Dynamic Programming

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Reuse a solved subproblem

Dynamic programming (DP) is a way to organise a computation whose subproblems recur. Instead of solving the same state again along every recursive path, we solve it once and retain its result. The result may be an optimum, a count, a Boolean answer, or some other summary; DP is not restricted to optimisation.

Two ideas have different roles:

- **overlapping subproblems** make storage useful because the same states would otherwise be recomputed; and
- **optimal substructure** makes an optimisation recurrence valid because an optimal solution contains optimal solutions to the subproblems selected by its first or last decision.

A table is only a representation. The algorithmic work is to define states that contain exactly the information required for future decisions.

2 Learning goals

After studying this note, you should be able to:

- define a DP state, transition, base cases, dependency order, and answer state;
- justify a recurrence by considering the first or last decision;
- implement top-down memoisation and bottom-up tabulation;
- reconstruct a witness rather than only its value;
- derive weighted edit distance, adjacent-transposition distance, and substring search;
- derive global and local biological sequence alignment with affine gap penalties;
- trace dynamic time warping (DTW) and explain window constraints; and
- derive and analyse matrix-chain multiplication.

3 Prerequisites

You should be comfortable with arrays, recursion, asymptotic notation, and directed acyclic graphs. We use one-based string notation such as $X[1..i]$ and reserve row or column zero for the empty prefix. No earlier string-matching material is assumed.

4 A design recipe

For each new problem, write down the following items before writing code.

1. **State.** Give one precise sentence defining every index and stored value.
2. **Transition.** Express a state using strictly smaller or earlier states.
3. **Boundaries.** Define empty, smallest, and impossible states.
4. **Order.** Ensure every dependency is available before it is used.
5. **Answer.** Identify the state or aggregate that answers the original question.
6. **Witness.** Store a predecessor or decision if an actual solution is required.
7. **Cost.** Count reachable states and the work and storage per state.

For example, Fibonacci numbers satisfy $F_i = F_{i-1} + F_{i-2}$ with $F_0 = 0$ and $F_1 = 1$. Direct recursion repeats the same calls. A bottom-up computation keeps only two previous values:

```
fib(n):
    if n = 0: return 0
    previous = 0
    current  = 1
    for i = 2 .. n:
        previous, current = current, previous + current
    return current
```

This uses $O(n)$ arithmetic operations and $O(1)$ stored values. Top-down memoisation instead keeps a map from an argument to its computed result. It mirrors a recursive specification and can avoid unreachable states; tabulation avoids recursive call overhead and often exposes a better memory order.

5 Why a recurrence is correct

A standard proof follows the dependency order. Prove the base states directly. Then assume every predecessor state already has its stated meaning. Show that every candidate in the transition constructs a feasible solution for the current state, and that every feasible solution must make one of the enumerated final decisions. Taking the best candidate therefore neither invents an impossible solution nor misses the optimum.

This is induction on an index, interval length, number of selected objects, or any other well-founded state order. Merely displaying a table is not a correctness argument.

6 Edit distance from the final operation

Let $D[i, j]$ be the minimum cost of transforming prefix $X[1..i]$ into prefix $Y[1..j]$. Under unit-cost insertion, deletion, and substitution, the last operation is exactly one of the following:

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 & \text{delete } X_i, \\ D[i,j-1]+1 & \text{insert } Y_j, \\ D[i-1,j-1]+[X_i \neq Y_j] & \text{match or substitute.} \end{cases}$$

The boundaries are $D[i, 0] = i$ and $D[0, j] = j$. Fill rows left-to-right and top-to-bottom, or use any order in which the upper, left, and diagonal predecessor have already been computed.

There are $(|X|+1)(|Y|+1)$ states and constant work per state, so the running time is $\Theta(|X||Y|)$ and the full table uses the same asymptotic space. If only the value is needed, one previous row and the current row use $O(\min\{|X|, |Y|\})$ space after orienting the shorter string along a row.

6.1 Weighted and symbol-dependent edits

Applications need not assign every operation cost 1. Let $c_{del}(x)$, $c_{ins}(y)$, and $c_{sub}(x, y)$ be nonnegative costs, with $c_{sub}(x, x) = 0$. Replace the three added constants in the recurrence by these functions. A case-only change may cost 0.1, for example, while an implausible letter change costs 1.

The chosen costs are part of the problem definition. The resulting function is not automatically symmetric or a metric. Symmetry, identity, and the triangle inequality require suitable relationships among insertion, deletion, and substitution costs.

An adjacent transposition can be added when $i, j \geq 2$, $X_i = Y_{j-1}$, and $X_{i-1} = Y_j$:

$$D[i,j] \leftarrow \min(D[i,j], D[i-2,j-2] + c_{tr}(X_{i-1}, X_i)).$$

This simple recurrence computes the **optimal string alignment** variant, in which a substring is not edited repeatedly. The unrestricted Damerau–Levenshtein distance needs additional last-occurrence information; the two models should not be named interchangeably.

7 Worked edit-distance trace

For `cat` and `cut`, rows are prefixes of `cat` and columns are prefixes of `cut`:

D	ε	c	u	t
ε	0	1	2	3
c	1	0	1	2
a	2	1	1	2
t	3	2	2	1

The bottom-right value is 1. A traceback chooses the diagonal predecessors: match `c`, substitute `a` by `u`, and match `t`. The recurrence also considered insertions and deletions; the trace is evidence of one optimum, not a greedy decision made at the first mismatch.

Store a predecessor direction in every cell to recover an edit script. If several predecessors tie, there may be several optimal scripts. Keeping all tied predecessors represents a DAG of optimal solutions, which can be exponentially numerous even though the value table is small.

8 Global and local biological sequence alignment

Edit distance minimises a nonnegative cost. Biological sequence comparison is usually written as **score maximisation**: a substitution score $s(a, b)$ rewards a match or a biologically plausible replacement, while gaps subtract penalties. DNA may use a simple match/mismatch score; protein alignment commonly uses an empirically derived substitution matrix.

Choose the alignment scope before choosing scores. A **global** alignment explains both complete sequences, a **local** alignment selects the best pair of substrings, and a **semi-global** alignment makes specified end gaps free. The recurrence, boundary values, traceback start, and traceback stop must all describe the same scope. A substitution matrix is simply a table giving $s(a, b)$ for every symbol pair in the chosen alphabet; it does not determine the gap model.

Let $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$. Let $g_o \geq 0$ be the gap-opening penalty and $g_e \geq 0$ the extension penalty. This note charges a gap of length $\ell \geq 1$ by

$$g_o + (\ell - 1)g_e.$$

Thus the first missing symbol costs g_o and each later symbol in the same gap costs g_e. Some software instead charges $g_o + \ell g_e$; scores are comparable only after the convention is stated.

A linear gap cost needs no memory of how the previous column was formed. An affine cost does: opening a gap and extending an existing run must use different predecessor states.

8.1 Three affine-gap states

For prefixes $A[1..i]$ and $B[1..j]$, define:

- $M[i, j]$: best score ending by aligning a_i with b_j;
- $G_B[i, j]$: best score ending with a_i aligned to a gap in B; and
- $G_A[i, j]$: best score ending with a gap in A aligned to b_j.

For $i, j > 0$ use

$$M[i, j] = s(a_i, b_j) + \max\{M[i-1, j-1], G_B[i-1, j-1], G_A[i-1, j-1]\},$$

$$G_B[i, j] = \max \begin{cases} M[i-1, j] - g_o, \\ G_A[i-1, j] - g_o, \\ G_B[i-1, j] - g_e, \end{cases} \qquad G_A[i, j] = \max \begin{cases} M[i, j-1] - g_o, \\ G_B[i, j-1] - g_o, \\ G_A[i, j-1] - g_e. \end{cases}$$

The first two candidates in each gap state open a new gap; the last extends the same gap. These recurrences permit adjacent gap runs in opposite sequences. A model that forbids such a switch omits the cross-gap opening candidate; that convention can change an optimum when mismatch scores are unusually severe.

8.2 Global alignment: Needleman–Wunsch with affine gaps

A global alignment consumes both complete sequences. Initialise impossible states to $-\infty$ and set

$$M[0,0] = 0, \qquad G_A[0,0] = G_B[0,0] = -\infty.$$

For $i > 0$,

$$G_B[i,0] = -g_o - (i-1)g_e, \qquad M[i,0] = G_A[i,0] = -\infty,$$

and for $j > 0$,

$$G_A[0,j] = -g_o - (j-1)g_e, \qquad M[0,j] = G_B[0,j] = -\infty.$$

The optimum global score is

$$\max\{M[n,m], G_B[n,m], G_A[n,m]\}.$$

Leading and trailing gaps are charged because no sequence end may be discarded. With a linear gap penalty this is the familiar Needleman–Wunsch grid; the three-state affine version is commonly called Gotoh's refinement.

8.3 Local alignment: Smith–Waterman with affine gaps

A local alignment chooses the best-scoring pair of substrings. Set $M[0,j] = M[i,0] = 0$ and keep the two gap-state borders at $-\infty$. Replace the paired-state recurrence by

$$M[i,j] = \max\left(0, s(a_i, b_j) + \max\{M[i-1,j-1], G_B[i-1,j-1], G_A[i-1,j-1]\}\right).$$

The gap recurrences are unchanged. The zero candidate starts a new local alignment, and the answer is the maximum of 0 and all three states over all cells. Traceback begins at that best cell and stops at the restart marker. If no nonempty alignment has positive score, the conventional answer is the empty alignment with score zero.

This is why local alignment is naturally expressed with scores rather than nonnegative edit costs: if empty substrings were allowed while minimising cost, zero would always be an uninformative optimum.

8.4 Pseudocode and traceback

```
affine_alignment(A, B, score, gap_open, gap_extend, local):
    fill M, GB, GA with -infinity
    create matching predecessor tables PM, PGB, PGA

    if local:
        M[0,j] = 0, PM[0,j] = STOP for every j
        M[i,0] = 0, PM[i,0] = STOP for every i
    else:
        M[0,0] = 0
        for i = 1 .. n:
            GB[i,0] = -gap_open-(i-1)*gap_extend
            PGB[i,0] = M if i = 1 else GB
        for j = 1 .. m:
            GA[0,j] = -gap_open-(j-1)*gap_extend
            PGA[0,j] = M if j = 1 else GA
```

With the boundaries in place, fill all interior cells and remember which state attained each maximum:

```
    best = (0, STOP, 0, 0)
    for i = 1 .. n:
        for j = 1 .. m:
            GB[i,j], PGB[i,j] = maximum of
                (M[i-1,j]-gap_open, M),
                (GA[i-1,j]-gap_open, GA),
                (GB[i-1,j]-gap_extend, GB)

            GA[i,j], PGA[i,j] = maximum of
                (M[i,j-1]-gap_open, M),
                (GB[i,j-1]-gap_open, GB),
                (GA[i,j-1]-gap_extend, GA)

            previous, PM[i,j] = maximum of
                (M[i-1,j-1], M),
                (GB[i-1,j-1], GB),
                (GA[i-1,j-1], GA)
            M[i,j] = previous + score(A[i], B[j])
            if local and M[i,j] < 0: M[i,j], PM[i,j] = 0, STOP

            if local: update best from M[i,j], GB[i,j], and GA[i,j]

    endpoint = best if local else maximum of M[n,m], GB[n,m], GA[n,m]
    traceback from endpoint, following the stored state:
        M  outputs A[i] with B[j] and moves diagonally
        GB outputs A[i] with '-'  and moves upward
        GA outputs '-'  with B[j] and moves leftward
        stop at (0,0) globally or at STOP locally
    reverse and return the two output rows and their score
```

One predecessor per state reconstructs one optimum. Keeping every tied predecessor represents all optimum alignments, whose number can be exponential.

8.5 Worked comparison

Use match score $+2$, mismatch score -1, $g_o = 2$, and $g_e = 1$ for

```
A = ACGTT
B = ACT
```

One optimum global alignment is

```
ACGTT
AC--T
```

It has three matches and one gap of length two, so its score is

$$3 \cdot 2 - (2 + (2 - 1) \cdot 1) = 3.$$

A local alignment may discard poorly scoring ends. Aligning `AC` with `AC` scores 4, which exceeds the best global score because the local problem need not explain every symbol.

For correctness, inspect the last alignment column. A paired column comes from one of the three diagonal states. A gap column either opens a new gap or extends the corresponding gap state. The recurrences therefore construct only valid alignments and enumerate every possible last case. Induction on $i + j$ proves every state optimal; the local restart and maximum over all cells additionally enumerate every substring start and end.

There are three states per prefix pair and constant work per transition, so both variants take $\Theta(nm)$ time. Full values and traceback pointers use $\Theta(nm)$ space. Scores alone need only $O(\min\{n, m\})$ working space, but ordinary traceback requires retained pointers or a more advanced reconstruction method.

9 Searching inside a longer text

Pairwise distance forces both complete strings to participate. To match pattern P against any substring ending in text prefix $T[1..j]$, keep the same recurrence but initialise

$$D[0, j] = 0 \quad \text{for every } j, \qquad D[i, 0] = i.$$

The free top row allows the alignment to start after any text prefix. A value $D[m, j] \leq k$ reports a match ending at j. Traceback until row zero recovers a corresponding start. Reporting every tied start may require following several predecessor paths and must include output size in the complexity. This initialisation will be used again in the approximate-matching note.

10 Dynamic time warping

DTW aligns numeric sequences that may progress at different speeds. Let local discrepancy be $d(x_i, y_j) \geq 0$, for example $|x_i - y_j|$ or squared Euclidean distance. For global alignment define

$$W[0, 0] = 0, \qquad W[i, 0] = W[0, j] = \infty \text{ for } i, j > 0,$$

and

$$W[i, j] = d(x_i, y_j) + \min\{W[i - 1, j], W[i, j - 1], W[i - 1, j - 1]\}.$$

The three moves produce a monotone path: each observation is used in order, while a horizontal or vertical move repeats an alignment partner. This is why DTW can align a stretched peak that Euclidean distance at equal timestamps treats as different.

For $X = (1, 2, 3)$ and $Y = (1, 1, 2, 3)$ with absolute difference, an optimal path pairs the first `1` in X with both initial `1`s in Y, then pairs `2` and `3` directly. Its total cost is 0. A traceback through the minimum predecessors reveals this warping path.

The unrestricted table takes $\Theta(|X||Y|)$ time. A **Sakoe–Chiba window** permits only cells satisfying $|i - j| \leq w$ after the sequences have been put on a compatible time scale. It reduces work to $O(w \max\{|X|, |Y|\})$ for comparable lengths, but it changes the feasible set. For global DTW one must have at least $w \geq \big||X| - |Y|\big|$; an arbitrary narrow window can exclude the correct alignment.

Raw DTW tends to grow with sequence length and is not generally a metric. Applications should specify local cost, permitted steps, window, endpoint rules, and any normalisation by path length. Subsequence DTW uses free or selected boundary conditions when a short query is sought inside a long signal; that is a different problem from global alignment.

11 Matrix-chain multiplication

The product $A_1 A_2 \cdots A_n$ has fixed matrix order, but associativity permits different parenthesisations with very different costs. If A_i has dimensions $p_{i-1} \times p_i$, define $M[i, j]$ as the minimum number of scalar multiplications needed for $A_i \cdots A_j$.

The base case is $M[i, i] = 0$. If the final multiplication splits after A_k, its cost is

$$M[i, k] + M[k + 1, j] + p_{i-1} p_k p_j.$$

Therefore

$$M[i, j] = \min_{i \le k < j} \{M[i, k] + M[k + 1, j] + p_{i-1} p_k p_j\}.$$

Fill intervals by increasing length and store the best split in $S[i, j]$:

```
for i = 1 .. n: M[i,i] = 0
for length = 2 .. n:
    for i = 1 .. n-length+1:
        j = i+length-1
        M[i,j] = infinity
        for k = i .. j-1:
            candidate = M[i,k] + M[k+1,j] + p[i-1]*p[k]*p[j]
            if candidate < M[i,j]:
                M[i,j], S[i,j] = candidate, k
```

For dimensions 10×30, 30×5, and 5×60, computing $(A_1 A_2) A_3$ costs $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 4500$. Computing $A_1 (A_2 A_3)$ costs $30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 27000$.

There are $O(n^2)$ intervals and $O(n)$ possible splits per interval, giving $O(n^3)$ time and $O(n^2)$ space. Correctness follows because the root multiplication of every full parenthesisation has one split k; if either side were not optimally parenthesised, replacing it would improve the whole solution.

12 Time, space, and reconstruction trade-offs

Rolling arrays are valid only when transitions use a bounded number of recent layers. They may destroy information needed to reconstruct a witness. Alternatives include retaining predecessor decisions, recomputing selected regions, or using divide-and-conquer reconstruction. State explicitly whether a bound is for the optimum value, one witness, or all witnesses.

Memoisation and tabulation both require a well-founded dependency relation in the standard DP setting. If a formulation has genuine cycles, a simple recursive table is not enough; it may instead be a shortest-path, fixed-point, or iterative-relaxation problem.

13 Connections

- Edit distance is a shortest path in the acyclic grid graph whose edges are edit operations.
- Needleman–Wunsch and Smith–Waterman use the same prefix grid with different boundary, restart, and endpoint rules; affine gaps add state memory.
- DTW uses the same grid geometry but different boundary conditions and step meaning.
- Matrix-chain multiplication is an interval DP; parsing and polygon triangulation use closely related split recurrences.

- The approximate-matching note turns pairwise edit distance into text search and then avoids evaluating every matrix cell.

14 Common mistakes

- Defining a state too vaguely to prove its transition.
- Treating overlapping subproblems as a requirement for correctness rather than for efficiency.
- Forgetting boundary or impossible states.
- Filling a state before its dependencies are available.
- Returning only an optimum value when a witness is requested.
- Calling the simple adjacent-transposition recurrence unrestricted Damerau–Levenshtein distance.
- Mixing edit-cost minimisation with alignment-score maximisation.
- Charging $g_o + \ell g_e$ after defining the gap cost as $g_o + (\ell - 1)g_e$.
- Using one DP state for an affine gap cost, or zero-initialising global borders.
- Returning the bottom-right cell for local alignment instead of the best restart-delimited region.
- Using a narrow DTW window without checking that the intended path remains feasible.

15 Self-check

1. What are the state, transition, boundaries, answer, and evaluation order for edit distance?
2. Why does changing the entire top row to zero turn pairwise distance into substring search?
3. Which assumptions make weighted edit distance a metric?
4. Why do affine gaps require separate paired and gap states?
5. Which initialisation and endpoint rules distinguish global from local alignment?
6. Why can DTW distance be zero for sequences of different lengths?
7. Derive both parenthesisation costs for matrices of dimensions 5×10, 10×3, and 3×12.
8. Give the induction parameter for a correctness proof of matrix-chain DP.

16 Revision summary

- DP stores results for precisely defined states and evaluates a well-founded dependency graph.
- Correctness comes from exhaustive final decisions and induction, not from the existence of a table.
- Edit distance, approximate substring alignment, and DTW share a grid but differ in boundaries and move semantics.
- Weighted edits and transpositions must be specified as an explicit model.
- Global alignment consumes complete sequences; local alignment can restart and end anywhere; affine gaps distinguish opening from extension.
- Matrix-chain multiplication illustrates interval states, split decisions, and witness reconstruction.
- Complexity is the number of reachable states times work per state, with reconstruction space counted separately.

17 Implementations and hands-on exploration

- **Python memoisation:** `functools.cache` turns a pure recursive recurrence into an unbounded memoised function. Arguments must be hashable, and the cache retains arguments and results until cleared; use `cache_info()` to measure hits and misses.
- **Python biological alignment:** Biopython `PairwiseAligner` exposes global and local alignment, substitution matrices, affine gaps, and separate end-gap scores. Its documented formula `open + (length-1)*extend` matches this note's convention when penalties are supplied as negative scores. Set every relevant score explicitly: its defaults do not represent a generally accepted biological model.
- **C biological alignment:** Parasail provides SIMD Needleman–Wunsch, Smith–Waterman, and semi-global alignment with affine gaps and optional traceback. It likewise applies only the opening penalty to the first gap symbol. Score-only, statistics, and traceback variants do different work, and fixed-width SIMD variants require attention to saturation.

- **C++ biological alignment:** SeqAn3 pairwise alignment supports global, local, semi-global, banded, and affine-gap configurations. Its current `gap_cost_affine` scores a gap as `open_score + length*extension_score`, not with this note's parameter names. To represent penalty $g_o+(\ell-1)g_e$, use `extension_score = -g_e` and `open_score = -g_o+g_e`, then configure free ends separately.
- **Python matrix chains:** `numpy.linalg.multi_dot` selects an efficient parenthesisation and performs the product. It returns the numerical result, not the DP cost table or split tree, so instrument your own recurrence when studying reconstruction.

17.1 Small experiments

1. Implement memoised and bottom-up unit-cost edit distance. For every pair of strings over `{a,b}` of length at most 4, assert equal answers and compare memoisation misses with the number of table cells actually filled.
2. Reproduce the `ACGTT` versus `ACT` example with Biopython or Parasail using the note's gap convention. Then vary only the extension penalty and record when the optimal alignment or score changes.

18 Sources and further study

- Dynamic Programming I and II, Jaak Vilo.
- A general method applicable to the search for similarities in the amino acid sequence of two proteins, Needleman and Wunsch, 1970.
- Identification of common molecular subsequences, Smith and Waterman, 1981.
- An improved algorithm for matching biological sequences, Gotoh, 1982.
- (Cormen et al. 2022), Chapter 14.
- (Kleinberg and Tardos 2006), Chapter 6.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.

Kleinberg, Jon, and Éva Tardos. 2006. *Algorithm Design*. Pearson.