

Dynamic Programming

Aayush Bajaj

July 10, 2026

Contents

1	bellman’s principle of optimality	1
2	the two ingredients	2
3	memoisation vs tabulation	2
4	the subproblem dag	3
5	worked problems	4
5.1	0/1 knapsack	4
5.2	longest common subsequence	6
5.3	edit distance	7
6	space optimisation	8
7	dp over time: value and policy iteration	8
8	see also	10

dynamic programming is two things wearing one name. to bellman it was a mathematical theory of *multistage decision processes* — sibling to [linear programming](#) in the “programming means planning” sense.¹ to a computer scientist it is a technique: solve a problem by combining solutions to subproblems, and never solve the same subproblem twice (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). the two are the same idea at different altitudes, and this page covers both. worked implementations also live in this [github repo](#).

bellman chose the name partly to hide the mathematics from a defence secretary who hated the word research; the story is in the footnote

1 bellman’s principle of optimality

bellman’s 1957 book states the principle that makes multistage optimisation tractable:

¹the name is a marketing triumph. bellman, in his autobiography *eye of the hurricane* (world scientific, 1984, p. 159), explains that the secretary of defense at the time, charles wilson, “had a pathological fear and hatred of the word research”, so a name was needed to “shield wilson and the air force from the fact that i was really doing mathematics... i thought, let’s kill two birds with one stone... it’s impossible to use the word dynamic in a pejorative sense... thus, i thought dynamic programming was a good name. it was something not even a congressman could object to.” historians note the timeline is shaky — the term appears in bellman’s papers by 1952, before wilson took office in 1953 (see stuart dreyfus, “richard bellman on the birth of dynamic programming”, *operations research* 50(1), 2002) — but the story is too good not to repeat, and the flavour is certainly authentic. the theory itself is laid out in bellman’s *dynamic programming* (princeton university press, 1957), which also states the principle of optimality quoted in the first section.

an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

in other words: tails of optimal solutions are optimal solutions of tails. whenever it holds, the value of a state depends only on the state — not on how you got there — and the whole problem collapses into a *functional equation*. for a finite-horizon deterministic process with state s , decisions $a \in A(s)$, transition $s' = f_k(s, a)$ and stage reward $r_k(s, a)$:

$$V_k(s) = \max_{a \in A(s)} \{r_k(s, a) + V_{k+1}(f_k(s, a))\}, \quad V_N(s) = r_N(s). \quad (1)$$

solve it backwards from the horizon and $V_0(s_0)$ is the optimal total reward; the maximising a at each state is the optimal policy. every recurrence on this page is an instance of this equation with different decorations — and when the transition becomes random the same equation grows an expectation, which is where this page ends up.

2 the two ingredients

DP applies when a problem has both of (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009):

- **optimal substructure** — an optimal solution is built from optimal solutions to subproblems. this is the principle of optimality in CS clothing. it fails when subproblems interact: the longest *simple* path in a general graph has no such structure (gluing two longest simple paths can revisit vertices), which is one reason it is NP-hard while shortest path is easy.
- **overlapping subproblems** — the naive recursion meets the *same* subproblem many times. the total number of *distinct* subproblems is small (typically polynomial), so caching turns an exponential tree into a polynomial table.

the contrast with divide-and-conquer is exactly the second ingredient. mergesort's two halves are disjoint — no subproblem recurs, so plain recursion is already optimal and a cache would be dead weight. fibonacci's two subcalls overlap almost entirely — the naive recursion does $2F(n+1) - 1$ calls, $\Theta(\varphi^n)$ with $\varphi \approx 1.618$, to fill what is really an $(n+1)$ -entry table.

greedy algorithms are the other neighbour: they also need optimal substructure but commit to a locally best choice without solving subproblems at all

3 memoisation vs tabulation

two mechanical ways to exploit overlap:

- **memoisation (top-down)**: keep the natural recursion, add a cache keyed by subproblem. **pro** — only reachable subproblems get solved; the recursion order finds dependencies for you. **con** — recursion overhead, stack depth, cache-hostile access patterns.
- **tabulation (bottom-up)**: enumerate subproblems in an order where dependencies come first, fill an explicit table. **pro** — no recursion, tight loops, and the table's shape exposes space optimisations. **con** — you must know the dependency order, and you fill every cell whether needed or not.

fibonacci both ways, instrumented:

```
1 calls = 0
2
3 def fib_naive(n):
4     global calls
5     calls += 1
6     return n if n < 2 else fib_naive(n - 1) + fib_naive(n - 2)
7
8 def fib_memo(n, memo=None):
9     global calls
10    if memo is None:
11        memo = {}
12    if n in memo:
13        return memo[n]
14    calls += 1
15    memo[n] = n if n < 2 else fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
16    return memo[n]
17
18 def fib_tab(n):
19     table = [0] * (n + 1)
20     if n >= 1:
21         table[1] = 1
22     for i in range(2, n + 1):
23         table[i] = table[i - 1] + table[i - 2]
24     return table[n]
25
26 def fib_rolling(n):
27     a, b = 0, 1 # F(i-2), F(i-1)
28     for _ in range(n):
29         a, b = b, a + b
30     return a
31
32 for f in (fib_naive, fib_memo):
33     calls = 0
34     print(f"{f.__name__}(30) = {f(30):>6} calls = {calls}")
35 print(f"fib_tab(30) = {fib_tab(30):>6} table cells = 31")
36 print(f"fib_rolling(90) = {fib_rolling(90)} space = 2 words")
```

```
fib_naive(30) = 832040    calls = 2692537
fib_memo(30) = 832040    calls = 31
fib_tab(30)   = 832040    table cells = 31
fib_rolling(90) = 2880067194370816120    space = 2 words
```

$2,692,537 = 2F(31) - 1$ calls collapse to $31 = n + 1$ — the count of distinct subproblems, exactly as predicted. the rolling version previews the space trick below: the recurrence only ever looks back two cells, so keep two cells.

4 the subproblem dag

every DP has a hidden graph: one node per distinct subproblem, one edge from each problem to each subproblem it consults. optimal substructure makes the values well-defined; the absence of cycles makes them computable. the two strategies are then just two graph traversals:

- **memoisation** = depth-first search of this dag from the goal node, caching post-order results.
- **tabulation** = processing nodes in a (reverse) topological order, so every edge points to an already-filled cell.

*call counts, not seconds:
python's big integers make
 $F(90)$ exact, but each addition
on n -bit numbers costs $O(n)$ bit
operations, so the ' $O(n)$
algorithm' is really $O(n^2)$ bit complexity*

running time reads straight off the graph: $\sum_{\text{nodes}}(\text{work per node})$, which for constant-time combination is $O(V + E)$ of the subproblem dag (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). fibonacci: $n + 1$ nodes, $2n - 2$ edges, linear. knapsack: $(n + 1)(W + 1)$ nodes, out-degree ≤ 2 . single-source shortest paths in a dag *is* this picture with nothing hidden — relax vertices in topological order — and conversely every DP can be read as a shortest (or longest) path computation over its subproblem dag.

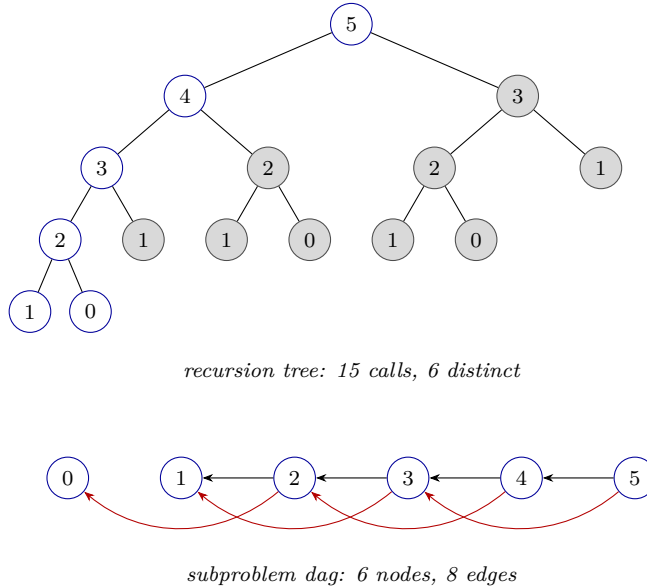


Figure 1: top: the recursion tree of `fib(5)` — 15 calls, and every grey node recomputes work already done. bottom: the subproblem dag those calls collapse onto — 6 nodes, 8 edges; arrows point from a problem to the subproblems it needs, so filling nodes left to right (a reverse topological order) is exactly tabulation.

5 worked problems

5.1 0/1 knapsack

n items with values v_i and weights w_i , capacity W ; take each item at most once, maximise value. as a mathematical program it is the one-constraint 0/1 integer program $\max v^\top x$ subject to $w^\top x \leq W$, $x \in \{0,1\}^n$ — the integer programming page attacks it with branch and bound driven by LP relaxation bounds; DP is the other classic exact method. subproblem: $\text{OPT}(i, w) =$ best value using only the first i items with capacity w . the decision for item i is binary — skip it or take it:

$$\text{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \text{OPT}(i - 1, w) & w_i > w \\ \max\{\text{OPT}(i - 1, w), v_i + \text{OPT}(i - 1, w - w_i)\} & w_i \leq w \end{cases} \quad (2)$$

```
1 from functools import lru_cache
2
3 def knapsack(values, weights, W):
```

```

4     n = len(values)
5     dp = [[0] * (W + 1) for _ in range(n + 1)]
6     for i in range(1, n + 1):
7         for w in range(W + 1):
8             dp[i][w] = dp[i - 1][w]                # skip item i
9             if weights[i - 1] <= w:                # or take it
10                dp[i][w] = max(dp[i][w], dp[i - 1][w - weights[i - 1]] +
11                               ↪ values[i - 1])
12    # traceback: which items were taken?
13    taken, w = [], W
14    for i in range(n, 0, -1):
15        if dp[i][w] != dp[i - 1][w]:
16            taken.append(i - 1)
17            w -= weights[i - 1]
18    return dp[n][W], sorted(taken)
19
20 def knapsack_rolling(values, weights, W):
21     dp = [0] * (W + 1)
22     for v, wt in zip(values, weights):
23         for w in range(W, wt - 1, -1): # reverse! forward would reuse item i
24             dp[w] = max(dp[w], dp[w - wt] + v)
25     return dp[W]
26
27 def knapsack_memo(values, weights, W):
28     @lru_cache(maxsize=None)
29     def best(i, w): # best value using items[0..i) with
30         ↪ capacity w
31         if i == 0:
32             return 0
33         skip = best(i - 1, w)
34         if weights[i - 1] <= w:
35             return max(skip, best(i - 1, w - weights[i - 1]) + values[i - 1])
36         return skip
37     ans = best(len(values), W)
38     return ans, best.cache_info().currsz
39
40 values = [ 60, 100, 120, 80, 30]
41 weights = [ 10, 20, 30, 15, 5]
42 W = 50
43
44 best, items = knapsack(values, weights, W)
45 print(f"optimal value = {best}, items taken = {items}")
46 print(f"weights used = {[weights[i] for i in items]}, total = {sum(weights[i]
47 ↪ for i in items)}")
48 print(f"rolling-array answer = {knapsack_rolling(values, weights, W)} (O(W)
49 ↪ space)")
50 memo_ans, touched = knapsack_memo(values, weights, W)
51 print(f"memoised answer = {memo_ans}, states touched = {touched} of
52 ↪ {(len(values)+1)*(W+1)} in the full table")

```

optimal value = 270, items taken = [0, 1, 3, 4]

weights used = [10, 20, 15, 5], total = 50

rolling-array answer = 270 (O(W) space)

memoised answer = 270, states touched = 36 of 306 in the full table

three points worth staring at:

- **complexity** is $O(nW)$ time and space — *pseudo-polynomial*. it is polynomial in the numeric value of W but exponential in its bit length, so this does not contradict knapsack being NP-hard. scale-and-round the values and the same DP yields an FPTAS.
- **memoisation earned its keep here**: only 36 of 306 table states are reachable

from (n, W) with these weights, and top-down solved exactly those. tabulation filled all 306 (cheap ones, admittedly).

- **the reverse loop** in the rolling version matters: iterating w downwards means $\text{dp}[w - \text{wt}]$ still holds the *previous* item's row. iterate upwards and each item can be taken repeatedly — which is not a bug so much as a different problem: that variant correctly solves *unbounded* knapsack.

5.2 longest common subsequence

given $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$, find the longest sequence appearing in both in order (not necessarily contiguously). subproblem: $c(i, j)$ = LCS length of the prefixes $x_1 \dots x_i$ and $y_1 \dots y_j$ (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009):

$$c(i, j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c(i - 1, j - 1) + 1 & x_i = y_j \\ \max\{c(i, j - 1), c(i - 1, j)\} & x_i \neq y_j \end{cases} \quad (3)$$

optimal substructure argument: if $x_i = y_j$, some LCS ends by matching them (exchange argument), and what precedes is an LCS of both prefixes shortened by one. if not, any LCS omits x_i or omits y_j , so it is an LCS of one of the two smaller problems. $O(mn)$ time and space; the answer string is recovered by walking the table backwards — no arrows need storing, since each cell's provenance can be re-derived by comparing it with its neighbours.

```

1 def lcs(X, Y):
2     m, n = len(X), len(Y)
3     dp = [[0] * (n + 1) for _ in range(m + 1)]
4     for i in range(1, m + 1):
5         for j in range(1, n + 1):
6             if X[i - 1] == Y[j - 1]:
7                 dp[i][j] = dp[i - 1][j - 1] + 1
8             else:
9                 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
10    # traceback
11    out, i, j = [], m, n
12    while i and j:
13        if X[i - 1] == Y[j - 1]:
14            out.append(X[i - 1]); i -= 1; j -= 1
15        elif dp[i - 1][j] >= dp[i][j - 1]:
16            i -= 1
17        else:
18            j -= 1
19    return dp, "".join(reversed(out))
20
21 X, Y = "ABCBDBAB", "BDCABA"
22 dp, s = lcs(X, Y)
23 print(f"lcs({X!r}, {Y!r}) = {s!r}, length {len(s)}")
24 print("      " + " ".join(" " + c for c in " " + Y))
25 for i, row in enumerate(dp):
26     label = (" " + X)[i]
27     print(f" {label} " + " ".join(f"{v:2d}" for v in row))

```

`lcs('ABCBDBAB', 'BDCABA') = 'BCBA', length 4`

	B	D	C	A	B	A
0	0	0	0	0	0	0
A	0	0	0	0	1	1

B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

the same table, drawn with its traceback:

$x_i \backslash y_j$		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

Figure 2: the LCS table for $X = \text{ABCB DAB}$, $Y = \text{BDCABA}$. arrows trace the recovery walk from $c(7, 6)$ back to the empty prefixes: diagonal steps land on matched characters (coloured cells, read bottom-up: B, C, B, A) and each up/left step drops an unmatched character.

5.3 edit distance

the levenshtein distance between $x_1 \dots x_m$ and $y_1 \dots y_n$: the minimum number of single-character insertions, deletions and substitutions turning one into the other. subproblem $d(i, j)$ = distance between the two prefixes; the last operation on the alignment's final column is one of three choices:

$$d(i, j) = \begin{cases} \max\{i, j\} & \min\{i, j\} = 0 \\ \min\{d(i-1, j) + 1, d(i, j-1) + 1, d(i-1, j-1) + [x_i \neq y_j]\} & \text{otherwise} \end{cases} \quad (4)$$

where $[\cdot]$ is the iverson bracket (a substitution is free when the characters already agree). structurally this is LCS with a min instead of a max and a third diagonal option; both are special cases of sequence alignment with general gap and mismatch costs, which is how bioinformatics consumes this recurrence by the terabyte.

```

1 def edit_distance(a, b):
2     m, n = len(a), len(b)
3     dp = [[0] * (n + 1) for _ in range(m + 1)]
4     for i in range(m + 1):
5         dp[i][0] = i # delete everything
6     for j in range(n + 1):
7         dp[0][j] = j # insert everything
8     for i in range(1, m + 1):
9         for j in range(1, n + 1):

```

```

10     sub = 0 if a[i - 1] == b[j - 1] else 1
11     dp[i][j] = min(dp[i - 1][j] + 1,          # delete a[i-1]
12                   dp[i][j - 1] + 1,          # insert b[j-1]
13                   dp[i - 1][j - 1] + sub)     # substitute (or match)
14     return dp[m][n]
15
16 pairs = [("kitten", "sitting"), ("sunday", "saturday"), ("dynamic",
17 ↪ "dynastic")]
18 for a, b in pairs:
19     print(f"edit({a!r}, {b!r}) = {edit_distance(a, b)}")

```

```

edit('kitten', 'sitting') = 3
edit('sunday', 'saturday') = 3
edit('dynamic', 'dynastic') = 2

```

$O(mn)$ time, and — remarkably — that is probably where it ends: any algorithm running in $O(n^{2-\epsilon})$ time for some $\epsilon > 0$ would refute the strong exponential time hypothesis.²

6 space optimisation

the subproblem dag tells you what you may forget: once every consumer of a cell has been filled, the cell is garbage. concretely:

- **look-back of depth 1:** fibonacci keeps two scalars; LCS and edit distance keep two rows (or one row plus one scalar with careful in-place updates), dropping $O(mn)$ space to $O(\min\{m, n\})$.
- **knapsack in one row:** shown above — one $O(W)$ array traversed in decreasing w , because the recurrence reads strictly left of the write position in the *previous* row.
- **the catch:** rolling arrays keep the optimal *value* but destroy the table you would traceback through. hirschberg’s trick recovers the solution anyway — divide and conquer on the midpoint of one string, computing forward and backward rows to find where the optimal path crosses the middle — same $O(mn)$ time, linear space.³

7 dp over time: value and policy iteration

bellman’s own setting was sequential decision-making, where the modern habitat is the markov decision process: states s , actions a , transition probabilities $P(s' | s, a)$, rewards, discount $\gamma \in [0, 1)$. the principle of optimality becomes the **bellman optimality equation**:

$$V^*(s) = \max_a \left\{ r(s, a) + \gamma \sum_{s'} P(s' | s, a) V^*(s') \right\}. \quad (5)$$

the right-hand side, viewed as an operator T on value functions, is a γ -contraction in the sup norm, so V^* exists, is unique, and **value iteration** $V_{k+1} = TV_k$ converges

²backurs and indyk, “edit distance cannot be computed in strongly subquadratic time (unless SETH is false)”, STOC 2015. one of the flagship results of fine-grained complexity: quadratic is not laziness, it is (conditionally) a law.

³hirschberg, “a linear space algorithm for computing maximal common subsequences”, *communications of the ACM* 18(6), 1975.

geometrically: $\|V_{k+1} - V^*\|_\infty \leq \gamma \|V_k - V^*\|_\infty$. **policy iteration** alternates exact policy evaluation (a linear solve) with greedy improvement, and terminates in finitely many steps because each iteration strictly improves over a finite policy set. this is DP with the table indexed by state rather than prefix — and the tabulation order replaced by iterate-to-fixpoint, since the “dag” now has cycles that the contraction tames.

```

1 import numpy as np
2
3 # a 4-state chain: states 0,1,2,3. state 3 is the goal (absorbing, reward 10
  ↳ on entry).
4 # actions: 0 = left, 1 = right. moves succeed with prob 0.8, stay put with
  ↳ prob 0.2.
5 # every non-terminal step costs -1. discount gamma = 0.9.
6 nS, nA, gamma = 4, 2, 0.9
7 P = np.zeros((nS, nA, nS)) # P[s, a, s']
8 R = np.zeros((nS, nA))
9 for s in range(3):
10     for a, d in ((0, -1), (1, +1)):
11         t = min(max(s + d, 0), 3)
12         P[s, a, t] += 0.8
13         P[s, a, s] += 0.2
14         R[s, a] = -1 + 0.8 * (10 if t == 3 else 0)
15 P[3, :, 3] = 1.0 # absorbing, zero reward forever
16
17 # value iteration: V ← max_a [ R(s,a) + gamma * sum_s' P(s,a,s') V(s') ]
18 V = np.zeros(nS)
19 for k in range(1, 200):
20     Q = R + gamma * P @ V # Q[s,a]
21     V_new = Q.max(axis=1)
22     if np.abs(V_new - V).max() < 1e-10:
23         break
24     V = V_new
25 policy = Q.argmax(axis=1)
26 print(f"converged after {k} sweeps")
27 print("V* =", np.round(V, 4))
28 print("pi* =", ["left" if a == 0 else "right" for a in policy[:3]], "(state 3
  ↳ terminal)")
29
30 # policy iteration: solve the linear system for each fixed policy, then
  ↳ greedify
31 pi = np.zeros(nS, dtype=int) # start: always left
32 for it in range(1, 50):
33     Ppi = P[np.arange(nS), pi] # nS x nS
34     Rpi = R[np.arange(nS), pi]
35     Vpi = np.linalg.solve(np.eye(nS) - gamma * Ppi, Rpi) # exact policy
  ↳ evaluation
36     pi_new = (R + gamma * P @ Vpi).argmax(axis=1)
37     if (pi_new == pi).all():
38         break
39     pi = pi_new
40 print(f"policy iteration: {it} iterations, V = {np.round(Vpi, 4)}")

```

converged after 21 sweeps

V* = [4.2911 6.276 8.5366 0.]

pi* = ['right', 'right', 'right'] (state 3 terminal)

policy iteration: 4 iterations, V = [4.2911 6.276 8.5366 0.]

both routes agree, and policy iteration gets there in 4 iterations against value iteration's 21 sweeps — typical, since each of its steps does much more work. everything above assumed the model P is known; strike that assumption and you must *sample* the bellman backup from experience instead of computing its expectation, which is

precisely **q-learning** — stochastic-approximation DP.

the same expectation-inside-a-max structure is what **stochastic programming** calls a *multistage problem with recourse*: decide now, watch randomness resolve, recover optimally, repeat. stochastic DP solves it by backward induction on states; stochastic programming unrolls the randomness into a scenario tree and hands one enormous deterministic program to a solver. same principle of optimality, opposite trade-off — DP scales in horizon but dies in state dimension (bellman also coined “the curse of dimensionality” for exactly this), while scenario methods tolerate rich state but explode with stages.

8 see also

- **linear programming** — the other “programming” that means planning; simplex vs bellman is 1947 vs 1953
- **integer programming** — knapsack by branch and bound; where DP’s pseudo-polynomial trick meets NP-hardness
- **stochastic programming** — recourse problems and scenario trees, the optimisation view of stochastic DP
- **classical algorithms** — bellman-ford, floyd-warshall and friends: shortest-path DP in the wild
- **q-learning** — the bellman optimality backup, sampled instead of computed

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.