u \rightarrow t$ (4), $v \rightarrow t$ (2)?

Solution. (a) The minimum cuts are $(\{s\}, \{u, v, t\})$, $(\{s, v\}, \{u, t\})$ and $(\{s, u, v\}, \{t\})$, each of capacity 2. Writing a cut by its source side A , the four cuts have capacities

$$\begin{aligned} c(\{s\}) &= c_{su} + c_{sv} = 2, & c(\{s, u\}) &= c_{sv} + c_{uv} + c_{ut} = 3, \\ c(\{s, v\}) &= c_{su} + c_{vt} = 2, & c(\{s, u, v\}) &= c_{ut} + c_{vt} = 2, \end{aligned}$$

(for $A = \{s, v\}$ the edge $u \rightarrow v$ enters A and does not count), so the minimum is 2, attained exactly by the three cuts listed.

(b) The minimum capacity is 4. The four cuts have capacities

$$\begin{aligned} c(\{s\}) &= 2 + 4 = 6, & c(\{s, u\}) &= 4 + 6 + 4 = 14, \\ c(\{s, v\}) &= 2 + 2 = 4, & c(\{s, u, v\}) &= 4 + 2 = 6, \end{aligned}$$

and the minimum is $c(\{s, v\}) = 4$ (again $u \rightarrow v$ enters A).

Problem 7.2. Figure 7.26 shows a flow network on which an s - t flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbers, specifically the four edges of capacity 3, have no flow being sent on them.) Call the unlabelled top, middle-left and middle-right nodes a, b, c ; the bottom node is d . The edges, written capacity / flow, are

edge	$s \rightarrow a$	$s \rightarrow b$	$s \rightarrow d$	$b \rightarrow a$	$b \rightarrow c$	$b \rightarrow d$	$a \rightarrow c$	$d \rightarrow c$	$a \rightarrow t$	$c \rightarrow t$	$d \rightarrow t$
c/f	10/5	8/8	5/5	3/0	10/8	3/0	3/0	3/0	5/5	8/8	10/5

(a) What is the value of this flow? Is this a maximum (s, t) flow in this graph?

(b) Find a minimum s - t cut in the flow network pictured in Figure 7.26, and also say what its capacity is.

Solution. (a) The value is $\nu(f) = 5 + 8 + 5 = 18$, and it is not maximum. In the residual graph G_f the path

$$s \xrightarrow{5} a \xrightarrow{3} c \xrightarrow{8} b \xrightarrow{3} d \xrightarrow{5} t$$

exists ($c \rightarrow b$ is the backward edge of $b \rightarrow c$; the labels are residual capacities), so augmenting by its bottleneck 3 gives, by (7.3), a flow f' of value 21: $f'(s, a) = 8$, $f'(a, c) = 3$, $f'(b, c) = 5$, $f'(b, d) = 3$, $f'(d, t) = 8$, all other edges unchanged.

(b) A minimum cut is $A = \{s, a\}$, $B = \{b, c, d, t\}$, of capacity

$$c(A, B) = c_{sb} + c_{sd} + c_{ac} + c_{at} = 8 + 5 + 3 + 5 = 21$$

($b \rightarrow a$ enters A and does not count). Since $\nu(f') = 21 = c(A, B)$, (7.8) shows that f' is a maximum flow and (A, B) a minimum cut; indeed A is exactly the set of nodes reachable from s in $G_{f'}$, as in (7.9).

Problem 7.3. Figure 7.27 shows a flow network on which an s - t flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbers have no flow being sent on them.) The nodes are s, a, b, c, d, t and the edges, written capacity / flow, are

edge	$s \rightarrow a$	$s \rightarrow b$	$s \rightarrow d$	$a \rightarrow b$	$a \rightarrow c$	$a \rightarrow t$	$b \rightarrow c$	$d \rightarrow b$	$d \rightarrow c$	$c \rightarrow t$	$d \rightarrow t$
c/f	10/6	3/3	1/1	1/1	2/0	5/5	6/5	3/1	3/0	5/5	10/0

(a) What is the value of this flow? Is this a maximum (s, t) flow in this graph?

(b) Find a minimum s - t cut in the flow network pictured in Figure 7.27, and also say what its capacity is.

Solution. (a) The value is $\nu(f) = 6 + 3 + 1 = 10$, and it is not maximum. In the residual graph G_f the path

$$s \xrightarrow{4} a \xrightarrow{2} c \xrightarrow{5} b \xrightarrow{1} d \xrightarrow{10} t$$

exists ($c \rightarrow b$ and $b \rightarrow d$ are the backward edges of $b \rightarrow c$ and $d \rightarrow b$; the labels are residual capacities), so augmenting by its bottleneck 1 gives, by (7.3), a flow f' of value 11: $f'(s, a) = 7$, $f'(a, c) = 1$, $f'(b, c) = 4$, $f'(d, b) = 0$, $f'(d, t) = 1$, all other edges unchanged.

(b) A minimum cut is $A = \{s, a, b, c\}$, $B = \{d, t\}$, of capacity

$$c(A, B) = c_{sd} + c_{at} + c_{ct} = 1 + 5 + 5 = 11$$

($d \rightarrow b$ and $d \rightarrow c$ enter A and do not count). Since $\nu(f') = 11 = c(A, B)$, (7.8) shows that f' is a maximum flow and (A, B) a minimum cut; A is exactly the set of nodes reachable from s in $G_{f'}$, as in (7.9).

Problem 7.4. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s , a sink t , and a positive integer capacity c_e on every edge e . If f is a maximum s - t flow in G , then f saturates every edge out of s with flow (i.e., for all edges e out of s , we have $f(e) = c_e$).

Solution. False. Take nodes s, v, t with edges $s \rightarrow v$ of capacity $c_{sv} = 2$ and $v \rightarrow t$ of capacity $c_{vt} = 1$. Conservation at v and the capacity of $v \rightarrow t$ force every flow to satisfy

$$f(s, v) = f(v, t) \leq 1 < 2 = c_{sv},$$

so the maximum flow (value 1, matching the cut $(\{s, v\}, \{t\})$ of capacity 1) does not saturate the only edge out of s .

Problem 7.5. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s , a sink t , and a positive integer capacity c_e on every edge e ; and let (A, B) be a minimum s - t cut with respect to these capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity; then (A, B) is still a minimum s - t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

Solution. False. Adding 1 to every capacity raises the capacity of a cut by the number of edges crossing it, so a minimum cut crossed by many edges can lose to one crossed by few. Take nodes s, a, b, t and edges

$$c_{sa} = 2, \quad c_{ab} = 1, \quad c_{at} = 1, \quad c_{bt} = 1.$$

The cuts, by source side, have capacities $c(\{s\}) = 2$, $c(\{s, a\}) = c_{ab} + c_{at} = 2$, $c(\{s, a, b\}) = c_{at} + c_{bt} = 2$, $c(\{s, b\}) = c_{sa} + c_{bt} = 3$, so $(A, B) = (\{s, a\}, \{b, t\})$ is a minimum cut. With capacities $1 + c_e$,

$$c'(\{s, a\}, \{b, t\}) = 2 + 2 = 4 > 3 = c'(\{s\}, \{a, b, t\}),$$

so (A, B) is no longer a minimum cut.

Problem 7.6. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the setup of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can *see* the light fixture being controlled.

Sometimes this is possible and sometimes it isn't. Figure 7.28 shows two simple staircase-shaped floor plans, each with three light fixtures a, b, c and three switches 1, 2, 3. In plan (a) one can wire switch 1 to a , switch 2 to b and switch 3 to c so that every switch has a line of sight to its fixture. In plan (b) switches 2 and 3 sit side by side in the bottom room, from which only fixture c is visible, so no such wiring is possible.

Let's call a floor plan, together with n light fixture locations and n switch locations, *ergonomic* if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i -th wall has endpoints $(x_i, y_i), (x'_i, y'_i)$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is *visible* from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan is ergonomic. The running time should be polynomial in m and n . You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

Solution. Build the bipartite "visibility graph" between switches and fixtures and test it for a perfect matching.

Let $X = \{p_1, \dots, p_n\}$ be the switches and $Y = \{q_1, \dots, q_n\}$ the fixtures, and put the edge (p_i, q_j) in G exactly when the segment $p_i q_j$ crosses none of the m walls; this takes m calls of the crossing subroutine per pair, $O(n^2 m)$ in total. Then compute a maximum matching M of G with the Ford-Fulkerson Algorithm as in (7.38), and answer "ergonomic" iff $|M| = n$.

A wiring in which each switch controls one fixture and each fixture is controlled by one switch is a bijection $X \rightarrow Y$; it is valid iff every pair (p_i, q_j) it uses has q_j visible from p_i , i.e. iff it is a set of n edges of G no two sharing an endpoint. So

$$\text{ergonomic} \iff G \text{ has a perfect matching} \iff |M| = n,$$

and by (7.37)-(7.38) Ford-Fulkerson returns a maximum matching, so the test is correct.

Running time: G has at most n^2 edges and $2n$ nodes, so (7.38) gives $O(n^2 \cdot n) = O(n^3)$ for the matching, and the total is $O(n^2 m + n^3)$.

Problem 7.7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a *range parameter* r — a client can only be connected to a base station that is within distance r . There is also a *load parameter* L — no more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

Solution. Compute a maximum flow in the network: source $\rightarrow$ each client (capacity 1), client $\rightarrow$ each base station within distance r (capacity 1), each base station $\rightarrow$ sink (capacity L); the answer is yes

iff the maximum flow value is n .

Concretely, G has nodes s, t , clients $u_1, \dots, u_n$ and stations $b_1, \dots, b_k$, edges (s, u_i) with capacity 1, (u_i, b_j) with capacity 1 whenever $\|u_i - b_j\| \leq r$, and (b_j, t) with capacity L . Run Ford-Fulkerson and answer yes iff $\nu(f) = n$.

($\Rightarrow$) Given a valid connection, send one unit along $s \rightarrow u_i \rightarrow b_j \rightarrow t$ for each client u_i and its station b_j . Each (s, u_i) and (u_i, b_j) carries at most 1, each (b_j, t) carries the number of clients at b_j , which is $\leq L$, and conservation holds at every client and station; the value is n .

($\Leftarrow$) If the maximum flow has value n , then by (7.14) there is an integer maximum flow f , and Ford-Fulkerson returns one. Since $c(\{s\}, V - \{s\}) = n$, every (s, u_i) carries exactly 1; by conservation and integrality each u_i sends its unit along exactly one edge (u_i, b_j) , and we connect u_i to that b_j . This b_j is within range by construction of the edges, and the number of clients connected to b_j is

$$f^{\text{in}}(b_j) = f^{\text{out}}(b_j) = f(b_j, t) \leq L.$$

Running time: the network has $n + k + 2$ nodes and at most $nk + n + k$ edges, built in $O(nk)$ time; the flow value is at most $C = n$, so by (7.5) Ford-Fulkerson runs in $O(n(nk + n + k)) = O(n^2k)$ time (using $k \geq 1$).

7.2 Exercises 7.8–7.14

Problem 7.8. Statistically, the arrival of spring typically results in increased accidents and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether its blood supply is sufficient.

The basic rule for blood donation is the following. A person's own blood supply has certain *antigens* present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four *types*: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.

(a) Let s_O, s_A, s_B , and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type d_O, d_A, d_B , and d_{AB} for the coming week. Give a polynomial-time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in U.S. patients is roughly 45 percent type O, 42 percent type A, 10 percent type B, and 3 percent type AB. The hospital wants to know if the blood supply it has on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

blood type	supply	demand
O	50	45
A	36	42
B	11	8
AB	8	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum-capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words *flow*, *cut*, or *graph* in the sense we use them in this book.)

Solution. (a) The supply suffices iff the following network has a flow of value $D = d_O + d_A + d_B + d_{AB}$: a source s , a node u_X for each blood type X on hand, a node w_Y for the patients of each type Y , and a sink t , with edges

$$\begin{aligned} (s, u_X) &\text{ of capacity } s_X, & (w_Y, t) &\text{ of capacity } d_Y, \\ (u_X, w_Y) &\text{ of capacity } s_X \text{ whenever} \\ &\text{type-}Y \text{ patients can receive type-}X \text{ blood,} \end{aligned}$$

the compatible pairs being $O \rightarrow O, A, B, AB$; $A \rightarrow A, AB$; $B \rightarrow B, AB$; $AB \rightarrow AB$.

($\Rightarrow$) If a_{XY} units of type X go to type- Y patients in an allocation meeting all demand, the flow $f(s, u_X) = \sum_Y a_{XY} \leq s_X$, $f(u_X, w_Y) = a_{XY} \leq s_X$, $f(w_Y, t) = \sum_X a_{XY} = d_Y$ is feasible of value D . ($\Leftarrow$) If the maximum flow value is D , by (7.14) there is an integer flow f of value D ; since $c(V - \{t\}, \{t\}) = D$, every (w_Y, t) is saturated, so giving $f(u_X, w_Y)$ whole units of type X to type- Y patients uses at most $f^{\text{out}}(u_X) = f(s, u_X) \leq s_X$ units of each type, only along compatible pairs, and meets each demand $d_Y = f^{\text{in}}(w_Y)$.

The network has 10 nodes and 17 edges, so the Scaling Max-Flow Algorithm runs in $O(m^2 \log_2 C) = O(\log C)$ time by (7.20), where $C = s_O + s_A + s_B + s_{AB}$ is the capacity out of s ; this is polynomial in the input size (the supplies and demands written in binary).

(b) No: at most 97 of the 98 patients in the table can be served. (The table's demands sum to 98, not 100; with the 10 percent figure, $d_B = 10$, the same argument gives at most 99 of 100.) An optimal allocation:

blood used	to O patients	to A patients	to B patients	to AB patients
O	45	5	0	0
A	0	36	0	0
B	0	0	8	0
AB	0	0	0	3

which serves $45 + 41 + 8 + 3 = 97$ patients, leaving one type-A patient without blood.

For the cut, take the source side $A = \{s, u_B, u_{AB}, w_B, w_{AB}\}$. The edges out of u_B and u_{AB} all end at w_B or w_{AB} , inside A , so the edges leaving A are (s, u_O) , (s, u_A) , (w_B, t) , (w_{AB}, t) , and

$$c(A, V - A) = s_O + s_A + d_B + d_{AB} = 50 + 36 + 8 + 3 = 97.$$

By (7.8) every flow has value at most $97 < 98$, so the allocation above is optimal and not all patients can be served.

For the administrators: type O and type A patients can only be given type O or type A blood. There are $45 + 42 = 87$ such patients but only $50 + 36 = 86$ units of type O and type A blood in stock, so at least one of them must go without, no matter how the blood is shared out; the surplus type B and AB blood cannot help them.

Problem 7.9. Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is *balanced*: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

Solution. Compute a maximum flow in the network: source $\rightarrow$ each person (capacity 1), person $\rightarrow$ each hospital within a half-hour's drive (capacity 1), each hospital $\rightarrow$ sink (capacity $\lceil n/k \rceil$); a balanced assignment exists iff the maximum flow value is n .

This is the network of Exercise 7.7 with clients replaced by people, base stations by hospitals, the range condition “within distance r ” by “within a half-hour's driving time”, and $L = \lceil n/k \rceil$; the correctness proof there (both directions, with (7.14) giving an integral flow of value n) applies word for word, the number of people sent to h_j being $f(h_j, t) \leq \lceil n/k \rceil$.

Running time: the network has $n + k + 2$ nodes and at most $nk + n + k$ edges, and the flow value is at most n , so by (7.5) Ford-Fulkerson runs in $O(n(nk + n + k)) = O(n^2k)$ time, after $O(nk)$ time to read off which hospitals are within reach of each person.

Problem 7.10. Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e , a source $s \in V$, and a sink $t \in V$. You are also given a maximum s - t flow in G , defined by a flow value f_e on each edge e . The flow f is *acyclic*: There is no cycle in G on which all edges carry positive flow. The flow f is also integer-valued.

Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

Solution. If e^* is not saturated, f is still maximum; otherwise cancel one unit of flow along an s - t path through e^* and then search once for an augmenting path.

Let $\nu = \nu(f)$ and write G' for the graph with $c'_{e^*} = c_{e^*} - 1$. Every flow in G' is a flow in G , so the maximum flow value ν' of G' satisfies $\nu' \leq \nu$.

(i) $f_{e^*} < c_{e^*}$. Then $f_{e^*} \leq c'_{e^*}$ by integrality, so f is feasible in G' with value $\nu \geq \nu'$, and f is maximum in G' .

(ii) $f_{e^*} = c_{e^*} \geq 1$, with $e^* = (u, v)$. Walk backwards from u : at a node $x \neq s$ with an outgoing edge of positive flow, conservation gives an entering edge (y, x) with $f_{(y,x)} > 0$; step to y . Since no edge leaves t , every node reached other than s has positive outflow, and since f is acyclic no node repeats, so the walk ends at s and gives a path P_1 from s to u of positive-flow edges. Symmetrically, walking forwards from v (no edge enters s) gives a path P_2 from v to t . If P_1 and P_2 shared a node x , then $x \rightsquigarrow u \rightarrow v \rightsquigarrow x$ would be a cycle of positive-flow edges; hence $P = P_1 e^* P_2$ is a simple s - t path with $f_e \geq 1$ on each edge. Let $f' = f - 1$ on the edges of P . Conservation is preserved at the interior nodes of P , $0 \leq f'_e \leq c'_e$ for all e (in particular $f'_{e^*} = c_{e^*} - 1$), and

$$\nu - 1 = \nu(f') \leq \nu' \leq \nu.$$

Now search $G'_{f'}$ once for an s - t path by breadth-first search. If there is one, augment along it as in (7.3); the bottleneck is a positive integer, so the result has value at least ν , hence exactly $\nu = \nu'$ and it is maximum. If there is none, f' is maximum in G' by (7.9).

Running time: each walk visits each node at most once and, scanning the edge lists of the nodes it visits, inspects each edge at most once, so P is found in $O(m + n)$; updating f' , building $G'_{f'}$, the breadth-first search and the single augmentation are each $O(m + n)$.

Problem 7.11. Some friends have written a maximum-flow program that repeatedly finds augmenting paths as in Section 7.1, but their residual graph omits backward edges: it looks for s - t paths only in the graph $\tilde{G}_f$ made up of the edges e with $f(e) < c_e$, and it halts as soon as no s - t path built entirely from such edges exists. Call this the Forward-Edge-Only Algorithm. How it picks its forward-edge paths is left unspecified; it may pick them any way it likes, as long as it stops only when no forward-edge path remains.

The friends claim the following: there is an absolute constant $b > 1$, independent of the input network, such that on every instance of the Maximum-Flow Problem, the Forward-Edge-Only Algorithm always finds a flow of value at least $1/b$ times the maximum-flow value, no matter how it picks its

forward-edge paths.

Decide whether this statement is true or false, and prove the statement or its negation.

Solution. False: for every n there is a network with maximum-flow value n on which the Forward-Edge-Only Algorithm can stop at value 1.

Take nodes $s, t, u_1, \dots, u_n, v_1, \dots, v_n$ and these edges, all with capacity 1:

$$\begin{aligned} (s, u_i), (u_i, v_i), (v_i, t) & \quad (1 \leq i \leq n), \\ (v_i, u_{i+1}) & \quad (1 \leq i \leq n-1). \end{aligned}$$

The three-edge paths $s \rightarrow u_i \rightarrow v_i \rightarrow t$ are pairwise edge-disjoint, so they carry a flow of value n . The cut $(\{s\}, V - \{s\})$ has capacity n , so by (7.8) the maximum-flow value is exactly n .

The algorithm may pick the forward-edge path

$$P = s \rightarrow u_1 \rightarrow v_1 \rightarrow u_2 \rightarrow v_2 \rightarrow \dots \rightarrow u_n \rightarrow v_n \rightarrow t$$

as its first augmenting path. Its bottleneck is 1, so afterwards every edge of P is saturated, and in particular every (u_i, v_i) is saturated. In $\tilde{G}_f$ the only edges leaving s go to $u_2, \dots, u_n$, and the only edge leaving each u_i is (u_i, v_i) , which is now saturated. So the set of nodes reachable from s in $\tilde{G}_f$ is $\{s, u_2, \dots, u_n\}$, which does not contain t , and the algorithm halts with a flow of value 1.

Now let $b > 1$ be any constant and choose an integer $n > b$. On this network the algorithm returns a flow of value $1 = n/n < n/b$, which is less than $1/b$ times the maximum. So no such constant b exists.

Problem 7.12. You are given a flow network in which every edge has unit capacity: a directed graph $G = (V, E)$ with source $s \in V$, sink $t \in V$, and $c_e = 1$ for all $e \in E$, together with an integer parameter k . The aim is to remove k edges so that the maximum s - t flow drops as much as possible; that is, find $F \subseteq E$ with $|F| = k$ such that the maximum s - t flow in $G' = (V, E - F)$ is as small as possible.

Give a polynomial-time algorithm for this problem.

Solution. Delete edges of a minimum s - t cut: the optimum value is exactly $\max(\nu - k, 0)$, where ν is the maximum flow value of G (we assume $k \leq |E|$, otherwise no such F exists).

Algorithm. Run Ford-Fulkerson to get a maximum flow f with $\nu(f) = \nu$, and from it a minimum cut (A^*, B^*) by (7.11). Let D be the set of edges leaving A^* ; since all capacities are 1, $|D| = c(A^*, B^*) = \nu$ by (7.13).

- (i) If $k \leq \nu$: let F be any k edges of D .
- (ii) If $k > \nu$: let F be D together with any $k - \nu$ further edges of E .

Upper bound. In G' the cut (A^*, B^*) has capacity $|D - F| = \max(\nu - k, 0)$, so by (7.8) every s - t flow in G' has value at most $\max(\nu - k, 0)$.

Lower bound. Let F be any set of k edges and (A, B) any s - t cut. Removing F removes at most k edges out of A , each of capacity 1, so

$$c_{G'}(A, B) \geq c_G(A, B) - k \geq \nu - k,$$

using (7.13) in G . Also $c_{G'}(A, B) \geq 0$. Hence the minimum cut of G' , which equals its maximum flow by (7.13), is at least $\max(\nu - k, 0)$ for every choice of F , and our F is optimal.

Running time. Every edge out of s has capacity 1 (and G has no parallel edges), so $C = \sum_{e \text{ out of } s} c_e \leq n - 1$ and Ford-Fulkerson runs in $O(mn)$ time by (7.5); the cut takes $O(m)$ by (7.11) and choosing F takes $O(m)$, for $O(mn)$ overall.

Problem 7.13. Ordinary s - t maximum flow puts capacities on edges and none on nodes. Here we study the opposite: maximum flow and minimum cut when the capacities sit on the nodes.

Let $G = (V, E)$ be a directed graph with source $s \in V$, sink $t \in V$, and a nonnegative capacity $c_v \geq 0$

on every node $v \in V$. For a flow f , the flow through node v is $f^{\text{in}}(v)$. A flow is feasible if it obeys the usual conservation constraints and the node-capacity constraints $f^{\text{in}}(v) \leq c_v$ for all v . Give a polynomial-time algorithm that finds a maximum s - t flow in such a node-capacitated network. Define an s - t cut for node-capacitated networks, and prove that the analogue of the Max-Flow Min-Cut Theorem holds.

Solution. Split every node v into an entry copy v_{in} and an exit copy v_{out} joined by an edge of capacity c_v , and run an ordinary maximum-flow algorithm on the result.

(As in Section 7.1, no edge enters s and none leaves t ; so $f^{\text{in}}(s) = 0$ and the constraint at s is vacuous.) Let $M = 1 + \sum_v c_v$. The network G' has nodes $v_{\text{in}}, v_{\text{out}}$ for every $v \in V$; a node edge $(v_{\text{in}}, v_{\text{out}})$ of capacity c_v for every v ; an edge $(u_{\text{out}}, v_{\text{in}})$ of capacity M for every $(u, v) \in E$; source s_{out} and sink t_{out} . *Flows correspond.* Given a feasible node-capacitated f in G , put

$$f'(u_{\text{out}}, v_{\text{in}}) = f(u, v), \quad f'(v_{\text{in}}, v_{\text{out}}) = f^{\text{in}}(v).$$

Conservation at v_{in} is immediate, and at v_{out} ($v \neq s, t$) it is $f^{\text{in}}(v) = f^{\text{out}}(v)$. The node edges obey $f^{\text{in}}(v) \leq c_v$, and the other edges obey

$$f(u, v) \leq \begin{cases} f^{\text{out}}(u) = f^{\text{in}}(u) \leq c_u < M, & u \neq s, \\ f^{\text{in}}(v) \leq c_v < M, & u = s. \end{cases}$$

Conversely, a flow f' in G' gives $f(u, v) = f'(u_{\text{out}}, v_{\text{in}})$. For $v \neq s, t$, conservation at v_{in} and at v_{out} gives

$$f^{\text{in}}(v) = f'(v_{\text{in}}, v_{\text{out}}) = f^{\text{out}}(v),$$

and the same identity $f^{\text{in}}(v) = f'(v_{\text{in}}, v_{\text{out}}) \leq c_v$ holds for every v . Both maps keep the value, since $f^{\text{out}}(s) = f'^{\text{out}}(s_{\text{out}})$. So a maximum flow of G' , mapped back, is a maximum feasible flow of G , and both have the same value.

Cuts. A node cut is a set $X \subseteq V \setminus \{s\}$ that contains a node of every s - t path in G (possibly t itself). Its capacity is $c(X) = \sum_{v \in X} c_v$.

Theorem. The maximum value of a feasible flow equals the minimum capacity of a node cut.

By (7.13) applied to G' , it suffices to show that the minimum capacity of a node cut equals the minimum capacity of an s_{out} - t_{out} cut in G' .

(i) Node cut to G' cut. Take a node cut X . Let A be the set of nodes reachable from s_{out} in G' once the node edges of X are deleted, and let B be the rest. Any s_{out} - t_{out} path in G' traces an s - t walk in G , and that walk contains an s - t path, so it must pass through a node edge of X . Hence $t_{\text{out}} \in B$. By the definition of A , the only edges leaving A are node edges of X , so

$$c(A, B) \leq c(X).$$

(ii) G' cut to node cut. A minimum cut (A, B) of G' has capacity at most $c(\{t\}) < M$, because by (i) the cut built from $X = \{t\}$ has capacity at most c_t . So every edge it cuts is a node edge. Put

$$X = \{v : v_{\text{in}} \in A, v_{\text{out}} \in B\},$$

so that $c(X) = c(A, B)$. For any s - t path $s = v_0, v_1, \dots, v_k = t$ in G , the path

$$s_{\text{out}} \rightarrow v_{1,\text{in}} \rightarrow v_{1,\text{out}} \rightarrow \dots \rightarrow t_{\text{in}} \rightarrow t_{\text{out}}$$

in G' goes from A to B , so at some point it crosses an edge from A to B . That edge is a node edge $(v_{i,\text{in}}, v_{i,\text{out}})$ with $i \geq 1$, so $v_i \in X$. Hence X is a node cut with $c(X) = c(A, B)$.

Together,

$$\begin{aligned} \max_f v(f) &= \max \text{ flow of } G' = \min \text{ cut of } G' \\ &= \min \{c(X) : X \text{ a node cut}\}. \end{aligned}$$

After the max flow is computed, (7.11) gives a minimum cut (A, B) of G' in linear time, and step (ii) reads off a minimum node cut X .

Running time. G' has $2n$ nodes and $m + n$ edges, and building it takes $O(m + n)$. Preflow-Push with maximum-height selection runs in $O(mn) + O(n^3) = O(n^3)$ on $2n$ nodes by (7.30) and (7.33), so the total is $O(n^3)$ (here $m = O(n^2)$).

Problem 7.14. The *Escape Problem*: we are given a directed graph $G = (V, E)$, a set $X \subset V$ of *populated nodes* and a disjoint set $S \subset V$ of *safe nodes*. A set of *evacuation routes* is a set of paths in G such that (i) every node of X is the tail (starting node) of one path, (ii) every path ends at a node of S , and (iii) no two paths share an edge.

- (a) Given G, X, S , decide in polynomial time whether a set of evacuation routes exists.
(b) Same problem, but with (iii) strengthened to: no two paths share a node. Decide in polynomial time whether such a set of evacuation routes exists. Also give an example with the same G, X, S for which the answer to (a) is yes but the answer to (b) is no.

Solution. (a) Add a super-source and super-sink and test whether the maximum flow equals $|X|$. Build the network G' : keep every edge of G with capacity 1; add a source s with an edge (s, x) of capacity 1 for each $x \in X$; add a sink t with an edge (v, t) of capacity $|X|$ for each $v \in S$. Answer yes iff the maximum s - t flow in G' has value $|X|$.

Routes $\Rightarrow$ flow. Given routes P_x ($x \in X$) ending at $v_x \in S$, send one unit along each $s \rightarrow x \xrightarrow{P_x} v_x \rightarrow t$. Every edge of G lies on at most one route, each (s, x) is used once, and each (v, t) is used at most $|X|$ times, so this is a feasible flow of value $|X|$.

Flow $\Rightarrow$ routes. All capacities are integers, so by (7.14) there is an integer maximum flow f , of value $|X|$. The tracing argument in the proof of (7.42) works for any integer flow if each traced path or cycle is lowered by 1 rather than set to 0. It splits f into $|X|$ s - t paths, and each edge e lies on at most $f(e)$ of them. Since $\nu(f) = |X| = f^{\text{out}}(s)$, every edge (s, x) carries one unit, so each $x \in X$ starts exactly one path. Each path has the form $s, x, \dots, v, t$ with $v \in S$, because only safe nodes have edges into t . Deleting s and t leaves a path in G from x to a safe node. Edges of G have capacity 1, so no two of these paths share an edge, and they form a set of evacuation routes.

Time. The capacity out of s is $C = |X| \leq n$, and G' has $m + |X| + |S| = O(m + n)$ edges. By (7.5), Ford-Fulkerson therefore runs in $O((m + n)n)$ time.

(b) Split every node, as for node capacities (Exercise 13), and make all capacities 1. Build G'' as follows. Replace each $v \in V$ by two nodes $v_{\text{in}}, v_{\text{out}}$ joined by an edge $(v_{\text{in}}, v_{\text{out}})$ of capacity 1. Replace each edge $(u, v) \in E$ by $(u_{\text{out}}, v_{\text{in}})$ with capacity 1. Add edges (s, x_{in}) for $x \in X$ and (v_{out}, t) for $v \in S$, all of capacity 1. Answer yes iff the maximum flow in G'' has value $|X|$.

Routes $\Rightarrow$ flow. For node-disjoint routes, send one unit along $s \rightarrow x_{\text{in}} \rightarrow x_{\text{out}} \rightarrow \dots \rightarrow v_{\text{in}} \rightarrow v_{\text{out}} \rightarrow t$ for each route. Every node of G lies on at most one route, so each edge $(v_{\text{in}}, v_{\text{out}})$ carries at most 1. The routes end at distinct safe nodes, so each (v_{out}, t) also carries at most 1. This gives a flow of value $|X|$.

Flow $\Rightarrow$ routes. By (7.14) there is a 0-1 maximum flow, of value $|X|$, and by (7.42) it contains $|X|$ edge-disjoint s - t paths. As in (a), each path uses exactly one edge (s, x_{in}) , with every $x \in X$ covered, and its last edge is some (v_{out}, t) with $v \in S$. A path through v_{in} must continue along $(v_{\text{in}}, v_{\text{out}})$, since that is the only edge leaving v_{in} . A path reaching v_{out} must arrive along that same edge, since it is the only edge entering v_{out} . Hence two paths through the same node v of G would share the edge $(v_{\text{in}}, v_{\text{out}})$, which is impossible. Contracting each pair $v_{\text{in}}, v_{\text{out}}$ back to v and deleting s, t therefore gives node-disjoint evacuation routes.

Time. G'' has $2n + 2$ nodes and $n + m + |X| + |S| = O(m + n)$ edges, and $C = |X| \leq n$. By (7.5) the running time is $O((m + n)n)$.

Example. Take $V = \{x_1, x_2, v\}$, $E = \{(x_1, v), (x_2, v)\}$, $X = \{x_1, x_2\}$ and $S = \{v\}$. The routes x_1v and x_2v share no edge, so the answer to (a) is yes. Every route must end at v , the only safe node, so the two routes always share v and the answer to (b) is no.

7.3 Exercises 7.15–7.21

Problem 7.15. A co-op has n residents $p_1, \dots, p_n$ (you, your friend Alanis, and $n - 2$ others) and there are n nights $d_1, \dots, d_n$; each resident must cook exactly once and someone must cook every night. Resident p_i cannot cook on the nights in a set $S_i \subset \{d_1, \dots, d_n\}$. A *feasible dinner schedule* assigns each resident to a different night so that every resident cooks exactly once, every night has

a cook, and whenever p_i cooks on d_j we have $d_j \notin S_i$.

(a) Describe a bipartite graph G that has a perfect matching if and only if a feasible dinner schedule exists.

(b) Alanis produces a schedule in which $n - 2$ residents are assigned to distinct nights on which they are available, but the remaining two residents p_i and p_j are both assigned to the same night d_k , while the remaining night d_ℓ has no cook. Using her almost-correct schedule, show how to decide in $O(n^2)$ time whether a feasible dinner schedule exists, and output one if it does.

Solution. (a) Take G with sides $X = \{p_1, \dots, p_n\}$, $Y = \{d_1, \dots, d_n\}$ and an edge (p_i, d_j) exactly when $d_j \notin S_i$. A perfect matching assigns each person to one night and each night to one person, using only allowed pairs, i.e. it is a feasible schedule; conversely the pairs (person, night cooked) of a feasible schedule are n disjoint edges of G , a perfect matching.

(b) Turn Alanis's schedule into a matching of size $n - 1$ or $n - 2$ and finish with at most two Ford-Fulkerson augmentations. Let G' be the flow network of Section 7.5 built from G (edges $s \rightarrow p$, $p \rightarrow d$ for each edge of G , $d \rightarrow t$, all of capacity 1); G has $m \leq n^2$ edges, so G' has $O(n^2)$ edges.

M = the $n-2$ correct (person, night) pairs of Alanis's schedule

if d_k not in S_i : add (p_i, d_k) to M

else if d_k not in S_j : add (p_j, d_k) to M

f = flow in G' sending 1 unit along $s \rightarrow p \rightarrow d \rightarrow t$ for each (p, d) in M

while $v(f) < n$:

 BFS from s in the residual graph G'_f

 if t unreachable: report "no feasible schedule"; stop

 augment f along the s - t path found

output the pairs (p, d) with $f(p, d) = 1$

M is a matching of G : the $n - 2$ correct pairs use distinct people and distinct nights other than d_k, d_ℓ (the n nights are all assigned except d_ℓ , and only d_k twice), and at most one pair at d_k is added, using an allowed edge. Hence f is a valid integer s - t flow with

$$v(f) = |M| \in \{n - 2, n - 1\}.$$

Each augmentation keeps f integral and raises $v(f)$ by exactly 1 (by (7.2) and (7.3), all capacities being 1), so the loop runs at most $n - v(f) \leq 2$ times. If it stops because t is unreachable in G'_f , then f is a maximum flow by (7.9), of value $< n$; by (7.37) the maximum matching of G has size $< n$, so by (a) there is no feasible schedule. If it reaches $v(f) = n$, then by (7.34)-(7.36) the edges $p \rightarrow d$ carrying flow form a matching of size n in G , which by (a) is a feasible schedule.

Building G and G' and f takes $O(n^2)$; each of the at most two iterations is one BFS plus one augmentation on a graph with $O(n)$ nodes and $O(n^2)$ edges, so $O(n^2)$ each. Total: $O(n^2)$.

Problem 7.16. A Web site has identified k distinct demographic groups $G_1, G_2, \dots, G_k$ (the groups may overlap; for example G_1 could be all residents of New York State and G_2 all people with a degree in computer science). The site has contracts with m different advertisers to show a certain number of copies of their ads to users of the site. The contract with the i th advertiser looks like this.

- For a subset $X_i \subseteq \{G_1, \dots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in X_i .
- For a number r_i , advertiser i wants its ads shown to at least r_i users each minute.

An advertising policy is a way to show a single ad to each user of the site. Suppose that at a given minute there are n users visiting the site, and from registration information we know that user j (for $j = 1, 2, \dots, n$) belongs to a subset $U_j \subseteq \{G_1, \dots, G_k\}$ of the demographic groups. Is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers are satisfied for this minute? (That is, for each $i = 1, 2, \dots, m$, can at least r_i of the n users, each belonging to at least one demographic group in X_i , be shown an ad provided by advertiser i ?)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

Solution. Compute a maximum flow in a bipartite network that sends r_i units from each advertiser to users eligible for that advertiser; a policy exists iff the flow saturates all of $R = \sum_i r_i$ and every user is eligible for some advertiser.

Call user j *eligible* for advertiser i if $U_j \cap X_i \neq \emptyset$. Build G' with source s , sink t , a node a_i per advertiser and a node u_j per user, and edges

$$(s, a_i) \text{ of capacity } r_i, \quad (u_j, t) \text{ of capacity } 1, \\ (a_i, u_j) \text{ of capacity } 1 \text{ whenever } j \text{ is eligible for } i.$$

Run Ford-Fulkerson to get an integer maximum flow f (7.14). If $\nu(f) < R$, or some user is eligible for no advertiser, report that no policy exists. Otherwise show user j the ad of the advertiser i with $f(a_i, u_j) = 1$ if there is one, and the ad of any advertiser j is eligible for if not.

Correctness. If $\nu(f) = R$, every edge (s, a_i) is saturated, so advertiser i has exactly r_i users u_j with $f(a_i, u_j) = 1$, all eligible; capacity 1 on (u_j, t) means each user is counted for at most one advertiser. So every user gets one ad, every ad goes to an eligible user, and advertiser i gets at least r_i users. Conversely, given a valid policy, every user is eligible for the advertiser whose ad it sees; pick for each i any r_i of the users shown ad i and send one unit along $s \rightarrow a_i \rightarrow u_j \rightarrow t$ for each. The capacities hold (each user is picked at most once), so this is a flow of value R , and $\nu(f) \geq R$. Since $c(\{s\}, V - \{s\}) = R$, we get $\nu(f) = R$ by (7.8).

Running time. Testing eligibility costs $O(k)$ per pair, so building G' takes $O(mnk)$, and G' has $O(mn)$ edges. The cut $(V - \{t\}, \{t\})$ has capacity n , so Ford-Fulkerson does at most n augmentations (each raises the value by at least 1), each costing $O(mn)$, as in (7.5). The total is $O(mnk + mn^2)$.

Problem 7.17. Some network administrators want to diagnose the extent of a failure in their network. The network carries traffic from a designated source node s to a designated target node t , so we model it as a directed graph $G = (V, E)$ in which every edge has capacity 1 and every node lies on at least one path from s to t .

When everything runs smoothly, the maximum s - t flow in G has value k . Now an attacker has destroyed some of the edges, so that there is no path from s to t using the surviving edges. The administrators believe, and we assume they are right, that the attacker has destroyed only k edges, the minimum number needed to separate s from t (the size of a minimum s - t cut).

A monitoring tool runs on node s : the command $\text{ping}(v)$, for a node v , reports whether there is currently a path from s to v . (So $\text{ping}(t)$ reports that no path exists, while $\text{ping}(s)$ always reports a path from s to itself.) Give an algorithm that issues a sequence of ping commands to nodes of the network and then reports the *full* set of nodes that are not currently reachable from s . Pinging every node would work, but you should use many fewer pings, given that only k edges were deleted. Your algorithm may choose which node to ping next based on the outcomes of earlier pings.

Give an algorithm that accomplishes this task using only $O(k \log n)$ pings.

Solution. Decompose a maximum flow into k edge-disjoint s - t paths and binary-search each path for its one destroyed edge. That finds all k destroyed edges with $k \lceil \log_2 n \rceil$ pings, and then a BFS with no further pings finds the unreachable nodes.

```
compute an integer max flow f in G (value k), and k edge-disjoint
s-t paths P_1..P_k from it by (7.42), cycles cut out
for each path P = (v_0 = s, v_1, ..., v_l = t):
    lo = 0, hi = l                                # ping(v_lo) yes, ping(v_hi) no
    while hi - lo > 1:
        mid = (lo + hi) / 2
        if ping(v_mid) then lo = mid else hi = mid
    add edge (v_lo, v_hi) to F
BFS from s in G - F; report every node not reached
```

Correctness. Let F^* be the set of destroyed edges and A the set of nodes reachable from s in $G - F^*$, with $B = V - A$. Every edge of G leaving A is in F^* , since otherwise its head would be reachable. Also $t \in B$, so (A, B) is an s - t cut and

$$k \leq c(A, B) \leq |F^*| = k,$$

where the first inequality is (7.8) applied to f . So the edges leaving A are exactly F^* , and $f^{\text{out}}(A) \leq c(A, B) = k$. Then (7.6) gives

$$k = \nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leq k - f^{\text{in}}(A),$$

so $f^{\text{in}}(A) = 0$, and no flow-carrying edge goes from B into A . Each P_r uses only edges with $f(e) = 1$, so once P_r leaves A it never comes back. Its nodes in A therefore form a prefix $v_0, \dots, v_j$, and the predicate $\text{ping}(v_i)$ is true for $i \leq j$ and false for $i > j$. The binary search keeps $v_{lo} \in A$ and $v_{hi} \in B$ while it shrinks $hi - lo$, so it stops at the edge (v_j, v_{j+1}) , which leaves A and so lies in F^* . The k paths are edge-disjoint, so they give k distinct edges of F^* , which means $F = F^*$. The BFS in $G - F$ then outputs exactly B .

Pings. Each path is simple, so $l \leq n - 1$, and the search makes at most $\lceil \log_2 n \rceil$ pings per path. That is $k \lceil \log_2 n \rceil = O(k \log n)$ pings in all.

Problem 7.18. Consider the Bipartite Matching Problem on a bipartite graph $G = (V, E)$, where V is partitioned into sets X and Y and each edge has one end in X and the other in Y . If M is a matching in G , a node $y \in Y$ is *covered* by M if y is an end of one of the edges in M .

(a) We are given G and a matching M in G . For a given number k , we want to decide if there is a matching M' in G so that (i) M' has k more edges than M does, and (ii) every node $y \in Y$ that is covered by M is also covered by M' . This is the *Coverage Expansion Problem* with input G , M and k , and such an M' is a *solution* to the instance. Give a polynomial-time algorithm that takes an instance of Coverage Expansion and either returns a solution M' or reports (correctly) that there is no solution. Include an analysis of the running time and a brief proof of correctness. (You may wish to look at part (b) first.)

Example (Figure 7.29). Let $X = \{x_1, x_2\}$ and $Y = \{y_1, \dots, y_5\}$, where x_1 is adjacent to y_1, y_2, y_3 and x_2 is adjacent to y_3, y_4, y_5 . Let $M = \{(x_1, y_2)\}$ and $k = 1$. The answer is yes: for example, $M' = \{(x_1, y_2), (x_2, y_4)\}$ has one more edge than M and still covers y_2 .

(b) Give an example of an instance of Coverage Expansion, specified by G , M and k , such that the instance has a solution, but in any solution M' the edges of M do not form a subset of the edges of M' .

(c) Let G be a bipartite graph and M any matching in G . Let K_1 be the size of the largest matching M' such that every node y covered by M is also covered by M' , and let K_2 be the size of the largest matching M'' in G . Clearly $K_1 \leq K_2$. Prove that $K_1 = K_2$: we can obtain a maximum matching even if we are constrained to cover all the nodes covered by the initial matching M .

Solution. (a) Run Ford-Fulkerson on the matching network G' of Section 7.5, starting from the flow given by M and stopping after k augmentations. If k augmentations are possible, return the resulting matching M' ; if the residual graph runs out of s - t paths first, report that there is no solution.

Key fact. An augmentation along an s - t path in the residual graph G'_f never uncovers a node of Y . Such a path has the form $s, x_1, y_1, x_2, y_2, \dots, x_r, y_r, t$, where x_1 and y_r are unmatched and each (y_i, x_{i+1}) is a backward edge for a matched edge. Augmenting replaces the $r - 1$ matched edges (x_{i+1}, y_i) by the r edges (x_i, y_i) , $i = 1, \dots, r$. Every y_i is matched afterwards, and nodes off the path keep their partner. So the set of covered nodes of Y only grows, and each augmentation adds exactly one edge to the matching (by (7.37) the matching edges are the X - Y edges carrying flow).

Correctness. If k augmentations succeed, the key fact shows that M' satisfies (i) and (ii). If they do not, the algorithm has stopped at a flow with no s - t path in the residual graph, which is a maximum flow by (7.9). By (7.37) its matching has size $K_2 < |M| + k$. Any solution would be a matching of size $|M| + k > K_2$, which is impossible, so the report is correct.

Running time. Each augmentation is a search in G'_f for a path, in $O(m + n)$ time, and there are at most $k \leq n$ of them. That is $O(k(m + n)) = O(mn)$ when G has no isolated nodes.

(b) Take $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, $E = \{(x_1, y_1), (x_1, y_2), (x_2, y_2)\}$, $M = \{(x_1, y_2)\}$ and $k = 1$. A solution must have 2 edges, so it is a perfect matching. The only perfect matching is $M' = \{(x_1, y_1), (x_2, y_2)\}$, because (x_1, y_2) leaves x_2 with no partner. This M' covers y_2 , so it is a solution,

and it does not contain (x_1, y_2) .

(c) Run the algorithm of (a) with no limit on k , that is, Ford-Fulkerson started from M until G'_f has no s - t path. It ends at a maximum flow by (7.9), so by (7.37) its matching M' has $|M'| = K_2$. By the key fact, M' covers every node covered by M . Hence

$$K_2 = |M'| \leq K_1 \leq K_2,$$

so $K_1 = K_2$.

Problem 7.19. For each of the next n days, a hospital has fixed the number of doctors it wants on hand: on day i it requires that *exactly* p_i doctors be present. There are k doctors, and doctor j provides a set L_j of days on which he or she is willing to work. The system should take these lists and try to return to each doctor j a list L'_j with the following properties.

(A) L'_j is a subset of L_j , so that doctor j only works on days he or she finds acceptable.

(B) Taken together, the lists $L'_1, \dots, L'_k$ cause exactly p_i doctors to be present on day i , for $i = 1, 2, \dots, n$.

(a) Give a polynomial-time algorithm that takes the numbers $p_1, p_2, \dots, p_n$ and the lists $L_1, \dots, L_k$, and either returns lists $L'_1, L'_2, \dots, L'_k$ satisfying (A) and (B), or reports (correctly) that no set of lists $L'_1, L'_2, \dots, L'_k$ satisfies both (A) and (B).

(b) The doctors' lists are often too restrictive, so the system often reports (correctly) that no acceptable set of lists exists. The hospital therefore relaxes the requirements: it adds a parameter $c > 0$, and the system should now try to return to each doctor j a list L'_j with the following properties.

(A*) L'_j contains at most c days that do not appear on the list L_j .

(B) (Same as before.) The lists $L'_1, \dots, L'_k$ cause exactly p_i doctors to be present on day i , for $i = 1, 2, \dots, n$.

Give a polynomial-time algorithm that takes $p_1, \dots, p_n$, the lists $L_1, \dots, L_k$ and the parameter $c > 0$, and either returns lists $L'_1, \dots, L'_k$ satisfying (A*) and (B), or reports (correctly) that no such set of lists exists.

Solution. (a) Find a maximum flow in a doctors-to-days network in which day i can absorb p_i units. The lists exist iff the flow has value $P = \sum_i p_i$.

Build G' with source s , sink t , a node u_j per doctor and a node w_i per day, and edges

$$\begin{aligned} (s, u_j) &\text{ of capacity } n, & (w_i, t) &\text{ of capacity } p_i, \\ (u_j, w_i) &\text{ of capacity } 1 \text{ for each } i \in L_j. \end{aligned}$$

Compute an integer maximum flow f with Ford-Fulkerson (7.14). If $\nu(f) = P$, return $L'_j = \{i : f(u_j, w_i) = 1\}$; otherwise report that no lists exist.

Correctness. If $\nu(f) = P$, every edge (w_i, t) is saturated, since the cut $(V - \{t\}, \{t\})$ has capacity P . By conservation at w_i , exactly p_i doctors j have $i \in L'_j$, which is (B). Each L'_j uses only edges (u_j, w_i) with $i \in L_j$, which is (A). Conversely, lists satisfying (A) and (B) give a flow of value P : put $f(u_j, w_i) = 1$ for $i \in L'_j$, $f(s, u_j) = |L'_j| \leq n$, and $f(w_i, t) = p_i$. So a maximum flow has value $\geq P$, and hence exactly P .

Running time. G' has $O(nk)$ edges and $\nu(f) \leq P \leq nk$ (if some $p_i > k$, report failure at once). Ford-Fulkerson therefore runs in $O(nk \cdot nk) = O(n^2 k^2)$ time, as in (7.5).

(b) Add to the network of (a) a node g_j for each doctor j that carries his or her “extra” days. Use the edges of (a), together with

$$\begin{aligned} (u_j, g_j) &\text{ of capacity } c, \\ (g_j, w_i) &\text{ of capacity } 1 \text{ for each } i \notin L_j. \end{aligned}$$

Compute an integer maximum flow f . If $\nu(f) = P$, return $L'_j = \{i : f(u_j, w_i) = 1 \text{ or } f(g_j, w_i) = 1\}$; otherwise report that no lists exist.

Correctness. For each pair (j, i) there is exactly one unit-capacity route from u_j to w_i : directly if $i \in L_j$, and through g_j if $i \notin L_j$. So f assigns doctor j to day i at most once. The days of L'_j outside L_j are exactly the units through g_j , and there are at most c of them, which is (A*). Saturation of

the edges (w_i, t) gives (B) exactly as in (a). Conversely, lists satisfying (A*) and (B) define a flow of value P : send one unit on the route for each pair with $i \in L'_j$, and set $f(u_j, g_j) = |L'_j - L_j| \leq c$ and $f(s, u_j) = |L'_j| \leq n$. So the maximum flow has value P .

Running time. The network has $O(nk)$ edges and maximum flow value $\leq P \leq nk$, so Ford-Fulkerson again runs in $O(n^2k^2)$ time.

Problem 7.20. An atmospheric science experiment needs measurements of a set S of n different conditions in the atmosphere, using m balloons. Each balloon can make at most two measurements. Not all balloons can measure all conditions: for each balloon $i = 1, \dots, m$ there is a set S_i of conditions that balloon i can measure. To make the results more reliable, each measurement should be taken by at least k different balloons. (A single balloon should not measure the same condition twice.)

Example. Let $k = 2$, let there be $n = 4$ conditions c_1, c_2, c_3, c_4 , and let there be $m = 4$ balloons with $S_1 = S_2 = \{c_1, c_2, c_3\}$ and $S_3 = S_4 = \{c_1, c_3, c_4\}$. One way to measure each condition at least $k = 2$ times is to have balloon 1 measure c_1, c_2 , balloon 2 measure c_2, c_3 , balloon 3 measure c_3, c_4 , and balloon 4 measure c_1, c_4 .

(a) Give a polynomial-time algorithm that takes an instance of this problem (the n conditions, the sets S_i for each of the m balloons, and the parameter k) and decides whether each condition can be measured by k different balloons while each balloon measures at most two conditions.

(b) There is an extra requirement. Each balloon is produced by one of three subcontractors, and there must be no condition for which all k measurements come from balloons produced by a single subcontractor. For example, suppose balloon 1 comes from the first subcontractor, balloons 2 and 3 from the second, and balloon 4 from the third. Then the solution above no longer works, because both measurements of c_3 come from the second subcontractor. However, balloons 1 and 2 could each measure c_1, c_2 , and balloons 3 and 4 could each measure c_3, c_4 . Explain how to modify your algorithm for (a) into a polynomial-time algorithm that decides whether there is a solution satisfying all the conditions of (a) plus the subcontractor requirement.

Solution. (a) A solution exists iff the network below has a maximum flow of value nk . Build G' with source s , sink t , a node b_i per balloon and a node c per condition, and edges

$$\begin{aligned} (s, b_i) &\text{ of capacity } 2, & (c, t) &\text{ of capacity } k, \\ (b_i, c) &\text{ of capacity } 1 \text{ for each } c \in S_i. \end{aligned}$$

Compute an integer maximum flow f with Ford-Fulkerson (7.14). If $\nu(f) = nk$, balloon i measures $\{c : f(b_i, c) = 1\}$; otherwise answer no.

Correctness. If $\nu(f) = nk$, every edge (c, t) is saturated, since the cut $(V - \{t\}, \{t\})$ has capacity nk . Conservation then gives each condition c k balloons i with $f(b_i, c) = 1$, all distinct and all able to measure c . Each balloon carries at most 2 units in, so it measures at most two conditions. Conversely, take a valid assignment and keep, for each condition, only k of the balloons measuring it. Sending one unit along $s \rightarrow b_i \rightarrow c \rightarrow t$ for each kept pair respects every capacity, so it is a flow of value nk .

Running time. G' has $O(mn)$ edges and $\nu(f) \leq 2m$, so Ford-Fulkerson takes $O(m \cdot mn) = O(m^2n)$ time, as in (7.5).

(b) Route each condition's measurements through one node per subcontractor, with capacity $k - 1$ into the condition. For each condition c and subcontractor $\ell \in \{1, 2, 3\}$, add a node c_ℓ . Replace each edge (b_i, c) by (b_i, c_ℓ) of capacity 1, where ℓ is balloon i 's subcontractor, and add the edges

$$(c_\ell, c) \text{ of capacity } k - 1, \quad \ell = 1, 2, 3.$$

Keep (s, b_i) of capacity 2 and (c, t) of capacity k , and answer yes iff the maximum flow is nk . This is for $k \geq 2$; for $k = 1$ a valid solution needs two balloons from different subcontractors on each condition, which is exactly the instance with $k = 2$, so run the $k = 2$ network.

Correctness. A flow of value nk gives, as in (a), exactly k distinct balloons per condition and at most two conditions per balloon. At most $k - 1$ of the k balloons for c pass through any one c_ℓ , so they do

not all come from one subcontractor. Conversely, suppose a valid assignment measures c by a set B_c of $\geq k$ balloons that is not all from one subcontractor. Choose $T_c \subseteq B_c$ with $|T_c| = k$: take two balloons of B_c from different subcontractors, and add any $k-2$ others. Then T_c uses at least two subcontractors, so at most $k-1$ of its balloons share a subcontractor. Sending one unit along $s \rightarrow b_i \rightarrow c_\ell \rightarrow c \rightarrow t$ for each $i \in T_c$ therefore respects all capacities and gives a flow of value nk .

Running time. The network has $m + 4n + 2$ nodes and $O(mn)$ edges, and the flow value is still at most $2m$, so the time is again $O(m^2n)$.

Problem 7.21. You're helping to organize a class on campus that has decided to give all its students wireless laptops for the semester. Thus there is a collection of n wireless laptops; there is also a collection of n wireless *access points*, to which a laptop can connect when it is in range.

The laptops are currently scattered across campus; laptop ℓ is within range of a *set* S_ℓ of access points. We will assume that each laptop is within range of at least one access point (so the sets S_ℓ are nonempty); we will also assume that every access point p has at least one laptop within range of it.

To make sure that all the wireless connectivity software is working correctly, you need to try having laptops make contact with access points in such a way that each laptop and each access point is involved in at least one connection. Thus we will say that a *test set* T is a collection of ordered pairs of the form (ℓ, p) , for a laptop ℓ and access point p , with the properties that

- (i) If $(\ell, p) \in T$, then ℓ is within range of p (i.e., $p \in S_\ell$).
- (ii) Each laptop appears in at least one ordered pair in T .
- (iii) Each access point appears in at least one ordered pair in T .

This way, by trying out all the connections specified by the pairs in T , we can be sure that each laptop and each access point have correctly functioning software.

The problem is: Given the sets S_ℓ for each laptop (i.e., which laptops are within range of which access points), and a number k , decide whether there is a test set of size at most k .

Example. Suppose that $n = 3$; laptop 1 is within range of access points 1 and 2; laptop 2 is within range of access point 2; and laptop 3 is within range of access points 2 and 3. Then the set of pairs (laptop 1, access point 1), (laptop 2, access point 2), (laptop 3, access point 3) would form a test set of size 3.

- (a) Give an example of an instance of this problem for which there is no test set of size n . (Recall that we assume each laptop is within range of at least one access point, and each access point p has at least one laptop within range of it.)
- (b) Give a polynomial-time algorithm that takes the input to an instance of this problem (including the parameter k) and decides whether there is a test set of size at most k .

Solution. (a) Take $n = 3$ with $S_1 = S_2 = \{1\}$ and $S_3 = \{1, 2, 3\}$. Any test set contains $(1, 1)$ and $(2, 1)$ (the only pairs for laptops 1, 2), and access points 2, 3 can only be covered by the two further pairs $(3, 2), (3, 3)$; so every test set has size at least $4 > n$ (and $\{(1, 1), (2, 1), (3, 2), (3, 3)\}$ is one).

(b) Compute a maximum matching M in the bipartite graph G with laptops on one side, access points on the other, and an edge (ℓ, p) for each $p \in S_\ell$; answer yes iff $2n - |M| \leq k$. A test set is exactly an edge cover of G , so it suffices to show that the minimum test set has size $2n - \nu$, where ν is the maximum matching size.

$\leq$: given a matching M , add for each of the $2n - 2|M|$ unmatched nodes one edge at it (one exists by the hypotheses on the S_ℓ and the access points); this is a test set of size $|M| + (2n - 2|M|) = 2n - |M|$.

$\geq$: let T be a minimum test set. Every pair of T has an endpoint of T -degree 1, since otherwise deleting it leaves a smaller test set. Hence T contains no path of three edges (its middle edge would have both endpoints of degree ≥ 2), so each connected component of (V, T) is a star with at least one edge; with c components covering all $2n$ nodes, $|T| = 2n - c$. One edge from each star gives a matching of size $c \leq \nu$, so

$$|T| = 2n - c \geq 2n - \nu.$$

By (7.38) the Ford-Fulkerson Algorithm finds M in $O(mn)$ time, where $m = \sum_\ell |S_\ell| \leq n^2$, so the algorithm runs in $O(mn) = O(n^3)$ time.

7.4 Exercises 7.22–7.28

Problem 7.22. Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j . A *diagonal entry* is one of the form m_{ii} for some i .

Swapping rows i and j of the matrix M denotes the following action: we swap the values m_{ik} and m_{jk} for $k = 1, 2, \dots, n$. Swapping two columns is defined analogously.

We say that M is *rearrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that, after all the swapping, all the diagonal entries of M are equal to 1.

(a) Give an example of a matrix M that is not rearrangeable, but for which at least one entry in each row and each column is equal to 1.

(b) Give a polynomial-time algorithm that determines whether a matrix M with 0-1 entries is rearrangeable.

Solution. (a) Take

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Every row and column contains a 1, but rows 2 and 3 have their only 1 in the same column, and swaps never separate entries of a column; so whichever positions rows 2, 3 end up in, at most one of them gets a 1 on the diagonal.

(b) M is rearrangeable iff the bipartite graph G with row nodes $r_1, \dots, r_n$, column nodes $c_1, \dots, c_n$ and an edge (r_i, c_j) for each $m_{ij} = 1$ has a perfect matching; decide this by computing a maximum matching. Row swaps permute rows and column swaps permute columns, independently of each other, and every permutation is a product of transpositions; so the matrices reachable by swaps are exactly

$$M' = (m_{\sigma(i)\tau(j)})_{i,j}, \quad \sigma, \tau \text{ permutations of } \{1, \dots, n\}.$$

If such an M' has all diagonal entries 1, then $m_{\sigma(i)\tau(i)} = 1$ for all i , so the edges $(r_{\sigma(i)}, c_{\tau(i)})$, $i = 1, \dots, n$, form a perfect matching of G . Conversely, if $\{(r_i, c_{\pi(i)})\}$ is a perfect matching, take $\sigma = \text{id}$, $\tau = \pi$: then $m'_{ii} = m_{i\pi(i)} = 1$ for all i .

G has at most n^2 edges, so by (7.38) the Ford-Fulkerson Algorithm finds a maximum matching in $O(n^2 \cdot n) = O(n^3)$ time; M is rearrangeable iff it has size n .

Problem 7.23. Suppose you're looking at a flow network G with source s and sink t , and you want to be able to express something like the following intuitive notion: Some nodes are clearly on the “source side” of the main bottlenecks; some nodes are clearly on the “sink side” of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

Here's one way to divide the nodes of G into three categories of this sort.

- We say a node v is *upstream* if, for all minimum s - t cuts (A, B) , we have $v \in A$ — that is, v lies on the source side of every minimum cut.
- We say a node v is *downstream* if, for all minimum s - t cuts (A, B) , we have $v \in B$ — that is, v lies on the sink side of every minimum cut.
- We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s - t cut (A, B) for which $v \in A$, and at least one minimum s - t cut (A', B') for which $v \in B'$.

Give an algorithm that takes a flow network G and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within a constant factor of the time required to compute a *single* maximum flow.

Solution. Compute one maximum flow f ; the upstream nodes are the set U of nodes reachable from s in G_f , the downstream nodes are the set D of nodes that can reach t in G_f , and all other nodes are central.

Every minimum cut (A, B) has $U \subseteq A$ and $D \subseteq B$. By (7.6) and (7.8),

$$c(A, B) = \nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leq f^{\text{out}}(A) \leq c(A, B),$$

so every edge out of A is saturated and every edge into A carries no flow; thus G_f has no edge from A to B . A residual path from $s \in A$ therefore stays in A , giving $U \subseteq A$; and a residual path from v to $t \in B$ cannot start in A , giving $D \subseteq B$.

Both $(U, V - U)$ and $(V - D, D)$ are minimum cuts. The first is the cut (A^*, B^*) of (7.9). For the second, $t \in D$ and $s \notin D$ (an s - t path in G_f would, by (7.3), give a flow of larger value), and G_f has no edge (u, w) with $u \notin D$, $w \in D$ (else u could reach t). So every edge leaving $V - D$ is saturated and every edge entering it carries no flow, whence by (7.6)

$$\nu(f) = f^{\text{out}}(V - D) - f^{\text{in}}(V - D) = c(V - D, D),$$

which is minimum by (7.8).

Hence a node of U is on the source side of every minimum cut (upstream); a node not in U is on the sink side of $(U, V - U)$ (not upstream); a node of D is downstream; and a node not in D is on the source side of $(V - D, D)$ (not downstream). Since $U \cap D = \emptyset$, the three classes are U , D and $V - (U \cup D)$, as claimed.

Running time: one maximum flow, plus building G_f and doing a breadth-first search from s in G_f and one into t (a search from t in G_f with edges reversed), all $O(m)$; computing a maximum flow already takes $\Omega(m)$ time, so the total is within a constant factor of one max-flow computation.

Problem 7.24. Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative edge capacities $\{c_e\}$. Give a polynomial-time algorithm to decide whether G has a *unique* minimum s - t cut (i.e., an s - t cut of capacity strictly less than that of all other s - t cuts).

Solution. Run the algorithm of Exercise 7.23: compute a maximum flow f , the set U of nodes reachable from s in G_f and the set D of nodes that can reach t in G_f ; the minimum cut is unique iff $U \cup D = V$, i.e. iff there are no central nodes.

By Exercise 7.23, every minimum cut (A, B) satisfies $U \subseteq A$ and $D \subseteq B$, and $(U, V - U)$ and $(V - D, D)$ are minimum cuts.

(i) If $U \cup D = V$, then $D = V - U$ (as $U \cap D = \emptyset$), so every minimum cut has $U \subseteq A \subseteq V - D = U$; thus $(A, B) = (U, V - U)$ is the only minimum cut.

(ii) If some $v \notin U \cup D$, then $v \in V - U$ and $v \in V - D$, so $(U, V - U)$ and $(V - D, D)$ are two distinct minimum cuts.

The running time is that of one maximum-flow computation plus $O(m)$, which is polynomial (e.g. $O(n^3)$ with the Preflow-Push Algorithm, (7.33)).

Problem 7.25. Suppose you live in a big apartment with a lot of friends. Over the course of a year, there are many occasions when one of you pays for an expense shared by some subset of the apartment, with the expectation that everything will get balanced out fairly at the end of the year. For example, one of you may pay the whole phone bill in a given month, another will occasionally make communal grocery runs to the nearby organic food emporium, and a third might sometimes use a credit card to cover the whole bill at the local Italian-Indian restaurant, Little Idli.

In any case, it's now the end of the year and time to settle up. There are n people in the apartment; and for each ordered pair (i, j) there's an amount $a_{ij} \geq 0$ that i owes j , accumulated over the course of the year. We will require that for any two people i and j , at least one of the quantities a_{ij} or a_{ji} is equal to 0. This can be easily made to happen as follows: If it turns out that i owes j a positive amount x , and j owes i a positive amount $y < x$, then we will subtract off y from both sides and declare $a_{ij} = x - y$ while $a_{ji} = 0$. In terms of all these quantities, we now define the *imbalance* of a person i to be the sum of the amounts that i is owed by everyone else, minus the sum of the amounts that i owes everyone else. (Note that an imbalance can be positive, negative, or zero.)

In order to restore all imbalances to 0, so that everyone departs on good terms, certain people will write checks to others; in other words, for certain ordered pairs (i, j) , i will write a check to j for an amount $b_{ij} > 0$.

We will say that a set of checks constitutes a *reconciliation* if, for each person i , the total value of the checks received by i , minus the total value of the checks written by i , is equal to the imbalance of i . Finally, you and your friends feel it is bad form for i to write j a check if i did not actually owe j money, so we say that a reconciliation is *consistent* if, whenever i writes a check to j , it is the case that $a_{ij} > 0$.

Show that, for any set of amounts a_{ij} , there is always a consistent reconciliation in which at most $n - 1$ checks get written, by giving a polynomial-time algorithm to compute such a reconciliation.

Solution. Start from the checks $b_{ij} = a_{ij}$ (for all $a_{ij} > 0$) and repeatedly cancel cycles of the underlying undirected graph until it is a forest. Writing $\beta_i(b)$ for the checks received by i minus those written by i , the starting set is a consistent reconciliation, since

$$\beta_i(a) = \sum_j a_{ji} - \sum_j a_{ij} = \text{imbalance of } i.$$

Let H be the undirected graph with an edge $\{i, j\}$ whenever $b_{ij} > 0$; it is simple, because $b_{ij} > 0$ only where $a_{ij} > 0$, and at most one of a_{ij}, a_{ji} is positive.

```

b = a  (one check i → j for each aij > 0)
while H contains a cycle C = v1, v2, ..., vr, v1:
    call a check on C forward if it goes vk → v(k+1)
        (in the traversal direction), backward otherwise
    if every check on C is forward: reverse the traversal
    d = min of b over the backward checks of C
    add d to every forward check, subtract d from every backward check
    discard checks whose amount became 0
return b

```

Invariants. Every remaining check $i \rightarrow j$ was in the starting set, so $a_{ij} > 0$ (consistency), and all amounts stay positive since d is the minimum backward amount. Each β_i is unchanged: a node v_k of C meets the two checks joining it to v_{k-1} and v_{k+1} , and in each of the four orientation cases

$$\begin{aligned} \Delta\beta_{v_k} &= (\text{change from } \{v_{k-1}, v_k\}) + (\text{change from } \{v_k, v_{k+1}\}) \\ &= (+d) + (-d) = 0 \end{aligned}$$

(on $\{v_{k-1}, v_k\}$ a forward check raises what v_k receives by d , a backward one lowers what v_k writes by d ; on $\{v_k, v_{k+1}\}$ a forward check raises what v_k writes, a backward one lowers what v_k receives). Nodes off C are untouched. So b remains a consistent reconciliation throughout.

Termination and size. Each iteration deletes at least one check (the backward check attaining d) and adds none, so there are at most $|E(H)| \leq \binom{n}{2}$ iterations; at the end H is a forest on n nodes, so at most $n - 1$ checks are written.

Running time. A cycle is found (or H shown acyclic) by depth-first search in $O(n^2)$, and the update is $O(n)$, so the total is $O(n^2 \cdot n^2) = O(n^4)$.

Problem 7.26. You can tell that cellular phones are at work in rural communities, from the giant microwave towers you sometimes see sprouting out of corn fields and cow pastures. Let's consider a very simplified model of a cellular phone network in a sparsely populated area.

We are given the locations of n *base stations*, specified as points $b_1, \dots, b_n$ in the plane. We are also given the locations of n cellular phones, specified as points $p_1, \dots, p_n$ in the plane. Finally, we are given a *range parameter* $\Delta > 0$. We call the set of cell phones *fully connected* if it is possible to assign each phone to a base station in such a way that

- each phone is assigned to a different base station, and
- if a phone at p_i is assigned to a base station at b_j , then the straight-line distance between the

points p_i and b_j is at most Δ .

Suppose that the owner of the cell phone at point p_1 decides to go for a drive, traveling continuously for a total of z units of distance due east. As this cell phone moves, we may have to update the assignment of phones to base stations (possibly several times) in order to keep the set of phones fully connected.

Give a polynomial-time algorithm to decide whether it is possible to keep the set of phones fully connected at all times during the travel of this one cell phone. (You should assume that all other phones remain stationary during this travel.) If it is possible, you should report a sequence of assignments of phones to base stations that will be sufficient in order to maintain full connectivity; if it is not possible, you should report a point on the traveling phone's path at which full connectivity cannot be maintained.

You should try to make your algorithm run in $O(n^3)$ time if possible.

Example. Suppose we have phones at $p_1 = (0, 0)$ and $p_2 = (2, 1)$; we have base stations at $b_1 = (1, 1)$ and $b_2 = (3, 1)$; and $\Delta = 2$. Now consider the case in which the phone at p_1 moves due east a distance of 4 units, ending at $(4, 0)$. Then it is possible to keep the phones fully connected during this motion: We begin by assigning p_1 to b_1 and p_2 to b_2 , and we reassign p_1 to b_2 and p_2 to b_1 during the motion (for example, when p_1 passes the point $(2, 0)$).

Solution. Cut the drive into at most $2n + 1$ pieces on which the set of base stations in range of the moving phone is constant, and keep a maximum bipartite matching up to date with one augmenting-path search per change.

Write $p_1 = (x_0, y_0)$, so at time $t \in [0, z]$ the phone is at $p_1(t) = (x_0 + t, y_0)$. For $b_j = (x_j, y_j)$ the times when b_j is in range form a closed interval, possibly empty:

$$\begin{aligned} J_j &= \{t \in [0, z] : |p_1(t) - b_j| \leq \Delta\} \\ &= [x_j - x_0 - w_j, x_j - x_0 + w_j] \cap [0, z], \end{aligned}$$

with $w_j = \sqrt{\Delta^2 - (y_j - y_0)^2}$ (and $J_j = \emptyset$ if $|y_j - y_0| > \Delta$). Let $0 = e_0 < e_1 < \dots < e_m = z$ be the distinct numbers among $0, z$ and the endpoints of the J_j , so $m \leq 2n + 1$. Let $G(t)$ be the bipartite graph on phones and bases with the fixed edges (p_i, b_j) , $i \geq 2$, $|p_i - b_j| \leq \Delta$, plus the edges (p_1, b_j) for $t \in J_j$. The phones are fully connected at time t exactly when $G(t)$ has a perfect matching.

compute the intervals J_j and the event times $e_0 < \dots < e_m$

$M =$ maximum matching of $G(\text{midpoint of } (e_0, e_1))$

for $k = 0, \dots, m-1$:

$t_k =$ midpoint of (e_k, e_{k+1})

 for each edge (p_1, b_j) present before but not in $G(t_k)$:

 delete it; if it was in M , remove it from M and

 run one augmenting-path search, augmenting if one is found

 for each edge (p_1, b_j) in $G(t_k)$ but not present before:

 add it; run one augmenting-path search, augmenting if found

 if $|M| < n$: report the point $p_1(t_k)$ and stop

$M_k = M$

report: use M_k on the time interval $[e_k, e_{k+1}]$, $k = 0, \dots, m-1$

Correctness. The algorithm keeps M a maximum matching. Deleting one edge lowers the maximum matching size by at most 1, and adding one edge raises it by at most 1. So after each change the current M has size at least the new maximum minus one. If one search finds no augmenting path, M is maximum by (7.9), applied to the flow network of Section 7.5 through (7.37). Hence $|M| = n$ at step k exactly when $G(t_k)$ has a perfect matching.

On the open interval (e_k, e_{k+1}) no endpoint of any J_j occurs, so $G(t) = G(t_k)$ there. Each J_j is closed, so at the endpoints $G(e_k)$ and $G(e_{k+1})$ contain every edge of $G(t_k)$. Thus:

(i) if the algorithm stops at step k , the phones are not fully connected at time t_k , and the reported point $p_1(t_k)$ is a correct witness;

(ii) otherwise each M_k is a valid assignment on all of $[e_k, e_{k+1}]$. The plan “use M_k on $[e_k, e_{k+1})$, switch to M_{k+1} at e_{k+1} ” keeps the phones fully connected throughout $[0, z]$. If $z = 0$, check $G(0)$ once instead. In the example, $J_1 = [0, 1 + \sqrt{3}]$ and $J_2 = [3 - \sqrt{3}, 4]$. The events are $0 < 3 - \sqrt{3} < 1 + \sqrt{3} < 4$, and the algorithm outputs $\{p_1 b_1, p_2 b_2\}$ on the first two pieces and $\{p_1 b_2, p_2 b_1\}$ on the last.

Running time. Computing the J_j and sorting the events takes $O(n \log n)$, and the initial matching takes $O(n^3)$ with Ford-Fulkerson (7.38). Each J_j adds its edge once and deletes it once, so there are at most $2n$ edge changes. Each costs one augmenting-path search in a graph with $O(n^2)$ edges, $O(n^2)$ time, for a total of $O(n^3)$.

Problem 7.27. Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems like the only option. So they decide to carpool to work.

Unfortunately, they all hate to drive, so they want to make sure that any carpool arrangement they agree upon is fair and doesn't overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

Here's one way to define fairness. Let the people be labeled $S = \{p_1, \dots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the i th day a subset $S_i \subseteq S$ of the people go to work. Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i: p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we'd like to require that p_j drives at most Δ_j times; unfortunately, Δ_j may not be an integer.

So let's say that a *driving schedule* is a choice of a driver for each day, that is, a sequence $p_{i_1}, p_{i_2}, \dots, p_{i_d}$ with $p_{i_t} \in S_t$, and that a *fair driving schedule* is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

(a) Prove that for any sequence of sets $S_1, \dots, S_d$, there exists a fair driving schedule.

(b) Give an algorithm to compute a fair driving schedule with running time polynomial in k and d .

Solution. Fair schedules are exactly the integer flows of value d in a day-person network. The uniform-random "fractional schedule" is a flow of value d , so integrality gives an integer one. (Days with $S_i = \emptyset$ need no driver and are dropped.)

(a) Build G with a source s , a node u_i for each day, a node v_j for each person, and a sink t , and the edges

$$\begin{aligned} (s, u_i) & \text{ capacity } 1, & i = 1, \dots, d, \\ (u_i, v_j) & \text{ capacity } 1, & p_j \in S_i, \\ (v_j, t) & \text{ capacity } \lceil \Delta_j \rceil, & j = 1, \dots, k. \end{aligned}$$

Put $f(s, u_i) = 1$, $f(u_i, v_j) = 1/|S_i|$ for $p_j \in S_i$, and $f(v_j, t) = \Delta_j$. Conservation holds at u_i ($1 = |S_i| \cdot \frac{1}{|S_i|}$) and at v_j (by the definition of Δ_j). The capacities hold because $1/|S_i| \leq 1$ and $\Delta_j \leq \lceil \Delta_j \rceil$. So f is a flow of value d . The cut $(\{s\}, V - \{s\})$ has capacity d , so by (7.8) the maximum flow value is exactly d . All capacities are integers, so by (7.14) there is an integer maximum flow f^* , of value d . In f^* each u_i receives one unit and sends it along exactly one edge (u_i, v_j) with $p_j \in S_i$; let p_j drive on day i . Then p_j drives on

$$\#\{i : f^*(u_i, v_j) = 1\} = f^*(v_j, t) \leq \lceil \Delta_j \rceil$$

days, so this is a fair driving schedule.

(b) Build G , run Ford-Fulkerson, and read off the schedule as in (a). By (a) and (7.10), Ford-Fulkerson returns a maximum flow of value d , and it is integer-valued by (7.2). G has $m \leq d + dk + k$ edges and the capacities out of s sum to $C = d$. By (7.5) the running time is $O(mC) = O(kd^2)$, plus $O(kd)$ to compute the Δ_j and build G .

Problem 7.28. A group of students has decided to add some features to Cornell's on-line Course Management System (CMS), to handle aspects of course planning that are not currently covered by the software. They're beginning with a module that helps schedule office hours at the start of the

semester.

Their initial prototype works as follows. The office hour schedule will be the same from one week to the next, so it's enough to focus on the scheduling problem for a single week. The course administrator enters a collection of nonoverlapping one-hour time intervals $I_1, I_2, \dots, I_k$ when it would be possible for teaching assistants (TAs) to hold office hours; the eventual office-hour schedule will consist of a subset of some, but generally not all, of these time slots. Then each of the TAs enters his or her weekly schedule, showing the times when he or she would be available to hold office hours.

Finally, the course administrator specifies, for parameters a , b , and c , that they would like each TA to hold between a and b office hours per week, and they would like a total of exactly c office hours to be held over the course of the week.

The problem, then, is how to assign each TA to some of the office-hour time slots, so that each TA is available for each of his or her office-hour slots, and so that the right number of office hours gets held. (There should be only one TA at each office hour.)

Example. Suppose there are five possible time slots for office hours: $I_1 = \text{Mon 3-4 P.M.}$; $I_2 = \text{Tue 1-2 P.M.}$; $I_3 = \text{Wed 10-11 A.M.}$; $I_4 = \text{Wed 3-4 P.M.}$; and $I_5 = \text{Thu 10-11 A.M.}$ There are two TAs; the first would be able to hold office hours at any time on Monday or Wednesday afternoons, and the second would be able to hold office hours at any time on Tuesday, Wednesday, or Thursday. Finally, each TA should hold between $a = 1$ and $b = 2$ office hours, and we want exactly $c = 3$ office hours per week total. One possible solution would be to have the first TA hold office hours in time slot I_1 , and the second TA hold office hours in time slots I_2 and I_5 .

(a) Give a polynomial-time algorithm that takes the input to an instance of this problem (the time slots, the TA schedules, and the parameters a , b , and c) and does one of the following two things:

- constructs a valid schedule for office hours, specifying which TA will cover which time slots, or
- reports (correctly) that there is no valid way to schedule office hours.

(b) This office-hour scheduling feature becomes very popular, and so course staffs begin to demand more. In particular, they observe that it's good to have a greater density of office hours closer to the due date of a homework assignment. So they want to be able to specify an *office-hour density* parameter for each day of the week: the number d_i specifies that they want to have at least d_i office hours on a given day i of the week.

For example, suppose that in the previous example we add the constraint that we want at least one office hour on Wednesday and at least one office hour on Thursday. Then the previous solution does not work; but there is a possible solution in which the first TA holds office hours in time slot I_1 , and the second TA holds office hours in time slots I_3 and I_5 . (Another solution would be to have the first TA hold office hours in time slots I_1 and I_4 , and the second TA hold office hours in time slot I_5 .)

Give a polynomial-time algorithm that computes office-hour schedules under this more complex set of constraints. The algorithm should either construct a schedule or report (correctly) that none exists.

Solution. Model the schedule as an integer circulation with demands and lower bounds (Section 7.7). Lower bounds enforce “at least a ” and “at least d_i ”, the capacities enforce “at most b ” and “one TA per slot”, and the demands $\mp c$ enforce “exactly c ”.

(a) Let the TAs be $T_1, \dots, T_r$. Build G with nodes s (demand $-c$), t (demand $+c$), a node T_x for each TA and a node I_ℓ for each slot (demand 0). The edges, with [lower bound, capacity], are

$$\begin{aligned} (s, T_x) & \quad [a, b], \quad x = 1, \dots, r, \\ (T_x, I_\ell) & \quad [0, 1], \quad T_x \text{ available during } I_\ell, \\ (I_\ell, t) & \quad [0, 1], \quad \ell = 1, \dots, k. \end{aligned}$$

Decide whether G has a feasible circulation by the reduction of (7.52) and then (7.50), that is, with one maximum-flow computation. If it does not, report that no schedule exists. If it does, take an integer-valued one f (it exists by (7.52), since all data are integers), and let T_x cover I_ℓ whenever $f(T_x, I_\ell) = 1$.

Correctness. Given a valid schedule, put one unit on (T_x, I_ℓ) for each assignment and on (I_ℓ, t) for each

covered slot, and put the number of hours of T_x on (s, T_x) . Each of these lies in $[a, b]$. The net flow out of s is the total number of hours, c , so this is a feasible circulation. Conversely, from an integer feasible circulation:

- (i) inflow to I_ℓ is at most 1, so each slot has at most one TA, and only an available one;
- (ii) T_x holds $f(s, T_x) \in [a, b]$ hours, by conservation at T_x ;
- (iii) the number of covered slots is $f^{\text{in}}(t) = c$, by the demand at t .

So it is a valid schedule.

(b) Insert a node D_i for each day i . Replace each edge (I_ℓ, t) by (I_ℓ, D_i) with $[0, 1]$, where i is the day of I_ℓ , and add (D_i, t) with $[d_i, k]$. The algorithm is the same. The flow on (D_i, t) equals the number of covered slots on day i , so the lower bound d_i is exactly the density constraint. Both directions of the argument in (a) go through unchanged. In the example, the flow realizing $T_1 \rightarrow I_1$, $T_2 \rightarrow I_3$, $T_2 \rightarrow I_5$ is feasible with $d_{\text{Wed}} = d_{\text{Thu}} = 1$.

Running time. First reject if $a > b$ (with $r \geq 1$), $ra > c$, $\sum_i d_i > c$ or $c > k$. Each is necessary, since no TA can hold between a and b hours when $a > b$, the TAs hold at least ra hours, the days need at least $\sum_i d_i$ hours, and at most k slots exist. Otherwise the auxiliary s^*-t^* flow of (7.50) has value $D = (c - \sum_i d_i) + \sum_i d_i = c \leq k$. The positive modified demands are $c - \sum_i d_i$ at t and d_i at D_i , while s and the T_x get $ra - c \leq 0$ and $-a$. The network has $m = O(rk)$ edges, so Ford-Fulkerson takes $O(mD) = O(rk^2)$ time by (7.5).

7.5 Exercises 7.29–7.35

Problem 7.29. Some of your friends have recently graduated and started a small company, which they are currently running out of their parents' garages in Santa Clara. They're in the process of porting all their software from an old system to a new, revved-up system; and they're facing the following problem.

They have a collection of n software applications, $\{1, 2, \dots, n\}$, running on their old system, and they'd like to port some of these to the new system. If they move application i to the new system, they expect a net (monetary) benefit of $b_i \geq 0$. The different software applications interact with one another: if applications i and j have extensive interaction, then the company will incur an expense if they move one of i or j to the new system but not both; let's denote this expense by $x_{ij} \geq 0$.

So, if the situation were really this simple, your friends would just port all n applications, achieving a total benefit of $\sum_i b_i$. Unfortunately, there's a problem. . . .

Due to small but fundamental incompatibilities between the two systems, there's no way to port application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay off to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems.

So this is the question they pose to you: Which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, 3, \dots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

Solution. Take a minimum cut in the segmentation network of Section 7.10: the source side is "new system", and application 1 is nailed to the sink side.

The objective is

$$\text{val}(S) = \sum_{i \in S} b_i - \sum_{\{i,j\}: |\{i,j\} \cap S| = 1} x_{ij}.$$

Let $B = \sum_{i \geq 2} b_i$. Build G on the nodes $\{s, t, 1, \dots, n\}$ with edges (s, i) of capacity b_i for $i \geq 2$, edges (i, j) and (j, i) each of capacity x_{ij} , and an edge $(1, t)$ of capacity $B + 1$. Compute a minimum s - t cut (A', B') and output $S = A' - \{s\}$.

Correctness. For any $S \subseteq \{2, \dots, n\}$, the cut $(\{s\} \cup S, \text{rest})$ is crossed by the edges (s, i) with $i \notin S$

and by exactly one of $(i, j), (j, i)$ for each pair split by S . So its capacity is

$$\begin{aligned} c(\{s\} \cup S, \text{rest}) &= \sum_{i \geq 2, i \notin S} b_i + \sum_{\text{split pairs}} x_{ij} \\ &= B - \text{val}(S). \end{aligned}$$

A cut with 1 on the source side contains $(1, t)$, so its capacity is at least $B+1 > B = c(\{s\}, V - \{s\})$, and such a cut is never minimum. Hence the minimum cuts are exactly the cuts $\{s\} \cup S$ with $S \subseteq \{2, \dots, n\}$ that minimize $B - \text{val}(S)$, that is, that maximize $\text{val}(S)$. This is the argument of (7.55), with the “background” benefit of every application set to 0.

Running time. One maximum flow and the cut from (7.11) on $n+2$ nodes, which is $O(n^3)$ with the Preflow-Push Algorithm by (7.33).

Problem 7.30. Consider a variation on the previous problem. In the new scenario, any application can potentially be moved, but now some of the benefits b_i for moving to the new system are in fact *negative*: if $b_i < 0$, then it is preferable (by an amount quantified in b_i) to keep i on the old system. Again, give a polynomial-time algorithm to find a set $S \subseteq \{1, 2, \dots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

Solution. Split each benefit into its positive and negative parts and charge a negative benefit on an edge to the sink. Then a minimum cut again maximizes the objective, as in Section 7.10.

Write $b_i^+ = \max(b_i, 0)$ and $b_i^- = \max(-b_i, 0)$, so $b_i = b_i^+ - b_i^-$, and let $P = \sum_i b_i^+$. Build G on $\{s, t, 1, \dots, n\}$ with edges (s, i) of capacity b_i^+ , edges (i, t) of capacity b_i^- , and edges (i, j) and (j, i) each of capacity x_{ij} . All capacities are nonnegative. Compute a minimum s - t cut (A', B') and output $S = A' - \{s\}$.

Correctness. Every $S \subseteq \{1, \dots, n\}$ gives the cut $(\{s\} \cup S, \text{rest})$, and every s - t cut arises this way. Its capacity is

$$\begin{aligned} c(\{s\} \cup S, \text{rest}) &= \sum_{i \notin S} b_i^+ + \sum_{i \in S} b_i^- + \sum_{\text{split pairs}} x_{ij} \\ &= P - \left(\sum_{i \in S} (b_i^+ - b_i^-) - \sum_{\text{split pairs}} x_{ij} \right) \\ &= P - \text{val}(S). \end{aligned}$$

Here $\text{val}(S)$ is the objective of Exercise 7.29. P does not depend on S , so a minimum cut gives a set S of maximum value.

Running time. One minimum-cut computation on $n+2$ nodes, which is $O(n^3)$ by (7.33).

Problem 7.31. There are n boxes; box i is a rectangular solid with side lengths (i_1, i_2, i_3) , and every side length lies strictly between $1/2$ and 1 meter. Box i *neests* inside box j if some permutation a, b, c of $\{1, 2, 3\}$ gives $i_a < j_1, i_b < j_2$ and $i_c < j_3$. A *nesting arrangement* is a sequence of operations, each putting a box i inside a box j in which i nests (whatever is already inside i goes along with it). Since every side exceeds half a meter and is below a meter, once a box i has been put inside j , nothing else can go into j . A box is *visible* if the arrangement never puts it inside another box. Give a polynomial-time algorithm that finds a nesting arrangement minimizing the number of visible boxes.

Example: for boxes $(.6, .6, .6)$, $(.75, .75, .75)$ and $(.9, .7, .7)$, the first box fits into either of the others, but the second and third boxes are both visible in every arrangement. So the minimum is two visible boxes, achieved for instance by putting the first box inside the second.

Solution. Build the bipartite graph G with sides $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_n\}$ and an edge (x_i, y_j) whenever box i nests in box j . The minimum number of visible boxes is $n - |M|$ for a maximum matching M of G , and an optimal arrangement can be read off from M .

Testing nesting. Write $a_1 \leq a_2 \leq a_3$ and $b_1 \leq b_2 \leq b_3$ for the sorted sides of boxes i and j . Then i nests

in j if and only if $a_k < b_k$ for $k = 1, 2, 3$. If the sorted order works, it is itself one of the permutations. Conversely, suppose some permutation works. Fix k and look at the k sides of j with values $b_1, \dots, b_k$. Each is matched to a different side of i that is strictly smaller, so i has k distinct sides all below b_k , and therefore $a_k < b_k$. Each pair is tested in $O(1)$ time, so G has $m \leq n^2$ edges and is built in $O(n^2)$ time.

Arrangements give matchings. In any arrangement, each box is put inside another box at most once (afterwards it sits inside that box). By the remark in the statement, each box directly receives at most one box. So the operations $(i \rightarrow j)$ form a matching M in G . The visible boxes are exactly those never moved, so

$$\# \text{visible} = n - \# \text{operations} = n - |M| \geq n - |M^*|,$$

where M^* is a maximum matching.

Matchings give arrangements. Let M be any matching, and let H be the directed graph on the boxes with an arc $i \rightarrow j$ for each $(x_i, y_j) \in M$. Every box has out-degree and in-degree at most 1 in H . Also, H is acyclic, since $i \rightarrow j$ forces

$$i_1 + i_2 + i_3 < j_1 + j_2 + j_3.$$

So H is a disjoint union of directed paths $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_r$. For each path, put v_1 into v_2 , then v_2 into v_3 , and so on up to v_r . Each step is legal: v_t nests in v_{t+1} , and v_{t+1} is still empty because v_t is the only box ever put into it. The visible boxes are the path ends v_r , and there are $n - |M|$ of them. So the minimum equals $n - |M^*|$, and the construction above with $M = M^*$ achieves it.

Algorithm.

```

sort the three sides of every box
build G: edge (x_i, y_j) iff sorted sides of i are
           coordinatewise strictly smaller than those of j
M = maximum matching of G (Ford-Fulkerson, Section 7.5)
for each path v1 -> v2 -> ... -> vr formed by the edges of M:
    put v1 into v2, then v2 into v3, ..., then v(r-1) into vr

```

Running time. Building G takes $O(n^2)$ time. By (7.37) and (7.38), Ford-Fulkerson finds a maximum matching in $O(mn) = O(n^3)$ time, and tracing the paths takes $O(n)$ time. The total is $O(n^3)$.

Problem 7.32. For a graph $G = (V, E)$ and a natural number k , define a relation $\xrightarrow{G,k}$ on pairs of vertices: for $x, y \in V$, write $x \xrightarrow{G,k} y$ when G contains k mutually edge-disjoint paths from x to y . Is $\xrightarrow{G,k}$ transitive for every G and every $k \geq 0$? That is, do $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$ always imply $x \xrightarrow{G,k} z$? Prove it or give a counterexample.

Solution. Yes, for directed and undirected G alike. By (7.45) (directed) and (7.48) (undirected), $s \xrightarrow{G,k} t$ holds exactly when deleting any set of fewer than k edges leaves t reachable from s .

For $k = 0$ there is nothing to prove. For $k \geq 1$ assume $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$, and take any $F \subseteq E$ with $|F| < k$. Then

$$\begin{aligned} x \xrightarrow{G,k} y, |F| < k &\Rightarrow y \text{ is reachable from } x \text{ in } G - F, \\ y \xrightarrow{G,k} z, |F| < k &\Rightarrow z \text{ is reachable from } y \text{ in } G - F. \end{aligned}$$

Joining the two walks, z is reachable from x in $G - F$. So no set of fewer than k edges separates x from z , and by (7.45) or (7.48) there are k edge-disjoint x - z paths, i.e. $x \xrightarrow{G,k} z$. ■

Problem 7.33. Let $G = (V, E)$ be a directed graph in which every node has in-degree equal to out-degree:

$$|\{(u, v) : (u, v) \in E\}| = |\{(v, w) : (v, w) \in E\}| \quad \text{for all } v \in V.$$

Let x, y be nodes of G and suppose there are k mutually edge-disjoint paths from x to y . Must there then be k mutually edge-disjoint paths from y to x ? Prove it or give a counterexample with explanation.

Solution. Yes: in a balanced graph every cut has as many edges leaving it as entering it, so every y - x cut is as large as some x - y cut.

Give every edge capacity 1. For $S \subseteq V$ write $e^{\text{out}}(S)$ and $e^{\text{in}}(S)$ for the number of edges leaving and entering S . An edge with both ends in S contributes +1 and -1 to the sum below, so summing the balance condition over S gives

$$0 = \sum_{v \in S} (\deg^{\text{out}}(v) - \deg^{\text{in}}(v)) = e^{\text{out}}(S) - e^{\text{in}}(S).$$

By (7.43) the maximum x - y flow has value at least k . By the Max-Flow Min-Cut Theorem (7.13), every x - y cut $(A, V - A)$ (so $x \in A, y \notin A$) then satisfies $e^{\text{out}}(A) \geq k$. Now take any y - x cut $(B, V - B)$ with $y \in B, x \notin B$, and put $A = V - B$. Then (A, B) is an x - y cut, and the edges leaving B are exactly the edges entering A , so

$$e^{\text{out}}(B) = e^{\text{in}}(A) = e^{\text{out}}(A) \geq k.$$

So every y - x cut has capacity at least k , and by (7.13) the maximum y - x flow has value at least k . By (7.43) there are k edge-disjoint paths from y to x . ■

Problem 7.34. Low-powered wireless devices are scattered over a disaster area, and some of them will fail. A device v that senses it is about to fail should send its state to another device, and it can only reach devices within d meters. For redundancy, each device v is to be assigned a *back-up set*: k other devices, each within d meters of v .

(a) You are given n devices, each with a position (x, y) , and parameters k, d, b . Design an algorithm that decides whether one can choose a back-up set for every device so that no device lies in the back-up sets of more than b other devices, and that outputs the back-up sets when they exist.

(b) Replace the sharp in-range/out-of-range rule by a decreasing power-decay function f : two devices at distance δ achieve signal strength $f(\delta)$. Given thresholds $p_1 \geq p_2 \geq \dots \geq p_k$, a back-up set for v whose members lie at distances $d_1 \leq d_2 \leq \dots \leq d_k$ from v is required to satisfy $f(d_j) \geq p_j$ for every j (one very strong link, one more at least moderately strong, and so on). Give an algorithm that decides whether back-up sets satisfying this condition can be chosen for all devices, still with no device in more than b back-up sets, and that outputs them when they exist.

Solution. (a) Reduce to a bipartite max-flow problem: each device sends k units of back-up demand to devices in range, and each device can absorb at most b .

Build G with source s , sink t , a node v (left copy) and a node v' (right copy) for every device. Add edges $s \rightarrow v$ of capacity k , edges $v \rightarrow u'$ of capacity 1 whenever $u \neq v$ and $\text{dist}(u, v) \leq d$, and edges $u' \rightarrow t$ of capacity b . Compute a maximum flow φ with Ford-Fulkerson. If its value is nk , output $B_v = \{u : \varphi(v \rightarrow u') = 1\}$; otherwise report that no choice exists.

Correctness. By (7.14) there is an integer maximum flow. Given an integer flow of value nk , every edge $s \rightarrow v$ is saturated. By conservation at v , exactly k unit edges $v \rightarrow u'$ carry flow, so $|B_v| = k$ and every member is a different device within d of v . By conservation at u' , the number of v with $u \in B_v$ equals $\varphi(u' \rightarrow t) \leq b$. Conversely, valid back-up sets give a flow of value nk : put 1 on $v \rightarrow u'$ for $u \in B_v$, k on each $s \rightarrow v$, and $|\{v : u \in B_v\}| \leq b$ on $u' \rightarrow t$. So the sets exist iff the maximum flow is nk .

Running time. G has $O(n)$ nodes and $m = O(n^2)$ edges, and building it takes $O(n^2)$ distance computations. The flow value is at most nk , so by (7.5) Ford-Fulkerson takes $O(m \cdot nk) = O(kn^3)$ time.

(b) Use the same network, but replace the single node v by a chain $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ whose capacities cap how many members may come from the weaker signal classes.

For $u \neq v$ put u in class $c_v(u) = \min\{j : f(\text{dist}(u, v)) \geq p_j\}$. Since the p_j are non-increasing, $f(\text{dist}(u, v)) \geq p_j$ iff $c_v(u) \leq j$. A device u with $f(\text{dist}(u, v)) < p_k$ gets no class and can never be used. If the hard range d is also kept, give no class to devices farther than d as well; nothing else changes. Let

$$S_j(v) = \{u : c_v(u) \leq j\}, \quad S_1(v) \subseteq S_2(v) \subseteq \dots \subseteq S_k(v).$$

Reformulation. A k -set B satisfies the condition iff $B \subseteq S_k(v)$ and $|B \setminus S_j(v)| \leq k - j$ for all j .

To see this, order B by distance as $u_1, \dots, u_k$. Since f is decreasing, $f(d_j) \geq p_j$ means $u_1, \dots, u_j \in S_j(v)$, so $|B \cap S_j(v)| \geq j$. Conversely, suppose $|B \cap S_j(v)| \geq j$, and let w be the farthest of j members of $B \cap S_j(v)$. Then $d_j \leq \text{dist}(w, v)$, so $f(d_j) \geq f(\text{dist}(w, v)) \geq p_j$. Finally, $|B \cap S_j(v)| \geq j$ iff $|B \setminus S_j(v)| \leq k - j$, because $|B| = k$ and $B \subseteq S_k(v)$.

Network. For each device v create chain nodes $v_1, \dots, v_k$, and for each device u a node u' . Add the edges

$$\begin{array}{ll} s \rightarrow v_1 & \text{capacity } k, \\ v_j \rightarrow v_{j+1} \ (1 \leq j < k) & \text{capacity } k - j, \\ v_{c_v(u)} \rightarrow u' \ (u \text{ classed for } v) & \text{capacity } 1, \\ u' \rightarrow t & \text{capacity } b. \end{array}$$

Compute an integer maximum flow φ (7.14). If its value is nk , output $B_v = \{u : v_{c_v(u)} \rightarrow u' \text{ carries flow}\}$; otherwise report infeasibility.

Correctness. Inside the gadget of v , the only way to reach v_{j+1} is along $v_j \rightarrow v_{j+1}$, and the edges leaving $v_{j+1}, \dots, v_k$ go exactly to the classes $> j$. So conservation gives

$$\varphi(v_j \rightarrow v_{j+1}) = |\{u \in B_v : c_v(u) > j\}| = |B_v \setminus S_j(v)|.$$

Take an integer flow of value nk . Each $s \rightarrow v_1$ is saturated, so $|B_v| = k$, with distinct members (unit edges) drawn from $S_k(v)$. The capacity $k - j$ then gives $|B_v \setminus S_j(v)| \leq k - j$, so B_v is valid by the reformulation, and as in (a) each u lies in at most b sets. Conversely, valid sets $\{B_v\}$ give a feasible flow of value nk : route one unit $s \rightarrow v_1 \rightarrow \dots \rightarrow v_{c_v(u)} \rightarrow u' \rightarrow t$ for each $u \in B_v$. By the display and the reformulation, chain edge j then carries $|B_v \setminus S_j(v)| \leq k - j$, and $u' \rightarrow t$ carries at most b . So the sets exist iff the maximum flow equals nk .

Running time. There are $O(nk)$ nodes and $m = O(nk + n^2) = O(n^2)$ edges, using $k \leq n - 1$ (if $k \geq n$ the answer is trivially no). The flow value is at most nk , so by (7.5) Ford-Fulkerson runs in $O(m \cdot nk) = O(kn^3)$ time.

Problem 7.35. Take the image segmentation setup of Section 7.10: n pixels, a set E of neighboring pairs, and parameters $\{a_i\}$, $\{b_i\}$, $\{p_{ij}\}$. Assume (1) every a_i , b_i , p_{ij} is a nonnegative integer in $\{0, 1, \dots, d\}$, and (2) every pixel has at most four neighbors (so each node of the resulting graph has at most four neighbor edges leaving it).

First an *initial segmentation* (A_0, B_0) maximizing $q(A_0, B_0)$ is computed. The user may then click a pixel v_1 to say it belongs in the foreground; the tool must then produce a segmentation (A_1, B_1) maximizing $q(A_1, B_1)$ subject to $v_1 \in A_1$ (not merely move v_1 across). More generally the user makes a sequence of clicks $v_1, v_2, \dots, v_t$, and after click v_i the tool must output (A_i, B_i) maximizing $q(A_i, B_i)$ subject to $v_1, \dots, v_i$ all lying in the foreground.

Give an algorithm whose initial segmentation costs a constant factor times one maximum-flow computation, and which handles each click in $O(dn)$ time. (Hint: Solved Exercise 1 of this chapter is a useful primitive. Forcing pixels into the background is symmetric and need not be treated.)

Solution. Force a clicked pixel v into the foreground by raising the capacity of the edge (s, v) to $a'_v = b_v + \sum_{j:(v,j) \in E} p_{vj} + 1$, and repair the current maximum flow with at most $5d + 1$ augmenting-path searches, as in Solved Exercise 1.

Build the network G' of Section 7.10 (edges (s, i) of capacity a_i , (i, t) of capacity b_i , and (i, j) , (j, i) of capacity p_{ij} for every neighboring pair). Compute an integer maximum flow f by Ford-Fulkerson (7.14), and output A_0 , the set of pixels reachable from s in G'_f (7.11). When pixel v is clicked, set

$$\Delta_v = \max\left(0, b_v + \sum_j p_{vj} + 1 - c(s, v)\right),$$

add Δ_v to $c(s, v)$, and then repeatedly search G'_f for an s - t path and augment along it, until a search fails. Output as A_i the pixels reachable from s in the final residual graph. (A pixel that was already

clicked has $\Delta_v = 0$.)

Write $q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{|A \cap \{i, j\}|=1} p_{ij}$, so $q(A, B) = \sum_i (a_i + b_i) - q'(A, B)$ and we must minimize q' under the constraints. Let H_i be the network after clicks $v_1, \dots, v_i$; only edges (s, v_k) have changed, and each now has capacity $c(s, v_k) > b_{v_k} + \sum_j p_{v_k j}$.

(i) In H_i , no minimum cut has some v_k on the t -side. If $v = v_k \in B$, moving v to the s -side changes the cut capacity by

$$\begin{aligned} b_v + \sum_{j \in B} p_{vj} - c(s, v) - \sum_{j \in A} p_{jv} \\ \leq b_v + \sum_j p_{vj} - c(s, v) < 0. \end{aligned}$$

(ii) For a cut with all $v_1, \dots, v_i$ on the s -side, no modified edge (s, v_k) crosses it, so its capacity in H_i equals its capacity in G' , which is $q'(A, B)$ by the computation in Section 7.10.

By (i) and (ii), the minimum cuts of H_i are exactly the constrained minimizers of q' , i.e. the constrained maximizers of q . The reachable set at termination is a minimum cut of H_i by (7.9), so (A_i, B_i) is correct. The repair is correct because the flow f from before the click stays feasible in H_i , and some minimum cut of H_{i-1} has capacity $\nu(f)$. That cut's capacity in H_i is at most $\nu(f) + \Delta_{v_i}$. Hence, exactly as in (7.68),

$$\nu(f) \leq \text{max-flow}(H_i) \leq \nu(f) + \Delta_{v_i}.$$

All capacities are integers, so each augmentation raises the flow value by at least 1. So after at most Δ_{v_i} augmentations a search fails, and the flow is then maximum by (7.9).

The initial phase is one maximum-flow computation plus an $O(m)$ search (7.11). For a click, the bounds $b_v \leq d$ and at most four neighbors with $p_{vj} \leq d$ give

$$\Delta_v \leq d + 4d + 1 = 5d + 1.$$

So there are at most $5d + 2$ searches. G' has $n + 2$ nodes and at most $2n + 4n = 6n$ edges, so each search and each augmentation in the residual graph (at most $12n$ edges) takes $O(n)$ time. The final reachability search is another $O(n)$. The total per click is

$$O((5d + 2)n) = O(dn)$$

(for $d \geq 1$; when $d = 0$ every segmentation is optimal).

7.6 Exercises 7.36–7.42

Problem 7.36. This is a multi-label version of the image segmentation problem of Section 7.10. Each pixel gets a label from $\{0, 1, \dots, M\}$, a rough estimate of its distance from the camera. Let $G = (V, E)$ be the graph whose nodes are the pixels and whose edges join neighbouring pixels. A *labeling* is a partition of V into sets $A_0, \dots, A_M$, where A_k holds the pixels labelled k . Assigning label k to pixel i costs $a_{i,k}$. If neighbours $(i, j) \in E$ get different labels, they pay a separation cost equal to p_{ij} times the difference of their labels. So the cost of a labeling is

$$q'(A_0, \dots, A_M) = \sum_{k=0}^M \sum_{i \in A_k} a_{i,k} + \sum_{k < \ell} \sum_{\substack{(i,j) \in E \\ i \in A_k, j \in A_\ell}} (\ell - k) p_{ij}.$$

We want a polynomial-time algorithm that finds a labeling of minimum cost, using the following partial flow network. There is a source s , a sink t , and for each pixel i there are nodes $v_{i,1}, \dots, v_{i,M}$. By convention $v_{i,0} = s$ and $v_{i,M+1} = t$ for every i . For $k = 0, \dots, M$ add an edge $(v_{i,k}, v_{i,k+1})$ of capacity $a_{i,k}$ and a reverse edge $(v_{i,k+1}, v_{i,k})$ of a very large capacity L . These nodes and edges form the *chain* of pixel i , a path $s \rightarrow v_{i,1} \rightarrow \dots \rightarrow v_{i,M} \rightarrow t$ with forward capacities $a_{i,0}, \dots, a_{i,M}$ and reverse capacities L (Figure 7.30 shows the case $M = 5$). The book notes that if L is large enough, no minimum cut (A, B) has an edge of capacity L leaving A (it asks how large L must

be). Then every chain has exactly one low-capacity edge leaving A . Complete the construction so that a minimum-cost labeling can be computed efficiently from a minimum s - t cut. Prove that the construction works, and show how to read the labeling off the cut.

Solution. For each $(i, j) \in E$ and each $k = 1, \dots, M$ add the edges $(v_{i,k}, v_{j,k})$ and $(v_{j,k}, v_{i,k})$, both of capacity p_{ij} , and take $L = 1 + \sum_i a_{i,0}$; the label of pixel i is the number of its chain nodes $v_{i,1}, \dots, v_{i,M}$ on the source side of a minimum cut. (We take $p_{ij} \geq 0$; if some $a_{i,k} < 0$, first add a constant to every $a_{i,k}$ of pixel i , which shifts every labeling's cost equally.)

Any $L > \sum_i a_{i,0} = c(\{s\}, V - \{s\})$ suffices: a cut with an L -edge leaving A has capacity at least L , so it is not minimum. If no L -edge leaves A , then $v_{i,k+1} \in A$ forces $v_{i,k} \in A$, so A meets each chain in a prefix

$$A \cap \{v_{i,0}, \dots, v_{i,M+1}\} = \{v_{i,0}, \dots, v_{i,k_i}\}, \quad 0 \leq k_i \leq M$$

($k_i \leq M$ as $t \notin A$). Conversely each labeling (k_i) gives the L -free cut $A = \{s\} \cup \{v_{i,m} : 1 \leq m \leq k_i\}$, so L -free cuts and labelings correspond bijectively. In such a cut the only chain edge of i leaving A is (v_{i,k_i}, v_{i,k_i+1}) , of capacity a_{i,k_i} ; and for $(i, j) \in E$ with $k_i < k_j$, the edge $(v_{j,m}, v_{i,m})$ leaves A exactly when $k_i < m \leq k_j$ while no edge $(v_{i,m}, v_{j,m})$ does (symmetrically for $k_i > k_j$; nothing is cut if $k_i = k_j$). Hence

$$c(A, B) = \sum_i a_{i,k_i} + \sum_{(i,j) \in E} |k_i - k_j| p_{ij} = q'(A_0, \dots, A_M),$$

with $A_k = \{i : k_i = k\}$. So compute a maximum flow, take the minimum cut (A, B) of nodes reachable from s in the residual graph by (7.11), and output $k_i = |\{m \geq 1 : v_{i,m} \in A\}|$: this cut is L -free, and every labeling is an L -free cut of equal capacity, so the output labeling minimizes q' .

The network has $N = nM + 2$ nodes ($n = |V|$) and $2n(M + 1) + 2M|E|$ edges, so by (7.33) the maximum-height Preflow-Push Algorithm runs in $O(N^3) = O(n^3 M^3)$ time, which dominates building the network and reading off the labels.

Problem 7.37. The usual minimum s - t cut problem requires every capacity to be nonnegative; with arbitrary signs the problem becomes much harder. Here we relax nonnegativity only slightly and ask for a polynomial-time algorithm anyway.

Let $G = (V, E)$ be a directed graph with source s , sink t and capacities $\{c_e\}$, where $c_e \geq 0$ for every edge e having neither s nor t as an endpoint; edges with at least one end in $\{s, t\}$ may have negative capacity. The value of an s - t cut (A, B) is still defined as the sum of c_e over all edges e whose tail lies in A and whose head lies in B . Give a polynomial-time algorithm that finds an s - t cut of minimum value in such a graph.

Solution. For each node v , subtract $\min(c_{sv}, c_{vt})$ from both of its terminal edges; this shifts every cut value by the same constant and makes all capacities nonnegative, so an ordinary minimum cut solves the problem.

Merge parallel edges by adding capacities, and for $v \in V - \{s, t\}$ write $\sigma_v = c_{(s,v)}$, $\tau_v = c_{(v,t)}$ (zero if the edge is absent) and $m_v = \min(\sigma_v, \tau_v)$. Build G' on the same nodes:

- every edge with neither end in $\{s, t\}$ is kept with its capacity $c_e \geq 0$;
- (s, v) gets capacity $\sigma_v - m_v \geq 0$ and (v, t) gets capacity $\tau_v - m_v \geq 0$;
- edges into s , edges out of t , and the edge (s, t) are deleted.

Compute a minimum s - t cut (A^*, B^*) of G' with a max-flow algorithm and output it.

Fix any s - t cut (A, B) , so $s \in A$, $t \in B$. An edge entering s or leaving t never goes from A to B , and (s, t) always does. Edge (s, v) is cut iff $v \in B$, and (v, t) is cut iff $v \in A$; exactly one of these holds, so

$$\sigma_v[v \in B] + \tau_v[v \in A] = m_v + (\sigma_v - m_v)[v \in B] + (\tau_v - m_v)[v \in A].$$

Summing over v and adding the unchanged interior edges,

$$c_G(A, B) = c_{G'}(A, B) + K, \quad K = c_{(s,t)} + \sum_{v \neq s,t} m_v,$$

where K does not depend on the cut. Both graphs have the same set of s - t cuts, so a cut minimizes $c_{G'}$ iff it minimizes c_G , and (A^*, B^*) is a minimum cut of G , with value $c_{G'}(A^*, B^*) + K$. Since G' has only nonnegative capacities, it is an ordinary flow network, so by (7.13) and (7.11) a minimum cut is read off the residual graph of a maximum flow in $O(m)$ time. The capacities may be arbitrary reals, so use the Preflow-Push Algorithm, which by (7.33) runs in $O(n^2m)$ time independent of the capacity values. Building G' takes $O(m + n)$, so the total running time is $O(n^2m)$.

Problem 7.38. A database table T has a set R of m rows (employees) and a set C of n columns (attributes). For example, the columns might be name, phone number, start date, manager's name, with five employees:

name	phone number	start date	manager's name
Alanis	3-4563	6/13/95	Chelsea
Chelsea	3-2341	1/20/93	Lou
Elrond	3-2345	12/19/01	Chelsea
Hal	3-9000	1/12/97	Chelsea
Raj	3-3453	7/1/96	Chelsea

For a subset $S \subseteq C$, the *projection* $T[S]$ is the table keeping only the columns in S . For a permutation p of the columns, the *permutation* T_p is T with its columns reordered by p .

You will work repeatedly with k column subsets $S_1, \dots, S_k$. Instead of storing all k projections, note that the database handles a subset efficiently if its members, in some order, form a prefix (the leftmost $|S_i|$ columns) of a stored table; the subsets carry no order of their own. Given a parameter $\ell < k$, decide whether there are ℓ permutations $p_1, \dots, p_\ell$ of the columns such that every S_i ($i = 1, \dots, k$) forms a prefix of at least one of $T_{p_1}, \dots, T_{p_\ell}$ (a *valid solution*). Give a polynomial-time algorithm that solves this and, when a valid solution exists, outputs such ℓ permutations.

Example: with the table above, $S_1 = \{\text{name, phone number}\}$, $S_2 = \{\text{name, start date}\}$, $S_3 = \{\text{name, manager's name, start date}\}$ and $\ell = 2$, the permutations $p_1 = (\text{name, phone number, start date, manager's name})$ and $p_2 = (\text{name, start date, manager's name, phone number})$ form a valid solution: S_1 is a prefix of T_{p_1} , and S_2, S_3 are prefixes of T_{p_2} .

Solution. Compute a minimum partition of the distinct subsets into chains under $\subsetneq$ by a bipartite matching (minimum chain cover = $d - \nu$); a valid solution exists iff this minimum is at most ℓ , and each chain becomes one permutation.

Let $\mathcal{D} = \{A_1, \dots, A_d\}$ be the distinct sets among $S_1, \dots, S_k$ ($d \leq k$). Build the bipartite graph B with a left node u_A and a right node $v_{A'}$ for each $A \in \mathcal{D}$, and an edge $(u_A, v_{A'})$ whenever $A \subsetneq A'$.

```

compute a maximum matching M of B          (Section 7.5)
for each matched edge (u_A, v_A'): set next(A) = A'
for each A in D with no matched edge into v_A:    // chain starts
    p = empty list; prev = empty set; X = A
    loop: append the columns of X - prev to p (any order)
        prev = X; if next(X) undefined, stop; X = next(X)
    append the columns of C - prev to p; output p
if the number of permutations output is <= ell:
    pad with arbitrary permutations to exactly ell; return them
else: report that no valid solution exists

```

Valid solutions are chain covers. The prefixes of one permutation are nested, so the sets S_i served by a single p_j form a chain under $\subseteq$; a valid solution with ℓ permutations therefore partitions $\mathcal{D}$ into at most ℓ chains. Conversely, for a chain $X_1 \subsetneq X_2 \subsetneq \dots \subsetneq X_r$, the permutation listing X_1 , then $X_2 \setminus X_1$, ..., then $X_r \setminus X_{r-1}$, then $C \setminus X_r$ has each X_j as its first $|X_j|$ columns. Hence a valid solution exists iff the minimum number c^* of chains partitioning $\mathcal{D}$ satisfies $c^* \leq \ell$, and the loop above turns any chain partition into that many valid permutations.

Chain partitions are matchings. (i) Given a matching M , every A has at most one successor $\text{next}(A)$ and at most one predecessor; since $A \subsetneq \text{next}(A)$ strictly increases $|A|$, following next never cycles, so the pointers split $\mathcal{D}$ into vertex-disjoint paths, each a chain. A forest of paths on d nodes with $|M|$ edges has $d - |M|$ components, so $c^* \leq d - |M|$. (ii) Given a partition into c chains, linking each set to

the next larger set in its chain uses each u_A and each v_A at most once, giving a matching of size $d - c$. Therefore

$$c^* = d - \nu(B),$$

and the matching found yields exactly c^* permutations, which (padded) is a valid solution whenever $c^* \leq \ell$, while for $c^* > \ell$ none exists.

Running time. Deduplicating and testing all $O(k^2)$ pairs for strict inclusion costs $O(k^2n)$ (sorted column lists or bit vectors); B has $2d \leq 2k$ nodes and $O(k^2)$ edges, so the Ford-Fulkerson bipartite matching of Section 7.5 runs in $O(k \cdot k^2) = O(k^3)$; writing the permutations costs $O(kn)$. Total $O(k^2n + k^3)$, polynomial.

Problem 7.39. An environmental statistics firm publishes population tables, with every entry measured in millions. A typical table (rows are population groups, columns are countries) is

Country	A	B	C	Total
grown-up men	11.998	9.083	2.919	24.000
grown-up women	12.983	10.872	3.145	27.000
children	1.019	2.045	0.936	4.000
Total	26.000	22.000	7.000	55.000

Assume all row sums and column sums are integers. The *Census Rounding Problem* asks to replace every entry by an integer, each fractional entry being rounded either up or down, so that no row sum and no column sum changes. For the table above one valid rounding is

Country	A	B	C	Total
grown-up men	11	10	3	24
grown-up women	13	10	4	27
children	2	2	0	4
Total	26	22	7	55

(a) First take the special case in which every entry lies between 0 and 1: each entry must be rounded to 0 or 1 while preserving all row and column sums. Use a flow computation to decide whether such a rounding exists.

(b) Now take the general problem: row and column sums are integers and each entry α must be rounded to $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to decide whether such a rounding exists.

(c) Prove that the roundings sought in (a) and (b) always exist.

Solution. A rounding exists iff the maximum flow in the bipartite network rows $\rightarrow$ columns (a unit edge per fractional entry, reduced row and column sums as source and sink capacities) saturates the source. Here $a = (a_{ij})$ is $m \times n$ with integer row sums r_i and column sums c_j ; an entry is *fractional* if $a_{ij} \notin \mathbb{Z}$ (integer entries are forced).

(a) Let r'_i (resp. c'_j) be r_i (resp. c_j) minus the number of entries equal to 1 in row i (resp. column j). Build G with source s , sink t , a node u_i per row and v_j per column, and edges

$$\begin{aligned} (s, u_i) &\text{ of capacity } r'_i, & (v_j, t) &\text{ of capacity } c'_j, \\ (u_i, v_j) &\text{ of capacity } 1 \text{ for each fractional entry } 0 < a_{ij} < 1. \end{aligned}$$

Let $D = \sum_i r'_i$; since $\sum_i r_i = \sum_j c_j$ and both sides subtract the total number of 1 entries, also $D = \sum_j c'_j$. Compute a maximum flow by Ford-Fulkerson; **the rounding exists iff its value is D** , and then $x_{ij} = f(u_i, v_j)$ for fractional entries (with 0 and 1 entries kept) is a rounding.

($\Rightarrow$) Given a rounding x , set $f(u_i, v_j) = x_{ij}$ on fractional cells, $f(s, u_i) = r'_i$, $f(v_j, t) = c'_j$. Conservation at u_i holds because the fractional cells of row i round to a total of $r_i - \#\{j : a_{ij} = 1\} = r'_i$ (entries equal to 0 stay 0); likewise at v_j . Capacities hold, and the value is D .

($\Leftarrow$) If the maximum flow value is D , by (7.14) there is an integer flow f of value D ; it saturates every (s, u_i) , and since $\sum_j c'_j = D$ it saturates every (v_j, t) . Integrality and capacity 1 give $f(u_i, v_j) \in \{0, 1\}$, and conservation at u_i and v_j says that x has row sums $r'_i + \#\{j : a_{ij} = 1\} = r_i$ and column sums c_j . The source capacities total $D \leq mn$, so Ford-Fulkerson runs in $O(|E|D) = O((mn)^2)$ time by (7.5), with $|E| \leq mn + m + n$.

(b) Reduce to (a). Write $a_{ij} = \lfloor a_{ij} \rfloor + b_{ij}$ with $b_{ij} \in [0, 1)$. Rounding a_{ij} to $\lfloor a_{ij} \rfloor$ or $\lceil a_{ij} \rceil$ is the same as rounding b_{ij} to 0 or 1 (if $b_{ij} = 0$ the entry is forced), and the row and column sums of b ,

$$\rho_i = r_i - \sum_j \lfloor a_{ij} \rfloor, \quad \gamma_j = c_j - \sum_i \lfloor a_{ij} \rfloor,$$

are integers. So x is a census rounding of a iff $x - \lfloor a \rfloor$ is a 0/1 rounding of b preserving the sums ρ_i, γ_j . Run (a) on b : the network has (s, u_i) of capacity $\rho_i \geq 0$, (v_j, t) of capacity $\gamma_j \geq 0$, and a unit edge (u_i, v_j) for each fractional a_{ij} ; the rounding exists iff the maximum flow equals $D = \sum_i \rho_i$, and then $x_{ij} = \lfloor a_{ij} \rfloor + f(u_i, v_j)$. Again $D = \sum_{ij} b_{ij}$ is at most the number of fractional entries, so the time is $O((mn)^2)$.

(c) The fractional table itself is a maximum flow. In the network of (b) (which is the network of (a) when all entries are in $[0, 1]$) define

$$f(s, u_i) = \rho_i, \quad f(u_i, v_j) = b_{ij}, \quad f(v_j, t) = \gamma_j.$$

Capacities hold since $0 \leq b_{ij} < 1$, and conservation holds since $\sum_j b_{ij} = \rho_i$ and $\sum_i b_{ij} = \gamma_j$ by definition. Its value is D , which is the capacity of the cut $(\{s\}, V - \{s\})$, so by (7.8) f is a maximum flow and the maximum flow value is D . All capacities are integers, so by (7.14) there is also an integer flow of value D , and by the ($\Leftarrow$) direction of (a) it yields a valid rounding. Hence the rounding always exists, in both (a) and (b), and the algorithm of (b) always finds one.

Problem 7.40. This exercise asks how robust a minimum spanning tree is. You are given a connected graph $G = (V, E)$ with a cost c_e on each edge e . The costs were measured experimentally and may contain errors, so the minimum spanning tree computed for G might not be the “real” one.

You are also given error parameters $\varepsilon > 0$ and $k > 0$ and a particular edge $e' = (u, v)$. The goal is to certify the following property.

(*) Suppose every edge cost is changed by at most ε (up or down), and then the costs of k edges other than e' are changed again, to completely arbitrary values. Even then, e' belongs to no minimum spanning tree of G .

Give a polynomial-time algorithm that takes G, e', ε and k and decides whether (*) holds for e' .

Solution. Let H be the graph on V whose edges are the edges $f \neq e'$ with $c_f < c_{e'} - 2\varepsilon$. Then (*) holds if and only if H contains at least $k+1$ edge-disjoint u - v paths. We test this by running Ford-Fulkerson on H with unit capacities (each undirected edge becomes two opposite arcs of capacity 1, as in Section 7.6), and we stop as soon as the flow value reaches $k+1$.

Lemma. For any costs c' , the edge e' lies in some minimum spanning tree if and only if there is no u - v path made only of edges $f \neq e'$ with $c'_f < c'_{e'}$.

(i) Suppose such a path P exists, and let T be a spanning tree that contains e' . Deleting e' from T splits V into two sides, one containing u and the other containing v . Some edge f of P crosses between them, and $T - e' + f$ is a spanning tree that is strictly cheaper than T . So T is not minimum.

(ii) Suppose no such path exists. Let S be the set of nodes that u can reach using edges cheaper than $c'_{e'}$. Then $v \notin S$, and every edge that leaves S costs at least $c'_{e'}$. Take any minimum spanning tree T that does not contain e' . The path in T from u to v leaves S through some edge $f \neq e'$, and $c'_f \geq c'_{e'}$. So $T - f + e'$ is a spanning tree that costs no more than T , which makes it a minimum spanning tree containing e' .

Reduction. An adversary picks a perturbation c' : it chooses a set $F \not\ni e'$ with $|F| \leq k$, sets the costs of F freely, and moves every other cost by at most ε . By the lemma, e' fails to be in any minimum spanning tree exactly when the cheap edges connect u to v .

Every edge $f \in H \setminus F$ stays cheaper than e' under every such c' :

$$c'_f \leq c_f + \varepsilon < c_{e'} - \varepsilon \leq c'_{e'}.$$

So if $H - F$ still has a u - v path, then e' is in no minimum spanning tree.

The adversary has a best response. It sets $c'_{e'} = c_{e'} - \varepsilon$, sets $c'_f = c_f + \varepsilon$ for every $f \notin F$, and makes

every edge of F more expensive than e' . Under this choice, the edges cheaper than e' are exactly the edges of $H - F$.

Therefore (*) fails if and only if some set F of at most k edges disconnects u from v in H . By the undirected Menger theorem (7.48), this happens if and only if H has at most k edge-disjoint u - v paths. By (7.47), that count is the maximum flow value computed above.

Running time. Building H takes $O(m)$ time, and each augmentation takes $O(m)$ time (proof of (7.5)); there are at most $k + 1$ augmentations, and at most $n - 1$ since the flow never exceeds $\deg(u)$, so the total is

$$O(m \cdot \min(k + 1, n)).$$

Problem 7.41. Suppose you're managing a collection of processors and must schedule a sequence of jobs over time.

The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be preempted and resumed from where it left off (possibly after a delay) on another processor.

Moreover, the collection of processors is not entirely static either: You have an overall pool of k possible processors; but for each processor i , there is an interval of time $[t_i, t'_i]$ during which it is available; it is unavailable at all other times.

Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time algorithm that either produces a schedule completing all jobs by their deadlines or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2 . J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2 . P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done.

- At time 0, we start job J_1 on processor P_1 .
- At time 1, we preempt J_1 to start J_2 on P_1 .
- At time 2, we resume J_1 on P_2 . (J_2 continues processing on P_1 .)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1 .

Notice that there is no solution that does not involve preemption and moving of jobs.

Solution. Cut time at every a_j, d_j, t_i, t'_i into elementary intervals, and route each job's ℓ_j units of work through a flow network into the intervals it may use; a flow of value $\sum_j \ell_j$ is then turned into a schedule interval by interval with a wrap-around rule.

Let $\tau_0 < \tau_1 < \dots < \tau_m$ be the distinct values among all a_j, d_j, t_i, t'_i ($m \leq 2n + 2k$), let $I_q = [\tau_{q-1}, \tau_q]$ with length $L_q = \tau_q - \tau_{q-1}$, and let c_q be the number of processors i with $t_i \leq \tau_{q-1}$ and $\tau_q \leq t'_i$. Because every endpoint is a breakpoint, each processor is available either throughout I_q or nowhere in its interior, and job j may run in I_q iff $a_j \leq \tau_{q-1}$ and $\tau_q \leq d_j$. Build G : a source s , a node J_j per job, a node I_q per interval, a sink t , with edges

$$\begin{aligned} (s, J_j) &\text{ of capacity } \ell_j, & (I_q, t) &\text{ of capacity } c_q L_q, \\ (J_j, I_q) &\text{ of capacity } L_q & \text{ whenever } [\tau_{q-1}, \tau_q] &\subseteq [a_j, d_j]. \end{aligned}$$

Compute a maximum flow f in G . If $\nu(f) < \sum_j \ell_j$, report that no schedule exists. Otherwise, for each q , take the c_q processors available on I_q , write the amounts $x_{jq} = f(J_j, I_q)$ one after another along a line of length $c_q L_q$, cut it into c_q consecutive pieces of length L_q , and run piece p on the p -th available processor during I_q (McNaughton's wrap-around rule).

Correctness. (i) A schedule gives a flow of value $\sum_j \ell_j$: let x_{jq} be the time job j is processed during I_q . Then $x_{jq} \leq L_q$ because j runs on at most one processor at a time; $x_{jq} = 0$ unless $I_q \subseteq [a_j, d_j]$;

$\sum_j x_{jq} \leq c_q L_q$ because at most c_q processors are working at any instant of I_q ; and $\sum_q x_{jq} = \ell_j$. So $f(s, J_j) = \ell_j$, $f(J_j, I_q) = x_{jq}$, $f(I_q, t) = \sum_j x_{jq}$ is a flow of that value.

(ii) A flow of value $\sum_j \ell_j$ gives a schedule: since $\sum_j x_{jq} \leq c_q L_q$, the wrap-around fills at most c_q processors, each only inside I_q , where it is available, and j gets exactly x_{jq} units in $I_q \subseteq [a_j, d_j]$, so ℓ_j units in total. A job's segment is split over at most two consecutive processors, say as $[\tau_{q-1} + u, \tau_q]$ on one and $[\tau_{q-1}, \tau_{q-1} + u']$ on the next with $(L_q - u) + u' = x_{jq} \leq L_q$; then $u' \leq u$, so the two pieces do not overlap in time and j is never on two processors at once.

Hence a schedule exists iff the maximum flow has value $\sum_j \ell_j$ (it can never exceed this, the capacity of the cut $(\{s\}, V - \{s\})$).

Running time. G has $O(n + k)$ nodes and $O(n(n + k))$ edges, and all capacities are integers, so the Preflow-Push Algorithm finds f in $O((n + k)^3)$ time by (7.33); building G and the wrap-around take $O(n(n + k) + k(n + k))$ more.

Problem 7.42. Give a polynomial-time algorithm for the following minimization analogue of the Maximum-Flow Problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $\ell(v, w)$ for each edge $(v, w) \in E$. We define a flow f , and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow — that is, they require that $f(v, w) \geq \ell(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

(a) Give a polynomial-time algorithm that finds a feasible flow of minimum possible value.

(b) Prove an analogue of the Max-Flow Min-Cut Theorem for this problem (i.e., does min-flow = max-cut?).

Solution. Find any feasible flow f_0 , then push as much flow as possible *backwards*, from t to s , through the network of allowed decreases and increases; the minimum equals the maximum of $\ell(A, B)$ over cuts that no edge enters. As in Section 7.1 we assume no edge enters s and none leaves t (otherwise a t - s path lets the value go to $-\infty$).

(a) *Feasible flow.* For each $e = (u, v)$ with $\ell(e) > 0$: if G has a v - u path, add $\ell(e)$ units around the cycle it closes with e ; else if G has an s - u path and a v - t path, add $\ell(e)$ units along the walk they form with e ; else declare the instance infeasible. Adding a constant along a closed walk or an s - t walk keeps conservation and only increases edge values, so the sum f_0 is feasible. Conversely let f be feasible with $f(e) > 0$ and no cycle through e ; let W be the set of nodes reachable from v by edges of positive flow, so $u \notin W$, $s \notin W$, and no positive-flow edge leaves W . If $t \notin W$, summing conservation over W gives

$$0 = f^{\text{in}}(W) - f^{\text{out}}(W) = f^{\text{in}}(W) \geq f(e) > 0,$$

a contradiction; so v reaches t , and symmetrically s reaches u . Hence the test is exact.

Minimization. Let $C = \nu(f_0) + 1$ and build H on V : for each $e = (u, v) \in E$ put an arc (u, v) of capacity C (room to increase) and an arc (v, u) of capacity $f_0(e) - \ell(e)$ (room to decrease). Compute a maximum t - s flow g in H and return

$$f(e) = f_0(e) + g(u, v) - g(v, u) \quad (e = (u, v) \in E).$$

Then $f(e) \geq f_0(e) - (f_0(e) - \ell(e)) = \ell(e)$, f is conserved at every node but s, t (as f_0 and g are), and $\nu(f) = \nu(f_0) - \nu(g)$; since the only H -arcs entering s are the decrease arcs of edges (s, w) , also $\nu(g) \leq \nu(f_0)$.

(b) For an s - t cut (A, B) with no edge from B to A , let $\ell(A, B) = \sum_{e \text{ out of } A} \ell(e)$ (such cuts exist, e.g. $A = V - \{t\}$). Claim: if a feasible flow exists, the minimum flow value equals the maximum of $\ell(A, B)$ over these cuts. For every feasible f' and such a cut, (7.6) gives

$$\nu(f') = f'^{\text{out}}(A) - f'^{\text{in}}(A) = f'^{\text{out}}(A) \geq \ell(A, B).$$

For the f of (a), let B be the set of nodes reachable from t in the residual graph H_g and $A = V - B$; $s \in A$ since g is maximum, and by (7.9) (applied to H with source t) the H -cut (B, A) has capacity

$\nu(g) \leq \nu(f_0) < C$, so no C -arc goes from B to A , i.e. no edge of G enters A . For $e = (u, v)$ with $u \in A$, $v \in B$, the residual capacity from v to u in H_g is

$$(f_0(e) - \ell(e) - g(v, u)) + g(u, v) = f(e) - \ell(e),$$

which is 0 since u is not reachable from t . Therefore $\nu(f) = f^{\text{out}}(A) - 0 = \ell(A, B)$. So f attains the lower bound, which proves both that f is a minimum feasible flow and that min-flow = max-cut.

Running time. The feasibility test is $O(m(m+n))$ by search, f_0 is built in $O(mn)$, and g comes from one maximum flow on n nodes and $2m$ arcs, $O(n^3)$ by the Preflow-Push Algorithm (7.33).

7.7 Exercises 7.43–7.49

Problem 7.43. You are trying to solve a circulation problem, but it is not feasible. The problem has demands, but no capacity limits on the edges. More formally, there is a graph $G = (V, E)$, and demands d_v for each node v (satisfying $\sum_{v \in V} d_v = 0$), and the problem is to decide if there is a flow f such that $f(e) \geq 0$ and $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ for all nodes $v \in V$. Note that this problem can be solved via the circulation algorithm from Section 7.7 by setting $c_e = +\infty$ for all edges $e \in E$. (Alternately, it is enough to set c_e to be an extremely large number for each edge — say, larger than the total of all positive demands d_v in the graph.)

You want to fix up the graph to make the problem feasible, so it would be very useful to know why the problem is not feasible as it stands now. On a closer look, you see that there is a subset U of nodes such that there is no edge into U , and yet $\sum_{v \in U} d_v > 0$. You quickly realize that the existence of such a set immediately implies that the flow cannot exist: The set U has a positive total demand, and so needs incoming flow, and yet U has no edges into it. In trying to evaluate how far the problem is from being solvable, you wonder how big the demand of a set with no incoming edges can be.

Give a polynomial-time algorithm to find a subset $S \subset V$ of nodes such that there is no edge into S and for which $\sum_{v \in S} d_v$ is as large as possible subject to this condition.

Solution. This is the Project Selection Problem of Section 7.11 with profits $p_v = d_v$ and the edges of G reversed as precedence constraints. “No edge into S ” says exactly

$$v \in S, (u, v) \in E \implies u \in S,$$

which is the precedence constraint “ v requires u ”. So build the graph G' of Section 7.11: a source s and a sink t ; an edge (s, v) of capacity d_v for each v with $d_v > 0$; an edge (v, t) of capacity $-d_v$ for each v with $d_v < 0$; and for each $(u, v) \in E$ an edge (v, u) of capacity $C + 1$, where $C = \sum_{v: d_v > 0} d_v$. Compute a minimum cut (A', B') of G' and output $S = A' - \{s\}$ if $c(A', B') < C$, and $S = \emptyset$ otherwise. By (7.56) and (7.57), the cuts of capacity at most C are exactly the cuts $(S \cup \{s\}, (V - S) \cup \{t\})$ with S having no entering edge, and their capacity is $C - \sum_{v \in S} d_v$. So by (7.58) the minimum cut gives a feasible S of maximum total demand, namely $C - c(A', B')$; when this is positive, $S \neq V$ because $\sum_{v \in V} d_v = 0$.

Running time. G' has $n + 2$ nodes and at most $m + n$ edges, so one run of the Preflow-Push Algorithm finds the minimum cut in $O(n^3)$ time (7.33), and (7.11) reads off A' in $O(m + n)$ more.

Problem 7.44. Suppose we are given a directed network $G = (V, E)$ with a root node r and a set of terminals $T \subseteq V$. We’d like to disconnect many terminals from r , while cutting relatively few edges. We make this trade-off precise as follows. For a set of edges $F \subseteq E$, let $q(F)$ denote the number of nodes $v \in T$ such that there is no r - v path in the subgraph $(V, E - F)$. Give a polynomial-time algorithm to find a set F of edges that maximizes the quantity $q(F) - |F|$. (Note that setting F equal to the empty set is an option.)

Solution. Add a new sink t with an edge (v, t) for every $v \in T$, give every edge capacity 1, compute a minimum r - t cut (A, B) , and output $F = \{(u, v) \in E : u \in A, v \in B\}$; the optimum is $|T| - c(A, B)$. For any cut (A, B) of this network G' (with $r \in A, t \in B$), the edges leaving A are the edges of E from

A to B and the edges (v, t) with $v \in T \cap A$, so

$$c(A, B) = |E(A, B)| + |T \cap A|.$$

(i) For every $F \subseteq E$, let A be the set of nodes reachable from r in $(V, E - F)$ and $B = (V - A) \cup \{t\}$. Every edge of E leaving A lies in F , and the disconnected terminals are exactly $T - A$, so

$$c(A, B) \leq |F| + |T| - q(F), \quad \text{i.e.} \quad q(F) - |F| \leq |T| - c(A, B).$$

(ii) For the minimum cut (A, B) and $F = E(A, B)$, every r - v path with $v \in B$ leaves A along an edge of F , so every terminal in B is disconnected in $(V, E - F)$ and

$$q(F) - |F| \geq |T \cap B| - |E(A, B)| = |T| - c(A, B).$$

Together, (i) applied to an optimal F^* and (ii) give $q(F) - |F| \geq |T| - c_{\min} \geq q(F^*) - |F^*|$, so F is optimal.

Running time. The cut $(\{r\}, (V - \{r\}) \cup \{t\})$ has capacity at most n , so the Ford-Fulkerson Algorithm performs at most n augmentations of $O(m + n)$ each (7.5), and (7.11) gives the cut in $O(m + n)$ more: $O(n(m + n))$ overall.

Problem 7.45. Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i , we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j , we have the total value e_{ij} of all exports from i to j ; this number is always nonnegative. We say that a subset S of the countries is *free-standing* if the sum of the budget surpluses of the countries in S , minus the total value of all exports from countries in S to countries not in S , is nonnegative. Give a polynomial-time algorithm that takes this data for a set of n countries and decides whether it contains a nonempty free-standing subset that is not equal to the full set.

Solution. Write $\phi(S) = \sum_{i \in S} s_i - \sum_{i \in S, j \notin S} e_{ij}$; we maximize ϕ by a minimum cut over sets forced to contain one country and exclude another, for the $2(n - 1)$ pairs that involve country 1, and answer yes iff some maximum is ≥ 0 .

Let $P = \sum_{i: s_i > 0} s_i$. Build G on the countries plus a source σ and a sink τ , with an edge (i, j) of capacity e_{ij} for each pair, an edge (σ, i) of capacity s_i for each $s_i > 0$, and an edge (i, τ) of capacity $-s_i$ for each $s_i < 0$. For the cut $(S \cup \{\sigma\}, (V - S) \cup \{\tau\})$,

$$\begin{aligned} c &= \sum_{i \in S, j \notin S} e_{ij} + \sum_{i \in S, s_i < 0} (-s_i) + \sum_{i \notin S, s_i > 0} s_i \\ &= \sum_{i \in S, j \notin S} e_{ij} - \sum_{i \in S} s_i + P = P - \phi(S), \end{aligned}$$

and every σ - τ cut has this form. For a pair (a, b) of distinct countries let G_{ab} be G plus edges (σ, a) and (b, τ) of capacity $M = P + \sum_{i, j} e_{ij} + \sum_i |s_i| + 1$. The cut with $S = \{a\}$ has capacity less than M , so a minimum cut of G_{ab} uses neither new edge, i.e. it has $a \in S$, $b \notin S$, and among such S it minimizes $P - \phi(S)$. Hence

$$\max_{S: a \in S, b \notin S} \phi(S) \geq 0 \iff c_{\min}(G_{ab}) \leq P.$$

A set is nonempty and proper iff it contains some a and misses some b ; and then either $1 \in S$ (take $a = 1$ and $b \notin S$) or $1 \notin S$ (take $b = 1$ and $a \in S$). So a nonempty proper free-standing set exists iff $c_{\min}(G_{1b}) \leq P$ for some $b \neq 1$ or $c_{\min}(G_{a1}) \leq P$ for some $a \neq 1$; the source side of that cut, minus σ , is such a set.

Running time. $2(n - 1)$ minimum cuts in graphs with $n + 2$ nodes and $O(n^2)$ edges, each $O(n^3)$ by the Preflow-Push Algorithm (7.33), for $O(n^4)$ in total.

Problem 7.46. Let $G = (V, E)$ be an undirected graph (people and friendships), with $n = |V|$ and $m = |E|$. For $S \subseteq V$ let $e(S)$ be the number of edges with both ends in S , and define the *cohesiveness* of a nonempty S to be $e(S)/|S|$. We want a set of maximum cohesiveness.

(a) Give a polynomial-time algorithm that, given a rational number α , decides whether some set S has cohesiveness at least α .

(b) Give a polynomial-time algorithm that finds a set S of maximum cohesiveness.

Solution. (a) Maximize $e(S) - \beta|S|$ as a Project Selection instance (Section 7.11) with β slightly below α , and answer yes iff the optimum is positive.

Write $\alpha = p/q$ with $q \geq 1$ and put $\beta = \alpha - \frac{1}{2qn}$. The projects are $E \cup V$: each edge e has profit $p_e = 1$, each node v has profit $p_v = -\beta$, and edge uv requires both u and v . For a feasible set A with $S = A \cap V$ we have $A \cap E \subseteq E(S)$, so

$$\text{profit}(A) = |A \cap E| - \beta|S| \leq e(S) - \beta|S|,$$

and equality holds for $A = S \cup E(S)$. So the optimum equals $\Phi(\beta) = \max_{S \subseteq V} (e(S) - \beta|S|)$, and if A is optimal then $S = A \cap V$ attains $\Phi(\beta)$. By (7.58), the source side A' of a minimum cut in the Section 7.11 network gives an optimal $A = A' - \{s\}$. Multiplying all profits by $2qn$ makes them integers ($2qn$ for an edge, $1 - 2pn$ for a node) without changing the optimal sets. Output yes iff $\Phi(\beta) > 0$.

Correctness. If some S has $e(S)/|S| \geq \alpha$, then

$$e(S) - \beta|S| = (e(S) - \alpha|S|) + \frac{|S|}{2qn} > 0.$$

Conversely, suppose $\Phi(\beta) > 0$ and take a maximizer S . Then $S \neq \emptyset$, since the empty set scores 0, and $e(S)/|S| > \beta$. Write $e(S)/|S| = a/k$ with $1 \leq k \leq n$. If $a/k < p/q$ we would get

$$\frac{p}{q} - \frac{a}{k} = \frac{pk - aq}{qk} \geq \frac{1}{qk} \geq \frac{1}{qn} > \frac{1}{2qn},$$

which gives $a/k < \beta$, a contradiction. So $e(S)/|S| \geq \alpha$.

Running time. The network has $N = m + n + 2$ nodes and $M = O(m + n)$ edges. Its integer capacities are at most $C = O(qnm + |p|n)$, and the edges of “infinite” capacity can be given capacity $C + 1$ as in Section 7.11. The Scaling Max-Flow Algorithm (Section 7.3) runs in $O(M^2 \log C)$ time, which is polynomial in n , m and the bit length of α .

(b) Binary search with (a) over the finitely many possible cohesiveness values, then read the set off the final cut.

Every cohesiveness value lies in $F = \{a/k : 0 \leq a \leq m, 1 \leq k \leq n\}$, a set of at most $(m + 1)n$ values, and the optimum α^* lies in F . The answer of (a) is yes exactly when $\alpha \leq \alpha^*$, so it is monotone in α . Sort F and binary search over it with (a). This finds α^* , the largest element of F that gets a yes, using $O(\log(mn))$ calls.

Now run the construction of (a) once more with $\alpha = \alpha^*$ and return $S = A' \cap V$ from the minimum cut. This S maximizes $e(S) - \beta|S|$, and $\Phi(\beta) > 0$ because α^* is attained. The converse argument of (a) then gives $S \neq \emptyset$ and $e(S)/|S| \geq \alpha^*$, so S has maximum cohesiveness.

Every $\alpha \in F$ has $q \leq n$ and $|p| \leq m$, so each flow computation takes $O((m + n)^2 \log(nm))$ time. Building and sorting F takes $O(mn \log(mn))$ time. The total is polynomial.

Problem 7.47. This exercise looks at variants of the Preflow-Push Algorithm that relabel less often while keeping its worst-case bounds. The algorithm maintains the invariant that $h(v) \leq h(w) + 1$ for every edge (v, w) of the residual graph of the current preflow. If f is a flow (not just a preflow) satisfying this invariant, then f is a maximum flow. Heights only ever increase, and the running-time analysis rests on bounding how many times heights can increase. In practice the algorithm runs much faster than its worst case, and the real bottleneck is relabeling, not the nonsaturating pushes

that drive the worst-case bound. So the aim here is to raise heights by more than one at a time. The input is a graph G with n nodes, m edges, capacities c , source s and sink t .

(a) The algorithm of Section 7.4 starts by putting flow c_e on every edge e out of s and flow 0 on every other edge, with $h(s) = n$ and $h(v) = 0$ for all other nodes v . Give an $O(m)$ procedure for setting the initial heights that improves on this. Each node v should get the largest height possible given the initial flow.

(b) Add a new step called *gap relabeling*. Let f be a preflow and h heights that satisfy the invariant. A *gap* is an integer $0 < h < n$ such that no node has height exactly h . For a gap value h , let A be the set of nodes v with $h < h(v) < n$. Gap relabeling sets the height of every node in A to n . Prove that Preflow-Push with gap relabeling is still a correct max-flow algorithm. (The only new thing to prove is that gap relabeling preserves the invariant $h(v) \leq h(w) + 1$ on residual edges.)

(c) Section 7.4 proves $h(v) \leq 2n - 1$ throughout. Now consider a variant in which $h(v) \leq n$ throughout: nodes are “frozen” once they reach height n . A node at height n no longer counts as active and is never chosen for push or relabel. So the algorithm ends with a preflow and a height function that satisfy the invariant, with all remaining excess at height n . Let B be the set of nodes v that have a path to t in the residual graph of the final preflow, and let $A = V - B$. Prove that at the end of the algorithm, (A, B) is a minimum-capacity s - t cut.

(d) The algorithm of part (c) finds a minimum cut, but it ends with a preflow that still has excesses, so it does not find a maximum flow. Give an algorithm that takes the final preflow f of part (c) and turns it into a maximum flow in $O(mn)$ time. (Hint: look at the nodes with excess, and try to send their excess back to s using only edges that the flow came in on.)

Solution. (a) Set each height to the residual distance: $h(v) = d_t(v)$, the number of edges on a shortest v - t path in the initial residual graph G_f , whenever that path exists. If v cannot reach t but can reach s , set $h(v) = n + d_s(v)$. Every remaining node gets $h(v) = 2n - 1$. Keep $h(s) = n$ and $h(t) = 0$.

To compute this, build G_f and run a BFS from t along reversed edges. Then run a second reversed BFS from s through the nodes that the first BFS did not reach. This is $O(m)$.

In the initial G_f , node s has no outgoing edges: every edge (s, v) is saturated, and every edge (u, s) carries flow 0. So no shortest path to t passes through s . Also, a node on a residual path to s that starts at a node unable to reach t cannot reach t itself. Hence $d_t(v) \leq n - 1$ and $n \leq n + d_s(v) \leq 2n - 1$. Compatibility: s has no residual out-edges and $h(t) = 0$, so let $(v, w) \in E_f$ with $v \neq s, t$.

(i) If v reaches t , then either w reaches t and $d_t(v) \leq d_t(w) + 1$, or $h(v) \leq n - 1 < h(w) + 1$ (this includes $w = s$).

(ii) If v reaches s but not t , then w cannot reach t . If w reaches s (this includes $w = s$, with $d_s(s) = 0$), then $n + d_s(v) \leq n + d_s(w) + 1$. Otherwise $h(v) \leq 2n - 1 = h(w) \leq h(w) + 1$.

(iii) If v reaches neither, then neither does w , and $2n - 1 \leq 2n$.

So (f, h) is compatible, and the analysis of Section 7.4 still applies. Every initial height is at most $2n - 1$, and the proof of (7.26) covers every node that is later relabeled, so (7.26)–(7.28) hold as stated. In (7.29), the starting potential $\sum_{e_f(v) > 0} h(v)$ grows by at most $2n^2$, so the $O(n^2m)$ bound is unchanged.

Maximality: let h' be any compatible labeling and $v = v_0, v_1, \dots, v_k$ a shortest residual path. Applying steepness along the path gives

$$h'(v) \leq h'(v_k) + k = \begin{cases} d_t(v), & v_k = t, \\ n + d_s(v), & v_k = s. \end{cases}$$

So every node that reaches s or t gets the largest height that any compatible labeling allows. For a node that reaches neither, compatibility puts no upper bound on its height, so we use the ceiling $2n - 1$ from (7.26).

(b) Gap relabeling only raises heights, and it leaves s (height n) and t (height $0 < h$) unchanged. So it can only break the invariant on a residual edge (v, w) with $v \in A$, $w \notin A$. For such an edge,

$$h(w) \geq h(v) - 1 \geq h, \quad h(w) \neq h \text{ (gap)}, \quad w \notin A \Rightarrow h(w) \geq n,$$

so the new height satisfies $h(v) = n \leq h(w) + 1$. If both $v, w \in A$, both heights become n . If $v \notin A$, then $h(v)$ is unchanged and $h(w)$ does not decrease.

Hence (7.24) still holds: f remains a preflow compatible with h . If the algorithm stops, (7.22) shows that f is a maximum flow.

Termination and the bounds are unaffected. Heights stay monotone, and by (7.25) and (7.26) they stay at most $2n - 1$ (a gap relabel sets heights to n). So the total height increase per node, from all causes, is at most $2n - 1$. The counting in (7.27)–(7.29) therefore goes through, and each gap relabel (applied only when $A \neq \emptyset$) uses up at least one unit of this increase.

(c) (A, B) is an s - t cut, because $t \in B$ trivially and $s \in A$ by (7.21). Freezing only restricts which push and relabel steps are allowed, so (7.24) still gives compatibility.

At termination, every node $v \neq s, t$ with $e_f(v) > 0$ has $h(v) = n$. Now suppose $v \in B - \{t\}$. Applying steepness along a simple residual v - t path, which has at most $n - 1$ edges, gives $h(v) \leq n - 1$. So $e_f(v) = 0$ for all $v \in B - \{t\}$.

No residual edge leaves A , since its tail would then reach t . So every edge from A to B is saturated, and every edge from B to A carries flow 0. Hence

$$c(A, B) = f^{\text{in}}(B) - f^{\text{out}}(B) = \sum_{v \in B} e_f(v) = e_f(t).$$

Now take any s - t cut (A', B') . Every $v \neq s$ has $e_f(v) \geq 0$, and $s \notin B'$, so

$$\begin{aligned} c(A, B) = e_f(t) &\leq \sum_{v \in B'} e_f(v) = f^{\text{in}}(B') - f^{\text{out}}(B') \\ &\leq f^{\text{in}}(B') \leq c(A', B'). \end{aligned}$$

So (A, B) has minimum capacity.

(d) Repeatedly take a node $v \neq s, t$ with excess and walk backward from it along edges that carry positive flow. If the walk reaches s , return excess along that path. If it closes a cycle, cancel the cycle.

```

for each node x: ptr(x) = first edge into x
for each node v other than s, t:
    while e(v) > 0:
        walk P = v; x = v
        while x != s:
            advance ptr(x) past edges with zero flow
            (u, x) = ptr(x)
            if u is already on P:
                C = cycle on P closed by (u, x)
                subtract min flow on C from every edge of C
                restart walk from v
            prepend u to P; x = u
        d = min(e(v), min flow on the edges of P)
        subtract d from every edge of P    (P runs from s to v)
return f

```

Correctness: whenever the walk is at a node $x \neq s$, some edge into x carries positive flow. This holds for $x = v$ because $e_f(v) > 0$. For an earlier node x of the walk, the flow on (x, y) is positive, so $f^{\text{out}}(x) > 0$, and $e_f(x) \geq 0$ then gives $f^{\text{in}}(x) > 0$. So the walk is always defined.

Subtracting d along an s - v path, or subtracting the bottleneck around a cycle, keeps every flow value between 0 and c_e . It also leaves the excess unchanged at every node except v (whose excess drops by d) and s .

No node with positive flow on an out-edge is ever t . In part (c), t is never active, so no push ever leaves t , and the initial flow is 0 there. So t lies on no walk, and $e_f(t)$ never changes. Excesses at nodes other than s never increase, so a node is finished once its excess is 0.

At the end, every $v \neq s, t$ has $e_f(v) = 0$, so f is a flow. Its value is $e_f(t)$, which by part (c) equals $c(A, B)$. By (7.8), this flow is maximum.

Running time: flows only decrease, so an edge whose flow becomes 0 stays at 0. Each pointer therefore moves forward only, for $O(m)$ pointer work in total.

Each walk visits at most n nodes, so it costs $O(n)$ apart from pointer moves. Each walk ends in one of two ways:

- a cycle cancellation, which zeroes at least one edge (at most m times overall);
- a path return, which zeroes an edge or the excess $e_f(v)$ (at most $m + n$ times overall).

So there are $O(m+n)$ walks, and the total is

$$O(n(m+n) + m) = O(mn),$$

using $m \geq n/2$ (the book assumes every node has an incident edge).

Problem 7.48. This exercise studies another node-selection rule for the Preflow-Push Algorithm of Section 7.4, together with versions that stop early and return a cut whose capacity is close to the minimum.

(a) Let f be an arbitrary preflow. Because f need not be a flow, $f^{out}(s)$ can be far larger than the maximum-flow value of G . Prove that $f^{in}(t)$, on the other hand, is always a lower bound on the maximum-flow value.

(b) Let f be a preflow with a compatible labeling h , let A be the set of nodes v for which G_f contains an s - v path, and let $B = V - A$; for such f the pair (A, B) is an s - t cut. Prove that its capacity is

$$c(A, B) = \sum_{v \in B} e_f(v).$$

(Together, (a) and (b) let the algorithm stop early and output (A, B) as a nearly minimum cut once $c(A, B) - f^{in}(t)$ is small. The rest of the exercise aims to shrink this gap by pushing flow out of nodes that have a lot of excess.)

(c) The scaling version of Preflow-Push keeps a parameter Δ , initially a large power of 2. At each step it chooses, among the nodes with excess at least Δ , one of smallest height. When no node other than t has excess at least Δ , it halves Δ and goes on. This is a legitimate implementation of the generic Preflow-Push Algorithm. A phase is a maximal stretch during which Δ does not change. Δ starts at the largest capacity C and the algorithm stops when $\Delta = 1$, so there are $O(\log C)$ phases. Show how to implement this variant so that it runs in $O(mn + n \log C + K)$ time, where K is the number of nonsaturating push operations.

(d) Prove that the algorithm performs $O(n^2 \log C)$ nonsaturating pushes. To bound their number within one phase, use the potential $\Phi = \sum_{v \in V} h(v)e_f(v)/\Delta$. How does a nonsaturating push change Φ ? Which operations can increase Φ ?

Solution. Throughout we use the convention of Section 7.1 that no edge enters s and no edge leaves t . Then $f^{in}(t) = e_f(t)$ and $e_f(s) = -f^{out}(s) \leq 0$. For a set B of nodes, the terms of edges with both ends in B cancel, so

$$\sum_{v \in B} e_f(v) = f^{in}(B) - f^{out}(B). \quad (\dagger)$$

(a) For every s - t cut (A, B) we have $f^{in}(t) \leq c(A, B)$; taking a minimum cut gives the claim by (7.13). Indeed, $t \in B$ and $s \notin B$, so every $v \in B - \{t\}$ has $e_f(v) \geq 0$ by preflow condition (ii). Hence

$$\begin{aligned} f^{in}(t) = e_f(t) &= \sum_{v \in B} e_f(v) = f^{in}(B) - f^{out}(B) \\ &\leq f^{in}(B) \leq c(A, B). \end{aligned}$$

(b) The edges from A to B are all saturated and the edges from B to A all carry zero flow. By (7.21), $t \notin A$, so (A, B) is an s - t cut.

- If $e = (u, w)$ with $u \in A$, $w \in B$ had $f(e) < c_e$, then the forward edge (u, w) would lie in G_f and would put w in A .
- If $e = (u, w)$ with $u \in B$, $w \in A$ had $f(e) > 0$, then the backward edge (w, u) would lie in G_f and would put u in A .

So $f^{in}(B) = c(A, B)$ and $f^{out}(B) = 0$, and $(\dagger)$ gives $\sum_{v \in B} e_f(v) = c(A, B)$.

(c) We use buckets of large-excess nodes indexed by height, a pointer L to the lowest nonempty bucket, and the current-edge pointers of (7.33). We also cap each push so that no receiving node ends up with more than 2Δ of excess (part (d) needs this cap). Let $\Delta_0 = 2^{\lceil \log_2 C \rceil}$. There are then $\lceil \log_2 C \rceil + 1$ phases, $\Delta = \Delta_0, \Delta_0/2, \dots, 1$. Call a node $v \neq s, t$ *big* if $e_f(v) \geq \Delta$.

Push rule. Suppose the selected node v is a big node of minimum height and $(v, w) \in E_f$ with $h(w) < h(v)$. Then w is in $\{s, t\}$ or has $e_f(w) < \Delta$, because w is lower than every big node. Push

$$\delta = \min(e_f(v), r_f(v, w), 2\Delta - e_f(w)),$$

leaving out the last term when $w \in \{s, t\}$. Since $0 < \delta \leq \min(e_f(v), r_f(v, w))$, the proof of (7.24) applies unchanged: f stays a preflow compatible with h . The proofs of (7.26)-(7.28) and (7.31)-(7.32) never use the exact value of δ , so they hold as well. The push rule keeps the invariant

(I) during phase Δ , $e_f(v) \leq 2\Delta$ for every $v \neq s, t$.

At the start of the first phase, $e_f(v) = c_{(s,v)} \leq C \leq \Delta_0$. At the start of any later phase, $e_f(v) < 2\Delta$, because the previous phase ended with no excess at least 2Δ . A push raises $e_f(w)$ to at most 2Δ .

Data structures. Keep the adjacency lists with $current(v)$ pointers as in (7.33). Keep doubly linked buckets $Q[0], \dots, Q[2n-1]$, where $Q[k]$ holds the big nodes of height k (heights are at most $2n-1$ by (7.26)). Each node stores a pointer to its own list cell, and L is an index with $Q[j]$ empty for all $j < L$. To take a step, increase L while $Q[L]$ is empty, then let v be the head of $Q[L]$. Advance $current(v)$ past inadmissible edges. If the list runs out, relabel v , which is legal by (7.32); then move v to $Q[L+1]$ and reset $current(v)$. Otherwise push on $current(v)$. If $e_f(v)$ drops below Δ , delete v from its bucket. If w becomes big, insert it into $Q[L-1]$ and set $L \leftarrow L-1$. When L passes $2n-1$, the phase ends: halve Δ and rebuild the buckets and L by one scan of the nodes. In the last phase $\Delta = 1$ and the excesses are integers (7.24), so the algorithm ends with zero excess at every $v \neq s, t$. Then f is a flow compatible with h (7.24), hence maximum by (7.22).

Running time.

- (i) Setting up the initial preflow costs $O(m)$, and rebuilding the buckets costs $O(n)$ per phase, which is $O(n \log C)$ in total.
- (ii) Each push and each relabel costs $O(1)$. There are fewer than $2n^2$ relabels by (7.27), at most $2nm$ saturating pushes by (7.28), and K nonsaturating pushes.
- (iii) By (7.32), $current(v)$ passes over each of the d_v edges at v at most once between relabels of v . This costs $O(\sum_v 2n d_v) = O(mn)$, as in (7.33).
- (iv) Within a phase, L decreases only when a push makes its receiver big, so it decreases by 1 at most once per push. A node becomes big only by receiving a push, and the receiver sits at height $L-1$. Since $0 \leq L \leq 2n$, the total increase of L is at most $2n$ plus the total decrease. Over all phases this is $O(n \log C + nm + K)$.

Adding these up, the running time is $O(mn + n \log C + K)$.

(d) Every nonsaturating push decreases Φ by at least 1. Within a phase, only a relabel can increase Φ , and by at most 2 each time; halving Δ doubles Φ , which we handle with a fresh bound at the start of each phase. Here we sum over $v \neq s$; the dropped term $h(s)e_f(s) \leq 0$ belongs to the source, and with it gone $\Phi \geq 0$.

- (i) A push of δ from v to w changes Φ by $\delta(h(w) - h(v))/\Delta = -\delta/\Delta$ when $w \neq s$, and by $-\delta h(v)/\Delta \leq -\delta/\Delta$ when $w = s$. A nonsaturating push has $\delta < r_f(v, w)$, so either $\delta = e_f(v) \geq \Delta$ (v was selected) or $\delta = 2\Delta - e_f(w) > \Delta$ (since $e_f(w) < \Delta$). In both cases Φ drops by at least 1.
- (ii) A relabel of v raises Φ by $e_f(v)/\Delta \leq 2$, by (I).
- (iii) At the start of a phase, $e_f(v) < 2\Delta$ for $v \neq s, t$ (and $\Phi = 0$ before the first push), $h(t) = 0$, and $h(v) \leq 2n-1$ by (7.26). Hence $\Phi < 2(n-2)(2n-1)$.

So if phase p has R_p relabels, it has at most $2(n-2)(2n-1) + 2R_p$ nonsaturating pushes. Since $\sum_p R_p < 2n^2$ by (7.27), the total is

$$\begin{aligned} K &\leq (\lceil \log_2 C \rceil + 1) \cdot 2(n-2)(2n-1) + 4n^2 \\ &= O(n^2 \log C). \end{aligned}$$

Combined with (c), the algorithm runs in $O(mn + n^2 \log C)$ time.

Problem 7.49. There are n stations (say cell towers) and a set U of k customers (requests, say cell phones). A bipartite graph G has an edge (x, y) between customer x and station y when x can be served from y , and each station serves at most one customer. (One max-flow computation tells whether every customer can be served, or finds an assignment serving as many customers as possible.)

Now each customer $x \in U$ will pay $v_x \geq 0$ for being served, and we want a subset $X \subseteq U$ maximizing $\sum_{x \in X} v_x$ such that the customers of X can all be assigned to distinct stations (call such a set *feasible*). Greedy rule: process the customers in order of decreasing value, breaking ties arbitrarily. Let X be the set of customers promised service so far. When customer x comes up, add x to X if $X \cup \{x\}$ is feasible; otherwise reject x for good. The actual assignment of customers to stations is fixed only after the whole set X has been chosen. Does this greedy approach always produce an optimal set of customers? Prove that it does, or give a counterexample.

Solution. Yes: the feasible sets satisfy the exchange property below, and greedy keeps the largest values that this property allows.

If A, B are feasible with $|A| < |B|$, then $A \cup \{b\}$ is feasible for some $b \in B \setminus A$. Indeed, take matchings M_A, M_B covering exactly A and B ; the components of $M_A \oplus M_B$ are alternating paths and cycles, and as $|M_B| > |M_A|$ one of them is a path P beginning and ending with M_B -edges. By bipartiteness its ends are a customer b and a station s , both uncovered by M_A (an M_A -edge there would extend P), so $b \notin A$ and $M_A \oplus P$ is a matching covering $A \cup \{b\}$.

Greedy's output $X = \{g_1, \dots, g_m\}$ (in acceptance order) is feasible by construction, and $v_{g_1} \geq \dots \geq v_{g_m}$. Let $O = \{o_1, \dots, o_r\}$ be an optimal feasible set with $v_{o_1} \geq \dots \geq v_{o_r}$. We claim that $m \geq r$ and that

$$v_{g_i} \geq v_{o_i} \quad (1 \leq i \leq r).$$

Suppose the claim fails, and let i be the first index where either g_i does not exist or $v_{g_i} < v_{o_i}$. Apply the exchange property to $A = \{g_1, \dots, g_{i-1}\}$ and $B = \{o_1, \dots, o_i\}$. Both are feasible (every subset of a feasible set is feasible), and $|A| = i - 1 < i = |B|$, so there is an $o_j \in B \setminus A$ with $A \cup \{o_j\}$ feasible. Since $j \leq i$,

$$v_{o_j} \geq v_{o_i} > v_{g_i} \quad (\text{when } g_i \text{ exists}),$$

so greedy examines o_j before g_i (or at some point, if g_i does not exist). At that moment the promised set is contained in A , since only $g_1, \dots, g_{i-1}$ have been accepted so far. Adding o_j to it gives a subset of the feasible set $A \cup \{o_j\}$, which is feasible, so greedy accepts o_j . Every customer greedy accepts before g_i is one of $g_1, \dots, g_{i-1}$, so $o_j \in A$, a contradiction. Summing the claim over i ,

$$\sum_{x \in X} v_x \geq \sum_{i=1}^r v_{g_i} \geq \sum_{i=1}^r v_{o_i} = \sum_{x \in O} v_x,$$

where the first inequality uses $v \geq 0$. Hence greedy is optimal. ■

7.8 Exercises 7.50–7.51

Problem 7.50. There are m machines, and each processes one job at a time. Every job must be assigned to exactly one machine, and the jobs on each machine must be put in an order, so as to minimize a cost function. All jobs have the same processing requirement, but machines differ in speed: machine i has a parameter ℓ_i , and any job run on machine i takes ℓ_i time. There are n jobs, which differ in urgency: job j comes with a cost function $c_j(t)$, the cost of finishing job j at time t . These costs are nonnegative and monotone in t . A schedule is an assignment of jobs to machines together with an order on each machine; the first job on machine i finishes at time ℓ_i , the second at time $2\ell_i$, and so on. For a schedule S let $t_S(j)$ be the completion time of job j , and define

$$\text{cost}(S) = \sum_j c_j(t_S(j)).$$

Give a polynomial-time algorithm that finds a schedule of minimum cost.

Solution. Reduce to the Minimum-Cost Perfect Matching Problem of Section 7.13, matching jobs to slots (i, k) , where slot (i, k) is the k -th position on machine i , finishing at $k\ell_i$. Here monotone is read as nondecreasing: finishing later never costs less.

Build a bipartite graph. On one side put the slots (i, k) for $1 \leq i \leq m$, $1 \leq k \leq n$ ($N = mn$ of them; no machine ever receives more than n jobs). On the other side put the n jobs plus $N - n$ dummy jobs. Join every job j to every slot (i, k) with cost $c_j(k\ell_i) \geq 0$, and join every dummy to every slot with cost 0. Both sides have size N , so the algorithm of Section 7.13 computes a minimum-cost perfect matching M . To read off a schedule, take the real jobs that M places on machine i , at slots $k_1 < k_2 < \dots < k_{r_i}$, and run them on machine i in that order.

Correctness. Let OPT be the minimum schedule cost.

(i) $\text{cost}(M) \leq \text{OPT}$. Any schedule S puts each job j in a distinct slot (i, k) with $t_S(j) = k\ell_i$, where $k \leq n$. Matching the dummies to the $N - n$ unused slots gives a perfect matching of cost $\sum_j c_j(t_S(j)) = \text{cost}(S)$.

(ii) The output schedule S_M satisfies $\text{cost}(S_M) \leq \text{cost}(M)$. The job matched to slot (i, k_q) is the q -th job on machine i , and $q \leq k_q$ because $k_1 < \dots < k_q$ are distinct positive integers. Monotonicity and zero-cost dummies then give

$$\text{cost}(S_M) = \sum_i \sum_{q=1}^{r_i} c_{j_{i,q}}(q\ell_i) \leq \sum_i \sum_{q=1}^{r_i} c_{j_{i,q}}(k_q\ell_i) = \text{cost}(M).$$

Combining (i) and (ii), $\text{OPT} \leq \text{cost}(S_M) \leq \text{cost}(M) \leq \text{OPT}$, so S_M is optimal.

Running time. The graph has $2N = 2mn$ nodes and $N^2 = m^2n^2$ edges, and building it takes mn^2 evaluations of the c_j . By (7.66), the matching costs N shortest-path computations with nonnegative lengths. Each one is a Dijkstra run in $O(N^2)$ time on this dense graph, so the total is $O(N^3) = O(m^3n^3)$, which is polynomial.

Problem 7.51. Some friends of yours have grown tired of the game “Six Degrees of Kevin Bacon” (after all, they ask, isn’t it just breadth-first search?) and decide to invent a game with a little more punch, algorithmically speaking. Here’s how it works.

You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1 . Player P_0 names an actress $x_1 \in X$, player P_1 names an actor y_1 who has appeared in a movie with x_1 , player P_0 names an actress x_2 who has appeared in a movie with y_1 , and so on. Thus, P_0 and P_1 collectively generate a sequence $x_1, y_1, x_2, y_2, \dots$ such that each actor/actress in the sequence has costarred with the actress/actor immediately preceding. A player P_i ($i = 0, 1$) loses when it is P_i ’s turn to move, and he/she cannot name a member of his/her set who hasn’t been named before.

Suppose you are given a specific pair of such sets X and Y , with complete information on who has appeared in a movie with whom. A *strategy* for P_i , in our setting, is an algorithm that takes a current sequence $x_1, y_1, x_2, y_2, \dots$ and generates a legal next move for P_i (assuming it’s P_i ’s turn to move). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

Solution. Build the bipartite costar graph G on $X \cup Y$ (edge xy iff x and y appeared in a movie together) and compute a maximum matching M ; P_1 can force a win iff M is perfect ($|M| = n$), and otherwise P_0 can.

(i) $|M| = n$. P_1 answers every x_i with $y_i = M(x_i)$. This is legal: $x_i y_i \in E$, and the earlier actors are $y_j = M(x_j)$ with $x_j \neq x_i$, so $y_i \neq y_j$ because M is a matching. Hence P_1 always has a move, and since the game ends after at most $2n$ moves, P_0 is the one who gets stuck.

(ii) $|M| < n$. Some $x_1 \in X$ is unmatched by M ; P_0 opens with x_1 and answers every y_i with $x_{i+1} = M(y_i)$. Suppose P_0 has played this way through y_i . If y_i were unmatched, then

$$P = x_1 y_1 x_2 y_2 \dots x_i y_i$$

has $y_j x_{j+1} \in M$ for $j < i$, and $x_j y_j \notin M$ for every j (x_1 is unmatched, and for $j \geq 2$ the partner of x_j is $y_{j-1} \neq y_j$). So P is an alternating path with both ends unmatched, and $M \oplus P$ is a matching with $|M| + 1$ edges, contradicting maximality. So $M(y_i)$ exists and is adjacent to y_i ; it is new because $M(y_i) \neq x_1$ (since x_1 is unmatched) and $M(y_i) \neq M(y_j) = x_{j+1}$ for $j < i$ (since $y_i \neq y_j$). Hence P_0 always has a move, and P_1 is the one who gets stuck.

By (7.38), Ford-Fulkerson on the flow network of Section 7.5 finds M in $O(mn)$ time, where m is the number of costar pairs; building G and comparing $|M|$ with n costs $O(m + n)$, so the algorithm runs in $O(mn)$ time.

8 NP and Computational Intractability

Contents

8.1 Exercises 8.1–8.7	131
8.2 Exercises 8.8–8.14	135
8.3 Exercises 8.15–8.21	140
8.4 Exercises 8.22–8.28	146
8.5 Exercises 8.29–8.35	150
8.6 Exercises 8.36–8.42	156

8.1 Exercises 8.1–8.7

Problem 8.1. For each of the two questions below, decide whether the answer is (i) “Yes,” (ii) “No,” or (iii) “Unknown, because it would resolve the question of whether $\mathcal{P} = \mathcal{NP}$.” Give a brief explanation of your answer.

(a) Let’s define the decision version of the Interval Scheduling Problem from Chapter 4 as follows: Given a collection of intervals on a time-line, and a bound k , does the collection contain a subset of nonoverlapping intervals of size at least k ?

Question: Is it the case that Interval Scheduling $\leq_P$ Vertex Cover?

(b) Question: Is it the case that Independent Set $\leq_P$ Interval Scheduling?

Solution. (a) Yes. Interval Scheduling is solvable in polynomial time: the greedy earliest-finish-time algorithm of Section 4.1 returns a maximum set of nonoverlapping intervals by (4.3), in $O(n \log n)$ time, and we answer “yes” iff that set has size at least k . An algorithm that solves the problem in polynomial time using zero calls to a black box for Vertex Cover is a reduction in the sense of Section 8.1, so Interval Scheduling $\leq_P$ Vertex Cover. (In Karp form: run the greedy algorithm, then output the fixed Vertex Cover instance (one vertex, no edges, $k = 0$) if the answer is yes and (one edge uv , $k = 0$) if it is no.)

(b) Unknown, because it would resolve $\mathcal{P}$ vs. $\mathcal{NP}$.

(i) If Independent Set $\leq_P$ Interval Scheduling, then since Interval Scheduling is in $\mathcal{P}$ (part (a)), (8.1) puts Independent Set in $\mathcal{P}$; Independent Set is NP-complete by (8.16), so $\mathcal{P} = \mathcal{NP}$ by (8.12).

(ii) If $\mathcal{P} = \mathcal{NP}$, then Independent Set $\in \mathcal{NP}$ is solvable in polynomial time, and as in (a) it reduces to any problem, Interval Scheduling included.

So the reduction exists if and only if $\mathcal{P} = \mathcal{NP}$.

Problem 8.2. A store trying to analyze the behavior of its customers will often maintain a two-dimensional array A , where the rows correspond to its customers and the columns correspond to the products it sells. The entry $A[i, j]$ specifies the quantity of product j that has been purchased by customer i .

Here’s a tiny example of such an array A .

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

One thing that a store might want to do with this data is the following. Let us say that a subset S of the customers is *diverse* if no two of the customers in S have ever bought the same product (i.e., for each product, at most one of the customers in S has ever bought it). A diverse set of customers can be useful, for example, as a target pool for market research.

We can now define the *Diverse Subset Problem* as follows: Given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is diverse?

Show that Diverse Subset is NP-complete.

Solution. Diverse Subset is Independent Set in disguise: customers are vertices, products are edges, and a customer buys exactly the edges incident to it.

Membership in NP. A certificate is the set S ; the checker verifies $|S| \geq k$ and that every column j has $A[i, j] > 0$ for at most one $i \in S$, in $O(mn)$ time.

Reduction. Independent Set is NP-complete by (8.16); we show Independent Set $\leq_P$ Diverse Subset. Given $G = (V, E)$ and k , we may assume $k \leq |V|$ (otherwise the answer is no, and we output the fixed no-instance with two customers who both bought one product and $k = 2$). Build the $|V| \times |E|$ array

$$A[v, e] = \begin{cases} 1 & v \text{ is an end of } e, \\ 0 & \text{otherwise,} \end{cases}$$

in $O(|V| |E|)$ time, and keep the same k . For $S \subseteq V$,

$$\begin{aligned} S \text{ diverse} &\iff \text{no column } e = uv \text{ has two buyers in } S \\ &\iff \text{no edge } uv \text{ has } u, v \in S \\ &\iff S \text{ independent in } G, \end{aligned}$$

since the buyers of product $e = uv$ are exactly u and v . Hence G has an independent set of size at least k iff A has a diverse subset of size at least k , and by (8.14) Diverse Subset is NP-complete.

Problem 8.3. Suppose you're helping to organize a summer sports camp, and the following problem comes up. The camp is supposed to have at least one counselor who's skilled at each of the n sports covered by the camp (baseball, volleyball, and so on). They have received job applications from m potential counselors. For each of the n sports, there is some subset of the m applicants qualified in that sport. The question is: For a given number $k < m$, is it possible to hire at most k of the counselors and have at least one counselor qualified in each of the n sports? We'll call this the *Efficient Recruiting Problem*. Show that Efficient Recruiting is NP-complete.

Solution. Efficient Recruiting contains Vertex Cover: counselors are vertices, sports are edges, and a counselor is qualified in exactly the edges incident to it.

Membership in NP. A certificate is the set H of hired counselors; check $|H| \leq k$ and that each of the n sports has a qualified member of H , in $O(nm)$ time.

Reduction. Vertex Cover is NP-complete by (8.16); we show $\text{Vertex Cover} \leq_P \text{Efficient Recruiting}$. Given $G = (V, E)$ and k : if $k \geq |V|$ the answer is yes (V itself is a cover), and we output the fixed yes-instance with $m = 2$ counselors, one sport in which both are qualified, and $k = 1$. Otherwise $k < |V|$, and we output the instance with counselors V (so $m = |V| > k$), one sport s_e for each $e \in E$, the qualified set of $s_e = uv$ being $\{u, v\}$, and the same k ; this takes $O(|V| + |E|)$ time. For $H \subseteq V$,

$$\begin{aligned} H \text{ staffs every sport} &\iff \forall e = uv \in E : H \cap \{u, v\} \neq \emptyset \\ &\iff H \text{ is a vertex cover of } G, \end{aligned}$$

so G has a vertex cover of size at most k iff at most k counselors can be hired to cover every sport. By (8.14), Efficient Recruiting is NP-complete.

Problem 8.4. Suppose you're consulting for a group that manages a high-performance real-time system in which asynchronous processes make use of shared resources. Thus the system has a set of n processes and a set of m resources. At any given point in time, each process specifies a set of resources that it requests to use. Each resource might be requested by many processes at once; but it can only be used by a single process at a time. Your job is to allocate resources to processes that request them. If a process is allocated all the resources it requests, then it is *active*; otherwise it is *blocked*. You want to perform the allocation so that as many processes as possible are active. Thus we phrase the *Resource Reservation Problem* as follows: Given a set of processes and resources, the set of requested resources for each process, and a number k , is it possible to allocate resources to processes so that at least k processes will be active?

Consider the following list of problems, and for each problem either give a polynomial-time algorithm or prove that the problem is NP-complete.

- The general Resource Reservation Problem defined above.
- The special case of the problem when $k = 2$.
- The special case of the problem when there are two types of resources — say, people and equipment — and each process requires at most one resource of each type. (In other words, each process requires one specific person and one specific piece of equipment.)
- The special case of the problem when each resource is requested by at most two processes.

Solution. Write R_p for the set of resources requested by process p . A set S of processes can be made simultaneously active iff the sets R_p , $p \in S$, are pairwise disjoint (give each resource of R_p to p ; conversely, a resource in $R_p \cap R_q$ cannot serve both p and q). So the question is whether some k of the R_p are pairwise disjoint.

(a) NP-complete. *Membership in NP:* the certificate is the set S of k processes, and pairwise disjointness of $\{R_p\}_{p \in S}$ is checked in $O(nm)$ time with a marker array over the resources. *Reduction:* Independent Set (NP-complete by (8.16)) $\leq_P$ Resource Reservation. Given $G = (V, E)$ and k , make one resource per edge and one process per vertex, with $R_v = \{e \in E : v \in e\}$, and keep k . For $S \subseteq V$,

$$\begin{aligned} \{R_v\}_{v \in S} \text{ pairwise disjoint} &\iff \text{no edge } e = uv \text{ has } u, v \in S \\ &\iff S \text{ is independent in } G, \end{aligned}$$

because $R_u \cap R_v$ is the set of edges joining u and v . So G has an independent set of size k iff k processes can be active, and (8.14) gives NP-completeness.

(b) Polynomial. Answer yes iff some pair $p \neq q$ has $R_p \cap R_q = \emptyset$, by the opening observation. There are $\binom{n}{2}$ pairs, each tested in $O(m)$ time with a marker array, for $O(n^2m)$ in total.

(c) Polynomial, by bipartite matching. Build a bipartite graph G with the people on one side and the equipment on the other; first set aside the processes with $R_p = \emptyset$ (always active; say z of them). A process requesting person a and equipment b becomes an edge ab . A process requesting only person a becomes an edge from a to a new dummy equipment node of its own, and symmetrically for one requesting only equipment. Two processes conflict iff their edges share an end (dummy nodes are private), so active sets of nonempty-request processes are exactly the matchings of G , and

$$\max |\text{active set}| = z + \nu(G),$$

where $\nu(G)$ is the maximum matching size. Answer yes iff $z + \nu(G) \geq k$. By (7.37) and (7.38), Ford-Fulkerson computes $\nu(G)$ (parallel edges, from processes with the same pair, are harmless); G has at most n edges and $m + n$ nodes, and the flow value is at most n , so there are at most n augmentations of $O(m + n)$ each, for $O(n(m + n))$ time.

(d) NP-complete. Membership follows from (a). The reduction in (a) already produces an instance of this special case: resource $e = uv$ is requested by exactly the two processes u and v . So Independent Set $\leq_P$ this special case, and by (8.14) it is NP-complete.

Problem 8.5. Consider a set $A = \{a_1, \dots, a_n\}$ and a collection $B_1, B_2, \dots, B_m$ of subsets of A (i.e., $B_i \subseteq A$ for each i).

We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \dots, B_m$ if H contains at least one element from each B_i — that is, if $H \cap B_i$ is not empty for each i (so H “hits” all the sets B_i).

We now define the *Hitting Set Problem* as follows. We are given a set $A = \{a_1, \dots, a_n\}$, a collection $B_1, B_2, \dots, B_m$ of subsets of A , and a number k . We are asked: Is there a hitting set $H \subseteq A$ for $B_1, B_2, \dots, B_m$ so that the size of H is at most k ?

Prove that Hitting Set is NP-complete.

Solution. Vertex Cover is the special case of Hitting Set in which every B_i is an edge.

Membership in NP. A certificate is H ; check $|H| \leq k$ and $H \cap B_i \neq \emptyset$ for each i , in $O(nm)$ time.

Reduction. Vertex Cover is NP-complete by (8.16); we show Vertex Cover $\leq_P$ Hitting Set. Given $G = (V, E)$ and k , set $A = V$, take one set $B_e = \{u, v\}$ for each edge $e = uv \in E$, and keep k ; this is $O(|V| + |E|)$ time. For $H \subseteq V$,

$$\begin{aligned} H \text{ is a hitting set} &\iff \forall e = uv \in E : H \cap \{u, v\} \neq \emptyset \\ &\iff H \text{ is a vertex cover of } G, \end{aligned}$$

so G has a vertex cover of size at most k iff the collection $\{B_e\}$ has a hitting set of size at most k . By (8.14), Hitting Set is NP-complete.

Problem 8.6. Consider an instance of the Satisfiability Problem, specified by clauses $C_1, \dots, C_k$ over a set of Boolean variables $x_1, \dots, x_n$. We say that the instance is *monotone* if each term in each clause consists of a nonnegated variable; that is, each term is equal to x_i , for some i , rather than $\bar{x}_i$. Monotone instances of Satisfiability are very easy to solve: they are always satisfiable, by setting each variable equal to 1.

For example, suppose we have the three clauses

$$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

This is monotone, and indeed the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1 and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

Given a monotone instance of Satisfiability, together with a number k , the problem of *Monotone Satisfiability with Few True Variables* asks: Is there a satisfying assignment for the instance in which at most k variables are set to 1? Prove this problem is NP-complete.

Solution. Reduce Vertex Cover (NP-complete by (8.16)) to it: one variable per vertex, one clause $(x_u \vee x_v)$ per edge.

Membership in NP. The certificate is an assignment; we check in polynomial time that every clause contains a variable set to 1 and that at most k variables are 1.

Reduction. Given an instance $(G = (V, E), k)$ of Vertex Cover, introduce a variable x_v for each $v \in V$ and a monotone clause $C_e = (x_u \vee x_v)$ for each edge $e = (u, v) \in E$; keep the same k . This takes $O(|V| + |E|)$ time. An assignment corresponds to the set $S = \{v : x_v = 1\}$, and

$$\begin{aligned} \text{all } C_e \text{ satisfied} &\iff \forall (u, v) \in E : x_u = 1 \text{ or } x_v = 1 \\ &\iff \forall (u, v) \in E : u \in S \text{ or } v \in S \iff S \text{ is a vertex cover,} \end{aligned}$$

with exactly $|S|$ variables true. Hence a satisfying assignment with at most k true variables exists if and only if G has a vertex cover of size at most k . So Vertex Cover $\leq_P$ Monotone Satisfiability with Few True Variables, and the problem, being in NP, is NP-complete by (8.14).

Problem 8.7. Since the 3-Dimensional Matching Problem is NP-complete, it is natural to expect that the corresponding 4-Dimensional Matching Problem is at least as hard. Let us define *4-Dimensional Matching* as follows. Given sets W, X, Y , and Z , each of size n , and a collection C of ordered 4-tuples of the form (w_i, x_j, y_k, z_ℓ) , do there exist n 4-tuples from C so that no two have an element in common?

Prove that 4-Dimensional Matching is NP-complete.

Solution. Reduce 3-Dimensional Matching (NP-complete by (8.20)) to it by prefixing every triple with every possible w .

Membership in NP. The certificate is a list of n tuples of C ; we check in polynomial time that they are pairwise disjoint.

Reduction. Given an instance of 3-Dimensional Matching, namely sets X, Y, Z of size n and a set $T \subseteq X \times Y \times Z$, let $W = \{w_1, \dots, w_n\}$ be n new elements and

$$C = \{(w_i, x, y, z) : 1 \leq i \leq n, (x, y, z) \in T\},$$

which has $n|T|$ tuples and is built in polynomial time.

(i) If $t_1, \dots, t_n \in T$ form a 3-dimensional matching, then $(w_1, t_1), \dots, (w_n, t_n)$ are n tuples of C that are pairwise disjoint: they differ in the W coordinate, and the triples are disjoint.

(ii) If n pairwise disjoint tuples (w_{i_r}, t_r) , $r = 1, \dots, n$, lie in C , then each $t_r \in T$ and the t_r are pairwise disjoint in their X, Y, Z coordinates, so $t_1, \dots, t_n$ is a 3-dimensional matching.

Thus 3-Dimensional Matching $\leq_P$ 4-Dimensional Matching, and 4-Dimensional Matching, being in NP,

| is NP-complete by (8.14).

8.2 Exercises 8.8–8.14

Problem 8.8. Your friends’ preschool-age daughter Madison has recently learned to spell some simple words. To help encourage this, her parents got her a colorful set of refrigerator magnets featuring the letters of the alphabet (some number of copies of the letter A, some number of copies of the letter B, and so on), and the last time you saw her the two of you spent a while arranging the magnets to spell out words that she knows.

Somehow with you and Madison, things always end up getting more elaborate than originally planned, and soon the two of you were trying to spell out words so as to use up all the magnets in the full set—that is, picking words that she knows how to spell, so that once they were all spelled out, each magnet was participating in the spelling of exactly one of the words. (Multiple copies of words are okay here; so for example, if the set of refrigerator magnets includes two copies each of C, A, and T, it would be okay to spell out CAT twice.)

This turned out to be pretty difficult, and it was only later that you realized a plausible reason for this. Suppose we consider a general version of the problem of *Using Up All the Refrigerator Magnets*, where we replace the English alphabet by an arbitrary collection of symbols, and we model Madison’s vocabulary as an arbitrary set of strings over this collection of symbols. The goal is the same as in the previous paragraph.

Prove that the problem of Using Up All the Refrigerator Magnets is NP-complete.

Solution. Reduce 3-Dimensional Matching (NP-complete by (8.20)): make every element a symbol with one magnet, and every triple a three-letter word.

Membership in NP. Let N be the total number of magnets and c_a the number of copies of symbol a . The empty word uses no magnet and can be discarded, so every word used has length at least 1 and appears at most N times. The certificate is a multiplicity $m_s \in \{0, \dots, N\}$ for each word s of the vocabulary. The verifier checks, for each symbol a ,

$$\sum_s m_s \cdot \#_a(s) = c_a,$$

where $\#_a(s)$ is the number of occurrences of a in s . This is polynomial in the input size even if the counts c_a are written in binary.

Reduction. Given an instance X, Y, Z (each of size n , pairwise disjoint) and $T \subseteq X \times Y \times Z$ of 3-Dimensional Matching, let the symbols be $X \cup Y \cup Z$, with exactly one magnet for each symbol. For each triple $(x, y, z) \in T$, put the three-letter string xyz in the vocabulary. This takes $O(n + |T|)$ time.

(i) If $t_1, \dots, t_n$ is a 3-dimensional matching, spelling the words of $t_1, \dots, t_n$ uses $3n$ magnets with no symbol twice, so it uses each of the $3n$ magnets exactly once.

(ii) Conversely, take a spelling that uses every magnet exactly once. Each symbol has a single copy, so no word is used twice and the words used are pairwise disjoint triples of T . Each uses 3 of the $3n$ magnets, so there are exactly n of them, and together they cover every element of $X \cup Y \cup Z$. That is a 3-dimensional matching.

Hence 3-Dimensional Matching $\leq_P$ Using Up All the Refrigerator Magnets, which is therefore NP-complete by (8.14).

Problem 8.9. Consider the following problem. You are managing a communication network, modeled by a directed graph $G = (V, E)$. There are c users who are interested in making use of this network. User i (for each $i = 1, 2, \dots, c$) issues a *request* to reserve a specific path P_i in G on which to transmit data.

You are interested in accepting as many of these path requests as possible, subject to the following restriction: if you accept both P_i and P_j , then P_i and P_j cannot share any nodes.

Thus, the *Path Selection Problem* asks: Given a directed graph $G = (V, E)$, a set of requests

$P_1, P_2, \dots, P_c$ —each of which must be a path in G —and a number k , is it possible to select at least k of the paths so that no two of the selected paths share any nodes? Prove that Path Selection is NP-complete.

Solution. Reduce Set Packing (NP-complete by (8.16)): each set becomes a path that starts at a private node and then runs through its elements in increasing order.

Membership in NP. The certificate is a list of k of the requests; we check in polynomial time that their node sets are pairwise disjoint.

Reduction. Let a Set Packing instance have ground set $U = \{u_1, \dots, u_n\}$, sets $S_1, \dots, S_m \subseteq U$ and a number k . Build the directed graph G on the nodes $U \cup \{s_1, \dots, s_m\}$. For each i , write $S_i = \{u_{a_1}, \dots, u_{a_r}\}$ with $a_1 < \dots < a_r$, and add the edges and the request

$$P_i = s_i \rightarrow u_{a_1} \rightarrow u_{a_2} \rightarrow \dots \rightarrow u_{a_r}.$$

(If $S_i = \emptyset$, then P_i is the single node s_i .) Keep the same k . All edges go from some s_i to some u , or from u_a to u_b with $a < b$, so each P_i is a genuine path in G . The construction takes time $O(m + n + \sum_i |S_i|)$. The node set of P_i is $\{s_i\} \cup S_i$, and the nodes s_i are distinct, so for $i \neq j$

$$P_i \cap P_j = S_i \cap S_j.$$

Hence a family of requests is pairwise node-disjoint exactly when the corresponding sets are pairwise disjoint, and k disjoint paths exist if and only if k disjoint sets exist. So Set Packing $\leq_P$ Path Selection, and Path Selection, being in NP, is NP-complete by (8.14).

Problem 8.10. Your friends at WebExodus have recently been doing some consulting work for companies that maintain large, publicly accessible Web sites—contractual issues prevent them from saying which ones—and they’ve come across the following *Strategic Advertising Problem*.

A company comes to them with the map of a Web site, which we’ll model as a directed graph $G = (V, E)$. The company also provides a set of t *trails* typically followed by users of the site; we’ll model these trails as directed paths $P_1, P_2, \dots, P_t$ in the graph G (i.e., each P_i is a path in G).

The company wants WebExodus to answer the following question for them: Given G , the paths $\{P_i\}$, and a number k , is it possible to place advertisements on at most k of the nodes in G , so that each path P_i includes at least one node containing an advertisement? We’ll call this the Strategic Advertising Problem, with input G , $\{P_i : i = 1, \dots, t\}$, and k .

Your friends figure that a good algorithm for this will make them all rich; unfortunately, things are never quite this simple.

(a) Prove that Strategic Advertising is NP-complete.

(b) Your friends at WebExodus forge ahead and write a pretty fast algorithm $\mathcal{S}$ that produces yes/no answers to arbitrary instances of the Strategic Advertising Problem. You may assume that the algorithm $\mathcal{S}$ is always correct.

Using the algorithm $\mathcal{S}$ as a black box, design an algorithm that takes input G , $\{P_i\}$, and k as in part (a), and does one of the following two things:

- Outputs a set of at most k nodes in G so that each path P_i includes at least one of these nodes, or
- Outputs (correctly) that no such set of at most k nodes exists.

Your algorithm should use at most a polynomial number of steps, together with at most a polynomial number of calls to the algorithm $\mathcal{S}$.

Solution. (a) Reduce Vertex Cover (NP-complete by (8.16)): orient each edge and make it a one-edge trail.

Membership in NP. The certificate is a set A of at most k nodes; we check in polynomial time that every P_i contains a node of A .

Reduction. Given an undirected graph $H = (V, E)$ and k , number the vertices $v_1, \dots, v_n$. Let G be the

directed graph on V with an edge $v_a \rightarrow v_b$ for each $\{v_a, v_b\} \in E$ with $a < b$. For each such edge, take the trail $P_e = v_a \rightarrow v_b$, a path in G , and keep the same k . This takes linear time. Since P_e has node set $\{v_a, v_b\} = e$,

$$A \text{ meets every } P_e \iff A \text{ meets every edge of } H \iff A \text{ is a vertex cover of } H.$$

So ad placements on at most k nodes exist if and only if H has a vertex cover of size at most k . Hence Vertex Cover $\leq_P$ Strategic Advertising, which is NP-complete by (8.14).

(b) Find the nodes one at a time: keep a node v exactly when $\mathcal{S}$ confirms that the trails missing v can still be hit with one fewer advertisement. Write $\mathcal{S}(Q, k')$ for the call on G , a subfamily Q of the trails, and the bound k' ; this is a valid instance of the problem.

```

if  $\mathcal{S}(\text{all trails}, k) = \text{no}$ : output "no such set"; stop
 $Q := \text{all trails}; k' := k; A := \text{empty}$ 
while  $Q$  is nonempty:
  for each node  $v$  lying on some trail of  $Q$ :
     $Q_v := \text{trails of } Q \text{ not containing } v$ 
    if  $\mathcal{S}(Q_v, k' - 1) = \text{yes}$ :
       $A := A + \{v\}; Q := Q_v; k' := k' - 1$ 
      break out of the for loop
output  $A$ 

```

Correctness. If the first call says no, the answer “no” is correct. Otherwise we maintain the invariant (I) Q can be hit by some set of at most k' nodes, and A hits every trail not in Q , with $|A| + k' = k$. It holds initially. It is preserved by an update, because $\mathcal{S}(Q_v, k' - 1) = \text{yes}$ says that Q_v can be hit by at most $k' - 1$ nodes, and v hits every trail of $Q \setminus Q_v$.

Each pass of the while loop does find a v . Since $Q \neq \emptyset$, (I) gives a set B , $1 \leq |B| \leq k'$, hitting Q ; make it minimal, so that every $v \in B$ lies on some trail of Q . For such a v , the set $B \setminus \{v\}$ of size at most $k' - 1$ hits Q_v , so $\mathcal{S}(Q_v, k' - 1)$ answers yes. (Also $k' \geq 1$, so the bound passed to $\mathcal{S}$ is never negative.) Each pass removes at least one trail from Q , because v lies on a trail of Q , so the loop ends with $Q = \emptyset$. By (I), A then hits every P_i and $|A| = k - k' \leq k$.

Running time. Each pass decreases k' by 1 and $k' \geq 0$ throughout, so there are at most $\min(k, t)$ passes. Each pass makes at most $|V|$ calls to $\mathcal{S}$ and does polynomial bookkeeping. In total there are at most $1 + |V| \min(k, t)$ calls to $\mathcal{S}$ and polynomially many other steps.

Problem 8.11. The idea of hypertext predates the World Wide Web by decades; even hypertext fiction is a relatively old idea. Rather than being constrained by the linearity of the printed page, you can plot a story that consists of a collection of interlocked virtual “places” joined by virtual “passages.” So a piece of hypertext fiction is really riding on an underlying directed graph, which we model as follows.

View the structure of a piece of hypertext fiction as a directed graph $G = (V, E)$. Each node $u \in V$ contains some text; when the reader is currently at u , he or she can choose to follow any edge out of u , and if the reader chooses $e = (u, v)$, he or she arrives next at the node v . There is a start node $s \in V$ where the reader begins, and an end node $t \in V$; when the reader first reaches t , the story ends. Thus any path from s to t is a valid *plot* of the story. Note that there is not necessarily a way to go back; once you have gone from u to v , you might not be able ever to return to u .

Consider a piece of hypertext fiction built on a graph $G = (V, E)$ in which there are certain crucial *thematic elements*: love, death, war, an intense desire to major in computer science, and so forth. Each thematic element i is represented by a set $T_i \subseteq V$ consisting of the nodes of G at which this theme appears. Given a directed graph G with start node s and end node t , and thematic elements represented by sets $T_1, T_2, \dots, T_k$, the *Plot Fulfillment Problem* asks: is there a path from s to t that contains at least one node from each of the sets T_i ?

Prove that Plot Fulfillment is NP-complete.

Solution. Reduce 3-SAT (NP-complete by (8.15)) to Plot Fulfillment with a “chain of binary choices” graph.

Membership in NP. The certificate is the s - t path; we check in polynomial time that consecutive nodes

are joined by edges, that t occurs only at the end, and that the path meets every T_i . (If repeated nodes are allowed, deleting cycles between consecutive theme hits leaves a witness of length at most $(k+1)|V|$.)

Reduction. Given a 3-SAT instance with variables $x_1, \dots, x_n$ and clauses $C_1, \dots, C_m$, build the directed graph with nodes $u_0, u_1, \dots, u_n$ and v_i^T, v_i^F for $1 \leq i \leq n$, edges

$$u_{i-1} \rightarrow v_i^T \rightarrow u_i, \quad u_{i-1} \rightarrow v_i^F \rightarrow u_i \quad (1 \leq i \leq n),$$

$s = u_0$, $t = u_n$, and one theme per clause:

$$T_j = \{v_i^T : x_i \in C_j\} \cup \{v_i^F : \neg x_i \in C_j\} \quad (1 \leq j \leq m).$$

This has $3n+1$ nodes and $4n$ edges. The graph is acyclic, so every s - t path is simple and meets t only at its end; the s - t paths are exactly the 2^n paths that pass, for each i , through exactly one of v_i^T, v_i^F . Match such a path P with the assignment $x_i = \text{true} \iff v_i^T \in P$. Then

$$\begin{aligned} P \cap T_j \neq \emptyset &\iff \exists i : (x_i \in C_j, v_i^T \in P) \text{ or } (\neg x_i \in C_j, v_i^F \in P) \\ &\iff \text{some literal of } C_j \text{ is true under the assignment.} \end{aligned}$$

So a path meeting every T_j exists if and only if some assignment satisfies every clause, which proves both directions. The construction takes polynomial time, so $3\text{-SAT} \leq_P \text{Plot Fulfillment}$, and Plot Fulfillment is NP-complete. ■

Problem 8.12. Some friends of yours maintain a popular news and discussion site on the Web, and they want to differentiate their visitors into paying and nonpaying customers. A standard way to do this is to make all the content available to customers who pay a monthly subscription fee, while visitors who don't subscribe can still view a subset of the pages. Two simple ways to control access for nonsubscribers are (1) to designate a fixed subset of pages as viewable by nonsubscribers, or (2) to allow any page in principle to be viewable but specify a maximum number of pages that a nonsubscriber can view in a single session. (Assume the site can track the path followed by a visitor through the site.)

Your friends are experimenting with a more subtle way of restricting access. They designate certain subsets of the pages as constituting particular *zones*: for example, a zone for pages on politics, a zone for pages on music, and so forth. A page may belong to more than one zone. As a nonsubscribing user passes through the site, the access policy allows him or her to visit one page from each zone, but an attempt to access a second page from the same zone later in the session is disallowed.

More formally, model the site as a directed graph $G = (V, E)$ whose nodes are Web pages and whose edges are directed hyperlinks. There is a distinguished *entry node* $s \in V$, and there are *zones* $Z_1, \dots, Z_k \subseteq V$. A path P taken by a nonsubscriber is restricted to include at most one node from each zone Z_i .

The *Evasive Path Problem* is: given $G, Z_1, \dots, Z_k, s \in V$, and a *destination node* $t \in V$, is there an s - t path in G that includes at most one node from each zone Z_i ? Prove that Evasive Path is NP-complete.

Solution. Reduce 3-SAT (NP-complete by (8.15)) to Evasive Path. The path picks one literal per clause, and two-node zones forbid it to pick both a literal and its negation.

Membership in NP. The certificate is the s - t path. Deleting any cycle from a walk does not increase how many nodes it has in any zone, so we may take a simple path, with at most $|V|$ nodes. We check its edges and count its nodes in each Z_i in polynomial time.

Reduction. Given clauses $C_1, \dots, C_m$ with $C_j = \ell_{j1} \vee \ell_{j2} \vee \ell_{j3}$, build nodes $u_0, \dots, u_m$ and c_{jp} for $1 \leq j \leq m, 1 \leq p \leq 3$, with edges

$$u_{j-1} \rightarrow c_{jp} \rightarrow u_j \quad (1 \leq j \leq m, 1 \leq p \leq 3),$$

$s = u_0$, $t = u_m$, and one zone $\{c_{jp}, c_{j'p'}\}$ for every pair of occurrences with $\ell_{j'p'} = \neg \ell_{jp}$. There are at

most $\binom{3m}{2}$ zones, so the construction is polynomial. The graph is acyclic, and the s - t paths are exactly the choices of one occurrence c_{jp_j} per clause.

($\Rightarrow$) Given a satisfying assignment, let p_j index a true literal of C_j , and take the path through $c_{1p_1}, \dots, c_{mp_m}$. Two chosen occurrences are both true, so they are never complementary, and the path has at most one node in each zone.

($\Leftarrow$) Given an evasive path through $c_{1p_1}, \dots, c_{mp_m}$, the chosen literals $\{\ell_{jp_j}\}$ contain no complementary pair, since such a pair would put two nodes of one zone on the path. So the assignment that makes every chosen literal true (and every other variable arbitrary) is well defined, and it satisfies every C_j through ℓ_{jp_j} .

Hence $3\text{-SAT} \leq_P \text{Evasive Path}$, and Evasive Path is NP-complete. ■

Problem 8.13. A *combinatorial auction* is a mechanism developed by economists for selling a collection of items to a collection of potential buyers. (The Federal Communications Commission has studied this type of auction for assigning stations on the radio spectrum to broadcasting companies.) Here is a simple type of combinatorial auction. There are n items for sale, labeled $I_1, \dots, I_n$. Each item is indivisible and can be sold to only one person. Now m different people place *bids*: the i th bid specifies a subset S_i of the items and an *offering price* x_i that the bidder is willing to pay for the items in S_i , as a single unit. (We represent this bid as the pair (S_i, x_i) .)

An auctioneer looks at the set of all m bids and chooses to *accept* some of them and to *reject* the others. Each person whose bid i is accepted gets all the items in the corresponding set S_i . So no two accepted bids can specify sets that contain a common item, since that would give the same item to two different people. The auctioneer collects the sum of the offering prices of all accepted bids (this is a one-shot auction, with no opportunity to place further bids), and her goal is to collect as much money as possible.

The problem of *Winner Determination for Combinatorial Auctions* asks: given items $I_1, \dots, I_n$, bids $(S_1, x_1), \dots, (S_m, x_m)$, and a bound B , is there a collection of bids that the auctioneer can accept so as to collect an amount of money that is at least B ?

Example. An auctioneer sells excess computer equipment: four items labeled PC, monitor, printer, and scanner, with three bids

$$\begin{aligned} S_1 &= \{\text{PC, monitor}\}, & S_2 &= \{\text{PC, printer}\}, \\ S_3 &= \{\text{monitor, printer, scanner}\}, \end{aligned}$$

$x_1 = x_2 = x_3 = 1$, and $B = 2$. The answer to this instance is no: any two bids share an item, so the auctioneer can accept at most one bid, for a total of only 1.

Prove that the problem of Winner Determination in Combinatorial Auctions is NP-complete.

Solution. Set Packing (NP-complete by (8.16), defined in Section 8.1) is the special case of Winner Determination in which every price is 1.

Membership in NP. The certificate is the set A of accepted bids. We check in polynomial time that the sets S_i , $i \in A$, are pairwise disjoint and that $\sum_{i \in A} x_i \geq B$.

Reduction. Given a Set Packing instance (a ground set U , subsets $S_1, \dots, S_m$, and a target k), take the items to be U , the bids to be $(S_i, 1)$ for $1 \leq i \leq m$, and $B = k$. A set A of bids is acceptable exactly when $\{S_i : i \in A\}$ is a packing, and its revenue is

$$\sum_{i \in A} x_i = |A|.$$

So revenue at least $B = k$ is possible if and only if there are at least k pairwise disjoint sets among the S_i , and both directions follow. The construction copies the input, so Set Packing $\leq_P$ Winner Determination, and Winner Determination is NP-complete. ■

Problem 8.14. We have seen the Interval Scheduling Problem in Chapters 1 and 4. Here we consider a computationally much harder version of it, which we call *Multiple Interval Scheduling*. As before, you have a processor that is available to run jobs over some period of time (e.g., 9 A.M. to 5 P.M.). People submit jobs to run on the processor, and the processor can work on only one job at any single point in time. Jobs in this model, however, are more complicated than before: each job requires a *set* of intervals of time during which it needs to use the processor. For example, a single job could require the processor from 10 A.M. to 11 A.M., and again from 2 P.M. to 3 P.M. If you accept this job, it ties up your processor during those two hours, but you could still accept jobs that need any other time periods (including the hours from 11 A.M. to 2 P.M.).

You are given a set of n jobs, each specified by a set of time intervals, and you want to answer the following question: for a given number k , is it possible to accept at least k of the jobs so that no two of the accepted jobs have any overlap in time?

Show that Multiple Interval Scheduling is NP-complete.

Solution. Reduce Independent Set (NP-complete by (8.16)) to Multiple Interval Scheduling: give each edge its own time slot, and let each vertex's job use the slots of its incident edges. (The book prints "11 A.M. to 2 A.M." in the statement; 2 P.M. is meant.)

Membership in NP. The certificate is a set of k accepted jobs. We check in polynomial time that no interval of one accepted job meets an interval of another.

Reduction. Given a graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$, $E = \{e_1, \dots, e_m\}$, and a target k , define pairwise disjoint intervals

$$\begin{aligned} I_{e_j} &= [2j, 2j + 1] & (1 \leq j \leq m), \\ J_{v_i} &= [2m + 2i, 2m + 2i + 1] & (1 \leq i \leq n), \end{aligned}$$

and for each vertex v the job

$$\text{job}(v) = \{J_v\} \cup \{I_e : e \in E, v \in e\},$$

with the same k . (J_v only keeps jobs of isolated vertices nonempty.) Because the intervals are pairwise disjoint and J_v belongs to one job only, for $u \neq v$

$$\begin{aligned} \text{job}(u) \text{ and } \text{job}(v) \text{ overlap} &\iff \exists e \in E : u \in e \text{ and } v \in e \\ &\iff (u, v) \in E. \end{aligned}$$

So a set $S \subseteq V$ is independent in G if and only if the jobs $\{\text{job}(v) : v \in S\}$ are pairwise non-overlapping. Hence G has an independent set of size at least k if and only if at least k jobs can be accepted, which gives both directions. The instance has n jobs and $2m + n$ intervals, so Independent Set $\leq_P$ Multiple Interval Scheduling, and Multiple Interval Scheduling is NP-complete. ■

8.3 Exercises 8.15–8.21

Problem 8.15. Your friend Neo needs your help with a problem: he and a few associates need to monitor radio signals at various points on the electromagnetic spectrum. There are n different frequencies that need monitoring, and to do this they have a collection of sensors available. The monitoring problem has two components.

- A set L of m geographic locations at which sensors can be placed.
- A set S of b *interference sources*, each of which blocks certain frequencies at certain locations. Each interference source i is specified by a pair (F_i, L_i) , where F_i is a subset of the frequencies and L_i is a subset of the locations. It signifies that, because of radio interference, a sensor placed at any location in L_i will not be able to receive signals on any frequency in F_i .

A subset $L' \subseteq L$ of locations is *sufficient* if, for each of the n frequencies j , there is some location in L' where frequency j is not blocked by any interference source. Thus, by placing a sensor at each

location of a sufficient set, you can monitor all n frequencies.

They have k sensors, so they want to know whether there is a sufficient set of locations of size at most k . We call this an instance of the *Nearby Electromagnetic Observation Problem*: given frequencies, locations, interference sources, and a parameter k , is there a sufficient set of size at most k ?

Example. Take four frequencies $\{f_1, f_2, f_3, f_4\}$, four locations $\{\ell_1, \ell_2, \ell_3, \ell_4\}$, and three interference sources

$$\begin{aligned}(F_1, L_1) &= (\{f_1, f_2\}, \{\ell_1, \ell_2, \ell_3\}), \\ (F_2, L_2) &= (\{f_3, f_4\}, \{\ell_3, \ell_4\}), \\ (F_3, L_3) &= (\{f_2, f_3\}, \{\ell_1\}).\end{aligned}$$

Then there is a sufficient set of size 2: choose locations ℓ_2 and ℓ_4 (since f_1 and f_2 are not blocked at ℓ_4 , and f_3 and f_4 are not blocked at ℓ_2).

Prove that Nearby Electromagnetic Observation is NP-complete.

Solution. Reduce Set Cover (NP-complete by (8.16)) to Nearby Electromagnetic Observation: the frequencies are the elements, location ℓ_i stands for the set S_i , and frequency j is blocked at exactly the locations whose set misses j .

Membership in NP. The certificate is the set L' itself. For each frequency j and each $\ell \in L'$, we test whether some source (F_i, L_i) has $j \in F_i$ and $\ell \in L_i$. This takes $O(nkb)$ membership tests, so $|L'| \leq k$ and sufficiency are checked in polynomial time.

Reduction. Given a Set Cover instance (a ground set $U = \{1, \dots, n\}$, subsets $S_1, \dots, S_m$, and a target k), take frequencies U , locations $\ell_1, \dots, \ell_m$, the same k , and one interference source per frequency:

$$(F_j, L_j) = (\{j\}, \{\ell_i : j \notin S_i\}) \quad (1 \leq j \leq n).$$

Only source j involves frequency j , so j is blocked at ℓ_i if and only if $j \notin S_i$. For $C \subseteq \{1, \dots, m\}$, this gives

$$\begin{aligned}\{\ell_i : i \in C\} \text{ is sufficient} &\iff \forall j \exists i \in C : j \text{ is not blocked at } \ell_i \\ &\iff \forall j \exists i \in C : j \in S_i \\ &\iff \bigcup_{i \in C} S_i = U.\end{aligned}$$

So a sufficient set of size at most k exists if and only if some at most k of the S_i cover U , which gives both directions. The instance has n sources of total size $O(nm)$, so Set Cover $\leq_P$ Nearby Electromagnetic Observation, and Nearby Electromagnetic Observation is NP-complete. ■

Problem 8.16. The task is to infer a set from how large its intersections with other sets are. The input is a finite set U with $|U| = n$, subsets $A_1, \dots, A_m \subseteq U$, and integers $c_1, \dots, c_m$. The question is whether some $X \subseteq U$ satisfies $|X \cap A_i| = c_i$ for every $i = 1, 2, \dots, m$. This is the *Intersection Inference Problem*, and its input is U , $\{A_i\}$ and $\{c_i\}$.

Prove that Intersection Inference is NP-complete.

Solution. Reduce from 3-Dimensional Matching. Make the triples the ground set, and require that the chosen set of triples contains exactly one triple through each element.

Membership in NP. The certificate is X itself. We can compute every $|X \cap A_i|$ and compare it with c_i in $O(mn)$ time.

Reduction. Take an instance of 3-Dimensional Matching: disjoint sets X_0, Y_0, Z_0 , each of size q , and a set $T \subseteq X_0 \times Y_0 \times Z_0$. This problem is NP-complete by (8.20) in Section 8.6. Build the following instance:

$$\begin{aligned}U &= T, \\ A_e &= \{t \in T : e \text{ is a coordinate of } t\} \quad (e \in X_0 \cup Y_0 \cup Z_0), \\ c_e &= 1.\end{aligned}$$

This gives $n = |T|$ and $m = 3q$ sets, and it is built in $O(q|T|)$ time.

Equivalence. For any $M \subseteq T$,

$$|M \cap A_e| = 1 \quad \forall e \iff \text{every } e \in X_0 \cup Y_0 \cup Z_0 \\ \text{lies in exactly one triple of } M.$$

($\Rightarrow$) Suppose M is a perfect three-dimensional matching. Every element lies in exactly one of its triples, so M itself is a feasible set X for the Intersection Inference instance.

($\Leftarrow$) Suppose $X \subseteq T$ satisfies $|X \cap A_e| = 1$ for all e . Then every element lies in exactly one triple of X . Each triple has exactly one coordinate in X_0 , so counting triples by their X_0 -coordinate gives $|X| = \sum_{e \in X_0} |X \cap A_e| = q$. Therefore X is a set of q triples covering each element exactly once, which is a perfect three-dimensional matching.

So 3-Dimensional Matching $\leq_P$ Intersection Inference. Since the problem is also in NP, Intersection Inference is NP-complete.

Problem 8.17. The input is a directed graph $G = (V, E)$ in which every edge $e \in E$ has a weight w_e , and the weights may be positive or negative. The *Zero-Weight-Cycle Problem* asks whether G has a simple cycle whose edge weights add up to exactly 0. Prove that this problem is NP-complete.

Solution. Reduce from Subset Sum, NP-complete by (8.23): each item becomes a choice between an edge of weight w_i and a detour of weight 0, and one back edge of weight $-W$ closes the cycle.

Membership in NP. The certificate is a sequence of vertices. In polynomial time we check that it is a simple cycle of G and that its weights add up to 0.

Reduction. Let natural numbers $w_1, \dots, w_n$ and a target W be an instance of Subset Sum. Build G with vertices $v_0, \dots, v_n, u_1, \dots, u_n$ and these edges:

$$\begin{aligned} (v_{i-1}, v_i) &\text{ of weight } w_i \quad (i = 1, \dots, n), \\ (v_{i-1}, u_i), (u_i, v_i) &\text{ of weight } 0 \quad (i = 1, \dots, n), \\ (v_n, v_0) &\text{ of weight } -W. \end{aligned}$$

This graph has $2n + 1$ vertices and $3n + 1$ edges. Each weight has the same bit length as an input number (or one bit more for $-W$), so the construction takes polynomial time.

Structure of the cycles. Delete the edge (v_n, v_0) . Every remaining edge goes from layer $i - 1$ to layer i or from u_i to v_i , so $G - (v_n, v_0)$ is acyclic. Hence every cycle of G uses (v_n, v_0) , and what is left of the cycle is a path from v_0 to v_n in that DAG. Such a path goes through every v_i exactly once, and for each i it reaches v_i from v_{i-1} in one of two ways: along the direct edge, or through u_i . Its vertices are all distinct, so the cycle is simple. The simple cycles of G therefore correspond one-to-one to subsets $S \subseteq \{1, \dots, n\}$, where S is the set of indices i at which the direct edge is taken. The cycle for S has weight

$$w(C_S) = \sum_{i \in S} w_i + \sum_{i \notin S} (0 + 0) - W = \sum_{i \in S} w_i - W.$$

Equivalence. (i) Suppose $\sum_{i \in S} w_i = W$ for some S . Then C_S is a simple cycle with $w(C_S) = 0$. (ii) Suppose C is a simple cycle of weight 0. By the structure above, $C = C_S$ for some S , and then $\sum_{i \in S} w_i = W$.

So Subset Sum $\leq_P$ Zero-Weight-Cycle. Since Zero-Weight-Cycle is also in NP, it is NP-complete.

Problem 8.18. A committee has a set $M = \{m_1, \dots, m_n\}$ of n members and has voted on t issues. On each issue every member votes Yes, No, or Abstain. The committee's decision on an issue is affirmative if the number of Yes votes is strictly greater than the number of No votes (abstentions count for neither side), and negative otherwise. A subset $M' \subseteq M$ is *decisive* if, counting only the votes of the members of M' by the same rule, the decision on every issue is the same as the decision of the whole committee.

The *Decisive Subset Problem*: given the full table of votes (each member on each issue) and a number

k , is there a decisive subset with at most k members?

Example: four members, three issues, $k = 2$, with these votes.

Issue	m_1	m_2	m_3	m_4
Issue 1	Yes	Yes	Abstain	No
Issue 2	Abstain	No	No	Abstain
Issue 3	Yes	Abstain	Yes	Yes

Here the answer is yes, because $\{m_1, m_3\}$ is decisive.

Prove that Decisive Subset is NP-complete.

Solution. We reduce Set Cover to it. Each set becomes a member who votes Yes on the issues for its elements and abstains on all the others.

Decisive Subset is in NP: the certificate is the subset M' itself. We check $|M'| \leq k$, then recount every issue using only the votes of M' and compare with the committee's decision, in $O(nt)$ time.

For hardness, Set Cover is NP-complete by (8.16). Take an instance of it: a ground set $U = \{u_1, \dots, u_t\}$, subsets $S_1, \dots, S_m \subseteq U$, and a bound k . If $\bigcup_j S_j \neq U$ the answer is no, and we output a fixed no-instance: one member, one issue on which that member votes Yes, and $k = 0$. The empty set decides that issue negatively, so it is not decisive. Otherwise we build a committee with one member m_j for each set S_j and one issue i for each element u_i :

$$\text{vote of } m_j \text{ on issue } i = \begin{cases} \text{Yes,} & u_i \in S_j, \\ \text{Abstain,} & u_i \notin S_j, \end{cases}$$

and we keep the same bound k . The table has mt entries, so this takes polynomial time.

Nobody ever votes No. So for any $M' \subseteq M$, the subset M' decides issue i affirmatively exactly when some member of M' votes Yes on it:

$$M' \text{ is affirmative on issue } i \iff \exists m_j \in M' \text{ with } u_i \in S_j.$$

Every u_i lies in some S_j , so the whole committee is affirmative on every issue. Hence

$$\begin{aligned} M' \text{ is decisive} &\iff \forall i \exists m_j \in M' : u_i \in S_j \\ &\iff \bigcup_{m_j \in M'} S_j = U. \end{aligned}$$

The map $\{S_j : j \in J\} \mapsto \{m_j : j \in J\}$ preserves size. It gives both directions:

(i) If $\{S_j\}_{j \in J}$ is a set cover with $|J| \leq k$, then $\{m_j\}_{j \in J}$ is a decisive subset of size at most k .

(ii) If M' is a decisive subset with $|M'| \leq k$, then $\{S_j : m_j \in M'\}$ is a set cover of size at most k .

So Set Cover $\leq_P$ Decisive Subset. Since Decisive Subset is in NP, it is NP-complete. ■

Problem 8.19. A port must unload n cannisters of hazardous material, each holding a different chemical, onto m trucks standing on the dock; each truck can carry at most k cannisters. For each of the following two versions, either give a polynomial-time algorithm or prove the problem NP-complete.

(a) Each cannister comes with a specified subset of the trucks that may safely carry it. Can all n cannisters be loaded so that no truck holds more than k and every cannister rides in one of its permitted trucks?

(b) Every cannister may go in any truck, but certain given pairs of cannisters must not share a truck (their contents could react explosively). Can all n cannisters be loaded so that no truck holds more than k and no forbidden pair ends up in the same truck?

| *Solution.* (a) Polynomial time, by a maximum-flow computation. Build a network with source s , a

node c_i for each cannister, a node t_j for each truck, and sink t , with edges

- (s, c_i) of capacity 1 for every cannister i ,
- (c_i, t_j) of capacity 1 whenever truck j may carry cannister i ,
- (t_j, t) of capacity k for every truck j .

Run Ford-Fulkerson and answer yes if and only if the maximum flow has value n .

If a valid loading exists, send one unit along $s \rightarrow c_i \rightarrow t_j \rightarrow t$ for each cannister i placed in truck j : each (s, c_i) and (c_i, t_j) carries at most 1, each (t_j, t) carries the number of cannisters in truck j , which is at most k , so this is a flow of value n . Conversely, if the maximum flow is n , then by (7.14) there is an integer flow f of value n . Each (s, c_i) then carries exactly 1 (the n source edges have total capacity n), so by conservation exactly one edge (c_i, t_j) out of c_i carries 1; put cannister i in that truck j , which is permitted because the edge exists. Truck j receives $f^{\text{in}}(t_j) = f(t_j, t) \leq k$ cannisters, so no truck is overloaded.

The network has $O(n + m)$ nodes and $E = O(nm)$ edges, and the capacities out of s sum to $C = n$, so by (7.5) Ford-Fulkerson runs in $O(EC) = O(n^2m)$ time.

(b) NP-complete. It is in NP: a certificate is the truck assigned to each cannister, and in time linear in n plus the number of forbidden pairs we check that every truck has at most k cannisters and no forbidden pair shares a truck.

For hardness we reduce from 3-Coloring, NP-complete by (8.22) (Section 8.7). Given a graph $G = (V, E)$, create one cannister per node, declare the pair $\{u, v\}$ forbidden for each edge $(u, v) \in E$, and take $m = 3$ trucks with capacity $k = n = |V|$. This is clearly polynomial.

($\Rightarrow$) If G has a 3-coloring $c: V \rightarrow \{1, 2, 3\}$, load cannister v into truck $c(v)$. Adjacent nodes get different colors, so no forbidden pair shares a truck, and each truck holds at most $n = k$ cannisters.

($\Leftarrow$) If a valid loading exists, color each node v by the index of the truck holding cannister v . For every edge $(u, v) \in E$ the pair $\{u, v\}$ is forbidden, so u and v are in different trucks and receive different colors; this is a proper 3-coloring of G .

Hence G is 3-colorable if and only if the constructed instance has a valid loading, so 3-Coloring $\leq_P$ problem (b), and with membership in NP, problem (b) is NP-complete.

Problem 8.20. A clustering instance consists of objects $p_1, \dots, p_n$ together with a distance $d(p_i, p_j)$ for every pair, subject only to $d(p_i, p_i) = 0$, $d(p_i, p_j) > 0$ whenever $i \neq j$, and symmetry $d(p_i, p_j) = d(p_j, p_i)$. Section 4.7 solved (via minimum spanning trees) the version that splits the objects into k sets so as to *maximize* the smallest distance between points lying in different clusters. Consider instead splitting into k sets so as to *minimize* the largest distance between two points lying in the same cluster, i.e. so that no cluster is too wide.

Decision version, the *Low-Diameter Clustering Problem*: given $p_1, \dots, p_n$ with distances as above, an integer k and a bound B , can the objects be partitioned into k sets such that any two points in the same set are at distance at most B ?

Prove that Low-Diameter Clustering is NP-complete.

Solution. Reduce 3-Coloring to it: make edges distance 2, non-edges distance 1, and ask for $k = 3$ clusters of diameter $B = 1$.

The problem is in NP: the certificate is the partition $S_1, \dots, S_k$; checking that it is a partition and that $d(p, q) \leq B$ for every pair inside a common set takes $O(n^2)$ time.

For hardness, 3-Coloring is NP-complete by (8.22); given a graph $G = (V, E)$, build the instance with one object per vertex, $k = 3$, $B = 1$, and

$$d(u, v) = \begin{cases} 0 & u = v, \\ 2 & (u, v) \in E, \\ 1 & u \neq v, (u, v) \notin E. \end{cases}$$

This d is symmetric and positive off the diagonal, so it is a legal instance, built in $O(|V|^2)$ time. For

any set $S \subseteq V$,

$$\max_{u,v \in S} d(u,v) \leq 1 \iff \text{no edge of } G \text{ has both ends in } S,$$

because the only distances exceeding 1 are the edge distances.

($\Rightarrow$) If G has a proper 3-coloring, take S_c to be the vertices of color c for $c = 1, 2, 3$. Each S_c is an independent set, so it has diameter at most $1 = B$.

($\Leftarrow$) If $V = S_1 \cup S_2 \cup S_3$ is a partition into sets of diameter at most 1, then by the display no edge lies inside any S_c . So coloring every vertex of S_c with color c is a proper 3-coloring. (If some S_c is empty, that color is simply unused.)

Hence 3-Coloring $\leq_P$ Low-Diameter Clustering, and Low-Diameter Clustering is NP-complete.

Problem 8.21. In configuring a new office computing system there are k components that must be selected (the operating system, the text-editing software, the e-mail program, and so forth). For the j th component there is a set A_j of options, and a configuration of the system consists of a selection of one element from each of the sets of options $A_1, A_2, \dots, A_k$.

Certain pairs of options from different sets may not be compatible: options $x_i \in A_i$ and $x_j \in A_j$ form an incompatible pair if a single system cannot contain them both. (For example, Linux as the operating system and Microsoft Word as the text editor form an incompatible pair.) A configuration is fully compatible if it consists of elements $x_1 \in A_1, x_2 \in A_2, \dots, x_k \in A_k$ such that none of the pairs (x_i, x_j) is an incompatible pair.

The Fully Compatible Configuration (FCC) Problem: an instance consists of disjoint sets of options $A_1, A_2, \dots, A_k$ and a set P of incompatible pairs (x, y) , where x and y are elements of different sets of options. The problem is to decide whether there exists a fully compatible configuration: a selection of an element from each option set so that no pair of selected elements belongs to P .

Example. Let $k = 3$, with A_1, A_2, A_3 the options for the operating system, the text editor and the e-mail program:

$$\begin{aligned} A_1 &= \{\text{Linux, Windows NT}\}, & A_2 &= \{\text{emacs, Word}\}, \\ A_3 &= \{\text{Outlook, Eudora, rmail}\}, \\ P &= \{(\text{Linux, Word}), (\text{Linux, Outlook}), (\text{Word, rmail})\}. \end{aligned}$$

The answer is yes: $\text{Linux} \in A_1, \text{emacs} \in A_2, \text{rmail} \in A_3$ is a fully compatible configuration. Prove that Fully Compatible Configuration is NP-complete.

Solution. Reduce 3-SAT to FCC by making each clause a component whose options are its literals, and declaring complementary literals incompatible.

FCC $\in \mathcal{NP}$: the certificate is the selection $(x_1, \dots, x_k)$, and we check $x_j \in A_j$ for all j and that none of the $\binom{k}{2}$ selected pairs is in P , in polynomial time.

Hardness: we show 3-SAT $\leq_P$ FCC, where 3-SAT is NP-complete by (8.15). Given clauses $C_1, \dots, C_m$ over variables $x_1, \dots, x_n$, write $C_j = \ell_{j1} \vee \ell_{j2} \vee \ell_{j3}$ and build

$$\begin{aligned} A_j &= \{(j, 1), (j, 2), (j, 3)\} & (j = 1, \dots, m), \\ P &= \{((j, s), (j', s')) : j \neq j', \ell_{js} = \neg \ell_{j's'}\}. \end{aligned}$$

The option sets are disjoint and every pair in P joins different sets; there are $3m$ options and $|P| \leq 9m^2$, so the instance is built in polynomial time.

(i) If ν satisfies every clause, select in each A_j an option (j, s) with ℓ_{js} true under ν . Two selected literals are both true, so they are not complementary, and no selected pair lies in P .

(ii) If $(j, s_j)_{j=1}^m$ is fully compatible, then no two of the selected literals $\ell_{1s_1}, \dots, \ell_{ms_m}$ are complementary (a complementary pair from different clauses would be in P). So we can set every selected literal true, and the remaining variables arbitrarily. Each clause C_j then contains the true literal ℓ_{js_j} , so the formula is satisfied.

Therefore the formula is satisfiable if and only if the FCC instance has a fully compatible configuration. By (8.14), FCC is NP-complete. ■

8.4 Exercises 8.22–8.28

Problem 8.22. Suppose someone gives you a black-box algorithm $\mathcal{A}$ that takes an undirected graph $G = (V, E)$ and a number k , and behaves as follows.

- If G is not connected, it simply returns “ G is not connected.”
- If G is connected and has an independent set of size at least k , it returns “yes.”
- If G is connected and does not have an independent set of size at least k , it returns “no.”

Suppose that $\mathcal{A}$ runs in time polynomial in the size of G and k .

Show how, using calls to $\mathcal{A}$, you could solve the Independent Set Problem in polynomial time: given an arbitrary undirected graph G and a number k , does G contain an independent set of size at least k ?

Solution. Add a new node v joined to every node of G , and ask $\mathcal{A}$ about the resulting connected graph G' with the same k .

```

IndepSet( $G = (V, E)$ ,  $k$ ):
  if  $k \leq 0$ : return yes
  if  $k == 1$ : return yes if  $V$  is nonempty, else no
  if  $k > |V|$ : return no
   $G' = (V + \{v\}, E + \{(v, u) : u \in V\})$     #  $v$  a new node
  return  $\mathcal{A}(G', k)$ 

```

The cases $k \leq 1$ and $k > |V|$ are answered correctly by inspection (the empty set, or any single node, is independent; no set exceeds $|V|$). For $2 \leq k \leq |V|$, G' is connected (every node of V is adjacent to v), so $\mathcal{A}$ answers yes or no, and its answer is the right one for G :

$$S \subseteq V \text{ independent in } G \iff S \text{ independent in } G',$$

$$S \text{ independent in } G', |S| \geq 2 \Rightarrow v \notin S,$$

the second because v is adjacent to every other node of G' . So the independent sets of size at least 2 in G' are exactly those of G , and G has one of size at least k iff G' does.

Running time: G' has $|V| + 1$ nodes and $|E| + |V|$ edges and is built in $O(|V| + |E|)$ time; then one call to $\mathcal{A}$ with $k \leq |V|$, which is polynomial in the size of G (the cap on k matters, since $\mathcal{A}$ is polynomial in the value of k).

Problem 8.23. Given a set of finite binary strings $S = \{s_1, \dots, s_k\}$, a string u is a concatenation over S if it is equal to $s_{i_1}s_{i_2}\dots s_{i_t}$ for some indices $i_1, \dots, i_t \in \{1, \dots, k\}$.

Consider the problem: given two sets of finite binary strings $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$, does there exist a string u that is both a concatenation over A and a concatenation over B ?

A friend claims the problem is in NP, “since I would just have to exhibit such a string u .” This is inadequate: the shortest such u might have length exponential in the input size, and then it would not be a polynomial-size certificate. However, the claim can be turned into a proof of membership in NP. Prove the following statement.

If there is a string u that is a concatenation over both A and B , then there is such a string whose length is bounded by a polynomial in the sum of the lengths of the strings in $A \cup B$.

Solution. A shortest common string u has length at most $L_A L_B$, where $L_A = \sum_i |a_i|$ and $L_B = \sum_j |b_j|$, because a repeated “overhang state” at two A -cut points could be cut out. (We take u nonempty and delete empty strings from A and B , which changes no nonempty concatenation.)

Fix a shortest nonempty common u , an A -factorization $u = a_{i_1} \dots a_{i_s}$ and a B -factorization of it. Let $0 < p_1 < \dots < p_{s-1} < |u|$ be the interior A -cut positions. For a cut p , let $\sigma(p) = 0$ if p is also a B -cut. Otherwise p falls strictly inside a B -piece b_j occupying $u[p - d, p - d + |b_j|)$ with $1 \leq d \leq |b_j| - 1$, and we set $\sigma(p) = (j, d)$.

(i) $\sigma(p) = 0$ is impossible: $u[0, p)$ would be a shorter nonempty common string.

(ii) $\sigma(p) = \sigma(q) = (j, d)$ with $p < q$ is impossible. Let $u = xyz$ with $x = u[0, p)$, $y = u[p, q)$ and $z = u[q, |u|)$. Both p and q are A -cuts, so xz is a concatenation over A . On the B side, $u[0, p - d)$ and

$u[q-d+|b_j|, |u|)$ are concatenations over B , and $u[p-d, p) = u[q-d, q)$ is the length- d prefix of b_j . Hence

$$\begin{aligned} xz &= u[0, p-d) u[p-d, p) u[q, q-d+|b_j|) u[q-d+|b_j|, |u|) \\ &= u[0, p-d) b_j u[q-d+|b_j|, |u|), \end{aligned}$$

which is a concatenation over B . It is nonempty ($p \geq 1$) and shorter than u , contradicting minimality. So the $s-1$ interior cuts carry distinct states of the form (j, d) , and there are only $\sum_j (|b_j|-1) \leq L_B-1$ such states. Therefore

$$s \leq L_B, \quad |u| = \sum_{r=1}^s |a_{i_r}| \leq s \max_i |a_i| \leq L_A L_B \leq \frac{1}{4}(L_A + L_B)^2.$$

which is quadratic in the total input length. ■

Problem 8.24. Let $G = (V, E)$ be a bipartite graph whose nodes are partitioned into sets X and Y so that each edge has one end in X and the other in Y . An (a, b) -skeleton of G is a set of edges $E' \subseteq E$ such that at most a nodes in X are incident to an edge in E' , and at least b nodes in Y are incident to an edge in E' .

Show that, given a bipartite graph G and numbers a and b , it is NP-complete to decide whether G has an (a, b) -skeleton.

Solution. Set Cover is the special case $b = |Y|$: put the sets on side X , the elements on side Y , and join each set to its elements.

Membership in $\mathcal{NP}$: the certificate is E' , and we count the endpoints of E' in X and in Y in $O(|E|)$ time.

Hardness: Set Cover $\leq_P (a, b)$ -Skeleton, where Set Cover is NP-complete by (8.16). Given a ground set U with $|U| = n$, subsets $S_1, \dots, S_m$ and a bound k , let

$$X = \{S_1, \dots, S_m\}, \quad Y = U, \quad E = \{(S_i, u) : u \in S_i\}, \quad a = k, \quad b = n.$$

This is built in time linear in $\sum_i |S_i|$.

(i) If $S_{i_1}, \dots, S_{i_\ell}$ with $\ell \leq k$ cover U , take E' to be all edges at these ℓ nodes. Exactly $\ell \leq a$ nodes of X are incident to E' , and every $u \in U$ lies in some chosen S_{i_r} , so all $n = b$ nodes of Y are incident to E' .

(ii) If E' is an (a, b) -skeleton, let C be the set of nodes of X incident to E' , so $|C| \leq k$. At least $b = n$, hence all, nodes $u \in Y$ are incident to some edge $(S_i, u) \in E'$, and then $S_i \in C$ and $u \in S_i$. So C is a set cover of size at most k .

So the Set Cover instance is a yes-instance iff G has an (a, b) -skeleton. By (8.14), the skeleton problem is NP-complete. ■

Problem 8.25. For functions $g_1, \dots, g_\ell$, define the function $\max(g_1, \dots, g_\ell)$ by

$$[\max(g_1, \dots, g_\ell)](x) = \max(g_1(x), \dots, g_\ell(x)).$$

You are given n piecewise linear, continuous functions $f_1, \dots, f_n$ defined over the interval $[0, t]$ for some integer t , and integers k and B . Decide: do there exist k of the functions $f_{i_1}, \dots, f_{i_k}$ such that

$$\int_0^t [\max(f_{i_1}, \dots, f_{i_k})](x) dx \geq B?$$

Prove that this problem is NP-complete.

Solution. Reduce from Set Cover by giving each element u its own unit interval, and letting f_i have a tent of area 1 over the interval of every element of S_i . Then the integral of the max counts the covered elements.

Membership in $\mathcal{NP}$: the certificate is the index set $\{i_1, \dots, i_k\}$. Let $x_0 < \dots < x_N$ be the union of the breakpoints of the chosen functions (with N at most the input size). On each $[x_{r-1}, x_r]$ all chosen functions are linear, and the at most $\binom{k}{2}$ crossing points of pairs of them, each the solution of a linear equation with rational input coefficients, split it into pieces on which the maximum is a single linear function. The integral is the sum of the exact trapezoid areas of these at most $N(k^2 + 1)$ pieces, computed in polynomial time with rational arithmetic, and we compare it to B .

Hardness: Set Cover $\leq_P$ this problem, where Set Cover is NP-complete by (8.16). Given $U = \{1, \dots, n\}$, sets $S_1, \dots, S_m$ and a bound k , let $t = n$, $B = n$, $k' = \min(k, m)$, and define the tent

$$T_u(x) = 2 \max(0, 1 - |2x - 2u + 1|)$$

on $[0, n]$. It is 0 outside $(u - 1, u)$, rises linearly to height 2 at $x = u - \frac{1}{2}$, and has $\int_0^n T_u = \frac{1}{2} \cdot 1 \cdot 2 = 1$. Set $f_i = \sum_{u \in S_i} T_u$. This function is continuous and piecewise linear with at most $3|S_i| + 2$ integer or half-integer breakpoints, so the instance is polynomial in size.

The tents have disjoint open supports, so on $[u - 1, u]$ each f_i equals T_u if $u \in S_i$ and 0 otherwise. For any index set I ,

$$\int_0^n \max_{i \in I} f_i = \sum_{u=1}^n \int_{u-1}^u \max_{i \in I} f_i = \left| \bigcup_{i \in I} S_i \right| \leq n.$$

So k' functions with integral $\geq B = n$ exist iff k' of the sets cover U . Since adding sets to a cover keeps it a cover, this holds iff some at most k' sets cover U , and since only m sets exist, iff some at most k sets cover U . By (8.14), the problem is NP-complete. ■

Problem 8.26. You and a friend have been trekking through various far-off parts of the world and have accumulated a big pile of souvenirs. At the time you weren't really thinking about which of these you were planning to keep and which your friend was going to keep, but now the time has come to divide everything up.

Here's a way you could go about doing this. Suppose there are n objects, labeled $1, 2, \dots, n$, and object i has an agreed-upon *value* x_i . (We could think of this, for example, as a monetary resale value; the case in which you and your friend don't agree on the value is something we won't pursue here.) One reasonable way to divide things would be to look for a *partition* of the objects into two sets, so that the total value of the objects in each set is the same.

This suggests solving the following *Number Partitioning Problem*. You are given positive integers $x_1, \dots, x_n$; you want to decide whether the numbers can be partitioned into two sets S_1 and S_2 with the same sum:

$$\sum_{x_i \in S_1} x_i = \sum_{x_j \in S_2} x_j.$$

Show that Number Partitioning is NP-complete.

Solution. Reduce Subset Sum (NP-complete by (8.23)) to Number Partitioning by adding two “balancing” numbers that are forced onto opposite sides.

Membership in NP. The certificate is the partition (S_1, S_2) ; adding up both sides in binary and comparing takes polynomial time.

Reduction. Let $w_1, \dots, w_n, W$ be an instance of Subset Sum. Discard every $w_i = 0$ (this changes no subset sum) and let $T = \sum_i w_i$. If $T = 0$, output the yes-instance $\{1, 1\}$ when $W = 0$ and the no-instance $\{1, 2\}$ otherwise; if $W > T$, output $\{1, 2\}$. Otherwise $T \geq 1$ and $0 \leq W \leq T$, and output the positive integers

$$w_1, \dots, w_n, \quad y = 2T - W, \quad z = T + W.$$

Their total is $T + (2T - W) + (T + W) = 4T$, so a valid partition has both sides equal to $2T$. Since $y + z = 3T > 2T$, the numbers y and z lie on opposite sides; the side containing y consists of y and a set A of the w_i , and

$$\sum_{i \in A} w_i + (2T - W) = 2T \iff \sum_{i \in A} w_i = W.$$

(i) If some A has $\sum_{i \in A} w_i = W$, then $S_1 = A \cup \{y\}$ and $S_2 = (\text{rest}) \cup \{z\}$ both sum to $2T$. (ii) If a partition exists, the display gives a subset A with sum W . The construction is polynomial (the new numbers have $O(\log T + \log W)$ bits), so Subset Sum $\leq_P$ Number Partitioning, and Number Partitioning is NP-complete by (8.14). ■

Problem 8.27. Consider the following problem. You are given positive integers $x_1, \dots, x_n$, and numbers k and B . You want to know whether it is possible to *partition* the numbers $\{x_i\}$ into k sets $S_1, \dots, S_k$ so that the squared sums of the sets add up to at most B :

$$\sum_{i=1}^k \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B.$$

Show that this problem is NP-complete.

Solution. Number Partitioning (NP-complete by Exercise 8.26) is the special case $k = 2$, $B = \lfloor S^2/2 \rfloor$ with $S = \sum_j x_j$.

Membership in NP. The certificate assigns each x_j to one of the k sets; the k sums, their squares and the comparison with B are computed in time polynomial in the input length (at most $\min(k, n)$ sets are nonempty, and empty sets contribute 0).

Reduction. Map the Number Partitioning instance $x_1, \dots, x_n$ to $(x_1, \dots, x_n; k = 2, B = \lfloor S^2/2 \rfloor)$. For a split into two sets with sums a and $b = S - a$,

$$a^2 + b^2 = \frac{S^2 + (a - b)^2}{2}.$$

(i) If $a = b$, then S is even and $a^2 + b^2 = S^2/2 = B$. (ii) If $a^2 + b^2 \leq B \leq S^2/2$, then $(a - b)^2 \leq 0$, so $a = b$.

So the numbers have an equal-sum partition if and only if they have a 2-set partition with squared sums at most B . The map is polynomial, so Number Partitioning $\leq_P$ this problem, which is therefore NP-complete by (8.14). ■

Problem 8.28. The following is a version of the Independent Set Problem. You are given a graph $G = (V, E)$ and an integer k . For this problem, we will call a set $I \subset V$ *strongly independent* if, for any two nodes $v, u \in I$, the edge (v, u) does not belong to E , and there is also no path of two edges from u to v , that is, there is no node w such that both $(u, w) \in E$ and $(w, v) \in E$. The *Strongly Independent Set Problem* is to decide whether G has a strongly independent set of size at least k . Prove that the Strongly Independent Set Problem is NP-complete.

Solution. Reduce Independent Set (NP-complete by (8.16)) by subdividing every edge of G and joining all the subdivision nodes into a clique.

Membership in NP. The certificate is I ; check $|I| \geq k$ and, for each pair $u, v \in I$, that $(u, v) \notin E$ and no w is adjacent to both, in $O(|V|^3)$ time.

Reduction. Given (G, k) with $G = (V, E)$, build G' on $V \cup \{w_e : e \in E\}$ with edges

$$\{(u, w_e), (v, w_e) : e = (u, v) \in E\} \cup \{(w_e, w_f) : e \neq f \in E\},$$

and ask for a strongly independent set of size at least k . In G' no two original nodes are adjacent, and the only common neighbours of $u, v \in V$ are nodes w_e with $e = (u, v)$, so

$$u, v \in V \text{ are at distance } \leq 2 \text{ in } G' \iff (u, v) \in E. \quad (*)$$

(i) If I is independent in G , then I is strongly independent in G' by $(*)$. (ii) Let I' be strongly independent in G' with $|I'| \geq k$. If $I' \subseteq V$, it is independent in G by $(*)$. Otherwise I' contains exactly one w_e (the w 's form a clique), say $e = (u, v)$; every node of positive degree in G is adjacent to some

w_f and so within distance 2 of w_e , so $I' \setminus \{w_e\}$ consists of isolated nodes of G , and $(I' \setminus \{w_e\}) \cup \{u\}$ is an independent set of G of the same size. G' has $|V| + |E|$ nodes and $O(|E|^2)$ edges, so Independent Set $\leq_P$ Strongly Independent Set, which is NP-complete by (8.14). ■

8.5 Exercises 8.29–8.35

Problem 8.29. You’re configuring a large network of workstations, which we’ll model as an undirected graph G ; the nodes of G represent individual workstations and the edges represent direct communication links. The workstations all need access to a common *core database*, which contains data necessary for basic operating system functions.

You could replicate this database on each workstation; this would make lookups very fast from any workstation, but you’d have to manage a huge number of copies. Alternately, you could keep a single copy of the database on one workstation and have the remaining workstations issue requests for data over the network G ; but this could result in large delays for a workstation that’s many hops away from the site of the database.

So you decide to look for the following compromise: You want to maintain a small number of copies, but place them so that any workstation either has a copy of the database or is connected by a direct link to a workstation that has a copy of the database. In graph terminology, such a set of locations is called a *dominating set*.

Thus we phrase the *Dominating Set Problem* as follows. Given the network G , and a number k , is there a way to place k copies of the database at k different nodes so that every node either has a copy of the database or is connected by a direct link to a node that has a copy of the database? Show that Dominating Set is NP-complete.

Solution. Reduce Vertex Cover (NP-complete by (8.16)) by attaching to every edge (u, v) a new node adjacent to exactly u and v . (A superset of a dominating set dominates, so for $k \leq |V|$ “exactly k copies” and “at most k copies” are the same question; for $k > |V|$ the answer is no.)

Membership in NP. The certificate is the set D of k nodes; checking that each node lies in D or has a neighbour in D takes $O(|V| + |E|)$ time.

Reduction. Let (G, k) ask for a vertex cover of size at most k . Deleting isolated nodes of G changes no vertex cover, so assume $G = (V, E)$ has none, and if $k \geq |V|$ output a trivial yes-instance. Otherwise build G' on $V \cup \{w_e : e \in E\}$ with edge set

$$E \cup \{(u, w_e), (v, w_e) : e = (u, v) \in E\},$$

and ask for a dominating set of size k in G' .

(i) Let C be a vertex cover of G with $|C| \leq k$. Each w_e is adjacent to an endpoint of e , which lies in C ; each $v \in V \setminus C$ has an edge (v, u) (no isolated nodes), and $u \in C$ since C covers it. So C dominates G' , and padding it gives k nodes. (ii) Let D dominate G' with $|D| = k$. Replace each $w_e \in D$, $e = (u, v)$, by u : since $N[w_e] = \{w_e, u, v\} \subseteq N[u]$, the result $D' \subseteq V$ still dominates and $|D'| \leq k$. For each $e = (u, v)$, the node $w_e \notin D'$ has neighbours only u, v , so $u \in D'$ or $v \in D'$; thus D' is a vertex cover of G of size at most k .

G' has $|V| + |E|$ nodes and $3|E|$ edges, so Vertex Cover $\leq_P$ Dominating Set, which is NP-complete by (8.14). ■

Problem 8.30. One thing that’s not always apparent when thinking about traditional “continuous math” problems is the way discrete, combinatorial issues of the kind we’re studying here can creep into what look like standard calculus questions.

Consider, for example, the traditional problem of minimizing a one-variable function like $f(x) = 3 + x - 3x^2$ over an interval like $x \in [0, 1]$. The derivative has a zero at $x = 1/6$, but this in fact is a maximum of the function, not a minimum; to get the minimum, one has to heed the standard warning to check the values on the boundary of the interval as well. (The minimum is in fact achieved

on the boundary, at $x = 1$.)

Checking the boundary isn't such a problem when you have a function in one variable; but suppose we're now dealing with the problem of minimizing a function in n variables $x_1, x_2, \dots, x_n$ over the unit cube, where each of $x_1, x_2, \dots, x_n \in [0, 1]$. The minimum may be achieved on the interior of the cube, but it may be achieved on the boundary; and this latter prospect is rather daunting, since the boundary consists of 2^n "corners" (where each x_i is equal to either 0 or 1) as well as various pieces of other dimensions. Calculus books tend to get suspiciously vague around here, when trying to describe how to handle multivariable minimization problems in the face of this complexity.

It turns out there's a reason for this: Minimizing an n -variable function over the unit cube in n dimensions is as hard as an NP-complete problem. To make this concrete, let's consider the special case of polynomials with integer coefficients over n variables $x_1, x_2, \dots, x_n$. To review some terminology, we say a *monomial* is a product of a real-number coefficient c and each variable x_i raised to some nonnegative integer power a_i ; we can write this as $cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n}$. (For example, $2x_1^2x_2x_3^4$ is a monomial.) A *polynomial* is then a sum of a finite set of monomials. (For example, $2x_1^2x_2x_3^4 + x_1x_3 - 6x_2^2x_3^2$ is a polynomial.)

We define the *Multivariable Polynomial Minimization Problem* as follows: Given a polynomial in n variables with integer coefficients, and given an integer bound B , is there a choice of real numbers $x_1, x_2, \dots, x_n \in [0, 1]$ that causes the polynomial to achieve a value that is $\leq B$?

Choose a problem Y from this chapter that is known to be NP-complete and show that

$$Y \leq_P \text{Multivariable Polynomial Minimization.}$$

Solution. Take $Y = \text{Independent Set}$ (NP-complete by (8.16)) and map (G, k) , $G = (V, E)$ with $V = \{1, \dots, n\}$, to the polynomial and bound

$$p(x) = \sum_{(i,j) \in E} x_i x_j - \sum_{i=1}^n x_i, \quad B = -k.$$

For $S \subseteq V$ with indicator vector χ_S , writing $e(S)$ for the number of edges with both ends in S ,

$$p(\chi_S) = e(S) - |S|.$$

(i) If G has an independent set I with $|I| \geq k$, then $p(\chi_I) = -|I| \leq -k = B$.

(ii) Suppose $p(x) \leq B$ for some $x \in [0, 1]^n$. Each monomial of p has degree at most 1 in each variable, so with the other coordinates fixed p is affine in x_i and is no larger at one of $x_i = 0$, $x_i = 1$ than at the current value. Rounding the coordinates one at a time this way gives a corner χ_S with $e(S) - |S| \leq p(x) \leq -k$. While S contains an edge, delete one of its endpoints: $|S|$ drops by 1 and $e(S)$ by at least 1, so $e(S) - |S|$ does not increase. The process ends at an independent set I with

$$-|I| = e(I) - |I| \leq e(S) - |S| \leq -k, \quad \text{i.e. } |I| \geq k.$$

The polynomial has integer coefficients, $|E| + n$ monomials, and is written down in linear time, so Independent Set $\leq_P$ Multivariable Polynomial Minimization. ■

Problem 8.31. Given an undirected graph $G = (V, E)$, a feedback set is a set $X \subseteq V$ with the property that $G - X$ has no cycles. The Undirected Feedback Set Problem asks: Given G and k , does G contain a feedback set of size at most k ? Prove that Undirected Feedback Set is NP-complete.

Solution. Reduce Vertex Cover to Undirected Feedback Set by putting every edge of G on its own triangle.

Membership in $\mathcal{NP}$: the certificate is X ; we check $|X| \leq k$ and that $G - X$ is acyclic (by breadth-first search, a graph is acyclic iff each component with c nodes has $c - 1$ edges), in polynomial time.

Hardness: Vertex Cover is NP-complete by (8.16). Given (G, k) with $G = (V, E)$, build G' by adding,

for each edge $e = (u, v) \in E$, a new node w_e joined to both u and v :

$$\begin{aligned} V' &= V \cup \{w_e : e \in E\}, \\ E' &= E \cup \{(u, w_e), (v, w_e) : e = (u, v) \in E\}, \end{aligned}$$

and ask for a feedback set of size at most k . This takes $O(|V| + |E|)$ time.

(i) If C is a vertex cover of G with $|C| \leq k$, then C is a feedback set of G' . In $G' - C$ the nodes of $V - C$ form an independent set of G (every edge of G has an end in C), so the only edges left are of the form (x, w_e) . Each w_e is adjacent only to the two ends of e , and at least one of them is in C , so w_e has degree at most 1 in $G' - C$. Every node on a cycle has degree at least 2, yet every edge of $G' - C$ has an end of degree at most 1, so $G' - C$ has no cycle.

(ii) If X is a feedback set of G' with $|X| \leq k$, set

$$C = (X \cap V) \cup \{u_e : w_e \in X\}, \quad u_e = \text{one fixed end of } e.$$

Then $|C| \leq |X| \leq k$. For each $e = (u, v) \in E$ the triangle u, v, w_e is a cycle of G' , so X contains u , v or w_e , and in each case C contains an end of e . Hence C is a vertex cover of G .

Therefore G has a vertex cover of size at most k if and only if G' has a feedback set of size at most k . By (8.14), Undirected Feedback Set is NP-complete. ■

Problem 8.32. Each of an organism's chromosomes can be viewed as an extremely long string over the four-letter alphabet $\{A, C, G, T\}$; one approach to genome mapping generates many short, overlapping snippets of a chromosome and infers the full string from them. Here is a simplified problem in this spirit.

Suppose we have a set $S = \{s_1, s_2, \dots, s_n\}$ of short DNA strings over a q -letter alphabet; each string s_i has length 2ℓ , for some number $\ell \geq 1$. We also have a library of additional strings $T = \{t_1, t_2, \dots, t_m\}$ over the same alphabet; each of these also has length 2ℓ . In trying to assess whether the string s_b might come directly after the string s_a in the chromosome, we look to see whether the library T contains a string t_k so that the first ℓ symbols in t_k are equal to the last ℓ symbols in s_a , and the last ℓ symbols in t_k are equal to the first ℓ symbols in s_b . If this is possible, we say that t_k corroborates the pair (s_a, s_b) . (In other words, t_k could be a snippet of DNA that straddled the region in which s_b directly followed s_a .)

Now we'd like to concatenate all the strings in S in some order, one after the other with no overlaps, so that each consecutive pair is corroborated by some string in the library T . That is, we'd like to order the strings in S as $s_{i_1}, s_{i_2}, \dots, s_{i_n}$, where $i_1, i_2, \dots, i_n$ is a permutation of $\{1, 2, \dots, n\}$, so that for each $j = 1, 2, \dots, n - 1$, there is a string t_k that corroborates the pair $(s_{i_j}, s_{i_{j+1}})$. (The same string t_k can be used for more than one consecutive pair in the concatenation.) If this is possible, we say that the set S has a perfect assembly.

Given sets S and T , the Perfect Assembly Problem asks: Does S have a perfect assembly with respect to T ? Prove that Perfect Assembly is NP-complete.

Example. Suppose the alphabet is $\{A, C, G, T\}$, the set $S = \{AG, TC, TA\}$, and the set $T = \{AC, CA, GC, GT\}$ (so each string has length $2\ell = 2$). Then the answer to this instance of Perfect Assembly is yes: we can concatenate the three strings in S in the order $TCAGTA$ (so $s_{i_1} = s_2$, $s_{i_2} = s_1$, and $s_{i_3} = s_3$). In this order, the pair (s_{i_1}, s_{i_2}) is corroborated by the string CA in the library T , and the pair (s_{i_2}, s_{i_3}) is corroborated by the string GT in the library T .

Solution. Reduce Hamiltonian Path to Perfect Assembly by making each node a one-letter symbol, each node the string of that letter doubled, and each directed edge a two-letter library string.

Membership in $\mathcal{NP}$: the certificate is the permutation $i_1, \dots, i_n$; for each of the $n - 1$ consecutive pairs we scan T for a corroborating string, in $O(nm\ell)$ time.

Hardness: Hamiltonian Path (on directed graphs) is NP-complete by (8.19). Given a directed graph $G = (V, E)$ with $V = \{1, \dots, n\}$, take the alphabet V (so $q = n$), $\ell = 1$, and

$$S = \{s_v = vv : v \in V\}, \quad T = \{t_{(u,v)} = uv : (u, v) \in E\}.$$

This is built in $O(|V| + |E|)$ time. With $\ell = 1$, a string $xy \in T$ corroborates (s_a, s_b) iff x equals the last symbol a of $s_a = aa$ and y equals the first symbol b of $s_b = bb$, so

$$\begin{aligned} \text{some } t \in T \text{ corroborates } (s_a, s_b) &\iff ab \in T \\ &\iff (a, b) \in E. \end{aligned}$$

Hence an ordering $s_{v_1}, s_{v_2}, \dots, s_{v_n}$ of S is a perfect assembly iff $(v_j, v_{j+1}) \in E$ for $j = 1, \dots, n-1$, that is, iff $v_1, v_2, \dots, v_n$ is a Hamiltonian path in G . Since orderings of S correspond one-to-one to orderings of V , S has a perfect assembly with respect to T if and only if G has a Hamiltonian path. By (8.14), Perfect Assembly is NP-complete. ■

Problem 8.33. In a barter economy, people trade goods and services directly, without money as an intermediate step. Trades happen when each party views the set of goods they're getting as more valuable than the set of goods they're giving in return. One difficulty with barter relative to money-based systems is the complexity of identifying opportunities for trading, even when these opportunities exist.

To model this, there is a set of n people $p_1, \dots, p_n$ and a set of m distinct objects $a_1, \dots, a_m$. Each object is owned by one of the people. Each person p_i has a valuation function v_i , defined so that $v_i(a_j)$ is a nonnegative number that specifies how much object a_j is worth to p_i (the larger the number, the more valuable the object is to the person). Everyone assigns a valuation to each object, including the ones they don't currently possess, and different people can assign very different valuations to the same object.

A two-person trade is possible when there are people p_i and p_j , and subsets of objects A_i and A_j possessed by p_i and p_j , respectively, so that each person would prefer the objects in the subset they don't currently have. More precisely,

- p_i 's total valuation for the objects in A_j exceeds his or her total valuation for the objects in A_i , and
 - p_j 's total valuation for the objects in A_i exceeds his or her total valuation for the objects in A_j .
- (A_i doesn't have to be all the objects possessed by p_i , and likewise for A_j ; A_i and A_j can be arbitrary subsets of their possessions that meet these criteria.)

Suppose you are given an instance of a barter economy, specified by the above data on people's valuations for objects, where each valuation is a natural number. Prove that the problem of determining whether a two-person trade is possible is NP-complete.

Solution. Reduce Subset Sum to the trade problem with one person who owns the numbers as objects and one person who owns a single object that both of them value near the target W .

Membership in $\mathcal{NP}$: the certificate is (i, j, A_i, A_j) ; we check $A_i \subseteq$ possessions of p_i , $A_j \subseteq$ possessions of p_j , and the two strict inequalities by adding at most $2m$ binary numbers, in polynomial time.

Hardness: Subset Sum (natural numbers $w_1, \dots, w_n$ and a target W : does some subset sum to exactly W ?) is NP-complete by (8.23). If $W = 0$ the answer is yes (empty subset), and we output a fixed instance that has a trade: p_1 owns a and p_2 owns b , with $v_1(a) = 0, v_1(b) = 1, v_2(a) = 1, v_2(b) = 0$. If $W \geq 1$, make two people, let p_1 own objects $a_1, \dots, a_n$ and p_2 own one object b , and set

$$v_1(a_i) = v_2(a_i) = w_i, \quad v_1(b) = W + 1, \quad v_2(b) = W - 1.$$

All values are natural numbers with $O(\log W + \log w_i)$ bits. Write $w(A) = \sum_{a_i \in A} w_i$. A trade gives some $A \subseteq \{a_1, \dots, a_n\}$ against some $B \subseteq \{b\}$. If $B = \emptyset$, then p_1 would need $0 > w(A)$, which is impossible; so $B = \{b\}$, and the trade conditions read

$$\begin{aligned} v_1(b) > v_1(A) &\iff W + 1 > w(A) \iff w(A) \leq W, \\ v_2(A) > v_2(b) &\iff w(A) > W - 1 \iff w(A) \geq W, \end{aligned}$$

using that $w(A)$ and W are integers. So a two-person trade is possible if and only if some A has $w(A) = W$, that is, iff the Subset Sum instance is a yes-instance. By (8.14), the trade problem is NP-complete. ■

Problem 8.34. Researchers in the mathematical social sciences (including Mark Granovetter and Thomas Schelling) proposed cascade processes as models of collective behavior: an individual's behavior is highly influenced by the behaviors of his or her friends, so if a few individuals instigate the process, it can spread to more and more people.

The most basic version of their models is the following. There is some underlying behavior, and at any point in time each person is either an adopter of the behavior or a nonadopter. We represent the population by a directed graph $G = (V, E)$ in which the nodes correspond to people and there is an edge (v, w) if person v has influence over the behavior of person w . Each person w also has a given threshold $\theta(w) \in [0, 1]$, with the following meaning: at any time when at least a $\theta(w)$ fraction of the nodes with edges to w are adopters of the behavior, the node w becomes an adopter as well. A node w with $\theta(w) = 0$ adopts the behavior immediately. Nodes with no incoming edges become adopters if $\theta(w) = 0$, and cannot become adopters if they have any larger threshold. The spread is simulated as follows.

```
Initially, set all nodes  $w$  with  $\theta(w) = 0$  to be adopters
(All other nodes start out as nonadopters)
Until there is no change in the set of adopters:
  For each nonadopter  $w$  simultaneously:
    If at least a  $\theta(w)$  fraction of nodes with edges to  $w$ 
      are adopters then
         $w$  becomes an adopter
    Endif
  Endfor
End
Output the final set of adopters
```

This process terminates, since there are only n individuals total, and at least one new person becomes an adopter in each iteration.

In viral marketing, we choose a set of nodes $S \subseteq V$ and reset the threshold of each node in S to 0. We then run the process above, and denote the size of the final set of adopters by $f(S)$, the influence of S . The Influence Maximization Problem is: given a directed graph $G = (V, E)$, with a threshold value at each node, and parameters k and b , is there a set S of at most k nodes for which $f(S) \geq b$? Prove that Influence Maximization is NP-complete.

Example. Suppose G has five nodes $\{a, b, c, d, e\}$, four edges $(a, b), (b, c), (e, d), (d, c)$, and all node thresholds equal to $2/3$. Then the answer to the Influence Maximization instance defined by G , with $k = 2$ and $b = 5$, is yes: we can choose $S = \{a, e\}$, and this will cause the other three nodes to become adopters as well. This is the only choice of S that works: if we choose $S = \{a, d\}$, then b and c become adopters, but e won't; if we choose $S = \{a, b\}$, then none of c, d , or e become adopters.

Solution. Reduce Vertex Cover to Influence Maximization with a two-layer graph: a node x_v of threshold 1 for each vertex, and a node y_e of threshold $1/2$ fed by the two ends of each edge, asking for $b = k + |E|$. Membership in $\mathcal{NP}$: the certificate is S ; we check $|S| \leq k$, run the simulation (at most n rounds, each a pass over the edges with one rational comparison per node) and check $f(S) \geq b$, all in polynomial time.

Hardness: Vertex Cover is NP-complete by (8.16). Let (H, k) be an instance with $H = (U, F)$, $|U| = n$, $|F| = m$. If $k \geq n$, then U is a cover and we output any fixed yes-instance; so assume $k \leq n$. Build the directed graph G on $X \cup Y$, where

$$\begin{aligned} X &= \{x_u : u \in U\}, & \theta(x_u) &= 1, \\ Y &= \{y_e : e \in F\}, & \theta(y_e) &= 1/2, \end{aligned}$$

with edges $(x_u, y_e), (x_v, y_e)$ for each $e = (u, v) \in F$, and ask whether some $|S| \leq k$ has $f(S) \geq k + m$. Each x_u has no incoming edges and positive threshold, so it is an adopter iff $x_u \in S$; each y_e has exactly two in-neighbours, so it adopts iff $y_e \in S$ or an end of e is in $S \cap X$. With $C_S = \{u : x_u \in S\}$ this gives

$$\begin{aligned} f(S) &= |S \cap X| + |\{e \in F : y_e \in S \text{ or } e \text{ has an end in } C_S\}| \\ &\leq |S \cap X| + m. \end{aligned}$$

- (i) If C is a vertex cover of H with $|C| \leq k$, enlarge it to a cover of size exactly k (possible as $k \leq n$) and take $S = \{x_u : u \in C\}$. Every y_e adopts, so $f(S) = k + m$.
- (ii) If $|S| \leq k$ and $f(S) \geq k + m$, the display gives $|S \cap X| \geq k$, so $S \subseteq X$ and $|C_S| = k$. Then no y_e is in S , and $f(S) = k + m$ forces every y_e to adopt, i.e. every edge has an end in C_S . So C_S is a vertex cover of size k .

The construction has $n + m$ nodes and $2m$ edges. Hence H has a vertex cover of size at most k iff the instance has a seed set of size at most k with influence at least $k + m$, and by (8.14) Influence Maximization is NP-complete. ■

Problem 8.35. Three of your friends work for a large computer-game company, and they've been working hard to get their proposal for a new game, Droid Trader!, approved by higher management. They are worried that the overall premise of the game is too simple, so that players get really good at it and become bored too quickly. Stripped down, the basic idea is as follows.

A player in the game controls a spaceship and is trying to make money buying and selling droids on different planets. There are n different types of droids and k different planets. Each planet p has the following properties: there are $s(j, p) \geq 0$ droids of type j available for sale, at a fixed price of $x(j, p) \geq 0$ each, for $j = 1, 2, \dots, n$; and there is a demand for $d(j, p) \geq 0$ droids of type j , at a fixed price of $y(j, p) \geq 0$ each. (We assume that a planet does not simultaneously have both a positive supply and a positive demand for a single type of droid; so for each j , at least one of $s(j, p)$ or $d(j, p)$ is equal to 0.)

The player begins on planet s with z units of money and must end at planet t ; there is a directed acyclic graph G on the set of planets, such that s - t paths in G correspond to valid routes by the player. For a given s - t path P in G , the player can engage in transactions as follows. Whenever the player arrives at a planet p on the path P , she can buy up to $s(j, p)$ droids of type j for $x(j, p)$ units of money each (provided she has sufficient money on hand) and/or sell up to $d(j, p)$ droids of type j for $y(j, p)$ units of money each (for $j = 1, 2, \dots, n$). The player's final score is the total amount of money she has on hand when she arrives at planet t .

Given an instance of this game, with a directed acyclic graph G on a set of planets, all the other parameters described above, and also a target bound B , is there a path P in G and a sequence of transactions on P so that the player ends with a final score that is at least B ? Call this the High-Score-on-Droid-Trader! Problem, or HSoDT! for short. Prove that HSoDT! is NP-complete, thereby guaranteeing (assuming $\mathcal{P} \neq \mathcal{NP}$) that there isn't a simple strategy for racking up high scores on your friends' game.

Solution. Reduce Subset Sum to HSoDT! with a single route on which droid i can be bought for w_i and later sold for $2w_i$, starting with $z = W$ and asking for $B = 2W$.

Membership in $\mathcal{NP}$: the certificate is the path P and, for each planet p on P and each type j , the numbers bought ($\leq s(j, p)$) and sold ($\leq d(j, p)$), all of polynomial bit-length. The verifier walks P , at each planet first performing the sales and then the purchases, and checks that holdings and money never go negative and that the final money is at least B . Performing the sales first loses nothing: a type sold at p is not bought at p , and sales only add money, so any feasible order of the same transactions at p remains feasible after the reordering.

Hardness: Subset Sum is NP-complete by (8.23). Given natural numbers $w_1, \dots, w_n$ and W , use n droid types and the path

$$s = p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n \rightarrow q = t,$$

with $s(i, p_i) = 1$, $x(i, p_i) = w_i$, and $d(i, q) = 1$, $y(i, q) = 2w_i$, all other supplies and demands 0, $z = W$ and $B = 2W$. This is a DAG with $n + 2$ planets, built in polynomial time, and no planet has both supply and demand for one type. The only route is this path. Let A be the set of types bought and $A' \subseteq A$ the set sold at q ; since money only decreases along $p_1, \dots, p_n$, buying A is feasible iff $w(A) \leq W$, and then

$$\text{score} = W - w(A) + 2w(A') \leq W + w(A).$$

- (i) If $w(A) = W$, buy A and sell it all: the score is $W + W = 2W = B$.

(ii) If the score is at least $2W$, then $W + w(A) \geq 2W$, so $w(A) \geq W$; with feasibility $w(A) \leq W$ this gives $w(A) = W$.
 So the player can reach score B iff some subset of $\{w_i\}$ sums to W , and by (8.14) HSoDT! is NP-complete. ■

8.6 Exercises 8.36–8.42

Problem 8.36. Your friends Raj and Alanis run a health-food restaurant and buy raw ingredients each morning. There is a set of n possible raw ingredients $I_1, I_2, \dots, I_n$. Ingredient I_j must be purchased in units of size $s(j)$ grams (any purchase must be for a whole number of units), and it costs $c(j)$ dollars per unit. Also, it remains safe to use for $t(j)$ days from the date of purchase.

Over the next k days, they want to make a set of k different daily specials, one each day. (The order in which they schedule the specials is up to them.) The i th daily special uses a subset $S_i \subseteq \{I_1, I_2, \dots, I_n\}$ of the raw ingredients; specifically, it requires $a(i, j)$ grams of ingredient I_j . Finally, for each daily special the ingredients S_i are partitioned into two subsets: those that must be purchased on the very day when the daily special is being offered, and those that can be used any day while they're still safe.

When they buy a unit of I_j they often don't need all of it for that day's special; if they quickly follow up with another special that uses I_j but doesn't require it to be fresh that day, they save money by not having to purchase I_j again.

The Daily Special Scheduling Problem: given data on ingredients and recipes as above, and a budget x , is there a way to schedule the k daily specials so that the total money spent on ingredients over the course of all k days is at most x ?

Prove that Daily Special Scheduling is NP-complete.

Solution. Reduce Hamiltonian Path to it: make each node a special and each edge (u, v) an ingredient that u needs fresh and v can take as u 's leftovers, but only on the following day.

We read “safe for $t(j)$ days” as: bought on day d , usable on days $d, \dots, d + t(j) - 1$ (under the other reading use $t(j) = 1$ below in place of 2).

Membership in $\mathcal{NP}$: the certificate is the order of the specials together with a purchase plan: for each day d and ingredient I_j , the number of units bought, and how many grams of each lot go to each special. WLOG no lot is bigger than the total need $\lceil \sum_i a(i, j) / s(j) \rceil$ units, so the plan has $O(k^2 n)$ numbers of polynomial bit length. Checking it is direct: each special gets its $a(i, j)$ grams, fresh ingredients come from that day's lots, no lot is used after it expires or beyond its size, and the total cost is $\leq x$.

Hardness: Hamiltonian Path (on directed graphs) is NP-complete by (8.19). Given $G = (V, E)$ with $|V| = n' \geq 2$, build this instance:

- one special for each $v \in V$, so $k = n'$;
- one ingredient I_{uv} for each edge $(u, v) \in E$, with $s = 2$ grams, $c = 1$, $t = 2$;
- special u needs 1 gram of I_{uv} , which must be fresh, and special v needs 1 gram of I_{uv} , which need not be fresh;
- budget $x = 2|E| - (n' - 1)$.

Fix an order π , and write $\text{pos}(v)$ for the day of special v . Ingredients are independent, so take one edge (u, v) . Special u must buy one unit of I_{uv} on day $\text{pos}(u)$; buying it any other day would not be fresh. Its leftover gram is usable only on days $\text{pos}(u)$ and $\text{pos}(u) + 1$, and no other purchase is free. Hence the cheapest cost of I_{uv} is

$$1 + [\text{pos}(v) \neq \text{pos}(u) + 1],$$

and so

$$\min \text{cost}(\pi) = 2|E| - \#\{i : (\pi_i, \pi_{i+1}) \in E\}.$$

The count on the right is at most $n' - 1$, with equality iff every consecutive pair of π is an edge, i.e. iff π is a Hamiltonian path of G . Therefore some schedule costs at most $x = 2|E| - (n' - 1)$ iff G has a Hamiltonian path.

The instance has n' specials and $|E|$ ingredients and is built in linear time, so Hamiltonian Path $\leq_P$ Daily Special Scheduling. By (8.14), Daily Special Scheduling is NP-complete.

Problem 8.37. View the galaxy as an undirected graph $G = (V, E)$, where each node is a star system and an edge (u, v) indicates that one can travel directly from u to v . The Death Star is a node s and the hidden Rebel base is a node t . Each edge e has an integer length $\ell_e \geq 0$ and an integer risk $r_e \geq 0$. For any path P from s to t , its total length is the sum of the lengths of its edges, and its total risk is the sum of the risks of its edges.

The Galactic Shortest-Path Problem: given a setup as above, and integer bounds L and R , is there a path from s to t whose total length is at most L , and whose total risk is at most R ?

Prove that Galactic Shortest Path is NP-complete.

Solution. Reduce Subset Sum to it: build a chain of n two-route segments, where the i th segment has a route of length w_i (take item i) and a route of risk w_i (skip item i).

Membership in $\mathcal{NP}$: the certificate is the path P . We check that it is an s - t path, then add up its lengths and risks and compare them with L and R .

Hardness: Subset Sum is NP-complete by (8.23). Given $w_1, \dots, w_n$ and W , let $N = \sum_i w_i$. Build G on the nodes $v_0, \dots, v_n, u_1, \dots, u_n$ with $s = v_0$ and $t = v_n$. For each i add the edges

$$\begin{aligned} (v_{i-1}, v_i) : \quad & \ell = w_i, \quad r = 0, \\ (v_{i-1}, u_i) : \quad & \ell = 0, \quad r = w_i, \\ (u_i, v_i) : \quad & \ell = 0, \quad r = 0, \end{aligned}$$

and set $L = W$ and $R = N - W$.

Each v_i with $0 < i < n$ separates s from t , so a simple s - t path passes $v_0, v_1, \dots, v_n$ in order. Between v_{i-1} and v_i it takes either the direct edge or the detour through u_i . Write A for the set of indices where it takes the direct edge. Then

$$\text{length}(P) = \sum_{i \in A} w_i, \quad \text{risk}(P) = N - \sum_{i \in A} w_i.$$

So $\text{length} \leq W$ and $\text{risk} \leq N - W$ hold together iff $\sum_{i \in A} w_i = W$. Every $A \subseteq \{1, \dots, n\}$ arises from such a path, so a feasible path exists iff some subset sums to exactly W .

The graph has $2n + 1$ nodes and $3n$ edges, and L and R have the bit length of the input, so the reduction runs in polynomial time. Hence Subset Sum $\leq_P$ Galactic Shortest Path, and by (8.14) Galactic Shortest Path is NP-complete.

Problem 8.38. Consider the following version of the Steiner Tree Problem, which we'll refer to as Graphical Steiner Tree. You are given an undirected graph $G = (V, E)$, a set $X \subseteq V$ of vertices, and a number k . You want to decide whether there is a set $F \subseteq E$ of at most k edges so that in the graph (V, F) , X belongs to a single connected component.

Show that Graphical Steiner Tree is NP-complete.

Solution. Reduce Set Cover to it: connect a root to one node per set, connect each set node to its elements, and ask for a tree joining the root to all the elements with $c + m$ edges.

Membership in $\mathcal{NP}$: the certificate is F . We check $|F| \leq k$ and, with one BFS in (V, F) , that all of X lies in one component.

Hardness: Set Cover is NP-complete by (8.16). Take an instance with universe $U = \{u_1, \dots, u_m\}$ ($m \geq 1$), sets $S_1, \dots, S_n$, and bound c . Build G on the nodes $\{r\} \cup \{u_1, \dots, u_m\} \cup \{\sigma_1, \dots, \sigma_n\}$ with these edges:

- (r, σ_j) for every j ;
- (σ_j, u_i) whenever $u_i \in S_j$.

Set $X = \{r, u_1, \dots, u_m\}$ and $k = c + m$.

($\Rightarrow$) Let $S_{j_1}, \dots, S_{j_q}$ be a cover with $q \leq c$. Take the q edges (r, σ_{j_p}) , and for each u_i one edge (σ_{j_p}, u_i) to a chosen set containing it. This gives $q + m \leq k$ edges, and every u_i is joined to r .

($\Leftarrow$) Let F have at most k edges, and let C be the component of (V, F) that contains X . Write

$C = X \cup \{\sigma_j : j \in Q\}$. A spanning tree of C uses edges of F , so

$$|X| + |Q| - 1 = m + |Q| \leq |F| \leq c + m, \quad \text{hence} \quad |Q| \leq c.$$

Each u_i lies in C together with $r \neq u_i$, so it has an F -edge inside C . All of its neighbours are set nodes, so $u_i \in S_j$ for some $j \in Q$. Thus $\{S_j : j \in Q\}$ is a cover of size at most c .

The construction has $1 + m + n$ nodes and at most $n + nm$ edges, so Set Cover $\leq_P$ Graphical Steiner Tree. By (8.14), Graphical Steiner Tree is NP-complete.

Problem 8.39. The Directed Disjoint Paths Problem is defined as follows. We are given a directed graph G and k pairs of nodes $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$. The problem is to decide whether there exist node-disjoint paths $P_1, P_2, \dots, P_k$ so that P_i goes from s_i to t_i . Show that Directed Disjoint Paths is NP-complete.

Solution. Reduce 3-SAT to it: each variable pair must route along a “true” chain or a “false” chain, and each clause pair must route through a node of an unused chain, which exists only when one of its literals is true.

Membership in $\mathcal{NP}$: the certificate is $P_1, \dots, P_k$. We check that each P_i is a directed s_i - t_i path and that no node appears twice.

Hardness: 3-SAT is NP-complete by (8.15). Take variables $x_1, \dots, x_n$ and clauses $C_1, \dots, C_m$, and build G as follows.

- For each variable x_i : nodes $s_i, t_i, a_{i1}, \dots, a_{im}, b_{i1}, \dots, b_{im}$, and the two chains
$$s_i \rightarrow a_{i1} \rightarrow a_{i2} \rightarrow \dots \rightarrow a_{im} \rightarrow t_i \quad (\text{true chain}),$$

$$s_i \rightarrow b_{i1} \rightarrow b_{i2} \rightarrow \dots \rightarrow b_{im} \rightarrow t_i \quad (\text{false chain}).$$
- For each clause C_j : nodes c_j, c'_j , and for each literal of C_j the edges $c_j \rightarrow b_{ij} \rightarrow c'_j$ if the literal is x_i , or $c_j \rightarrow a_{ij} \rightarrow c'_j$ if it is $\neg x_i$.

The pairs are (s_i, t_i) for $i \leq n$ and (c_j, c'_j) for $j \leq m$.

The nodes t_i and c'_j have no out-edges, and c_j has no in-edges. So an s_i - t_i path cannot pass through any clause terminal, and it must be one whole chain. Now take a c_j - c'_j path. Its second node is some a_{ij} or b_{ij} . From there it can only move forward along the chain, to larger second index, or step to the clause sink of the current index. The only node with an edge into c'_j has second index j , so the path must step straight to c'_j . Hence it is $c_j \rightarrow y \rightarrow c'_j$, where y is the node of one literal of C_j .

($\Leftarrow$) Given disjoint paths, set x_i true iff P_i is the true chain. The clause path for C_j uses $y \in \{a_{ij}, b_{ij}\}$ off the chain P_i takes:

- (i) $y = b_{ij}$: then P_i is the true chain, so x_i is true, and $x_i \in C_j$.
- (ii) $y = a_{ij}$: then P_i is the false chain, so x_i is false, and $\neg x_i \in C_j$.

Either way C_j is satisfied.

($\Rightarrow$) Given a satisfying assignment, route P_i along the true chain if x_i is true and along the false chain otherwise. For each C_j pick a true literal and route $c_j \rightarrow y \rightarrow c'_j$, where y is b_{ij} (if the literal is x_i) or a_{ij} (if it is $\neg x_i$). Such a y lies on the chain P_i does not use. Different clauses use nodes with different second index j , so all the paths are node-disjoint.

G has $O(nm)$ nodes and edges, so 3-SAT $\leq_P$ Directed Disjoint Paths. By (8.14), Directed Disjoint Paths is NP-complete.

Problem 8.40. We are given a directed graph $G = (V, E)$, with a designated node $r \in V$ and a designated set of target nodes $T \subseteq V - \{r\}$. Each node v has a switching time s_v , which is a positive integer.

At time 0, the node r generates a message that it would like every node in T to receive. A broadcast scheme works as follows. Node r may send a copy of the message to one of its neighbors at time 0; this neighbor will receive the message at time 1. In general, at time $t \geq 0$, any node v that has already received the message may send a copy of the message to one of its neighbors, provided it has not sent a copy of the message in any of the time steps $t - s_v + 1, t - s_v + 2, \dots, t - 1$. (So v needs

a pause of $s_v - 1$ steps between successive sendings; if $s_v = 1$, no restriction is imposed.)

The completion time of the broadcast scheme is the minimum time t by which all nodes in T have received the message. The Broadcast Time Problem: given the input described above, and a bound b , is there a broadcast scheme with completion time at most b ?

Prove that Broadcast Time is NP-complete.

Example. Take $V = \{r, a, b, c\}$, edges $(r, a), (a, b), (r, c)$, $T = \{b, c\}$, and $s_v = 2$ for each $v \in V$. A broadcast scheme with minimum completion time:

- r sends to a at time 0;
- a sends to b at time 1;
- r sends to c at time 2;
- the scheme completes at time 3, when c receives the message.

Here a may send as soon as it receives at time 1, since this is its first sending. But r cannot send at time 1, since $s_r = 2$ and it sent at time 0.

Solution. Reduce 3-Dimensional Matching to it, already with all $s_v = 1$: r feeds triple nodes, which feed elements. Three private targets of r use up its spare sending slots, so that exactly n triples get informed and they must cover all $3n$ elements.

Membership in $\mathcal{NP}$: a node needs to receive the message only once, so WLOG a scheme makes at most $|V| - 1$ sends. The certificate is this list of (time, sender, receiver) triples, with times $\leq b$ written in binary. We check four things: each sender has already received the message, each send is along an edge, the switching gaps are respected, and every node of T has received the message by time b .

Hardness: 3-Dimensional Matching is NP-complete by (8.20). Take disjoint sets X, Y, Z of size n and triples $\tau_1, \dots, \tau_m \subseteq X \times Y \times Z$. Build G as follows:

- nodes r , one node τ_q for each triple, one node for each element of $X \cup Y \cup Z$, and dummies d_1, d_2, d_3 ;
- edges $r \rightarrow \tau_q$ for all q , $r \rightarrow d_1, d_2, d_3$, and $\tau_q \rightarrow$ each of its three elements;
- $T = X \cup Y \cup Z \cup \{d_1, d_2, d_3\}$, $s_v = 1$ for all v , and $b = n + 3$.

($\Rightarrow$) Let $\tau_1, \dots, \tau_n$ be a perfect matching. Node r sends to τ_p at time $p - 1$, and to d_1, d_2, d_3 at times $n, n + 1, n + 2$. Node τ_p receives at time p and sends to its three elements at times $p, p + 1, p + 2$, so they receive by time $p + 3 \leq n + 3$. Every node sends at most once per step, which $s_v = 1$ allows, so the completion time is at most b .

($\Leftarrow$) In a scheme with completion time $\leq n + 3$, the useful sends of r happen at times $0, \dots, n + 2$, so there are at most $n + 3$ of them. Each dummy can only hear from r , which uses three sends. Hence at most n triple nodes are informed. Each element can hear only from the triple nodes containing it. Each triple node has just three out-neighbours, so the informed triples cover at most $3 \cdot n$ elements, with equality only if they are pairwise disjoint and there are exactly n of them. All $3n$ elements are reached, so equality holds, and the informed triples form a perfect three-dimensional matching.

The graph has $m + 3n + 4$ nodes and $4m + 3$ edges, so 3-Dimensional Matching $\leq_P$ Broadcast Time. By (8.14), Broadcast Time is NP-complete.

Problem 8.41. Given a directed graph G , a *cycle cover* is a set of node-disjoint cycles so that each node of G belongs to a cycle. The *Cycle Cover Problem* asks whether a given directed graph has a cycle cover.

(a) Show that the Cycle Cover Problem can be solved in polynomial time. (*Hint:* Use Bipartite Matching.)

(b) Suppose we require each cycle to have at most three edges. Show that determining whether a graph G has such a cycle cover is NP-complete.

Solution. (a) G has a cycle cover if and only if the bipartite graph B with sides $V^{out} = \{v^{out} : v \in V\}$ and $V^{in} = \{v^{in} : v \in V\}$ and an edge (u^{out}, v^{in}) for each $(u, v) \in E$ has a perfect matching.

If G has a cycle cover, every node u has exactly one successor $\sigma(u)$ on its cycle and exactly one predecessor, so $\{(u^{out}, \sigma(u)^{in}) : u \in V\}$ is a perfect matching of B . Conversely, a perfect matching M of B defines $\sigma(u) =$ the node with $(u^{out}, \sigma(u)^{in}) \in M$; each u has one image and one preimage, so σ is a permutation of V with $(u, \sigma(u)) \in E$ for all u . The cycles of the permutation σ are node-disjoint,

cover V , and each is a directed cycle of G because every step $u \rightarrow \sigma(u)$ is an edge.

B has $2n$ nodes and m edges, so by (7.38) Ford-Fulkerson finds a maximum matching in $O(mn)$ time, and we answer yes iff it has size n .

(b) Reduce 3-Dimensional Matching (NP-complete by (8.20)) by replacing each triple with a 9-node gadget that can be tiled by directed triangles in exactly two ways: one that also absorbs the triple's three elements, and one that absorbs none.

Membership in NP: the certificate is the list of cycles; we check in polynomial time that each is a cycle of G with at most three edges and that every node lies on exactly one of them.

Construction: given $X = \{x_i\}$, $Y = \{y_i\}$, $Z = \{z_i\}$ of size n and triples T , put the $3n$ element nodes in G , and for each $t = (x, y, z) \in T$ add new nodes $a_1, \dots, a_9$ (private to t) and the 18 edges

$$\begin{aligned} x &\rightarrow a_1 \rightarrow a_2 \rightarrow x, & a_2 &\rightarrow a_3 \rightarrow a_1, \\ y &\rightarrow a_4 \rightarrow a_5 \rightarrow y, & a_5 &\rightarrow a_6 \rightarrow a_4, \\ z &\rightarrow a_7 \rightarrow a_8 \rightarrow z, & a_8 &\rightarrow a_9 \rightarrow a_7, \\ a_3 &\rightarrow a_6 \rightarrow a_9 \rightarrow a_3. \end{aligned}$$

No pair of edges is antiparallel and there are no loops, so the allowed cycles are exactly the directed triangles. Element nodes have out-edges only to a_1, a_4, a_7 of gadgets and in-edges only from a_2, a_5, a_8 , so every triangle lies in one gadget plus possibly its own element nodes, and the triangles of gadget t are

$$\begin{aligned} \text{outer: } &\{x, a_1, a_2\}, \{y, a_4, a_5\}, \{z, a_7, a_8\}, \\ \text{middle: } &\{a_1, a_2, a_3\}, \{a_4, a_5, a_6\}, \{a_7, a_8, a_9\}, \\ \text{inner: } &\{a_3, a_6, a_9\}. \end{aligned}$$

In any triangle cover, consider the inner triangle of t : (i) it is used: then a_3, a_6, a_9 are taken, no middle triangle is available, and a_1, a_4, a_7 must lie in the three outer triangles (“on”: x, y, z are covered by t); (ii) it is not used: a_3 lies only in the inner triangle and $\{a_1, a_2, a_3\}$, so that middle triangle is used, and likewise for a_6, a_9 ; the gadget covers no element (“off”).

Hence a cover of G by cycles of length at most 3 is exactly a choice of on-triples covering each element node exactly once, i.e. a set of disjoint triples covering $X \cup Y \cup Z$, a perfect 3-dimensional matching.

Conversely, a perfect matching M gives a cover: turn $t \in M$ on and every other gadget off. G has $3n + 9|T|$ nodes and $18|T|$ edges, so the reduction is polynomial, and the problem is NP-complete by (8.14).

Problem 8.42. Suppose you’re consulting for a company in northern New Jersey that designs communication networks, and they come to you with the following problem. They’re studying a specific n -node communication network, modeled as a directed graph $G = (V, E)$. For reasons of fault tolerance, they want to divide up G into as many virtual “domains” as possible: A *domain* in G is a set X of nodes, of size at least 2, so that for each pair of nodes $u, v \in X$ there are directed paths from u to v and from v to u that are contained entirely in X .

Show that the following *Domain Decomposition Problem* is NP-complete. Given a directed graph $G = (V, E)$ and a number k , can V be *partitioned* into at least k sets, each of which is a domain?

Solution. Reuse the 3-Dimensional Matching reduction of Exercise 8.41(b) with $k = |V|/3$: in a graph with no antiparallel edges, a partition into $|V|/3$ domains is exactly a partition into directed triangles. Membership in NP: the certificate is the partition; we check that there are at least k parts, each of size at least 2, and that each induced subgraph $G[X]$ is strongly connected (two BFS runs from one node of X , in $G[X]$ and its reverse), all in polynomial time.

Hardness: from an instance of 3-Dimensional Matching (NP-complete by (8.20)) build the graph G of Exercise 8.41(b), with $N = 3n + 9|T|$ nodes, and set $k = N/3$; G has no loops and no antiparallel pair of edges, and G can be partitioned into directed triangles iff the 3DM instance has a perfect matching. Key fact: in G every domain has at least 3 nodes, and a 3-node domain X spans a directed triangle. A 2-node domain $\{u, v\}$ would need both (u, v) and (v, u) . If $|X| = 3$, each node of X needs an out-edge

and an in-edge inside X , so $G[X]$ has at least 3 edges; with no antiparallel pairs it has at most $\binom{3}{2} = 3$, so each node has in- and out-degree exactly 1 in $G[X]$, which is a directed 3-cycle.

($\Rightarrow$) If V is partitioned into at least $N/3$ domains, each of size at least 3, then

$$N = \sum_X |X| \geq 3 \cdot \#\text{domains} \geq 3 \cdot \frac{N}{3} = N,$$

so every domain has exactly 3 nodes and spans a directed triangle. This is a partition of G into triangles, so the 3DM instance has a perfect matching.

($\Leftarrow$) A perfect matching gives a partition of G into $N/3$ directed triangles by Exercise 8.41(b), and each triangle is a domain.

The construction is polynomial, so Domain Decomposition is NP-complete by (8.14).

9 PSPACE: A Class of Problems beyond NP

Contents

9.1 Exercises 9.1–9.3	163
---------------------------------	-----

9.1 Exercises 9.1–9.3

Problem 9.1. Let's consider a special case of Quantified 3-SAT in which the underlying Boolean formula has no negated variables. Specifically, let $\Phi(x_1, \dots, x_n)$ be a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \dots \wedge C_k,$$

where each C_i is a disjunction of three terms. We say Φ is *monotone* if each term in each clause consists of a nonnegated variable — that is, each term is equal to x_i , for some i , rather than $\overline{x_i}$. We define Monotone QSAT to be the decision problem

$$\exists x_1 \forall x_2 \dots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \dots, x_n)?$$

where the formula Φ is monotone.

Do one of the following two things: (a) prove that Monotone QSAT is PSPACE-complete; or (b) give an algorithm to solve arbitrary instances of Monotone QSAT that runs in time polynomial in n . (Note that in (b), the goal is polynomial *time*, not just polynomial space.)

Solution. We do (b): the answer is “yes” if and only if every clause C_j contains at least one existentially quantified variable, i.e. some x_i with i odd; this is checked by one scan of the clauses in $O(k)$ time.

(i) Suppose every clause contains some x_i with i odd. Let the $\exists$ quantifiers always choose $x_i = 1$. Whatever values $x_2, x_4, \dots, x_{n-1}$ take, every clause then contains a true term, so

$$\Phi(1, x_2, 1, x_4, \dots, x_{n-1}, 1) = 1 \quad \text{for all } x_2, x_4, \dots, x_{n-1} \in \{0, 1\},$$

and the quantified formula is true.

(ii) Suppose some clause C_j contains only variables x_i with i even. Let the $\forall$ quantifiers always choose $x_i = 0$. Whatever values $x_1, x_3, \dots, x_n$ take, all three terms of C_j are 0, so $C_j = 0$, $\Phi = 0$, and the quantified formula is false.

Problem 9.2. Consider the following word game, which we'll call *Geography*. You have a set of names of places, like the capital cities of all the countries in the world. The first player begins the game by naming the capital city c of the country the players are in; the second player must then choose a city c' that starts with the letter on which c ends; and the game continues in this way, with each player alternately choosing a city that starts with the letter on which the previous one ended. The player who loses is the first one who cannot choose a city that hasn't been named earlier in the game.

For example, a game played in Hungary would start with “Budapest,” and then it could continue (for example), “Tokyo, Ottawa, Ankara, Amsterdam, Moscow, Washington, Nairobi.”

This game is a good test of geographical knowledge, of course, but even with a list of the world's capitals sitting in front of you, it's also a major strategic challenge. Which word should you pick next, to try forcing your opponent into a situation where they'll be the one who's ultimately stuck without a move?

To highlight the strategic aspect, we define the following abstract version of the game, which we call *Geography on a Graph*. Here, we have a directed graph $G = (V, E)$, and a designated *start node* $s \in V$. Players alternate turns starting from s ; each player must, if possible, follow an edge out of the current node to a node that hasn't been visited before. The player who loses is the first one who cannot move to a node that hasn't been visited earlier in the game. (There is a direct analogy to *Geography*, with nodes corresponding to words.) In other words, a player loses if the game is currently at node v , and for edges of the form (v, w) , the node w has already been visited.

Prove that it is PSPACE-complete to decide whether the first player can force a win in Geography on a Graph.

Solution. Membership: the first player wins from the position (current node v , visited set S) iff

$$\text{Win}(v, S) = \bigvee_{(v,w) \in E, w \notin S} \neg \text{Win}(w, S \cup \{w\}),$$

and evaluating this recursively, exactly as in the algorithm of (9.4), uses recursion depth at most $|V|$ (each move visits a new node) with $O(|V|)$ bits per frame, so $O(|V|^2)$ space; hence the problem is in PSPACE.

Hardness: we show Competitive 3-SAT $\leq_P$ Geography on a Graph; since QSAT $\leq_P$ Competitive 3-SAT by (9.10) and QSAT is PSPACE-complete by (9.9), this suffices. Let $\Phi = C_1 \wedge \dots \wedge C_k$ over $x_1, \dots, x_n$ (n odd). Build G :

- for each i a diamond: nodes a_i, t_i, f_i, b_i with edges $(a_i, t_i), (a_i, f_i), (t_i, b_i), (f_i, b_i)$, and edges (b_i, a_{i+1}) for $i < n, (b_n, c)$;
- a node C_j for each clause with edge (c, C_j) ; for each term of C_j , an edge (C_j, f_i) if the term is x_i and (C_j, t_i) if it is $\bar{x}_i$;
- start node $s = a_1$.

This has $4n + k + 1$ nodes and is built in polynomial time. Read “the game passes through t_i ” as $x_i = 1$ and “through f_i ” as $x_i = 0$.

Every node other than a_i, c, C_j has out-degree 1, and in the first $3n$ moves its successor is still unvisited, so these moves are forced. Each diamond takes three moves, so the choice at a_i is made at move $3i - 2$: by player 1 for i odd and player 2 for i even, exactly as in Competitive 3-SAT. Move $3n$ (player 1) reaches c ; player 2 picks a clause C_j ; then player 1 must move from C_j to a node t_i or f_i , which is unvisited exactly when the corresponding term of C_j is true under the chosen assignment, since the game visited t_i iff $x_i = 1$:

$$\begin{aligned} (C_j, f_i) \text{ usable} &\iff f_i \text{ unvisited} \iff x_i = 1, \\ (C_j, t_i) \text{ usable} &\iff t_i \text{ unvisited} \iff x_i = 0. \end{aligned}$$

If player 1 can make this move, player 2 must then leave t_i or f_i , whose only successor b_i is visited, so player 2 loses. Hence, once the assignment is fixed, player 1 wins iff every clause has a true term, i.e. iff $\Phi = 1$ (otherwise player 2 picks a false clause and player 1 is stuck at it).

So the game on G is Competitive 3-SAT on Φ with forced moves added, and player 1 has a forced win on (G, s) iff player 1 has a forced win in Competitive 3-SAT on Φ . Geography on a Graph is therefore PSPACE-complete.

Problem 9.3. Give a polynomial-time algorithm to decide whether a player has a forced win in *Geography on a Graph*, in the special case when the underlying graph G has no directed cycles (in other words, when G is a DAG).

Solution. In a DAG no walk revisits a node, so the visited set never matters and the player to move at v wins iff some successor of v is a losing node; compute this label for every node in reverse topological order.

```

compute a topological order v_1, ..., v_n of G
for v = v_n down to v_1:
    W[v] = true if some edge (v, w) has W[w] = false, else false
first player has a forced win iff W[s] = true
(otherwise the second player has a forced win)
```

Correctness: if the game ever moved from v back to an earlier node, G would contain a directed cycle; so every node reachable in one move is unvisited, and the position is determined by the current node v alone. Every successor w of v comes after v in the topological order, so $W[w]$ is already computed when $W[v]$ is. By induction on this order, $W[v]$ is true iff the player to move at v has a forced win: (i) v a sink: the mover is stuck and loses, and $W[v] = \text{false}$; (ii) some $W[w] = \text{false}$: the mover goes to w , where the opponent moves from a losing node, so the mover wins; (iii) all $W[w] = \text{true}$: every move hands the opponent a winning node, so the mover loses. The game ends within n moves, so one of the two players has a forced win, and it is player 1 iff $W[s] = \text{true}$.

Running time: topological sorting takes $O(m + n)$ (Section 3.6) and the loop examines each edge once, so the total is $O(m + n)$.

10 Extending the Limits of Tractability

Contents

10.1 Exercises 10.1–10.7	167
10.2 Exercises 10.8–10.9	173

10.1 Exercises 10.1–10.7

Problem 10.1. In Exercise 5 of Chapter 8, we claimed that the Hitting Set Problem was NP-complete. To recap the definitions, consider a set $A = \{a_1, \dots, a_n\}$ and a collection $B_1, B_2, \dots, B_m$ of subsets of A . We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \dots, B_m$ if H contains at least one element from each B_i — that is, if $H \cap B_i$ is not empty for each i . (So H “hits” all the sets B_i .)

Now suppose we are given an instance of this problem, and we’d like to determine whether there is a hitting set for the collection of size at most k . Furthermore suppose that each set B_i has at most c elements, for a constant c . Give an algorithm that solves this problem with a running time of the form $O(f(c, k) \cdot p(n, m))$, where $p(\cdot)$ is a polynomial function, and $f(\cdot)$ is an arbitrary function that depends only on c and k , not on n or m .

Solution. Branch on which element of an unhit set goes into H , exactly as the Vertex Cover Algorithm of Section 10.1 branches on the ends of an edge (an edge is a set of size $c = 2$).

```

HS(collection B, k):
  if B is empty: return "yes"
  if k = 0: return "no"
  let B1 be any set of B
  for each element a of B1:
    B(a) = the sets of B that do not contain a
    if HS(B(a), k - 1) = "yes": return "yes"
  return "no"

```

Correctness. The analogue of (10.3): for any set B_1 of the collection $\mathcal{B}$,

$$\mathcal{B} \text{ has a hitting set of size } \leq k \iff \exists a \in B_1 : \mathcal{B}(a) \text{ has a hitting set of size } \leq k - 1.$$

($\Rightarrow$) H hits B_1 , so pick $a \in H \cap B_1$; then $H - \{a\}$ hits every set not containing a , and $|H - \{a\}| \leq k - 1$.

($\Leftarrow$) If H' hits $\mathcal{B}(a)$ then $H' \cup \{a\}$ hits $\mathcal{B}(a)$ and every set containing a , and has size $\leq k$. The base cases are immediate (the empty collection is hit by $\emptyset$; a nonempty one is not hit by a set of size 0; an empty B_1 correctly gives “no”), so induction on k shows HS answers correctly.

Running time. The recursion tree has depth at most k and every node has at most c children, so it has at most $\sum_{j=0}^k c^j \leq (k+1)c^k$ nodes. Building one reduced collection $\mathcal{B}(a)$ means testing $a \in B_j$ for each of the m sets, $O(cm)$ time, and each node builds at most c of them. Hence the total time is

$$O((k+1)c^k \cdot c^2m) = O(f(c, k) \cdot m), \quad f(c, k) = (k+1)c^{k+2},$$

plus $O(n + cm)$ to read the input, which is of the required form $O(f(c, k) \cdot p(n, m))$.

Problem 10.2. The difficulty in 3-SAT comes from the fact that there are 2^n possible assignments to the input variables $x_1, x_2, \dots, x_n$, and there’s no apparent way to search this space in polynomial time. This intuitive picture, however, might create the misleading impression that the fastest algorithms for 3-SAT actually require time 2^n . In fact, though it’s somewhat counterintuitive when you first hear it, there are algorithms for 3-SAT that run in significantly less than 2^n time in the worst case; in other words, they determine whether there’s a satisfying assignment in less time than it would take to enumerate all possible settings of the variables.

Here we’ll develop one such algorithm, which solves instances of 3-SAT in $O(p(n) \cdot (\sqrt{3})^n)$ time for some polynomial $p(n)$. Note that the main term in this running time is $(\sqrt{3})^n$, which is bounded by 1.74^n .

(a) For a truth assignment Φ for the variables $x_1, \dots, x_n$, we use $\Phi(x_i)$ to denote the value assigned by Φ to x_i (either 0 or 1). If Φ and Φ' are each truth assignments, we define the *distance* between Φ and Φ' to be the number of variables x_i for which they assign different values, and we denote this distance by $d(\Phi, \Phi')$. In other words, $d(\Phi, \Phi') = |\{i : \Phi(x_i) \neq \Phi'(x_i)\}|$.

A basic building block for our algorithm will be the ability to answer the following kind of question: Given a truth assignment Φ and a distance d , we’d like to know whether there exists a satisfying

assignment Φ' such that the distance from Φ to Φ' is at most d . Consider the following algorithm, $\text{Explore}(\Phi, d)$, that attempts to answer this question.

```

Explore(Phi, d):
  If Phi is a satisfying assignment then return "yes"
  Else if d = 0 then return "no"
  Else
    Let C_i be a clause that is not satisfied by Phi
    (i.e., all three terms in C_i evaluate to false)
    Let Phi_1 denote the assignment obtained from Phi by taking the
    variable that occurs in the first term of clause C_i and
    inverting its assigned value
    Define Phi_2 and Phi_3 analogously in terms of the second and
    third terms of the clause C_i
    Recursively invoke:
      Explore(Phi_1, d - 1)
      Explore(Phi_2, d - 1)
      Explore(Phi_3, d - 1)
    If any of these three calls returns "yes" then return "yes"
    Else return "no"

```

Prove that $\text{Explore}(\Phi, d)$ returns “yes” if and only if there exists a satisfying assignment Φ' such that the distance from Φ to Φ' is at most d . Also, give an analysis of the running time of $\text{Explore}(\Phi, d)$ as a function of n and d .

(b) Clearly any two assignments Φ and Φ' have distance at most n from each other, so one way to solve the given instance of 3-SAT would be to pick an arbitrary starting assignment Φ and then run $\text{Explore}(\Phi, n)$. However, this will not give us the running time we want.

Instead, we will need to make several calls to Explore , from different starting points Φ , and search each time out to more limited distances. Describe how to do this in such a way that you can solve the instance of 3-SAT in a running time of only $O(p(n) \cdot (\sqrt{3})^n)$.

Solution. (a) We prove the equivalence by induction on d , and the running time is $O(3^d \cdot m) = O(3^d n^3)$ for m clauses.

If Φ is satisfying, “yes” is correct ($\Phi' = \Phi$). If Φ is not satisfying and $d = 0$, the only assignment at distance 0 is Φ itself, so “no” is correct. Otherwise $d \geq 1$ and C_i is false under Φ ; note $d(\Phi, \Phi_j) = 1$ for $j = 1, 2, 3$.

(Soundness) If $\text{Explore}(\Phi_j, d - 1)$ says “yes”, the induction hypothesis gives a satisfying Φ' with

$$d(\Phi, \Phi') \leq d(\Phi, \Phi_j) + d(\Phi_j, \Phi') \leq 1 + (d - 1) = d.$$

(Completeness) Suppose Φ' is satisfying with $d(\Phi, \Phi') \leq d$. Then Φ' makes some term of C_i true, say the j -th, whose variable x Φ makes false; so $\Phi'(x) \neq \Phi(x) = \neg \Phi_j(x)$, i.e. Φ_j agrees with Φ' on x and with Φ elsewhere. Hence

$$d(\Phi_j, \Phi') = d(\Phi, \Phi') - 1 \leq d - 1,$$

and by induction $\text{Explore}(\Phi_j, d - 1)$ returns “yes”, so $\text{Explore}(\Phi, d)$ does too.

Running time. Each call makes at most 3 calls with parameter one smaller, so the recursion tree has at most $\sum_{j=0}^d 3^j < \frac{3}{2} \cdot 3^d$ nodes. Each node scans the m clauses once (to test satisfaction and find C_i) and copies an n -bit assignment three times, $O(m + n)$ work. With no repeated clause, $m \leq (2n)^3$, so

$$T(n, d) = O(3^d(m + n)) = O(3^d n^3).$$

(b) Run $\text{Explore}(\mathbf{0}, \lfloor n/2 \rfloor)$ and $\text{Explore}(\mathbf{1}, \lfloor n/2 \rfloor)$, where $\mathbf{0}$ and $\mathbf{1}$ set every variable to 0, resp. 1, and report “satisfiable” iff either says “yes”.

Each variable disagrees under Φ' with exactly one of $\mathbf{0}, \mathbf{1}$, so

$$d(\mathbf{0}, \Phi') + d(\mathbf{1}, \Phi') = n \Rightarrow \min\{d(\mathbf{0}, \Phi'), d(\mathbf{1}, \Phi')\} \leq \lfloor n/2 \rfloor.$$

So if a satisfying Φ' exists, one of the two calls says “yes” by (a); and if either says “yes”, a satisfying assignment exists by (a). By (a) the two calls cost

$$2 \cdot O(3^{n/2} n^3) = O(n^3 \cdot (\sqrt{3})^n),$$

which is the bound with $p(n) = n^3$.

Problem 10.3. Suppose we are given a directed graph $G = (V, E)$, with $V = \{v_1, v_2, \dots, v_n\}$, and we want to decide whether G has a Hamiltonian path from v_1 to v_n . (That is, is there a path in G that goes from v_1 to v_n , passing through every other vertex exactly once?)

Since the Hamiltonian Path Problem is NP-complete, we do not expect that there is a polynomial-time solution for this problem. However, this does not mean that all nonpolynomial-time algorithms are equally “bad.” For example, here’s the simplest brute-force approach: For each permutation of the vertices, see if it forms a Hamiltonian path from v_1 to v_n . This takes time roughly proportional to $n!$, which is about 3×10^{17} when $n = 20$.

Show that the Hamiltonian Path Problem can in fact be solved in time $O(2^n \cdot p(n))$, where $p(n)$ is a polynomial function of n . This is a much better algorithm for moderate values of n ; for example, 2^n is only about a million when $n = 20$.

Solution. Dynamic programming over subsets: for $S \subseteq V$ with $v_1 \in S$ and $v \in S$, let $H[S, v]$ be true iff G has a path from v_1 to v whose vertex set is exactly S . Then

$$H[\{v_1\}, v_1] = \text{true}, \quad H[S, v_1] = \text{false} \quad (|S| \geq 2),$$

$$H[S, v] = \bigvee_{u \in S - \{v\}, (u,v) \in E} H[S - \{v\}, u] \quad (v \neq v_1),$$

filled in order of increasing $|S|$; the answer is $H[V, v_n]$ (for $n = 1$ the answer is trivially yes).

Correctness, by induction on $|S|$. A path P from v_1 to $v \neq v_1$ with vertex set S has a last edge (u, v) , and P minus v is a path from v_1 to u with vertex set $S - \{v\}$; so $H[S, v]$ true forces the right side true. Conversely, if $H[S - \{v\}, u]$ is witnessed by a path P' and $(u, v) \in E$, then $v \notin S - \{v\}$, so P' followed by (u, v) is still a path, with vertex set S . (A path from v_1 back to v_1 through other vertices would repeat v_1 , hence the false entries.) A Hamiltonian path from v_1 to v_n is exactly a path from v_1 to v_n with vertex set V .

Running time. There are at most $2^{n-1} \cdot n$ entries, and each is an OR over at most $n - 1$ predecessors u (with an adjacency matrix and S stored as a bitmask, $O(1)$ per term), so the total is

$$O(2^n \cdot n^2),$$

i.e. $p(n) = n^2$.

Problem 10.4. We say that a graph $G = (V, E)$ is a *triangulated cycle graph* if it consists of the vertices and edges of a triangulated convex n -gon in the plane — in other words, if it can be drawn in the plane as follows.

The vertices are all placed on the boundary of a convex set in the plane (we may assume on the boundary of a circle), with each pair of consecutive vertices on the circle joined by an edge. The remaining edges are then drawn as straight line segments through the interior of the circle, with no pair of edges crossing in the interior. We require the drawing to have the following property. If we let S denote the set of all points in the plane that lie on vertices or edges of the drawing, then each bounded component of the plane after deleting S is bordered by exactly three edges. (This is the sense in which the graph is a “triangulation.”)

A triangulated cycle graph is pictured in Figure 10.12: five vertices $v_1, \dots, v_5$ in order around a circle, the five cycle edges (v_i, v_{i+1}) (indices mod 5), and the two chords (v_1, v_3) and (v_1, v_4) , which cut the pentagon into three triangles.

Prove that every triangulated cycle graph has a tree decomposition of width at most 2, and describe an efficient algorithm to construct such a decomposition.

Solution. Peel off an “ear” (a degree-2 vertex whose two neighbours are adjacent), decompose the rest by induction, and hang the ear’s triangle off a piece containing the two neighbours. All pieces are triangles, so the width is $3 - 1 = 2$.

Ears exist. Let $n \geq 4$. Some chord exists, since otherwise the interior is one bounded face bordered by

$n > 3$ edges. Among all chords (a, b) and the two arcs of the circle they cut off, choose a pair whose arc has the fewest vertices strictly inside it; there is at least one such vertex. Any other chord with both ends on this arc (ends included) would cut off a smaller arc, so there is none. Any chord from an inner vertex of the arc to a vertex off the arc has interleaved endpoints with (a, b) , so by convexity it crosses (a, b) in the interior, which is not allowed. So the region bounded by (a, b) and the arc is a single face, bordered by (number of inner vertices) + 2 edges, which must equal 3. Hence the arc has exactly one inner vertex v , v has degree 2 with neighbours a, b , and $(a, b) \in E$. Deleting v leaves the vertices on the circle with consecutive ones adjacent (the chord (a, b) becomes a side), merges the face avb into the outer region, and leaves every other bounded face as it was; so $G - v$ is a triangulated cycle graph on $n - 1$ vertices.

The decomposition, by induction on n . For $n = 3$, take one piece V . For $n \geq 4$, take an ear v with neighbours a, b and a width-2 decomposition $(T', \{V_t\})$ of $G - v$. By Edge Coverage some piece $V_s \ni a, b$; add a new leaf t_v adjacent to s with $V_{t_v} = \{a, v, b\}$. Then: (i) Node and Edge Coverage: the only new node and edges, $v, (a, v), (v, b)$, lie in V_{t_v} . (ii) Coherence for v : v lies in the single piece V_{t_v} . (iii) Coherence for a (and b): its subtree in T' contains s , and we add t_v , which is adjacent to s . (iv) Coherence for any other x : its subtree is unchanged.

Algorithm. Keep adjacency sets, current degrees, and a stack of degree-2 vertices. While more than 3 vertices remain, pop a vertex v that is still present with degree 2; with $n' \geq 4$ vertices left, any degree-2 vertex is an ear (its neighbours are its two circle neighbours a, b , and the bounded face at v is a triangle with sides $(a, v), (v, b)$, so $(a, b) \in E$). Record (v, a, b) , delete v , and push a or b if its degree drops to 2. The ear argument above shows the stack never runs dry. Then build the tree in reverse: start with the final triangle as one piece, keep a hash map from each edge to some piece containing it, and for each recorded ear (v, a, b) , in reverse order, attach $\{a, v, b\}$ to the piece stored for (a, b) and store (a, v) and (v, b) under the new piece. This is exactly the induction above, so it produces a valid decomposition of width 2. A triangulated n -gon has $2n - 3$ edges, and each deletion and each attachment takes $O(1)$ time, so the algorithm runs in $O(n)$ time.

Problem 10.5. The *Minimum-Cost Dominating Set Problem* is specified by an undirected graph $G = (V, E)$ and costs $c(v)$ on the nodes $v \in V$. A subset $S \subset V$ is said to be a *dominating set* if all nodes $u \in V - S$ have an edge (u, v) to a node v in S . (Note the difference between dominating sets and vertex covers: in a dominating set, it is fine to have an edge (u, v) with neither u nor v in the set S as long as both u and v have neighbors in S .)

- (a) Give a polynomial-time algorithm for the Dominating Set Problem for the special case in which G is a tree.
- (b) Give a polynomial-time algorithm for the Dominating Set Problem for the special case in which G has tree-width 2, and we are also given a tree decomposition of G with width 2.

Solution. (a) Root the tree at r and compute, bottom-up, three values per node, as in the tree dynamic program of (10.7). For a node v with subtree T_v and children $u_1, \dots, u_d$: (i) $A[v]$: min cost of $S \subseteq T_v$ with $v \in S$ and every node of T_v dominated; (ii) $B[v]$: same, but $v \notin S$ (so v is dominated by a child in S); (iii) $C[v]$: $v \notin S$, v not dominated inside T_v , every other node of T_v dominated. Writing $\mu_u = \min(A[u], B[u])$,

$$\begin{aligned} A[v] &= c(v) + \sum_u \min(A[u], B[u], C[u]), & C[v] &= \sum_u B[u], \\ B[v] &= \sum_u \mu_u + \max(0, \min_u (A[u] - \mu_u)) & (B[v] &= \infty \text{ if } v \text{ is a leaf}), \end{aligned}$$

and the answer is $\min(A[r], B[r])$; the sets are recovered by tracing back.

The subtrees T_{u_i} are disjoint and their only edges to the rest of T_v go to v , so the conditions on the parts $S \cap T_{u_i}$ separate child by child, and only through whether $u_i \in S$ and whether u_i is dominated. If $v \in S$, each u_i is dominated by v , so any of the three states is allowed for it. If $v \notin S$, no u_i can be in state (iii), since u_i has no other neighbor outside T_{u_i} . In case (iii) every u_i must also avoid S . In case (ii) at least one u_i must be in S , and the cheapest way is to give each child its best of A, B and then, if no child chose A , switch the child with the smallest extra cost $A[u] - \mu_u$. Each node takes $O(\deg v)$ time, so the total is $O(n)$.

(b) Run a dynamic program over the decomposition $(T, \{V_t\})$, as in (10.18), where the state records each piece node's role. We may make it nonredundant (Section 10.4), so by (10.16) it has at most n pieces. Root T at r and use G_t as in Section 10.4. A labelling $\sigma : V_t \rightarrow \{1, D, 0\}$ means 1: in S ; D : not in S , must have a neighbor in S ; 0: not in S , no requirement. Let $f_t(\sigma)$ be the minimum of $c(S)$ over $S \subseteq V(G_t)$ such that $S \cap V_t = \sigma^{-1}(1)$, every node of $G_t - V_t$ is dominated by S , and every x with $\sigma(x) = D$ has a neighbor in S . The answer is $\min\{f_r(\sigma) : \sigma(V_r) \subseteq \{1, D\}\}$. Each piece has $|V_t| \leq 3$, so there are at most 27 labellings.

Separation facts. Let $t_1, \dots, t_d$ be the children of t . By (10.13) the sets $G_{t_i} - V_t$ are disjoint with no edges between them. By (10.14) applied to the edge (t, t_i) , each $y \in V(G_{t_i}) - V_t$ has all its neighbors in G_{t_i} . A node $x \in V_t - V_{t_i}$ has no neighbor in $G_{t_i} - V_t$, and by Coherence $V_t \cap V(G_{t_i}) = V_t \cap V_{t_i}$.

Recurrence. Let $U = \sigma^{-1}(1)$, and let X_σ be the set of x with $\sigma(x) = D$ and no neighbor in U ; each must be dominated from below. For a child t_i and $R \subseteq X_\sigma \cap V_{t_i}$, let

$$g_i(\sigma, R) = \min_{\sigma_i} f_{t_i}(\sigma_i) - c(U \cap V_{t_i}),$$

over labellings σ_i of V_{t_i} with: $\sigma_i = 1$ exactly on $U \cap V_{t_i}$ among nodes of V_{t_i} ; $\sigma_i = D$ on R ; $\sigma_i = 0$ on the other nodes of V_{t_i} ; and $\sigma_i \in \{1, D\}$ on $V_{t_i} - V_t$. Then

$$f_t(\sigma) = c(U) + \min \left\{ \sum_{i=1}^d g_i(\sigma, R_i) : R_1, \dots, R_d \text{ partition } X_\sigma \right\},$$

where an empty minimum is ∞ (for a leaf, $f_t(\sigma) = c(U)$ if $X_\sigma = \emptyset$ and ∞ otherwise).

Proof of the recurrence. ($\geq$) Take an optimal S , and assign each $x \in X_\sigma$ to one child whose $G_{t_i} - V_t$ contains a neighbor of x in S . One exists by the separation facts, and it has $x \in V_{t_i}$. Let R_i be the nodes assigned to t_i . Then $S_i = S \cap V(G_{t_i})$ is feasible for the labelling σ_i above, and the nodes of $V_{t_i} - V_t$ really are dominated because their neighbors lie in G_{t_i} . Since the sets $S_i - V_t$ are disjoint,

$$c(S) = c(U) + \sum_i (c(S_i) - c(U \cap V_{t_i})) \geq \text{RHS}.$$

($\leq$) Conversely, let S_i attain $g_i(\sigma, R_i)$ and put $S = U \cup \bigcup_i S_i$. Then $S \cap V_t = U$, every node below t is dominated inside its own G_{t_i} , every D -node outside X_σ is dominated by U , and every node of X_σ is dominated by its responsible child. The same display gives $c(S) = \text{RHS}$.

Running time. For fixed σ , compute the partition minimum by folding in children one at a time over the at most 8 subsets $Y \subseteq X_\sigma$:

$$\Phi_0(\emptyset) = 0, \quad \Phi_j(Y) = \min_{R \subseteq Y \cap V_{t_j}} \Phi_{j-1}(Y - R) + g_j(\sigma, R),$$

with $\Phi_0(Y) = \infty$ for $Y \neq \emptyset$, so that $f_t(\sigma) = c(U) + \Phi_d(X_\sigma)$. Each $g_j(\sigma, R)$ scans the 27 labellings of V_{t_j} . A node t therefore costs $O(27 \cdot 27 \cdot 27 \cdot d_t) = O(d_t)$, and summed over the at most n pieces of a nonredundant decomposition the DP takes $O(n)$ time, which is polynomial.

Problem 10.6. The *Node-Disjoint Paths Problem* is given by an undirected graph G and k pairs of nodes (s_i, t_i) for $i = 1, \dots, k$. The problem is to decide whether there are node-disjoint paths P_i so that path P_i connects s_i to t_i . Give a polynomial-time algorithm for the Node-Disjoint Paths Problem for the special case in which G has tree-width 2, and we are also given a tree decomposition T of G with width 2.

Solution. Run a dynamic program up the tree decomposition whose subproblems are indexed by the possible “traces” of a solution on a piece V_t ; since $|V_t| \leq 3$, there are only $O(k^3)$ traces, even though k is part of the input.

Preprocessing. If a node lies in two different pairs the answer is no; a pair with $s_i = t_i$ is served by the one-node path, so delete s_i and the pair. So the $2k$ terminals are now distinct. A solution is the same thing as an edge set F such that (V, F) has no cycle, every terminal has F -degree exactly 1, every other

node has F -degree 0 or 2, and each component of (V, F) with edges joins s_i to t_i for some i . Make T nonredundant, so it has at most n pieces by (10.16), and root it at r . Assign each edge $e = (u, v)$ to the node of T closest to r whose piece contains both u and v (the nodes whose pieces contain u and v form a nonempty subtree, so this node is unique), and let E_t be the edges assigned to t and F_t the part of F assigned to nodes of the subtree T_t . A node of $G_{T_t} - V_t$ has, by (10.14), all its incident edges assigned inside T_t , so its F -degree is already final in F_t .

Subproblems. The trace of F_t on V_t records: the F_t -degree of each $v \in V_t$; the partition of V_t into the components of (G_{T_t}, F_t) ; and, for each such component, the set of terminals it contains. A trace is *realizable* if some $F_t \subseteq$ (edges assigned in T_t) is acyclic, has degree ≤ 1 at terminals and ≤ 2 elsewhere, has final degrees (1 at terminals, 0 or 2 elsewhere) at every node of $G_{T_t} - V_t$, and no component of it contains two terminals of different pairs. There are 3^3 degree vectors, at most 5 partitions of V_t , and at most $3k + 1$ terminal sets per block (none, one of the $2k$ terminals, or one of the k pairs), so $O(k^3)$ traces per piece.

Recurrence. For t with children $t_1, \dots, t_d$, start from the trace of $F_t = \emptyset$ on V_t (degrees 0, singletons, each block labelled by its own terminal), branch on including each edge of E_t , and then absorb the children one at a time: a realizable trace of t_j is first restricted to $V_t \cap V_{t_j}$, keeping it only if every node of $V_{t_j} - V_t$ has final degree and every component disjoint from V_t holds a complete pair $\{s_i, t_i\}$ (or is an unused non-terminal); then it is glued on by adding degrees, merging blocks that share a node and taking unions of terminal sets. A glued trace is discarded if a degree bound is exceeded, if two terminals of different pairs land in one block, or if gluing creates a cycle, which is visible on the traces alone: merging the child's blocks into the current blocks one at a time, a cycle appears exactly when a child block contains two nodes that already lie in one merged block. Because the edge sets assigned to different subtrees are disjoint and, by (10.13), $G_{T_{t_1}} - V_t, \dots, G_{T_{t_d}} - V_t$ share no nodes and no edges, every union of realizations of the glued traces is a realization of the result, and conversely every F_t splits into $F \cap E_t$ and the F_{t_j} , whose traces are realizable. Induction up the tree therefore shows the recurrence produces exactly the realizable traces.

Answer. The instance is a yes-instance iff the root r has a realizable trace in which every node of V_r has final degree and no block holds a single terminal: then (V, F) is acyclic with degrees 1 at terminals and 0, 2 elsewhere, so its non-trivial components are paths whose two ends are terminals of the same pair, giving all k paths P_i ; the converse is immediate. Following back-pointers recovers the paths.

Running time. Each of the $\leq n$ pieces has $O(k^3)$ traces; absorbing one child compares $O(k^3) \cdot O(k^3)$ pairs in constant time each, and there are $n - 1$ child absorptions in total, so the algorithm runs in $O(nk^6) = O(n^7)$ time.

Problem 10.7. The *chromatic number* of a graph G is the minimum k such that it has a k -coloring. As we saw in Chapter 8, it is NP-complete for $k \geq 3$ to decide whether a given input graph has chromatic number $\leq k$.

(a) Show that for every natural number $w \geq 1$, there is a number $k(w)$ so that the following holds. If G is a graph of tree-width at most w , then G has chromatic number at most $k(w)$. (The point is that $k(w)$ depends only on w , not on the number of nodes in G .)

(b) Given an undirected n -node graph $G = (V, E)$ of tree-width at most w , show how to compute the chromatic number of G in time $O(f(w) \cdot p(n))$, where $p(\cdot)$ is a polynomial but $f(\cdot)$ can be an arbitrary function.

Solution. (a) Take $k(w) = w + 1$. Induct on n , the case $n = 1$ being clear. Take a nonredundant tree decomposition of G of width $\leq w$ and a leaf t of T ; as in the proof of (10.16), some node $v \in V_t$ lies in no other piece. Every edge at v lies in a piece containing v , i.e. in V_t , so

$$\deg(v) \leq |V_t| - 1 \leq w.$$

The graph $G - v$ has tree-width $\leq w$ (a subgraph never has larger tree-width, as shown in the proof of (10.15)), so by induction it has a $(w + 1)$ -coloring; at most w colors appear on the neighbors of v , so a free color remains for v .

(b) Find a decomposition, then for $c = 1, 2, \dots, w + 1$ test c -colorability by dynamic programming over

it, in the style of (10.18); the answer is the first c that succeeds, and some $c \leq w + 1$ succeeds by (a).
Decomposition. Run the algorithm of (10.21) with parameter $w + 1$. Since G has tree-width $\leq w < w + 1$, it must return a tree decomposition $(T, \{V_t\})$ of width $< 4(w + 1)$, so $|V_t| \leq 4w + 4$; make it nonredundant, so T has at most n nodes by (10.16), and root it.

Subproblems. For a node t and a proper c -coloring φ of $G[V_t]$, let $g_t(\varphi)$ be true iff φ extends to a proper c -coloring of G_{T_t} , the subgraph induced by the pieces in the subtree T_t . If t has children $t_1, \dots, t_d$,

$$g_t(\varphi) = \bigwedge_{i=1}^d \bigvee \left\{ g_{t_i}(\psi) : \psi = \varphi \text{ on } V_t \cap V_{t_i} \right\}.$$

Necessity is clear (restrict an extension to each $G_{T_{t_i}}$). Sufficiency: pick such a ψ_i with an extension χ_i for each i ; by (10.13) the sets $V(G_{T_{t_i}}) - V_t$ are disjoint, so φ and the χ_i combine into one coloring χ of G_{T_t} , which agrees with χ_i on all of $G_{T_{t_i}}$. Every edge of G_{T_t} lies in V_t (properly colored by φ) or meets some $G_{T_{t_i}} - V_t$; in the latter case it lies in a piece containing that end, which is in T_{t_i} , so it is an edge of $G_{T_{t_i}}$ properly colored by χ_i . Hence χ is proper. At the root, G is c -colorable iff $g_r(\varphi)$ holds for some φ .

Running time. Each piece has at most $c^{|V_t|} \leq (w + 1)^{4w+4}$ colorings, each checked for properness in $O(w^2)$ time, and evaluating the recurrence across one tree edge compares $O((w + 1)^{8w+8})$ pairs at $O(w)$ cost each. Over $\leq n$ tree edges and $w + 1$ values of c , the dynamic program takes $O(f_1(w)n)$ time; step (10.21) takes $O(f_2(w)mn)$ and $m \leq wn$ by (a)'s degree bound applied repeatedly. The total is $O(f(w) \cdot n^2)$.

10.2 Exercises 10.8–10.9

Problem 10.8. Consider the class of 3-SAT instances in which each of the n variables occurs (counting positive and negated appearances combined) in exactly three clauses. Show that any such instance of 3-SAT is in fact satisfiable, and that a satisfying assignment can be found in polynomial time.

Solution. Match each clause to a distinct variable occurring in it, and set that variable so that its literal in the matched clause is true.

Let the clauses be $C_1, \dots, C_m$, each with exactly three literals. Counting literal occurrences two ways gives $3m = 3n$, so $m = n$. Build the bipartite graph H with sides $X = \{C_1, \dots, C_m\}$ and $Y = \{x_1, \dots, x_n\}$ and an edge (C_j, x_i) whenever x_i or $\bar{x}_i$ occurs in C_j . For $A \subseteq X$ let $\Gamma(A)$ be its set of neighbors. The $3|A|$ literal occurrences in the clauses of A are all occurrences of variables in $\Gamma(A)$, and each variable has only 3 occurrences in total, so

$$3|A| \leq 3|\Gamma(A)|, \quad \text{i.e.} \quad |\Gamma(A)| \geq |A|.$$

Since $|X| = |Y|$, (7.40) rules out a set A with $|\Gamma(A)| < |A|$, so H has a perfect matching M , and (7.40) finds it in $O(mn) = O(n^2)$ time. For each $(C_j, x_i) \in M$ set $x_i = 1$ if x_i occurs in C_j and $x_i = 0$ otherwise (then $\bar{x}_i$ occurs in C_j). Each variable is matched to exactly one clause, so this is a well-defined assignment, and each clause contains a literal made true by its matched variable. Hence every such instance is satisfiable, and the assignment is found in $O(n^2)$ time.

Problem 10.9. Give a polynomial-time algorithm for the following problem. We are given a binary tree $T = (V, E)$ with an even number of nodes, and a nonnegative weight on each edge. We wish to find a partition of the nodes V into two sets of *equal* size so that the weight of the cut between the two sets is as large as possible (i.e., the total weight of edges with one end in each set is as large as possible). Note that the restriction that the graph is a tree is crucial here, but the assumption that the tree is binary is not. The problem is NP-hard in general graphs.

Solution. Root T and compute, bottom-up, the best cut inside each subtree for every possible number of its nodes on side A and every side of its root.

Root T at r ; let T_v be the subtree at v , $n_v = |T_v|$, and w_e the weight of edge e . For $0 \leq j \leq n_v$ and $b \in \{A, B\}$ let $\text{OPT}_v(j, b)$ be the maximum, over partitions of T_v into (A_v, B_v) with $|A_v| = j$ and v on side b , of the weight of edges of T_v cut by the partition ($-\infty$ if no such partition exists). Process the children $c_1, \dots, c_d$ of v one at a time: let $S_i = \{v\} \cup T_{c_1} \cup \dots \cup T_{c_i}$ and $g_i(j, b)$ the analogous optimum on S_i , so $g_0(1, A) = g_0(0, B) = 0$ and all other g_0 values are $-\infty$. Then

$$g_i(j, b) = \max_{\ell, b'} \left\{ g_{i-1}(j - \ell, b) + \text{OPT}_{c_i}(\ell, b') \right. \\ \left. + w_{(v, c_i)} \cdot [b \neq b'] \right\},$$

and $\text{OPT}_v = g_d$. This is correct because the edges of the tree on S_i are the edges on S_{i-1} , the edges of T_{c_i} , and the single edge (v, c_i) , while S_{i-1} and T_{c_i} are disjoint. So a partition of S_i is exactly a partition of S_{i-1} together with one of T_{c_i} , its A -count is the sum of the two counts, and its cut weight is the sum of the two cut weights plus $w_{(v, c_i)}$ when v and c_i are on different sides. Each part can be optimized independently once $(j - \ell, b)$ and (ℓ, b') are fixed, so induction on n_v gives correctness. The answer is

$$\max \{ \text{OPT}_r(n/2, A), \text{OPT}_r(n/2, B) \},$$

and the partition is recovered by back-pointers.

Running time. Absorbing c_i costs $O(|S_{i-1}| \cdot n_{c_i})$, one unit for each pair consisting of a node already in S_{i-1} and a node of T_{c_i} . Each pair of nodes of T is counted this way at most once, at their lowest common ancestor, so the total is $O(n^2)$.

11 Approximation Algorithms

Contents

11.1 Exercises 11.1–11.7	176
11.2 Exercises 11.8–11.12	180

11.1 Exercises 11.1–11.7

Problem 11.1. You are acting as a consultant for the Port Authority of a small Pacific Rim nation. They do a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port. A ship arrives with n containers of weights $w_1, w_2, \dots, w_n$. On the dock is a set of trucks, each of which can hold K units of weight (K and every w_i are integers). You may stack several containers in a truck as long as the total weight is at most K . The goal is to minimize the number of trucks needed to carry all the containers. This problem is NP-complete (you do not have to prove this).

Consider this greedy algorithm. Start with an empty truck and pile containers $1, 2, 3, \dots$ into it until you reach a container that would exceed the weight limit. Declare this truck “loaded” and send it off, then continue with a fresh truck. Because it considers the trucks one at a time, this algorithm may not pack the containers as efficiently as possible.

- Give a set of weights and a value of K for which this algorithm does not use the minimum possible number of trucks.
- Show that for any set of weights and any value of K , the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number.

Solution. (a) Take $K = 2$ and $(w_1, w_2, w_3) = (1, 2, 1)$. The greedy algorithm loads $\{1\}$ and sends it off because $1 + 2 > 2$. It then loads $\{2\}$ and sends it off because $2 + 1 > 2$, and finally loads $\{1\}$. That is 3 trucks, but the packing $\{w_1, w_3\}, \{w_2\}$ uses only 2.

(b) Any two consecutive greedy loads exceed K : if the greedy algorithm uses k trucks with loads $L_1, \dots, L_k$, truck $j+1$ was started only because its first container did not fit on top of L_j , so $L_j + L_{j+1} > K$ for $1 \leq j < k$. (Every $w_i \leq K$, or no packing exists.) With $W = \sum_i w_i$ and k^* the optimum, summing over the $k-1$ consecutive pairs gives

$$(k-1)K < \sum_{j=1}^{k-1} (L_j + L_{j+1}) \leq 2 \sum_{j=1}^k L_j = 2W \leq 2Kk^*,$$

where the last step holds because each of the k^* optimal trucks carries at most K . Hence $k^* > (k-1)/2$. Since k^* is an integer, $k^* \geq k/2$, that is, $k \leq 2k^*$.

Problem 11.2. At a computational biology conference, a protein chemist proposed building a “representative set” for a large collection of protein molecules whose properties we do not understand. The proteins in the representative set would be studied intensively, and by inference we would learn about every protein in the full collection. To be useful, the representative set must have two properties.

- It should be relatively small, so that studying it is not too expensive.
- Every protein in the full collection should be “similar” to some protein in the representative set, so that the set gives some information about all the proteins.

More concretely, there is a large set P of n proteins. Similarity is defined by a distance function d : for two proteins p and q it returns a number $d(p, q) \geq 0$. The distance used here is the sequence alignment measure from Chapter 6. A distance cut-off Δ is given as part of the input, and two proteins p and q are “similar” if and only if $d(p, q) \leq \Delta$. A subset of P is a *representative set* if for every protein p there is a protein q in the subset with $d(p, q) \leq \Delta$. The goal is to find a representative set that is as small as possible.

- Give a polynomial-time algorithm that approximates the minimum representative set within a factor of $O(\log n)$. That is, if the smallest possible representative set has size s^* , your algorithm should return a representative set of size $O(s^* \log n)$.
- This problem is closely related to the Center Selection Problem, for which Section 11.2 gives approximation algorithms. Why does the algorithm described there not solve the current problem?

Solution. (a) Run Greedy-Set-Cover (Section 11.3) with unit weights on the universe $U = P$ and the sets

$$S_q = \{p \in P : d(p, q) \leq \Delta\}, \quad q \in P.$$

A subset $R \subseteq P$ is representative exactly when $\bigcup_{q \in R} S_q = P$, so representative sets correspond one-to-one to set covers $\{S_q : q \in R\}$ of the same size, and the optimal cover has s^* sets. The instance is feasible, because $d(p, p) = 0$ (an exact self-alignment costs nothing) and so P itself is representative. Every set has $|S_q| \leq n$, so by (11.11) the greedy cover has size at most

$$H(n) s^* \leq (1 + \ln n) s^* = O(s^* \log n).$$

For the running time, the n^2 distances are each computed by the polynomial-time sequence alignment dynamic program of Section 6.6. Building the sets and running Greedy-Set-Cover then takes polynomial time.

(b) The Center Selection algorithm fixes the number of centers k and approximates the radius, and it does so only within a factor of 2. By (11.6), a run with radius $r = \Delta$ that succeeds with at most k centers only guarantees that every protein lies within 2Δ of a center. Here the cut-off Δ is a hard constraint and the number of centers is what we must keep small, so a cover at radius 2Δ is not a representative set at all. The analysis in (11.6) and (11.7) also relies on the triangle inequality, which a sequence alignment distance with arbitrary gap and mismatch costs need not satisfy.

Problem 11.3. You are given a set of positive integers $A = \{a_1, a_2, \dots, a_n\}$ and a positive integer B . A subset $S \subseteq A$ is *feasible* if the sum of its numbers does not exceed B :

$$\sum_{a_i \in S} a_i \leq B.$$

The sum of the numbers in S is called the *total sum* of S . You want to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, the optimal solution is $S = \{8, 2\}$.

(a) Here is an algorithm for this problem.

```
Initially S = empty set
Define T = 0
For i = 1, 2, ..., n
  If T + a_i <= B then
    S <- S union {a_i}
    T <- T + a_i
  Endif
Endfor
```

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A .

(b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: it returns a feasible set $S \subseteq A$ whose total sum is at least half the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should run in time at most $O(n \log n)$.

Solution. (a) Take $A = \{1, 10\}$ with $a_1 = 1$, $a_2 = 10$, and $B = 10$. The algorithm takes a_1 , then rejects a_2 because $1 + 10 > 10$, and returns total sum 1. The feasible set $\{10\}$ has total sum 10, and $1 < 10/2$. (b) Discard every $a_i > B$, sort the remaining numbers in decreasing order, and run the algorithm of (a) on them. Let T be the total it returns and OPT the best feasible total. No discarded number can belong to a feasible set.

(i) If every remaining number is accepted, then S contains every number that can appear in a feasible set, so $T = \text{OPT}$.

(ii) Otherwise some a_j is rejected, and let T_j be the running total at that moment. The largest remaining number $a_1 \leq B$ is always accepted (it is examined first, with $T = 0$), and $a_1 \geq a_j$ by the sorting, so $T_j \geq a_1 \geq a_j$. Therefore

$$2T \geq 2T_j \geq T_j + a_j > B \geq \text{OPT}.$$

In both cases $T \geq \text{OPT}/2$. The sort takes $O(n \log n)$ time and the filter and the single pass take $O(n)$, for $O(n \log n)$ in total.

Problem 11.4. Consider the following optimization version of the Hitting Set Problem. We are given a set $A = \{a_1, \dots, a_n\}$ and a collection $B_1, B_2, \dots, B_m$ of subsets of A , and each element $a_i \in A$ has a weight $w_i \geq 0$. The problem is to find a hitting set $H \subseteq A$ whose total weight $\sum_{a_i \in H} w_i$ is as small as possible. (As in Exercise 5 of Chapter 8, H is a hitting set if $H \cap B_i$ is nonempty for each i .) Let $b = \max_i |B_i|$ be the maximum size of any of the sets $B_1, \dots, B_m$. Give a polynomial-time approximation algorithm for this problem that finds a hitting set whose total weight is at most b times the minimum possible.

Solution. Use the pricing method of Section 11.4. Here the sets B_i play the role of the edges and the elements play the role of the vertices. Each set B_i pays a price $p_i \geq 0$. The prices are *fair* if $\sum_{i: a \in B_i} p_i \leq w_a$ for every element a , and an element is *tight* when equality holds. (Assume each $B_i \neq \emptyset$; otherwise no hitting set exists.)

```

Set p_i = 0 for all i
While some set B_i contains no tight element
    Raise p_i by min over a in B_i of (w_a - sum of p_k with a in B_k)
EndWhile
Return H = the set of tight elements

```

Each raise is the largest increase that keeps the prices fair, so the prices stay fair throughout and at least one element of B_i becomes tight. When the loop stops, every B_i contains a tight element, so H is a hitting set.

Fair prices give a lower bound on every hitting set H^* , exactly as in (11.13). Every B_i meets H^* and all prices are nonnegative, so

$$\sum_{i=1}^m p_i \leq \sum_{i=1}^m \sum_{a \in H^* \cap B_i} p_i = \sum_{a \in H^*} \sum_{i: a \in B_i} p_i \leq w(H^*).$$

All elements of H are tight, and a price p_i is counted once for each element of $H \cap B_i$, which happens at most $|B_i| \leq b$ times. So, as in (11.14),

$$w(H) = \sum_{a \in H} \sum_{i: a \in B_i} p_i = \sum_{i=1}^m |H \cap B_i| p_i \leq b \sum_{i=1}^m p_i \leq b w(H^*).$$

Taking H^* optimal gives the factor b . Each iteration makes some new set B_i contain a tight element, and tight elements stay tight, so there are at most m iterations. Each iteration takes $O(\sum_i |B_i|)$ time, so the total time is $O(m \sum_i |B_i|)$, which is polynomial.

Problem 11.5. You are consulting for a business where clients bring in jobs each day for processing. Each job has a processing time t_i that is known when the job arrives. The company has ten machines, and each job can be processed on any of them.

The business currently runs the simple Greedy-Balance Algorithm from Section 11.1. They like its simplicity: each job is assigned to a machine as soon as it arrives, without waiting for later jobs. They have heard that this algorithm can produce a makespan as large as twice the minimum possible. In their experience, though, after running it every day for the last month, they have never seen it produce a makespan more than 20 percent above the average load $\frac{1}{10} \sum_i t_i$.

When you ask about their jobs and loads, you learn that job sizes range between 1 and 50, that is, $1 \leq t_i \leq 50$ for all jobs i . You also learn that the total load $\sum_i t_i$ is always at least 3,000 each day. Prove that on inputs of this kind, the Greedy-Balance Algorithm always finds a solution whose makespan is at most 20 percent above the average load.

Solution. The makespan satisfies $T \leq \bar{L} + 50 \leq \frac{7}{6}\bar{L} < 1.2\bar{L}$, where $\bar{L} = \frac{1}{10} \sum_k t_k \geq 300$ is the average load.

Let machine M_i attain the makespan $T = T_i$, and let j be the last job assigned to it. As in the proof of (11.3), M_i had the smallest load of the ten machines when j was assigned. Its load then was $T_i - t_j$, so ten times that load is at most the total load at that moment, which is at most the final total. Hence $T_i - t_j \leq \frac{1}{10} \sum_k t_k = \bar{L}$. With $t_j \leq 50$ and $\bar{L} \geq 3000/10 = 300$, this gives

$$T = (T_i - t_j) + t_j \leq \bar{L} + 50 \leq \bar{L} + \frac{1}{6}\bar{L} = \frac{7}{6}\bar{L} < 1.2\bar{L}.$$

Problem 11.6. Recall that in the basic Load Balancing Problem from Section 11.1, we're interested in placing jobs on machines so as to minimize the *makespan* — the maximum load on any one machine. In a number of applications, it is natural to consider cases in which you have access to machines with different amounts of processing power, so that a given job may complete more quickly on one of your machines than on another. The question then becomes: How should you allocate jobs to machines in these more heterogeneous systems?

Here's a basic model that exposes these issues. Suppose you have a system that consists of m *slow* machines and k *fast* machines. The fast machines can perform twice as much work per unit time as the slow machines. Now you're given a set of n jobs; job i takes time t_i to process on a slow machine and time $\frac{1}{2}t_i$ to process on a fast machine. You want to assign each job to a machine; as before, the goal is to minimize the makespan — that is the maximum, over all machines, of the total processing time of jobs assigned to that machine.

Give a polynomial-time algorithm that produces an assignment of jobs to machines with a makespan that is at most three times the optimum.

Solution. Run Greedy-Balance with “earliest finish”: take the jobs in any order and put job i on the machine on which it would finish soonest, i.e. on a slow machine q minimizing $L_q + t_i$ or a fast machine q minimizing $L_q + \frac{1}{2}t_i$, whichever is smaller (L_q the current load).

Let T^* be the optimal makespan and $W = \sum_i t_i$. In time T^* a slow machine does at most T^* units of (slow-machine) work and a fast one at most $2T^*$, and job i takes at least $\frac{1}{2}t_i$ wherever it runs, so (the analogues of (11.1) and (11.2))

$$T^* \geq \frac{W}{m+2k}, \quad T^* \geq \frac{1}{2} \max_i t_i.$$

Let machine q attain the greedy makespan F , and let j be the last job placed on it. When j was placed, its finish time on q was F and this was the minimum over all machines, so every slow machine then had load $\geq F - t_j$ and every fast machine load $\geq F - \frac{1}{2}t_j$; a fast machine with load L holds $2L$ units of work. All these jobs differ from j , so

$$\begin{aligned} W - t_j &\geq m(F - t_j) + 2k(F - \tfrac{1}{2}t_j) = (m+2k)F - (m+k)t_j, \\ F &\leq \frac{W}{m+2k} + \frac{m+k-1}{m+2k}t_j \\ &\leq T^* + t_j \leq T^* + 2T^* = 3T^*. \end{aligned}$$

Each job is placed by scanning the $m+k$ machines, so the running time is $O(n(m+k))$ (or $O(n \log(m+k))$ with two priority queues, one per machine type).

Problem 11.7. You're consulting for an e-commerce site that receives a large number of visitors each day. For each visitor i , where $i \in \{1, 2, \dots, n\}$, the site has assigned a value v_i , representing the expected revenue that can be obtained from this customer.

Each visitor i is shown one of m possible ads $A_1, A_2, \dots, A_m$ as they enter the site. The site wants a selection of one ad for each customer so that *each* ad is seen, overall, by a set of customers of reasonably large total weight. Thus, given a selection of one ad for each customer, we will define the

spread of this selection to be the minimum, over $j = 1, 2, \dots, m$, of the total weight of all customers who were shown ad A_j .

Example. Suppose there are six customers with values 3, 4, 12, 2, 4, 6, and there are $m = 3$ ads. Then, in this instance, one could achieve a spread of 9 by showing ad A_1 to customers 1, 2, 4, ad A_2 to customer 3, and ad A_3 to customers 5 and 6.

The ultimate goal is to find a selection of an ad for each customer that maximizes the spread. Unfortunately, this optimization problem is NP-hard (you don't have to prove this). So instead, we will try to approximate it.

(a) Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s , then your algorithm should produce a selection of one ad for each customer that has spread at least $s/2$. In designing your algorithm, you may assume that no single customer has a value that is significantly above the average; specifically, if $\bar{v} = \sum_{i=1}^n v_i$ denotes the total value of all customers, then you may assume that no single customer has a value exceeding $\bar{v}/(2m)$.

(b) Give an example of an instance on which the algorithm you designed in part (a) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your sample instance, and what your algorithm finds.

Solution. (a) Greedy: take the customers in order $1, \dots, n$ and show each one the ad whose current total weight W_j is smallest (ties to the lowest index).

Let $W_1, \dots, W_m$ be the final weights and $W = \min_j W_j$ the greedy spread. Some ad has weight at most the average, so $s \leq \bar{v}/m$. Fix any ad A_j with $W_j > 0$ and let i be the last customer shown A_j . When i arrived, A_j had the minimum weight, which is at most the then-current (hence at most the final) weight of the minimum ad, so

$$W_j \leq W + v_i \leq W + \frac{\bar{v}}{2m},$$

and this holds trivially when $W_j = 0$. Summing over the m ads, one of which has weight exactly W ,

$$\begin{aligned} \bar{v} &= \sum_j W_j \leq mW + (m-1) \frac{\bar{v}}{2m} \leq mW + \frac{\bar{v}}{2}, \\ \text{so } W &\geq \frac{\bar{v}}{2m} \geq \frac{s}{2}. \end{aligned}$$

With a heap keyed on W_j the algorithm runs in $O(n \log m)$ time.

(b) Take $m = 2$ and values 1, 1, 2, 2, 2 in this order; $\bar{v} = 8$ and every $v_i \leq 2 = \bar{v}/(2m)$. Greedy gives customer 1 to A_1 , 2 to A_2 , 3 to A_1 , 4 to A_2 (weights 3, 3), and 5 to A_1 , ending with weights 5 and 3: spread 3. The optimum shows A_1 to customers 1, 2, 3 and A_2 to customers 4, 5, with weights 4, 4: spread 4 (optimal since $s \leq \bar{v}/m = 4$).

11.2 Exercises 11.8–11.12

Problem 11.8. Some friends of yours are working with a system that performs real-time scheduling of jobs on multiple servers, and they've come to you for help in getting around an unfortunate piece of legacy code that can't be changed.

Here's the situation. When a batch of jobs arrives, the system allocates them to servers using the simple Greedy-Balance Algorithm from Section 11.1, which provides an approximation to within a factor of 2. In the decade and a half since this part of the system was written, the hardware has gotten faster to the point where, on the instances that the system needs to deal with, your friends find that it's generally possible to compute an optimal solution.

The difficulty is that the people in charge of the system's internals won't let them change the portion of the software that implements the Greedy-Balance Algorithm so as to replace it with one that finds the optimal solution. (Basically, this portion of the code has to interact with so many other parts of

the system that it's not worth the risk of something going wrong if it's replaced.)

After grumbling about this for a while, your friends come up with an alternate idea. Suppose they could write a little piece of code that takes the description of the jobs, computes an optimal solution (since they're able to do this on the instances that arise in practice), and then feeds the jobs to the Greedy-Balance Algorithm *in an order that will cause it to allocate them optimally*. In other words, they're hoping to be able to reorder the input in such a way that when Greedy-Balance encounters the input in this order, it produces an optimal solution.

So their question to you is simply the following: Is this always possible? Their conjecture is,

For every instance of the load balancing problem from Section 11.1, there exists an order of the jobs so that when Greedy-Balance processes the jobs in this order, it produces an assignment of jobs to machines with the minimum possible makespan.

Decide whether you think this conjecture is true or false, and give either a proof or a counterexample.

Solution. True: simulate Greedy-Balance and always feed it a job that an optimal schedule (re-targeted as we go) still wants on the machine Greedy-Balance is about to load.

Fix an optimal assignment with job sets $B_1, \dots, B_m$ and makespan T^* . We maintain *target* sets $B_1, \dots, B_m$ partitioning the jobs, with $\sum_{j \in B_q} t_j \leq T^*$ for every q , such that the jobs Greedy-Balance has so far put on machine q form a subset $C_q \subseteq B_q$; let $R_q = B_q \setminus C_q$ and $L_q = \sum_{j \in C_q} t_j$. Initially $C_q = \emptyset$. While jobs remain, let q be the machine Greedy-Balance will load next (a machine of minimum L_q , under its own tie-breaking rule, which we can simulate).

(i) $R_q \neq \emptyset$: feed any job of R_q ; it goes to q , and the invariant is kept.

(ii) $R_q = \emptyset$: pick p with $R_p \neq \emptyset$ and swap the remainders, setting $B_q := C_q \cup R_p$ and $B_p := C_p$; then feed a job of the new R_q as in (i). The new targets still have load at most T^* , because $L_q \leq L_p$:

$$\begin{aligned}\sum_{B_q} t_j &= L_q + \sum_{R_p} t_j \leq L_p + \sum_{R_p} t_j \leq T^*, \\ \sum_{B_p} t_j &= L_p \leq T^*.\end{aligned}$$

Each step assigns one job and keeps the invariant, so after n steps every R_q is empty and Greedy-Balance's machine q holds exactly B_q , of load at most T^* ; its makespan is therefore the optimum T^* .

Problem 11.9. Consider the following maximization version of the 3-Dimensional Matching Problem. Given disjoint sets X , Y , and Z , and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subseteq T$ is a *3-dimensional matching* if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The *Maximum 3-Dimensional Matching Problem* is to find a 3-dimensional matching M of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

Give a polynomial-time algorithm that finds a 3-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

Solution. Greedily build a maximal matching: scan the triples of T once, adding a triple to M whenever none of its three elements is already covered by M .

M is a 3-dimensional matching by construction, and it is maximal: a triple rejected during the scan shares an element with a triple already in M . Let M^* be a maximum matching. Each $t \in M^*$ meets some triple of M (if $t \in M$ it meets itself), so we may charge t to one element $e(t) \in t$ covered by M . The triples of M^* are disjoint, so $t \mapsto e(t)$ is injective into the $3|M|$ elements covered by M :

$$|M^*| \leq 3|M|, \quad \text{i.e.} \quad |M| \geq \frac{1}{3}|M^*|.$$

With a covered-flag per element of $X \cup Y \cup Z$, each triple is tested in $O(1)$, so the running time is $O(|T| + |X| + |Y| + |Z|)$.

Problem 11.10. Suppose you are given an $n \times n$ grid graph G , as in Figure 11.13 (nodes at the points (r, c) , $1 \leq r, c \leq n$, with an edge between two nodes exactly when they are horizontally or vertically adjacent, so every node has at most four neighbors).

Associated with each node v is a *weight* $w(v)$, which is a nonnegative integer. You may assume that the weights of all nodes are distinct. Your goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its *total weight*.)

Consider the following greedy algorithm for this problem.

```
The "heaviest-first" greedy algorithm:
  Start with S equal to the empty set
  While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
  Endwhile
  Return S
```

(a) Let S be the independent set returned by the “heaviest-first” greedy algorithm, and let T be any other independent set in G . Show that, for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w(v) \leq w(v')$ and (v, v') is an edge of G .

(b) Show that the “heaviest-first” greedy algorithm returns an independent set of total weight at least $\frac{1}{4}$ times the maximum total weight of any independent set in the grid graph G .

Solution. (a) If $v \in T \setminus S$, the node v' chosen in the iteration that deleted v works. Since $v \notin S$, v was deleted as a neighbor of the chosen node v' , so (v, v') is an edge and $v' \in S$; and v was still present when v' was picked as a node of maximum weight, so $w(v) \leq w(v')$.

(b) Map each $v \in T$ to $f(v) \in S$: $f(v) = v$ if $v \in S$, and otherwise $f(v) = v'$ from (a), so $w(v) \leq w(f(v))$ always. If $u \in S \cap T$, then $f^{-1}(u) = \{u\}$, because a $v \in T$ with $f(v) = u \neq v$ would be adjacent to $u \in T$, contradicting independence of T . If $u \in S \setminus T$, then $f^{-1}(u)$ consists of neighbors of u , so $|f^{-1}(u)| \leq 4$. Hence, for T a maximum-weight independent set,

$$w(T) = \sum_{u \in S} \sum_{v \in f^{-1}(u)} w(v) \leq \sum_{u \in S} |f^{-1}(u)| w(u) \leq 4 w(S).$$

Problem 11.11. Recall that in the Knapsack Problem, we have n items, each with a weight w_i and a value v_i . We also have a weight bound W , and the problem is to select a set of items S of highest possible value subject to the condition that the total weight does not exceed W – that is, $\sum_{i \in S} w_i \leq W$. Here’s one way to look at the approximation algorithm that we designed in this chapter. If we are told there exists a subset $\mathcal{O}$ whose total weight is $\sum_{i \in \mathcal{O}} w_i \leq W$ and whose total value is $\sum_{i \in \mathcal{O}} v_i = V$ for some V , then our approximation algorithm can find a set $\mathcal{A}$ with total weight $\sum_{i \in \mathcal{A}} w_i \leq W$ and total value at least $\sum_{i \in \mathcal{A}} v_i \geq V/(1+\epsilon)$. Thus the algorithm approximates the best value, while keeping the weights strictly under W . (Of course, returning the set $\mathcal{O}$ is always a valid solution, but since the problem is NP-hard, we don’t expect to always be able to find $\mathcal{O}$ itself; the approximation bound of $1 + \epsilon$ means that other sets $\mathcal{A}$, with slightly less value, can be valid answers as well.)

Now, as is well known, you can always pack a little bit more for a trip just by “sitting on your suitcase” – in other words, by slightly overflowing the allowed weight limit. This too suggests a way of formalizing the approximation question for the Knapsack Problem, but it’s the following, different, formalization.

Suppose, as before, that you’re given n items with weights and values, as well as parameters W and V ; and you’re told that there is a subset $\mathcal{O}$ whose total weight is $\sum_{i \in \mathcal{O}} w_i \leq W$ and whose total value is $\sum_{i \in \mathcal{O}} v_i = V$ for some V . For a given fixed $\epsilon > 0$, design a polynomial-time algorithm that finds a subset of items $\mathcal{A}$ such that $\sum_{i \in \mathcal{A}} w_i \leq (1+\epsilon)W$ and $\sum_{i \in \mathcal{A}} v_i \geq V$. In other words, you want $\mathcal{A}$ to achieve at least as high a total value as the given bound V , but you’re allowed to exceed the

weight limit W by a factor of $1 + \epsilon$.

Example. Suppose you're given four items, with weights and values as follows:

$$(w_1, v_1) = (5, 3), \quad (w_2, v_2) = (4, 6), \quad (w_3, v_3) = (1, 4), \quad (w_4, v_4) = (6, 11).$$

You're also given $W = 10$ and $V = 13$ (since, indeed, the subset consisting of the first three items has total weight at most 10 and has value 13). Finally, you're given $\epsilon = .1$. This means you need to find (via your approximation algorithm) a subset of weight at most $(1 + .1) \cdot 10 = 11$ and value at least 13. One valid solution would be the subset consisting of the first and fourth items, with value $14 \geq 13$. (Note that this is a case where you're able to achieve a value strictly greater than V , since you're allowed to slightly overfill the knapsack.)

Solution. Round the *weights* (not the values) up to multiples of $b = \epsilon W/n$ and solve the rounded instance exactly with the Knapsack dynamic program of Section 6.4, whose capacity is now only $O(n/\epsilon)$.

Take $W > 0$ (if $W = 0$, return the zero-weight items). Discard every item with $w_i > W$ (none of them can lie in $\mathcal{O}$). For the remaining items put

$$\tilde{w}_i = \lceil w_i/b \rceil, \quad C = \lfloor W/b \rfloor + n = \lfloor n/\epsilon \rfloor + n,$$

and run the recurrence (6.11) on weights $\tilde{w}_i$, values v_i and capacity C ; return the optimal set $\mathcal{A}$ it finds.

Value. Every $i \in \mathcal{O}$ survives the discard, and since $\sum_{\mathcal{O}} \tilde{w}_i$ is an integer,

$$\sum_{i \in \mathcal{O}} \tilde{w}_i \leq \sum_{i \in \mathcal{O}} \left(\frac{w_i}{b} + 1 \right) \leq \frac{W}{b} + n \Rightarrow \sum_{i \in \mathcal{O}} \tilde{w}_i \leq C,$$

so $\mathcal{O}$ is feasible for the rounded instance and the optimum $\mathcal{A}$ has $\sum_{\mathcal{A}} v_i \geq \sum_{\mathcal{O}} v_i = V$.

Weight. Since $w_i \leq b\tilde{w}_i$,

$$\begin{aligned} \sum_{i \in \mathcal{A}} w_i &\leq b \sum_{i \in \mathcal{A}} \tilde{w}_i \leq bC \\ &\leq b \left(\frac{W}{b} + n \right) = W + \epsilon W = (1 + \epsilon)W. \end{aligned}$$

Running time. By (6.12) the dynamic program takes $O(nC) = O(n^2/\epsilon)$ time (plus $O(n)$ for the rounding), polynomial for fixed ϵ .

Example. Here $b = 0.25$, $(\tilde{w}_1, \dots, \tilde{w}_4) = (20, 16, 4, 24)$ and $C = 44$; the rounded optimum is $\{2, 3, 4\}$ (rounded weight 44), of true weight $11 \leq 11$ and value $21 \geq 13$.

Problem 11.12. Consider the following problem. There is a set U of n nodes, which we can think of as users (e.g., these are locations that need to access a service, such as a Web server). You would like to place servers at multiple locations. Suppose you are given a set S of possible sites that would be willing to act as locations for the servers. For each site $s \in S$, there is a fee $f_s \geq 0$ for placing a server at that location. Your goal will be to approximately minimize the cost while providing the service to each of the customers. So far this is very much like the Set Cover Problem: The places s are sets, the weight of set s is f_s , and we want to select a collection of sets that covers all users. There is one extra complication: Users $u \in U$ can be served from multiple sites, but there is an associated cost d_{us} for serving user u from site s . When the value d_{us} is very high, we do not want to serve user u from site s ; and in general the service cost d_{us} serves as an incentive to serve customers from “nearby” servers whenever possible.

So here is the question, which we call the Facility Location Problem: Given the sets U and S , and costs f and d , you need to select a subset $A \subseteq S$ at which to place servers (at a cost of $\sum_{s \in A} f_s$), and assign each user u to the active server where it is cheapest to be served, $\min_{s \in A} d_{us}$. The goal

is to minimize the overall cost $\sum_{s \in A} f_s + \sum_{u \in U} \min_{s \in A} d_{us}$. Give an $H(n)$ -approximation for this problem.

(Note that if all service costs d_{us} are 0 or infinity, then this problem is exactly the Set Cover Problem: f_s is the cost of the set named s , and d_{us} is 0 if node u is in set s , and infinity otherwise.)

Solution. Run Greedy-Set-Cover (Section 11.3) on the implicit Set Cover instance whose sets are all “stars” (s, X) with $s \in S$, $\emptyset \neq X \subseteq U$, of weight $w(s, X) = f_s + \sum_{u \in X} d_{us}$, and open the sites of the chosen stars.

Implementing the greedy step. With R the uncovered users, Greedy-Set-Cover picks the star minimizing $w(s, X)/|X \cap R|$. Replacing X by $X \cap R$ keeps the denominator and does not raise the weight (all $d_{us} \geq 0$), so we may take $X \subseteq R$; and for fixed s and $|X| = k$ the weight is smallest when X is the k users of R with the smallest d_{us} . Hence the step is an exact minimization over $|S| \cdot |R|$ candidates:

```

R = U ; A = empty
while R is nonempty:
    for each site s:
        sort R by d[u][s] into u_1, u_2, ...
        for k = 1..|R|:
            ratio(s,k) = (f_s + d[u_1][s] + ... + d[u_k][s]) / k
        take (s,k) of minimum ratio
    A = A + {s} ; R = R - {u_1, ..., u_k}
return A

```

Approximation. Let $\mathcal{C}$ be the chosen stars and A^* an optimal site set of cost OPT. Assigning each user to its cheapest site of A^* partitions U into stars (s, X_s) , $s \in A^*$, a set cover of weight exactly OPT, so the optimal cover has weight $w^* \leq \text{OPT}$. Conversely, in the facility solution A each user u covered by $(s, X) \in \mathcal{C}$ pays at most d_{us} , and each f_s is paid once even if s heads several stars. With (11.11) and $d^* \leq |U| = n$,

$$\begin{aligned}
 \sum_{s \in A} f_s + \sum_{u \in U} \min_{s \in A} d_{us} &\leq \sum_{(s,X) \in \mathcal{C}} \left(f_s + \sum_{u \in X} d_{us} \right) \\
 &\leq H(d^*) w^* \leq H(n) \text{OPT}.
 \end{aligned}$$

Running time. Each iteration covers at least one user, so there are at most n iterations, each costing $O(|S|n \log n)$ for the sorts and prefix sums; the total is $O(|S|n^2 \log n)$.

12 Local Search

Contents

12.1 Exercises 12.1–12.4	186
------------------------------------	-----

12.1 Exercises 12.1–12.4

Problem 12.1. Take a Hopfield network (Section 12.3) in which every edge weight is positive, so that, as in the Maximum-Cut problem, the two ends of each edge want opposite states. Assume further that the graph G is connected and bipartite with sides X and Y . Then setting every node of X to $+1$ and every node of Y to -1 makes every edge good, which is plainly the best configuration. Does the State-Flipping Algorithm from the text (while some node is unsatisfied, pick one and flip its state) always reach this configuration? Either prove that it does, or give an input instance, a starting configuration and a run of the State-Flipping Algorithm that stops at a configuration where some edge is not good.

Solution. No. Let G be the 6-cycle $v_1v_2v_3v_4v_5v_6v_1$ with every weight $w_e = 1$. This graph is connected and bipartite, with $X = \{v_1, v_3, v_5\}$ and $Y = \{v_2, v_4, v_6\}$. By the definition in Section 12.3, node u is satisfied exactly when $\sum_{v: e=(u,v)} w_e s_u s_v \leq 0$. For a node of degree 2 with unit weights, this means at least one of its two edges is good.

Start with every node at $+1$. All six edges are bad, so every node is unsatisfied. The algorithm may run as follows.

(i) Flip v_1 , which is unsatisfied. Now $s = (-1, +1, +1, +1, +1, +1)$, and v_4 still has two bad edges, so it is unsatisfied.

(ii) Flip v_4 . Now $s = (-1, +1, +1, -1, +1, +1)$.

In this configuration the edges $v_1v_2, v_6v_1, v_3v_4, v_4v_5$ are good and the edges v_2v_3, v_5v_6 are bad. For each node, $\sum_v s_u s_v$ is

$$\begin{aligned} v_1, v_4 : -1 - 1 &= -2 \leq 0, \\ v_2, v_3, v_5, v_6 : +1 - 1 &= 0 \leq 0, \end{aligned}$$

so every node is satisfied and the State-Flipping Algorithm stops here. Two of the six edges are not good, so the algorithm did not reach the all-good configuration. ■

Problem 12.2. For a maximization problem, the analogue of gradient descent is gradient ascent (hill climbing): repeatedly move from the current solution to one of strictly larger value. As with Maximum-Cut and Labeling, some problems have the property that every local optimum is close in value to the global optimum; this exercise shows that Bipartite Matching is one of them. Consider the following gradient ascent algorithm for a bipartite graph: while some edge has both endpoints unmatched, add it to the current matching; when no such edge remains, stop and output the resulting locally optimal matching.

(a) Give a bipartite graph G on which this algorithm can fail to return a maximum matching.

(b) Let M and M' be matchings in a bipartite graph G with $|M'| > 2|M|$. Show that some edge $e' \in M'$ has the property that $M \cup \{e'\}$ is a matching in G .

(c) Use (b) to show that every locally optimal matching returned by the algorithm on a bipartite graph G has size at least half the size of a maximum matching of G .

Solution. (a) Take left side $\{x_1, x_2\}$, right side $\{y_1, y_2\}$ and edges x_1y_1, x_1y_2, x_2y_1 (a path $y_2 - x_1 - y_1 - x_2$). If the algorithm adds x_1y_1 first, then every remaining edge (x_1y_2 and x_2y_1) has a matched endpoint, so it stops with $\{x_1y_1\}$ of size 1, whereas $\{x_1y_2, x_2y_1\}$ is a matching of size 2.

(b) Let $V(M)$ be the set of endpoints of edges of M , so $|V(M)| \leq 2|M|$. The edges of M' are pairwise vertex-disjoint, so each vertex of $V(M)$ lies on at most one edge of M' ; hence

$$\#\{e' \in M' : e' \cap V(M) \neq \emptyset\} \leq |V(M)| \leq 2|M| < |M'|.$$

So some $e' \in M'$ has neither endpoint in $V(M)$. Then $e' \notin M$, and e' shares no endpoint with any edge of M , so $M \cup \{e'\}$ is a matching.

(c) Let M be the output and M^* a maximum matching. On termination no edge has both endpoints unmatched by M , i.e. there is no edge $e \notin M$ with $M \cup \{e\}$ a matching. If $|M^*| > 2|M|$, then (b) with

$M' = M^*$ produces exactly such an edge, which is impossible. Therefore

$$|M^*| \leq 2|M|, \quad \text{i.e.} \quad |M| \geq \frac{1}{2}|M^*|.$$

Problem 12.3. Two identical machines M_1, M_2 must share n jobs; job j has processing time $t_j \geq 0$. We split the jobs into sets A (run on M_1) and B (run on M_2), giving loads $T_1 = \sum_{j \in A} t_j$ and $T_2 = \sum_{j \in B} t_j$, and we want to minimize $|T_1 - T_2|$ (this is NP-hard, by a reduction from Subset Sum). Consider the following local search: start from an arbitrary assignment (say jobs $1, \dots, n/2$ on M_1 and the rest on M_2); a local move transfers a single job to the other machine, and a move is made only if it strictly decreases $|T_1 - T_2|$.

(a) Suppose no job dominates, i.e. $t_j \leq \frac{1}{2} \sum_{i=1}^n t_i$ for every j . Prove that every locally optimal solution satisfies $\frac{1}{2}T_1 \leq T_2 \leq 2T_1$.

(b) Modify the algorithm so that, whenever several jobs could make an improving move, it moves the one with the largest t_j . Prove that under this rule each job moves at most once, so the local search stops after at most n moves.

(c) Give an instance on which the local search does not reach an optimal solution.

Solution. The improving moves are exactly the transfers of a job with $0 < t_j < d$ off the heavier machine H , where $d = |T_1 - T_2|$ and $T = T_1 + T_2$: moving a job of size t off H changes the difference to $|d - 2t|$, and moving one onto H changes it to $d + 2t$; since

$$|d - 2t| < d \iff 0 < t < d,$$

which is the claim. (*)

(a) WLOG $T_1 \geq T_2$, so $H = M_1$ (if $d = 0$ there is nothing to prove). At a local optimum, (*) says every job $j \in A$ has $t_j = 0$ or $t_j \geq d$. Let $A^+ = \{j \in A : t_j > 0\}$.

- (i) $A^+ = \emptyset$: then $T_1 = 0 \geq T_2$, so $T_1 = T_2 = 0$.
- (ii) $A^+ = \{j\}$: then $T_1 = t_j \leq T/2$, so $T_1 \leq T_2$, i.e. $T_1 = T_2$.
- (iii) $|A^+| \geq 2$: the smallest job of A^+ is at most $T_1/2$, so $T_1 - T_2 = d \leq T_1/2$, i.e. $T_1 \leq 2T_2$.

In every case $T_2 \leq T_1 \leq 2T_2$, which gives $\frac{1}{2}T_1 \leq T_2 \leq 2T_1$.

(b) Let $d_0 > d_1 > d_2 > \dots$ be the differences before successive moves (strictly decreasing by definition of a move; the run stops if $d = 0$), and let j_k be the job moved at step k .

Sizes of moved jobs are non-increasing. At step k , j_k is the largest job on H with $t < d_k$. If H is still heavier after the move, every job eligible at step $k+1$ is on H with $t < d_{k+1} < d_k$ and is not j_k , so it was eligible at step k and $t_{j_{k+1}} \leq t_{j_k}$. If the move flips the heavier side, then $t_{j_k} > d_k/2$ and

$$d_{k+1} = 2t_{j_k} - d_k < t_{j_k},$$

so by (*) every later eligible job has size $< d_{k+1} < t_{j_k}$. (†)

No job moves twice. Suppose job j moves at step k (off H_k , onto the lighter machine L_k) and again at step $m > k$. It stays on L_k in between, so by (*) L_k is the heavier machine at step m and $t_j < d_m$. Since H_k is heavier at step k and L_k at step m , some move at a step $s-1 \in [k, m-1]$ flips the heavier side; call its job i . Then $d_s = 2t_i - d_{s-1} < t_i$ by (†), and $t_i \leq t_j$ because sizes are non-increasing and $s-1 \geq k$. Hence

$$t_j < d_m \leq d_s < t_i \leq t_j,$$

a contradiction. So each job moves at most once and there are at most n moves.

(c) Take $n = 4$ with $t = (2, 2, 1, 1)$; it satisfies the hypothesis of (a), since $2 \leq \frac{1}{2} \cdot 6$. The initial assignment is $A = \{1, 2\}$, $B = \{3, 4\}$, so $T_1 = 4$, $T_2 = 2$, $d = 2$. Both jobs on the heavier machine have size $2 \not< d$, so by (*) no move improves and the algorithm (in either variant) stops with $|T_1 - T_2| = 2$. The partition $\{1, 3\}$, $\{2, 4\}$ achieves $|T_1 - T_2| = 0$, so this local optimum is not optimal.

Problem 12.4. Take the Load Balancing Problem of Section 11.1: m machines $M_1, \dots, M_m$, jobs with loads t_j , each job assigned to one machine, and the makespan is the maximum load on any machine,

$$\max_{\text{machines } M_i} \sum_{\text{jobs } j \text{ assigned to } M_i} t_j.$$

A local search heuristic starts from an arbitrary assignment and repeatedly performs *swap moves*. If $A(i)$ and $A(j)$ are the job sets of M_i and M_j , a swap move on these two machines picks subsets $B(i) \subseteq A(i)$ and $B(j) \subseteq A(j)$ (either may be empty or the whole set) and exchanges them: $A(i)$ becomes $A(i) \cup B(j) - B(i)$ and $A(j)$ becomes $A(j) \cup B(i) - B(j)$. If the loads of M_i, M_j are T_i, T_j before and T'_i, T'_j after, the move is *improving* when $\max(T'_i, T'_j) < \max(T_i, T_j)$, i.e. the larger of the two loads strictly drops. An assignment is *stable* if no improving swap move exists. The heuristic performs improving swap moves until it reaches a stable assignment and returns it.

(For example, with two machines, M_1 holding jobs of sizes 1, 3, 5, 8 and M_2 holding 2, 4, swapping the 8 for the 2 is improving; the result, $\{1, 2, 3, 5\}$ on M_1 and $\{4, 8\}$ on M_2 , is stable and has the optimal makespan 12.)

(a) Nothing in the description guarantees termination: could the heuristic cycle forever among unstable assignments? Prove that it always terminates after finitely many steps, with a stable assignment, on every instance.

(b) Prove that every stable assignment has makespan at most twice the minimum possible makespan.

Solution. The book prints $B(i) \subseteq A(j)$; the intended condition is $B(i) \subseteq A(i)$, which is what we use.

(a) The potential $\Phi = \sum_k T_k^2$ (sum of squared machine loads) strictly decreases with every improving move. A swap on M_i, M_j leaves every other load unchanged and preserves $S = T_i + T_j = T'_i + T'_j$. For any $x + y = S$ we have $|x - y| = 2 \max(x, y) - S$, so

$$\begin{aligned} T_i'^2 + T_j'^2 &= \frac{1}{2}(S^2 + (2 \max(T'_i, T'_j) - S)^2) \\ &< \frac{1}{2}(S^2 + (2 \max(T_i, T_j) - S)^2) = T_i^2 + T_j^2, \end{aligned}$$

where the strict inequality uses $S/2 \leq \max(T'_i, T'_j) < \max(T_i, T_j)$, on which range $u \mapsto (2u - S)^2$ is strictly increasing. Hence Φ strictly decreases at each step, so no assignment is ever visited twice. There are only m^n assignments of n jobs, so the heuristic stops after at most $m^n - 1$ moves, and it stops exactly when no improving move exists, i.e. at a stable assignment.

(b) Let T^* be the optimal makespan and let M_i be a machine of maximum load $T_i = T$ in a stable assignment. If $T = 0$ there is nothing to prove; otherwise M_i holds a job ℓ with $t_\ell > 0$. For any $j \neq i$, the swap with $B(i) = \{\ell\}$, $B(j) = \emptyset$ yields loads $T - t_\ell$ and $T_j + t_\ell$, and it is not improving, so (as $T - t_\ell < T = \max(T_i, T_j)$)

$$T_j + t_\ell \geq T, \quad \text{i.e.} \quad T_j \geq T - t_\ell \quad (j \neq i),$$

and trivially $T_i \geq T - t_\ell$. Summing over all machines and using (11.1),

$$m(T - t_\ell) \leq \sum_k T_k = \sum_j t_j \leq mT^*,$$

so $T - t_\ell \leq T^*$. By (11.2), $t_\ell \leq \max_j t_j \leq T^*$. Therefore

$$T = (T - t_\ell) + t_\ell \leq T^* + T^* = 2T^*.$$

13 Randomized Algorithms

Contents

13.1 Exercises 13.1–13.7	190
13.2 Exercises 13.8–13.14	194
13.3 Exercises 13.15–13.18	198

13.1 Exercises 13.1–13.7

Problem 13.1. *3-Coloring* is a yes/no question, but it has a natural optimization version. Given a graph $G = (V, E)$, we color every node with one of three colors, without requiring that adjacent nodes get different colors; an edge (u, v) is called *satisfied* when u and v receive different colors. Let c^* be the largest number of edges satisfied by any 3-coloring. Give a polynomial-time algorithm that outputs a 3-coloring satisfying at least $\frac{2}{3}c^*$ edges. A randomized algorithm is acceptable, in which case the *expected* number of satisfied edges must be at least $\frac{2}{3}c^*$.

Solution. Color every node independently and uniformly at random with one of the three colors. For an edge $e = (u, v)$ let X_e be the indicator that e is satisfied, and $X = \sum_{e \in E} X_e$. Conditioning on the color of u , the color of v is uniform and independent, so it agrees with probability $\frac{1}{3}$. By linearity of expectation,

$$\begin{aligned} E[X] &= \sum_{e \in E} E[X_e] = \sum_{e \in E} \Pr[e \text{ satisfied}] = \frac{2}{3}|E| \\ &\geq \frac{2}{3}c^*, \end{aligned}$$

since no coloring satisfies more than all $|E|$ edges. The algorithm takes $O(|V|)$ time.

Method (2) (deterministic). Order the nodes $v_1, \dots, v_n$ and color them in turn, giving v_i the color used least often among its already-colored neighbours v_j , $j < i$. If v_i has d_i such neighbours, the least-used of the three colors occurs at most $d_i/3$ times among them, so at least $\frac{2}{3}d_i$ of these edges are satisfied, and they stay satisfied since colors are never changed. Each edge is counted exactly once, at its later endpoint, so

$$\#\text{satisfied} \geq \sum_i \frac{2}{3}d_i = \frac{2}{3}|E| \geq \frac{2}{3}c^*,$$

in $O(|V| + |E|)$ time.

Problem 13.2. In a county, 100,000 people vote in an election with exactly two candidates on the ballot, a Democrat D and a Republican R . Of the voters, 80,000 arrive intending to vote for D and 20,000 arrive intending to vote for R . Because the ballot is confusing, each voter, independently and with probability $\frac{1}{100}$, votes for the candidate he or she did not intend to vote for (with only two candidates, this means the other one).

Let X be the random variable giving the number of votes received by D under this error process. Determine $E[X]$, and explain how you derived it.

Solution. $E[X] = 79,400$.

For each voter i let X_i be the indicator that i votes for D , so $X = \sum_{i=1}^{100000} X_i$. A voter intending D votes for D unless an error occurs, so $E[X_i] = \Pr[X_i = 1] = \frac{99}{100}$; a voter intending R votes for D only by error, so $E[X_i] = \frac{1}{100}$. By linearity of expectation (13.8), which needs no independence,

$$\begin{aligned} E[X] &= \sum_{i \text{ intends } D} E[X_i] + \sum_{i \text{ intends } R} E[X_i] \\ &= 80,000 \cdot \frac{99}{100} + 20,000 \cdot \frac{1}{100} \\ &= 79,200 + 200 = 79,400. \end{aligned}$$

Problem 13.3. Section 13.1 used randomization for contention resolution; here it is used to build an independent set without central control. There are n processes, and some pairs are in *conflict* (both need the same shared resource). We want to schedule a large set S of processes so that no two conflicting processes are both in S ; such an S is called *conflict-free*. Model this as a graph $G = (V, E)$ with one node per process and an edge for each conflicting pair, so that S is conflict-free exactly when it is an independent set of G . Since finding a maximum independent set is hard in general,

we look for a simple distributed heuristic in which each process talks only to a few others and then decides whether to join S . Assume every node has exactly d neighbours in G .

(a) Consider this protocol: each process P_i independently sets $x_i = 1$ with probability $\frac{1}{2}$ and $x_i = 0$ with probability $\frac{1}{2}$, and joins S if and only if $x_i = 1$ and every process it conflicts with chose 0. Prove that the resulting S is conflict-free, and give a formula for $E[|S|]$ in terms of n and d .

(b) The value $\frac{1}{2}$ was an arbitrary choice. Generalize it: each P_i independently sets $x_i = 1$ with probability p and $x_i = 0$ with probability $1 - p$, $0 \leq p \leq 1$, and joins S under the same rule. In terms of the parameters of G , find the value of p that maximizes $E[|S|]$, and give a formula for $E[|S|]$ at that value.

Solution. (a) S is conflict-free and $E[|S|] = n/2^{d+1}$. Suppose (P_i, P_j) is a conflicting pair with both $P_i, P_j \in S$. Then $x_i = 1$, because P_i joined. But P_j joined too, and P_j joins only if every neighbour chose 0, so $x_i = 0$. This is a contradiction.

For the expectation, use the general protocol of part (b) with $p = \frac{1}{2}$. Let X_i be the indicator of $P_i \in S$. The event $P_i \in S$ says that $d + 1$ distinct processes, namely P_i and its d neighbours, made prescribed independent choices. So

$$\Pr[X_i = 1] = p(1 - p)^d, \quad E[|S|] = \sum_{i=1}^n E[X_i] = np(1 - p)^d$$

by linearity of expectation (13.8). At $p = \frac{1}{2}$ this gives $E[|S|] = n \cdot 2^{-(d+1)}$.

(b) Take $p = 1/(d + 1)$. The derivation of (a) holds for any p , so $E[|S|] = nf(p)$ with $f(p) = p(1 - p)^d$ on $[0, 1]$. If $d = 0$, then $f(p) = p$ is maximized at $p = 1 = 1/(d + 1)$. If $d \geq 1$,

$$f'(p) = (1 - p)^{d-1}((1 - p) - dp) = (1 - p)^{d-1}(1 - (d + 1)p),$$

which is positive on $(0, \frac{1}{d+1})$ and negative on $(\frac{1}{d+1}, 1)$. Hence f is unimodal and attains its maximum at $p = \frac{1}{d+1}$. Then

$$E[|S|] = \frac{n}{d+1} \left(\frac{d}{d+1} \right)^d.$$

Problem 13.4. Consider a simple model of how a peer-to-peer overlay network grows. It starts with a single node v_1 . Nodes then arrive one by one, and when node v_k arrives (with $v_1, \dots, v_{k-1}$ already present) it adds a directed link from v_k to one of $v_1, \dots, v_{k-1}$, chosen uniformly at random and independently of all other choices. Run the process until the nodes are $v_1, \dots, v_n$. Every node except v_1 then has exactly one outgoing edge, while a node can have any number of incoming links, including none. Many incoming links mean a heavy load on a node, and since early nodes tend to collect more links than late ones, the network is unbalanced. The exercise asks you to measure this imbalance.

(a) Find the expected number of incoming links to node v_j in the final network. Give an exact formula in n and j , and also an asymptotic expression in $\Theta(\cdot)$ notation that contains no large summations.

(b) Another sign of imbalance is that many nodes are expected to end up with no incoming links at all. Give a formula for the expected number of nodes with no incoming links.

Solution. (a) $E[\text{in}(v_j)] = H_{n-1} - H_{j-1} = \Theta(\log(n/j))$, where $H_m = \sum_{i=1}^m 1/i$ (with $H_0 = 0$).

Let X_k be the indicator that v_k links to v_j . This is possible only when $k > j$, and then $\Pr[X_k = 1] = 1/(k - 1)$. By linearity of expectation,

$$E[\text{in}(v_j)] = \sum_{k=j+1}^n \frac{1}{k-1} = \sum_{i=j}^{n-1} \frac{1}{i} = H_{n-1} - H_{j-1}.$$

For the asymptotics, $1/i \geq \int_i^{i+1} dx/x$ and $1/i \leq 2/(i+1) \leq 2 \int_i^{i+1} dx/x$, so

$$\ln \frac{n}{j} = \int_j^n \frac{dx}{x} \leq \sum_{i=j}^{n-1} \frac{1}{i} \leq 2 \int_j^n \frac{dx}{x} = 2 \ln \frac{n}{j}.$$

So $E[\ln(v_j)] = \Theta(\log(n/j))$.

(b) For $n \geq 2$ the expected number of nodes with no incoming links is exactly $n/2$ (for $n = 1$ it is 1). Node v_j has no incoming link exactly when none of $v_{j+1}, \dots, v_n$ chooses it. These choices are independent, so

$$\Pr[\ln(v_j) = 0] = \prod_{k=j+1}^n \left(1 - \frac{1}{k-1}\right) = \prod_{m=j}^{n-1} \frac{m-1}{m} = \frac{j-1}{n-1}.$$

The product telescopes, and the formula also covers $j = 1$: there the factor with $m = 1$ is 0, and v_1 is certainly chosen by v_2 . Summing these indicator probabilities by linearity of expectation gives

$$E[\#\{j : \ln(v_j) = 0\}] = \sum_{j=1}^n \frac{j-1}{n-1} = \frac{1}{n-1} \cdot \frac{n(n-1)}{2} = \frac{n}{2}.$$

Problem 13.5. There are n towns $T_1, \dots, T_n$ on one highway leading to an Internet switching hub, with T_i located i miles from the hub (Figure 13.6 shows the hub followed by $T_1, T_2, \dots, T_n$ in a path). A fiber cable from the hub out to T_n costs k dollars per mile, so kn dollars in total, and the towns must split this cost. Three proposals are on the table.

- **Proposal A.** Each town pays the same amount, k .
- **Proposal B.** Town T_i pays an amount proportional to its distance i .
- **Proposal C.** Cut the cable into unit edges $e_1, \dots, e_n$, where e_1 joins the hub to T_1 and e_i ($i > 1$) joins T_{i-1} to T_i . The cost k of e_i is split equally among the towns $T_i, T_{i+1}, \dots, T_n$ lying beyond it.

The **Shapley value** is defined as follows. For an ordering $\mathcal{O}$ in which the towns “arrive”, the marginal cost of T_i is ki if T_i comes first. Otherwise let T_j be the farthest town from the hub among those arriving before T_i ; the cable is assumed to reach exactly to T_j , so the marginal cost of T_i is 0 if $j > i$ and $k(i-j)$ if $j < i$. (For $n = 3$ and order T_1, T_3, T_2 , the marginal costs are $k, 2k, 0$.) Let X_i be the marginal cost of T_i when $\mathcal{O}$ is a uniformly random permutation; the Shapley value assigns T_i the share $E[X_i]$.

Which of Proposals A, B, C, if any, coincides with the Shapley value? Prove your answer.

Solution. Proposal C, and (for $n \geq 2$) only Proposal C, gives the Shapley value.

For $\ell \leq i$, let $Y_{i\ell}$ be the indicator that T_i comes first in $\mathcal{O}$ among $T_\ell, T_{\ell+1}, \dots, T_n$. Write j for the largest index arriving before T_i ($j = 0$ if there is none). If $j > i$, then T_i is never first among $T_\ell, \dots, T_n$ for any $\ell \leq i$, because T_j is in that set and came earlier. If $j < i$, then T_i is first among $T_\ell, \dots, T_n$ exactly when $j < \ell$. In either case

$$X_i = k |\{\ell : j < \ell \leq i\}| = k \sum_{\ell=1}^i Y_{i\ell}.$$

That is, T_i pays for e_ℓ exactly when it is the first town beyond e_ℓ to arrive. In a uniformly random order, each of the $n-\ell+1$ towns $T_\ell, \dots, T_n$ is equally likely to come first among them, so $E[Y_{i\ell}] = 1/(n-\ell+1)$. Linearity of expectation then gives

$$E[X_i] = \sum_{\ell=1}^i \frac{k}{n-\ell+1}.$$

This is exactly Proposal C's charge to T_i : of each edge e_ℓ with $\ell \leq i$, T_i pays the share $k/(n-\ell+1)$, and it pays nothing toward edges e_ℓ with $\ell > i$.

Proposals A and B differ from C already when $n = 2$ (total cost $2k$):

town	A	B	C = Shapley
T_1	k	$2k/3$	$k/2$
T_2	k	$4k/3$	$3k/2$

Problem 13.6. (Genome mapping.) We are given n markers $\mu_1, \dots, \mu_n$ (positions on a chromosome) and must output a linear ordering of them. There are k constraints, each a triple (μ_i, μ_j, μ_k) demanding that μ_j lie *between* μ_i and μ_k in the ordering; the constraint does not say which of μ_i, μ_k comes first. Since all constraints may not be simultaneously satisfiable, we want an ordering satisfying as many as possible; deciding whether some ordering satisfies at least k' of the k constraints is NP-complete (no proof required).

Give a constant $\alpha > 0$, independent of n , and an algorithm such that whenever k^* constraints can be satisfied simultaneously, the algorithm outputs an ordering satisfying at least αk^* constraints. The algorithm may be randomized, in which case the *expected* number of satisfied constraints must be at least αk^* .

Solution. Take $\alpha = \frac{1}{3}$ and output a uniformly random ordering of the n markers.

For a constraint $C = (\mu_i, \mu_j, \mu_k)$ (three distinct markers), the relative order of μ_i, μ_j, μ_k in a uniformly random permutation is uniform over the $3! = 6$ orders, by symmetry. Exactly two of them, $\mu_i \mu_j \mu_k$ and $\mu_k \mu_j \mu_i$, put μ_j in the middle, so with X_C the indicator that C is satisfied,

$$\Pr[X_C = 1] = \frac{2}{6} = \frac{1}{3}.$$

By linearity of expectation, with $X = \sum_C X_C$ the number of satisfied constraints,

$$\mathbf{E}[X] = \sum_C \mathbf{E}[X_C] = \frac{k}{3} \geq \frac{k^*}{3},$$

since $k^* \leq k$ (a triple with a repeated marker is never satisfiable, so discard it first; this leaves k^* unchanged). The shuffle takes $O(n)$ time (Fisher-Yates).

Problem 13.7. Section 13.4 gave a $7/8$ -approximation for MAX 3-SAT, where every clause has terms on three distinct variables. Now consider general MAX SAT: given clauses $C_1, \dots, C_k$ over variables $X = \{x_1, \dots, x_n\}$, find a truth assignment satisfying as many clauses as possible. Every clause has at least one term, and the variables within a single clause are distinct, but clause lengths are otherwise arbitrary (some clauses may be very long, others may be a single term).

(a) Take the MAX 3-SAT algorithm: set each variable independently to true or false with probability $1/2$ each. Show that the expected number of satisfied clauses is at least $k/2$. Give an example of a MAX SAT instance in which no assignment satisfies more than half the clauses.

(b) A clause with a single term (such as $\{x_1\}$ or $\{\bar{x}_2\}$) can be satisfied in only one way. Suppose the instance has no pair of “conflicting clauses”, that is, no variable x_i has both a clause $C = \{x_i\}$ and a clause $C' = \{\bar{x}_i\}$. Modify the randomized procedure so that the expected number of satisfied clauses is at least $.6k$.

(c) Give a randomized polynomial-time algorithm for general MAX SAT whose expected number of satisfied clauses is at least $.6$ times the maximum number that any assignment can satisfy. (By the example in (a), one cannot always satisfy more than $k/2$ clauses; the goal is a $.6$ fraction of the optimum.)

Solution. (a) A clause with $\ell \geq 1$ terms on distinct variables is false only when all ℓ independent terms are false, so with Z the number of satisfied clauses, linearity of expectation gives

$$E[Z] = \sum_{j=1}^k \left(1 - 2^{-|C_j|}\right) \geq \sum_{j=1}^k \frac{1}{2} = \frac{k}{2}.$$

Example: the two clauses $\{x_1\}$ and $\{\bar{x}_1\}$ ($k = 2$). Every assignment makes exactly one of them true, so at most $1 = k/2$ clause can be satisfied.

(b) Independently, set x_i true with probability 0.6 if $\{x_i\}$ is a clause, false with probability 0.6 if $\{\bar{x}_i\}$ is a clause, and true with probability 1/2 otherwise (no conflicting pairs, so at most one of the first two cases applies).

Under this rule every single term is false with probability at most 0.6: a term is false with probability 0.4 if it agrees with a unit clause, 0.6 if it is the negation of a unit clause's term, and 0.5 if its variable has no unit clause. Hence: (i) A single-term clause $\{t\}$ has t biased in its favour, so $\Pr[\text{satisfied}] = 0.6$. (ii) A clause with $\ell \geq 2$ terms is false only if all of its terms are false. These involve distinct, independently set variables, so $\Pr[\text{satisfied}] \geq 1 - 0.6^\ell \geq 1 - 0.36 = 0.64$.

By linearity of expectation, $E[Z] \geq 0.6k$.

(c) First remove the conflicts, then run (b). For each variable x_i , let a_i be the number of copies of the clause $\{x_i\}$ and b_i the number of copies of $\{\bar{x}_i\}$. If $a_i, b_i \geq 1$, delete the $\min(a_i, b_i)$ copies of whichever unit clause is less frequent (break ties arbitrarily). Let $t = \sum_i \min(a_i, b_i)$ be the total number of deleted clauses. The remaining $k - t$ clauses contain no conflicting pair. Run the algorithm of (b) on them and output the resulting assignment for the original instance.

For each variable i , pair the $\min(a_i, b_i)$ deleted clauses with the same number of copies of the opposite unit clause. This gives t disjoint pairs $\{x_i\}, \{\bar{x}_i\}$, and every assignment falsifies exactly one clause of each pair, so

$$\text{OPT} \leq k - t.$$

Deleted clauses that happen to be satisfied only increase the count. Applying (b) to the conflict-free instance with $k - t$ clauses therefore gives

$$E[Z] \geq 0.6(k - t) \geq 0.6 \text{OPT}.$$

Counting and deleting unit clauses is one pass over the input and sampling is $O(n)$, so the algorithm runs in linear time.

13.2 Exercises 13.8–13.14

Problem 13.8. Let $G = (V, E)$ be an undirected graph with n nodes and m edges, and for $X \subseteq V$ let $G[X]$ be the subgraph induced on X : node set X , together with every edge of G whose two ends both lie in X .

Given a natural number $k \leq n$, we want k nodes inducing a “dense” subgraph, in the following precise sense. Design a polynomial-time algorithm that, given $k \leq n$, outputs a set $X \subseteq V$ with $|X| = k$ such that $G[X]$ has at least

$$\frac{mk(k-1)}{n(n-1)}$$

edges. Either (a) a deterministic algorithm or (b) a randomized algorithm whose expected running time is polynomial and which never outputs a wrong answer is acceptable.

Solution. Repeatedly delete a node of minimum degree in the current induced subgraph until k nodes remain; this deterministic peeling never lowers the edge density $|E(G[X])|/\binom{|X|}{2}$.

```

X = V
while |X| > k:
    v = a node of minimum degree in G[X]
    X = X - {v}
return X

```

If $k \leq 1$ (or $n \leq 1$) the required bound is 0 and any k -set works, so assume $k \geq 2$. We prove by induction on the number of deletions that

$$|E(G[X])| \geq \frac{m|X|(|X| - 1)}{n(n - 1)},$$

which holds with equality at the start ($X = V$). Suppose it holds for a set X with $|X| = N > k$, so $N \geq 3$, and let $M = |E(G[X])|$. The degrees in $G[X]$ sum to $2M$, so the minimum degree is at most

$2M/N$, and deleting that node v leaves

$$\begin{aligned} |E(G[X - \{v\}])| &\geq M - \frac{2M}{N} = M \frac{N-2}{N} \\ &\geq \frac{mN(N-1)}{n(n-1)} \cdot \frac{N-2}{N} = \frac{m(N-1)(N-2)}{n(n-1)}, \end{aligned}$$

which is the invariant for the set of size $N-1$ (we used $N-2 > 0$). Once $|X| = k$ the invariant is exactly the required bound.

Running time: there are $n-k$ rounds. Keep each node's degree in the current $G[X]$; a round finds the minimum in $O(n)$ time, and deleting v lowers its neighbours' degrees in $O(\deg v)$ time. The total is $O(n^2 + m)$, which is polynomial.

Problem 13.9. A seller offers one item in a one-pass auction: buyers arrive one after another, and buyer i makes a bid $b_i > 0$, which the seller must accept or refuse on the spot and irrevocably. Accepting sells the item and sends away every later buyer; refusing makes buyer i leave with the bid withdrawn, and only after that does the seller see the next buyer. There are n buyers with pairwise distinct bids, they arrive in a uniformly random order, and the seller knows n . A strategy is a rule that decides whether to accept each bid using only n and the bids seen so far. (For example, always accepting the first bid wins the highest bid with probability only $1/n$.)

Give a strategy under which the seller accepts the highest of the n bids with probability at least $1/4$, for every n (you may assume n is even), and prove the guarantee.

Solution. Refuse the first $n/2$ bids, and let M be the largest of them. Then accept the first later bid that is larger than M . If no such bid arrives, the item is not sold.

The seller wins on the event

$$E = \{\text{the 2nd-highest bid is among the first } n/2, \text{ and the highest is among the last } n/2\}.$$

On E , M is the second-highest bid overall. So the only bid after position $n/2$ that beats M is the highest bid, and the strategy accepts exactly that bid.

Because the order is uniformly random, the second-highest bid lands in the first half with probability $\frac{n/2}{n}$. Given that, the highest bid is equally likely to be in any of the remaining $n-1$ positions, and $n/2$ of those are in the second half. Hence

$$\begin{aligned} \Pr[\text{accept the highest bid}] &\geq \Pr[E] = \frac{n/2}{n} \cdot \frac{n/2}{n-1} \\ &= \frac{n}{4(n-1)} > \frac{1}{4}. \end{aligned}$$

Problem 13.10. A simple online auction has n bidding agents; agent i holds a bid b_i , a positive natural number, and the bids are pairwise distinct. The agents arrive in a uniformly random order and each announces its bid in turn; the system keeps a variable b^* equal to the largest bid seen so far, starting from $b^* = 0$. As a function of the problem's parameters, what is the expected number of times b^* gets updated?

Example: with $b_1 = 20$, $b_2 = 25$, $b_3 = 10$ and arrival order 1, 3, 2, the variable b^* is updated when agents 1 and 2 arrive, but not when agent 3 arrives.

Solution. If X counts the updates, $E[X] = H(n) = \sum_{k=1}^n 1/k$, so $\ln(n+1) < E[X] < 1 + \ln n$ by (13.10); it depends only on n , not on the bid values.

Write the bids in decreasing order. For $k = 1, \dots, n$, let X_k be the indicator that the agent whose bid is the k -th largest causes an update, and let $X = \sum_k X_k$ count the updates. That agent updates b^* exactly when its bid is larger than everything seen before it. Since all bids are positive and b^* starts at 0, this happens exactly when it arrives before each of the $k-1$ agents with larger bids. The arrival

order is uniformly random, so its restriction to these k agents is also uniform, and each of the k is equally likely to come first. By linearity of expectation (13.8),

$$\begin{aligned} E[X] &= \sum_{k=1}^n E[X_k] = \sum_{k=1}^n \Pr[\text{rank-}k \text{ agent precedes the } k-1 \text{ higher ones}] \\ &= \sum_{k=1}^n \frac{1}{k} = H(n). \end{aligned}$$

Problem 13.11. *Load balancing* algorithms for parallel or distributed systems seek to spread out collections of computing jobs over multiple machines. In this way, no one machine becomes a “hot spot.” If some kind of central coordination is possible, then the load can potentially be spread out almost perfectly. But what if the jobs are coming from diverse sources that can’t coordinate? As we saw in Section 13.10, one option is to assign them to machines at random and hope that this randomization will work to prevent imbalances. Clearly, this won’t generally work as well as a perfectly centralized solution, but it can be quite effective. Here we try analyzing some variations and extensions on the simple load balancing heuristic we considered in Section 13.10.

Suppose you have k machines, and k jobs show up for processing. Each job is assigned to one of the k machines independently at random (with each machine equally likely).

(a) Let $N(k)$ be the expected number of machines that do not receive any jobs, so that $N(k)/k$ is the expected fraction of machines with nothing to do. What is the value of the limit $\lim_{k \rightarrow \infty} N(k)/k$? Give a proof of your answer.

(b) Suppose that machines are not able to queue up excess jobs, so if the random assignment of jobs to machines sends more than one job to a machine M , then M will do the first of the jobs it receives and reject the rest. Let $R(k)$ be the expected number of rejected jobs; so $R(k)/k$ is the expected fraction of rejected jobs. What is $\lim_{k \rightarrow \infty} R(k)/k$? Give a proof of your answer.

(c) Now assume that machines have slightly larger buffers; each machine M will do the first two jobs it receives, and reject any additional jobs. Let $R_2(k)$ denote the expected number of rejected jobs under this rule. What is $\lim_{k \rightarrow \infty} R_2(k)/k$? Give a proof of your answer.

Solution. The limits are $\frac{1}{e}$, $\frac{1}{e}$ and $\frac{3}{e} - 1 \approx 0.104$.

Let X_j be the number of jobs sent to machine j ; each job lands on j independently with probability $\frac{1}{k}$, so

$$\begin{aligned} p_0 &= \Pr[X_j = 0] = \left(1 - \frac{1}{k}\right)^k, \\ p_1 &= \Pr[X_j = 1] = k \cdot \frac{1}{k} \left(1 - \frac{1}{k}\right)^{k-1} \\ &= \left(1 - \frac{1}{k}\right)^{k-1}, \end{aligned}$$

and both $p_0 \rightarrow \frac{1}{e}$ and $p_1 \rightarrow \frac{1}{e}$ as $k \rightarrow \infty$ by (13.1).

(a) By Linearity of Expectation (13.8) over the indicators of the events $X_j = 0$, $N(k) = kp_0$, so $N(k)/k = p_0 \rightarrow \frac{1}{e}$.

(b) Machine j rejects $\max(X_j - 1, 0) = X_j - 1 + [X_j = 0]$ jobs. Since $\sum_j X_j = k$,

$$R(k) = \sum_{j=1}^k E[X_j - 1 + [X_j = 0]] = k - k + kp_0 = N(k),$$

so $R(k)/k = p_0 \rightarrow \frac{1}{e}$.

(c) Machine j rejects $\max(X_j - 2, 0) = X_j - 2 + 2[X_j = 0] + [X_j = 1]$ jobs (check the cases $X_j = 0, 1, \geq 2$), so by (13.8)

$$\begin{aligned} \frac{R_2(k)}{k} &= \frac{k - 2k + 2kp_0 + kp_1}{k} = 2p_0 + p_1 - 1 \\ &\rightarrow \frac{2}{e} + \frac{1}{e} - 1 = \frac{3}{e} - 1. \end{aligned}$$

Problem 13.12. Consider the following analogue of Karger's algorithm for finding minimum s - t cuts. We will contract edges iteratively using the following randomized procedure. In a given iteration, let s and t denote the possibly contracted nodes that contain the original nodes s and t , respectively. To make sure that s and t do not get contracted, at each iteration we delete any edges connecting s and t and select a random edge to contract among the remaining edges. Give an example to show that the probability that this method finds a minimum s - t cut can be exponentially small.

Solution. Take nodes $s, t, v_1, \dots, v_n$ with one edge (s, v_i) and two parallel edges (v_i, t) for each i (a multigraph, as in the Contraction Algorithm of Section 13.2); the method then succeeds with probability exactly $(2/3)^n$.

Cuts: if S is the s -side, each $v_i \in S$ contributes its 2 edges to t and each $v_i \notin S$ contributes its 1 edge to s , so

$$c(S) = n + |S \cap \{v_1, \dots, v_n\}|,$$

and the unique minimum s - t cut is $S = \{s\}$, of capacity n .

The process: there are no edges between the v_i , so every contraction merges some uncontracted v_i into the supernode s or into the supernode t ; its remaining edges then join s to t and are deleted. After n contractions only s and t remain, and the cut returned is $\{s\}$ together with the v_i merged into s . Hence the method finds the minimum cut iff every contraction uses an edge to t . When m of the v_i are uncontracted, the remaining edges are m edges to s and $2m$ edges to t , so the chosen edge goes to t with probability $\frac{2m}{3m} = \frac{2}{3}$, independently of the history. Therefore

$$\Pr[\text{min } s\text{-}t \text{ cut found}] = \left(\frac{2}{3}\right)^n,$$

which is exponentially small in the number $n + 2$ of nodes.

Problem 13.13. Consider a balls-and-bins experiment with $2n$ balls but only two bins. As usual, each ball independently selects one of the two bins, both bins equally likely. The expected number of balls in each bin is n . In this problem, we explore the question of how big their difference is likely to be. Let X_1 and X_2 denote the number of balls in the two bins, respectively. (X_1 and X_2 are random variables.) Prove that for any $\varepsilon > 0$ there is a constant $c > 0$ such that the probability $\Pr[X_1 - X_2 > c\sqrt{n}] \leq \varepsilon$.

Solution. Take $c = \sqrt{2/\varepsilon}$.

Write $D = X_1 - X_2 = \sum_{j=1}^{2n} Y_j$, where $Y_j = +1$ if ball j goes to bin 1 and $Y_j = -1$ otherwise. The Y_j are independent with $E[Y_j] = 0$ and $Y_j^2 = 1$, so by Linearity of Expectation (13.8)

$$E[D^2] = \sum_j E[Y_j^2] + \sum_{j \neq l} E[Y_j] E[Y_l] = 2n.$$

Since $D^2 \geq c^2 n \cdot [D^2 > c^2 n]$ pointwise, taking expectations (Markov's inequality) gives

$$\begin{aligned} \Pr[X_1 - X_2 > c\sqrt{n}] &\leq \Pr[D^2 > c^2 n] \\ &\leq \frac{E[D^2]}{c^2 n} = \frac{2}{c^2} = \varepsilon, \end{aligned}$$

for every $n \geq 1$.

Problem 13.14. Some people designing parallel physical simulations come to you with the following problem. They have a set P of k basic processes and want to assign each process to run on one of two machines, M_1 and M_2 . They are then going to run a sequence of n jobs, $J_1, \dots, J_n$. Each job J_i is represented by a set $P_i \subseteq P$ of exactly $2n$ basic processes which must be running (each on its assigned machine) while the job is processed. An assignment of basic processes to machines will be

called *perfectly balanced* if, for each job J_i , exactly n of the basic processes associated with J_i have been assigned to each of the two machines. An assignment of basic processes to machines will be called *nearly balanced* if, for each job J_i , no more than $\frac{4}{3}n$ of the basic processes associated with J_i have been assigned to the same machine.

(a) Show that for arbitrarily large values of n , there exist sequences of jobs $J_1, \dots, J_n$ for which no perfectly balanced assignment exists.

(b) Suppose that $n \geq 200$. Give an algorithm that takes an arbitrary sequence of jobs $J_1, \dots, J_n$ and produces a nearly balanced assignment of basic processes to machines. Your algorithm may be randomized, in which case its expected running time should be polynomial, and it should always produce the correct answer.

Solution. (a) For every odd $n \geq 3$, take disjoint sets A, B, C of n processes each, $P = A \cup B \cup C$, and jobs $P_1 = A \cup B$, $P_2 = B \cup C$, $P_3 = A \cup C$, $P_4 = \dots = P_n = A \cup B$; each has exactly $2n$ processes. If a, b, c are the numbers of processes of A, B, C placed on M_1 , perfect balance of J_1, J_2, J_3 requires

$$\begin{aligned} a + b = b + c = a + c = n &\implies 2(a + b + c) = 3n \\ &\implies a = b = c = \frac{n}{2}, \end{aligned}$$

impossible for odd n .

(b) Assign every process to M_1 or M_2 independently with probability $\frac{1}{2}$ each, check every job, and repeat until the assignment is nearly balanced; this is a Las Vegas algorithm, so its output is always correct.

Fix a job J_i and let X be the number of its $2n$ processes on M_1 , a sum of $2n$ independent 0-1 variables with $E[X] = n$. By the Chernoff bound (13.42) with $\mu = n$ and $\delta = \frac{1}{3}$,

$$\Pr[X > \frac{4}{3}n] < \left[\frac{e^{1/3}}{(4/3)^{4/3}} \right]^n < 0.951^n,$$

and the same bound holds for $2n - X$, the count on M_2 . A union bound over the n jobs and two machines gives

$$\begin{aligned} \Pr[\text{not nearly balanced}] &< 2n \cdot 0.951^n \\ &\leq 400 \cdot 0.951^{200} < 0.02 \quad (n \geq 200), \end{aligned}$$

since $2n \cdot 0.951^n$ is decreasing for $n \geq 20$ (the ratio of consecutive terms is $\frac{n+1}{n} \cdot 0.951 < 1$). So each round succeeds independently with probability $p > 0.98$, and the number of rounds is geometric with mean $1/p < 2$.

Running time: a round costs $O(k)$ to assign and $O(n \cdot 2n) = O(n^2)$ to count the $2n$ processes of each job on M_1 , so the expected running time is $O(k + n^2)$.

13.3 Exercises 13.15–13.18

Problem 13.15. Suppose you are presented with a very large set S of real numbers, and you'd like to approximate the median of these numbers by sampling. You may assume all the numbers in S are distinct. Let $n = |S|$; we will say that a number x is an ε -approximate median of S if at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are less than x , and at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are greater than x . Consider an algorithm that works as follows. You select a subset $S' \subseteq S$ uniformly at random, compute the median of S' , and return this as an approximate median of S . Show that there is an absolute constant c , independent of n , so that if you apply this algorithm with a sample S' of size c , then with probability at least .99, the number returned will be a (.05)-approximate median of S . (You may consider either the version of the algorithm that constructs S' by sampling with replacement, so that an element of S can be selected multiple times, or one without replacement.)

Solution. Sampling with replacement, $c = 3201$ works for every $n \geq 100$; for some small n (every $n \leq 10$, and $n = 12, 14, 16, 18$) no element of S is a (.05)-approximate median, so the returned number never is one, and “very large” is read as $n \geq 100$.

Draw $c = 2m + 1$ elements independently and uniformly from S and return the $(m + 1)$ -st smallest draw x . Let L be the set of $y \in S$ with fewer than $0.45n$ elements of S below y : these are the $\lceil 0.45n \rceil$ smallest elements, so

$$p := \frac{|L|}{n} \leq 0.45 + \frac{1}{n} \leq 0.46.$$

Because L is an initial segment of S , $x \in L$ iff at least $m + 1$ draws lie in L . The number Y of draws in L is a sum of c independent 0-1 variables with $E[Y] = pc \leq 0.46c =: \mu$, so (13.42) with $(1 + \delta)\mu = \frac{c}{2}$, i.e. $\delta = \frac{0.5}{0.46} - 1 = \frac{2}{23}$, gives

$$\begin{aligned} \Pr[x \in L] &= \Pr[Y > \tfrac{c}{2}] < \left[\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right]^{0.46c} \\ &= \exp(0.46c(\delta - (1 + \delta)\ln(1 + \delta))) \\ &< e^{-0.00169c} \leq e^{-5.40} < 0.0046. \end{aligned}$$

Symmetrically, with H the $\lceil 0.45n \rceil$ largest elements (those with fewer than $0.45n$ elements above), x is also the $(m + 1)$ -st largest draw, so $\Pr[x \in H] < 0.0046$. If $x \notin L \cup H$, then at least $0.45n$ elements of S are below x and at least $0.45n$ above, i.e. x is a (.05)-approximate median; hence

$$\Pr[x \text{ is a (.05)-approximate median}] \geq 1 - 2 \cdot 0.0046 > 0.99.$$

Problem 13.16. Consider the following (partially specified) method for transmitting a message securely between a sender and a receiver. The message will be represented as a string of bits. Let $\Sigma = \{0, 1\}$, and let Σ^* denote the set of all strings of 0 or more bits (e.g., $0, 00, 1110001 \in \Sigma^*$). The “empty string,” with no bits, will be denoted $\lambda \in \Sigma^*$.

The sender and receiver share a secret function $f : \Sigma^* \times \Sigma \rightarrow \Sigma$. That is, f takes a word and a bit, and returns a bit. When the receiver gets a sequence of bits $\alpha \in \Sigma^*$, he or she runs the following method to decipher it.

```

Let alpha = alpha_1 alpha_2 ... alpha_n, where n is the number of bits in alpha
The goal is to produce an n-bit deciphered message,
  beta = beta_1 beta_2 ... beta_n
Set beta_1 = f(lambda, alpha_1)
For i = 2, 3, 4, ..., n
  Set beta_i = f(beta_1 beta_2 ... beta_{i-1}, alpha_i)
Endfor
Output beta

```

One could view this as a type of “stream cipher with feedback.” One problem with this approach is that, if any bit α_i gets corrupted in transmission, it will corrupt the computed value of β_j for all $j \geq i$.

We consider the following problem. A sender S wants to transmit the same (plain-text) message β to each of k receivers $R_1, \dots, R_k$. With each one, he shares a different secret function $f^{(i)}$. Thus he sends a different encrypted message $\alpha^{(i)}$ to each receiver, so that $\alpha^{(i)}$ decrypts to β when the above algorithm is run with the function $f^{(i)}$.

Unfortunately, the communication channels are very noisy, so each of the n bits in each of the k transmissions is *independently* corrupted (i.e., flipped to its complement) with probability $1/4$. Thus no single receiver on his or her own is likely to be able to decrypt the message correctly. Show, however, that if k is large enough as a function of n , then the k receivers can jointly reconstruct the plain-text message in the following way. They get together, and without revealing any of the $\alpha^{(i)}$ or the $f^{(i)}$, they interactively run an algorithm that will produce the correct β with probability at least $9/10$. (How large do you need k to be in your algorithm?)

Solution. Decode bit by bit with a majority vote, every receiver feeding back the *jointly agreed* prefix; an odd $k \geq 10.4 \ln(10n)$, i.e. $k = O(\log n)$, suffices.

```

gamma = empty string
For i = 1, 2, ..., n
    each receiver R_j privately computes b_j = f^(j)(gamma, received bit i of alpha^(j))
    each receiver announces only the bit b_j
    gamma_i = majority of b_1, ..., b_k ; append gamma_i to gamma
Output gamma

```

Only the bits b_j are made public, never an $\alpha^{(j)}$ or an $f^{(j)}$.

Let C_i be the number of receivers whose i -th bit was corrupted; C_i is a sum of k independent 0-1 variables with $E[C_i] = k/4$. Suppose $\gamma_1 \cdots \gamma_{i-1} = \beta_1 \cdots \beta_{i-1}$. A receiver whose i -th bit is intact computes $f^{(j)}(\beta_1 \cdots \beta_{i-1}, \alpha_i^{(j)}) = \beta_i$, by the definition of $\alpha^{(j)}$; so at most C_i votes are wrong, and $\gamma_i = \beta_i$ unless $C_i > k/2$ (k odd). By induction on i , if $C_i \leq k/2$ for every i then $\gamma = \beta$. The Chernoff bound (13.42) with $\mu = k/4$, $\delta = 1$ and the Union Bound (13.2) give

$$\Pr[\gamma \neq \beta] \leq \sum_{i=1}^n \Pr[C_i > 2 \cdot \frac{k}{4}] < n \left(\frac{e}{4}\right)^{k/4} \leq \frac{1}{10}$$

as soon as

$$k \geq \frac{4 \ln(10n)}{\ln 4 - 1} \approx 10.36 \ln(10n).$$

Problem 13.17. Consider the following simple model of gambling in the presence of bad odds. At the beginning, your net profit is 0. You play for a sequence of n rounds; and in each round, your net profit increases by 1 with probability $1/3$, and decreases by 1 with probability $2/3$. Show that the expected number of steps in which your net profit is positive can be upper-bounded by an absolute constant, independent of the value of n .

Solution. The expected number is less than 28 for every n .

Let U_t be the number of winning rounds among the first t , a sum of t independent 0-1 variables with $E[U_t] = t/3$; the profit after round t is $U_t - (t - U_t) = 2U_t - t$, which is positive iff $U_t > t/2 = (1 + \frac{1}{2})\frac{t}{3}$. Let Z be the number of rounds $t \in \{1, \dots, n\}$ after which the profit is positive. By Linearity of Expectation (13.8) and the Chernoff bound (13.42) with $\mu = t/3$, $\delta = \frac{1}{2}$,

$$\begin{aligned}
 E[Z] &= \sum_{t=1}^n \Pr[U_t > \frac{3}{2} \cdot \frac{t}{3}] < \sum_{t=1}^n \left[\frac{e^{1/2}}{(3/2)^{3/2}} \right]^{t/3} \\
 &< \sum_{t=1}^{\infty} q^t = \frac{q}{1-q}, \quad q = \left(\frac{e^{1/2}}{(3/2)^{3/2}} \right)^{1/3} \approx 0.9646,
 \end{aligned}$$

since $e^{1/2} \approx 1.649 < 1.837 \approx (3/2)^{3/2}$ gives $q < 1$. Numerically $q/(1-q) \approx 27.2 < 28$, independent of n .

Problem 13.18. In this problem, we will consider the following simple randomized algorithm for the Vertex Cover Algorithm.

```

Start with S = empty set
While S is not a vertex cover,
    Select an edge e not covered by S
    Select one end of e at random (each end equally likely)
    Add the selected node to S
Endwhile

```

We will be interested in the expected cost of a vertex cover selected by this algorithm.

- Is this algorithm a c -approximation algorithm for the Minimum Weight Vertex Cover Problem for some constant c ? Prove your answer.
- Is this algorithm a c -approximation algorithm for the Minimum Cardinality Vertex Cover Problem

for some constant c ? Prove your answer.

(*Hint:* For an edge, let p_e denote the probability that edge e is selected as an uncovered edge in this algorithm. Can you express the expected value of the solution in terms of these probabilities? To bound the value of an optimal solution in terms of the p_e probabilities, try to bound the sum of the probabilities for the edges incident to a given vertex v , namely, $\sum_{e \text{ incident to } v} p_e$.)

Solution. (a) No. Given any c , take the graph with one edge (u, v) , weights $w_u = 1$ and $w_v = W > 2c - 1$. The algorithm returns $\{u\}$ or $\{v\}$ with probability $\frac{1}{2}$ each, so

$$E[w(S)] = \frac{1 + W}{2} > c = c \cdot w(\{u\}) = c \cdot \text{OPT}.$$

(b) Yes, with $c = 2$: $E[|S|] \leq 2|S^*|$ for a minimum vertex cover S^* .

A selected edge is uncovered, so neither end is in S and each iteration adds a new node; thus $|S|$ equals the number of selected edges and, by Linearity of Expectation (13.8), $E[|S|] = \sum_e p_e$.

Fix v and let N_v be the number of selected edges incident to v . Each time such an edge is selected, $v \notin S$ and v is added with probability $\frac{1}{2}$, independently of the past; once $v \in S$, no edge at v is ever selected again. Hence $\Pr[N_v \geq m + 1] \leq \frac{1}{2} \Pr[N_v \geq m]$, so $\Pr[N_v \geq m] \leq 2^{-(m-1)}$ and

$$\sum_{e \ni v} p_e = E[N_v] = \sum_{m \geq 1} \Pr[N_v \geq m] \leq \sum_{m \geq 1} 2^{-(m-1)} = 2.$$

Every edge has an end in S^* , so

$$E[|S|] = \sum_e p_e \leq \sum_{v \in S^*} \sum_{e \ni v} p_e \leq 2|S^*|.$$