Sequence Similarity and Global Alignment

CMSC 423

How far apart are two strings — and which edits get from one to the other?



UNIVERSITY OF MARYLAND

Follows the treatments of Carl Kingsford and Ben Langmead (langmead-lab.org/teaching-materials). Figures redrawn.

From exact matching to similarity

Lectures 5–7

Where does P occur exactly?

suffix tree · suffix array · FM index

→

Today

How similar are two strings?

edit distance · alignment · DP

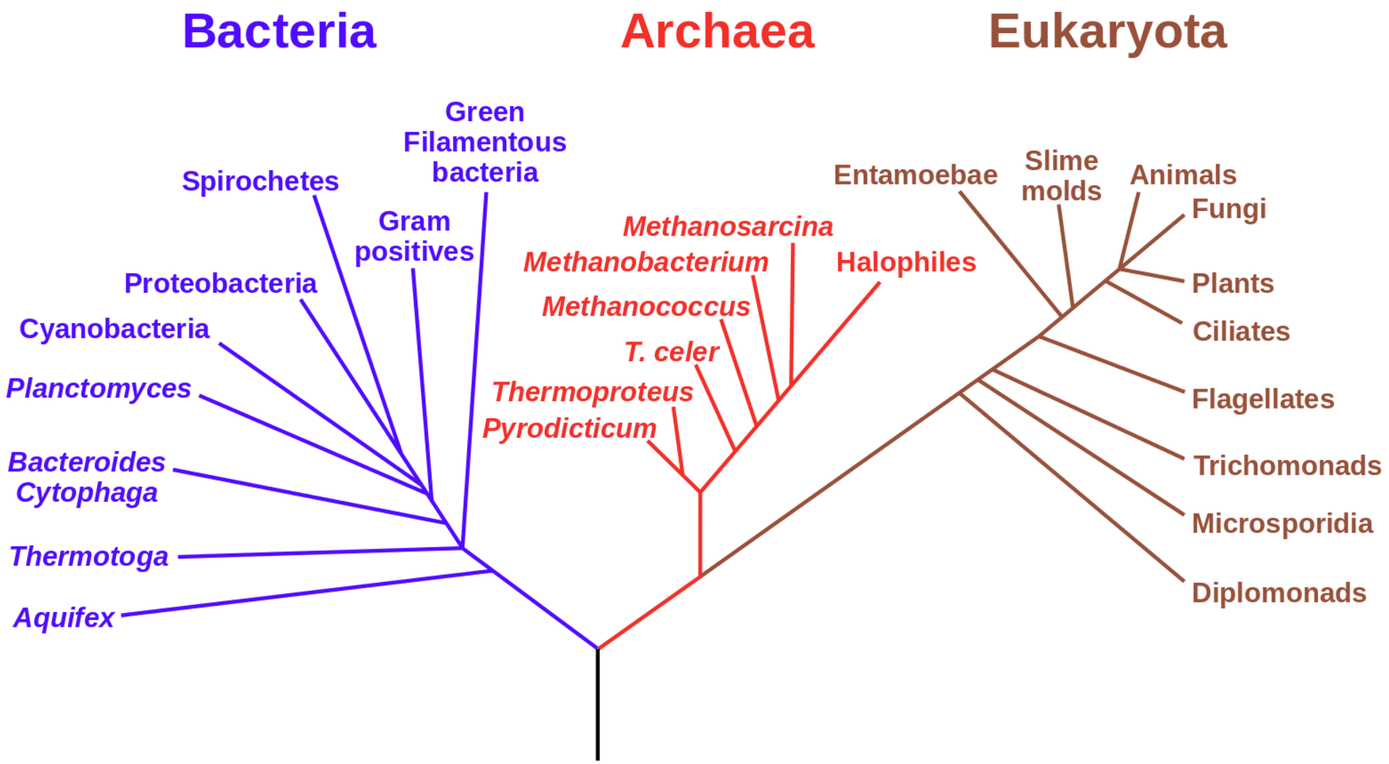
- Exact matching assumes the pattern is *in* the text, letter for letter. Biology almost never promises that: reads carry sequencing errors, and every genome differs from the reference.
- So we need a **number** that says how far apart two strings are, and the **edits** that realise it.
- *Today: the global version — compare two whole strings. Local and banded variants, which read aligners actually run, are built on the same table.*

Why compare sequences?

Similarity is evidence

Why compare – Strings & edits – Dynamic programming – The alignment DP – Traceback & graphs

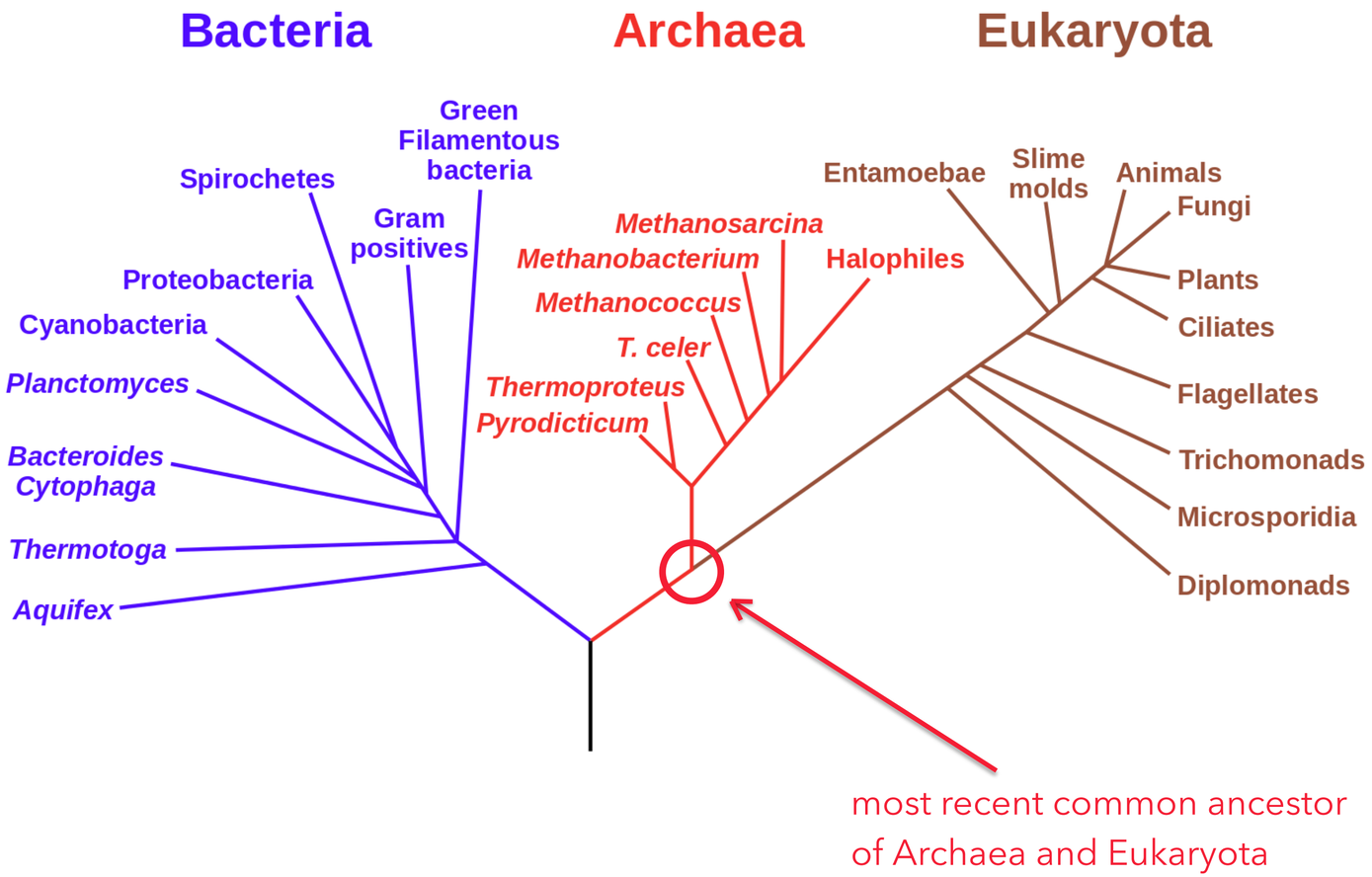
Relatedness of biological sequences



- **Descent with modification.** Organisms inherit their genome from their ancestors, then lineages change independently.
- Two species' versions of a gene are two copies of one ancestral sequence, each edited since they split.
- The more time since the split, the more edits – so *how different* two sequences are estimates *how long ago* their lineages parted.

Tree: Wikimedia Commons, “Phylogenetic tree” (public domain)

Relatedness of biological sequences



- Every branch point is a common ancestor; the one where two lineages last met is their **most recent common ancestor**.
- We cannot sequence it, but we can compare its descendants: what they share was probably inherited, what differs happened since.
- *Comparing sequences is how trees like this one are built in the first place.*

Tree: Wikimedia Commons, “Phylogenetic tree” (public domain)

An analogy: copying a text by hand

THE ORIGINAL

Mix two cups of flour with warm water and knead for ten minutes.

COPY 1 – a substitution

Mix two cups of flour with **worm** water and knead for ten minutes.

COPY 2a – a deletion (the final "s")

Mix two cups of flour with worm water and knead for ten **minute**.

COPY 2b – an insertion (4 characters)

Mix two **big** cups of flour with worm water and knead for ten minutes.

Every copy inherits its parent's mistakes. The two siblings at the bottom are **5** character edits apart – more than either is from its parent (1 and 4). Distance accumulates along the tree.

Sequence tells a story

- If two sequences are *similar*, that is evidence they descend from a common ancestor — sequences this alike rarely arise by chance.
- Sequences are *conserved* at different rates. Changes that break a function are removed by selection, so important regions change slowly.
- So a very similar sequence means either a *recent common ancestor* or a *strongly conserved function* — and the pattern of where the differences fall tells you which.

Why compare DNA or protein sequences?

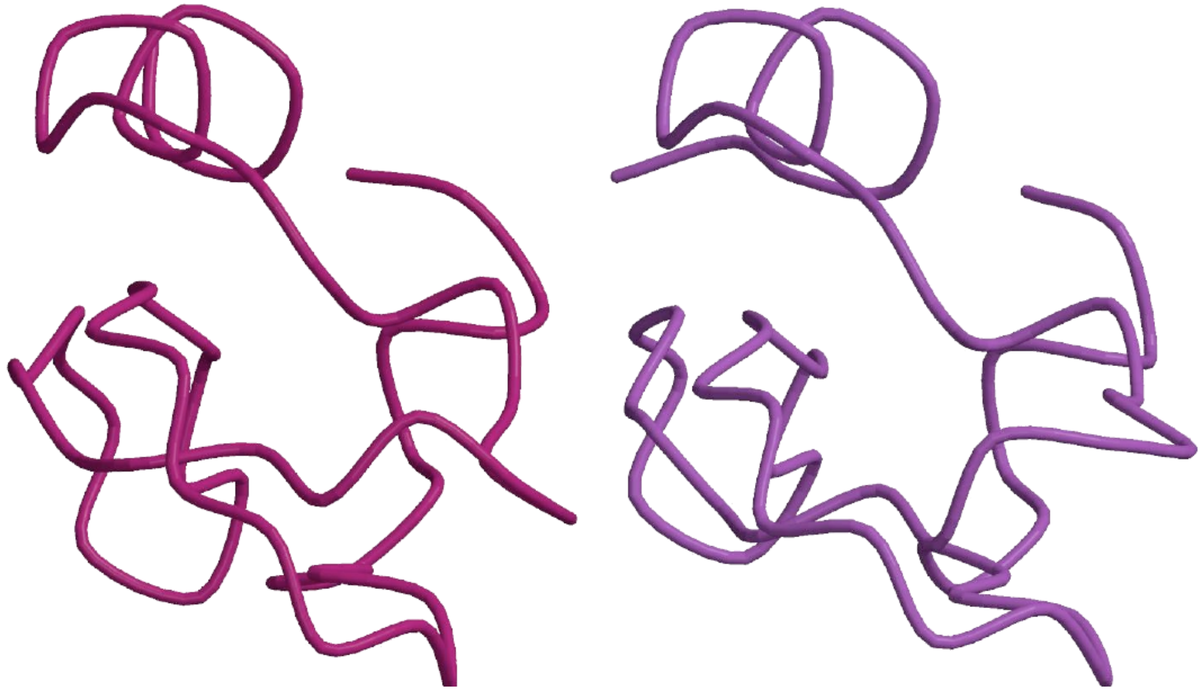
*Part of the CTCF protein in eight species – conserved columns marked **

```
H. sapiens        -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
P. troglodytes    -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
C. lupus          -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
B. taurus         -EDSSDS-ENAEPDLDDNEDEEEPAVEIEPEPE----------PQPVTPA
M. musculus       -EDSSDSEENAEPDLDDNEEEEEPAVEIEPEPE--PQPQPPPPPQPVAPA
R. norvegicus     -EDSSDS-ENAEPDLDDNEEEEEPAVEIEPEPEPQPQPQPQPQPQPVAPA
G. gallus         -EDSSDSEENAEPDLDDNEDEEETAVEIEAEPE----------VSAEAPA
D. rerio          DDDDDSDEHGEPDLDDIDEEDEDDL-LDEDQMGLLDQAPPSVPIP-APA
                    *   ** *   ******    * *                      **
```

- Find **conserved** regions – they are likely to matter
- Find genes similar to known genes, and guess their function
- Measure evolutionary distance – zebrafish (*D. rerio*) is further from us than chicken (*G. gallus*)
- Search databases of sequences (BLAST)
- Assemble genomes and align reads to a reference
- Predict protein structure and function (next slide)

CTCF alignment: en.wikipedia.org/wiki/CTCF (CC BY-SA 3.0)

Sequence can reveal structure



1dtk **5pti**



Dendrotoxin K – black-mamba venom (1dtk)



BPTI – bovine pancreatic trypsin inhibitor (5pti)

Only **42 %** identical (24 of 57 aligned positions) – yet the same fold. All six cysteines line up: they form the three disulfide bridges that hold the fold together.

```
1dtk  XAKYCKLPLRIGPCKRKIPSFYYKWKAKQCLPFDYSGCGGNANRFKTIEECRRTCVG-
      ××××|××|×××|||||××|××××|××||××|××|×|×||××××|×||××|×|×|||×|
5pti  RPDFCLEPPYTGPCKARIIRYFYNAKAGLCQTFVYGGCRAKRNNFKSAEDCMRTCGGA
```

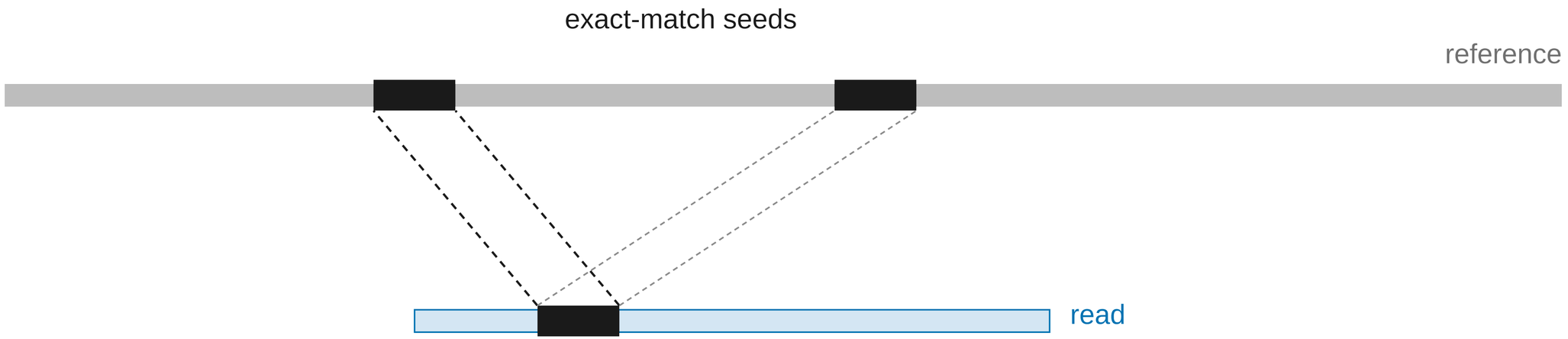
Why exact matching is not enough

Suffix trees, suffix arrays and the FM index find exact occurrences fast. But reads carry **errors**, and the sample truly **differs** from the reference (SNPs, insertions, deletions).

technology	dominant error	error rate	read length
Illumina	substitution	~0.1–0.5 %	50–300 bp
PacBio HiFi	indel	< 1 %	10–25 kb
Oxford Nanopore	indel	~1–5 %	10 kb – 1 Mb+

Rates are approximate and keep falling: HiFi is typically ~0.1 %, and the newest Nanopore chemistry approaches 1 %; older long-read chemistries ran at 10–15 %. Either way, a 10 kb read is all but guaranteed to contain differences.

The typical strategy: seed and extend



- **Seed**: find short substrings of the read that occur exactly in the reference — the FM index's job.
- **Filter / vote**: keep only the candidate locations supported by enough seeds.
- **Extend**: run an alignment DP around each surviving location — *today's algorithm*, restricted to a small window.
- *A heuristic: a read with an error in every seed is missed. It works because errors are rare enough that some seed is usually clean.*

Strings and edits

Making “similar” precise

Why compare – Strings & edits – Dynamic programming – The alignment DP – Traceback & graphs

The language of strings

- A **string** *s* is a finite sequence of characters; |*s*| is its **length**.
- Characters come from an **alphabet** Σ:

Σ_{DNA} = {A, C, G, T} Σ_{RNA} = {A, C, G, U}

$\Sigma_{protein}$ = {A,R,N,D,C,E,Q,G,H,I,L,K,M,F,P,S,T,W,Y,V}

- The **empty string** is written ε, and |ε| = 0.
- Concatenation: *st* is *s* followed by *t*.

adapted from a slide by Ben Langmead

Substrings, subsequences, prefixes, suffixes

- *s* is a **substring** of *t* if *t* = *usv* for some (possibly empty) strings *u* and *v* – the characters are *consecutive* in *t*.
- *s* is a **subsequence** of *t* if its characters appear in *t* in order, but not necessarily consecutively.
- *s* is a **prefix** of *t* if *t* = *sv*, and a **suffix** if *t* = *us*. It is *proper* if it is not *t* itself.

v a **c a t** i o n cat is a substring of vacation

v a **c a** t i o **n** can is a subsequence of vacation

v a c a t i o n vac is a prefix of vacation

adapted from a slide by Ben Langmead

The simplest string comparison problem

Given two strings over an alphabet Σ

$a = a_1 a_2 \ldots a_m \qquad b = b_1 b_2 \ldots b_n$

compute how *similar* they are.

The **edit distance** between *a* and *b* is the smallest number of these operations that turns *a* into *b*:

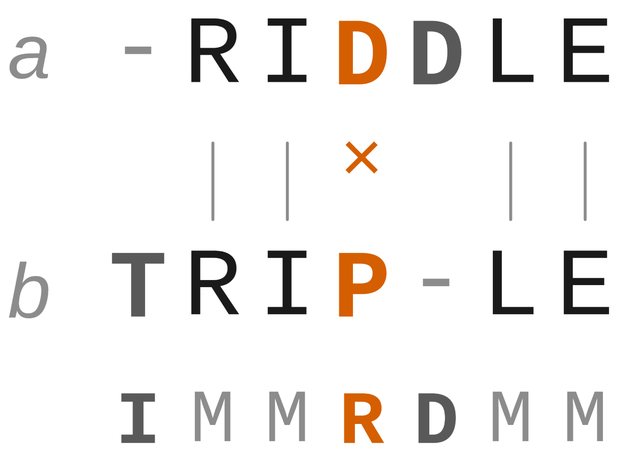
- **substitute** (mutate) one character for another
- **delete** a character
- **insert** a character



Three edits suffice. Can it be done in two? Today's algorithm answers that.

An alignment writes the edits down

Stack the strings, inserting gaps "–" so that every column holds a character of *a* over a character of *b*, or one of them over a gap.



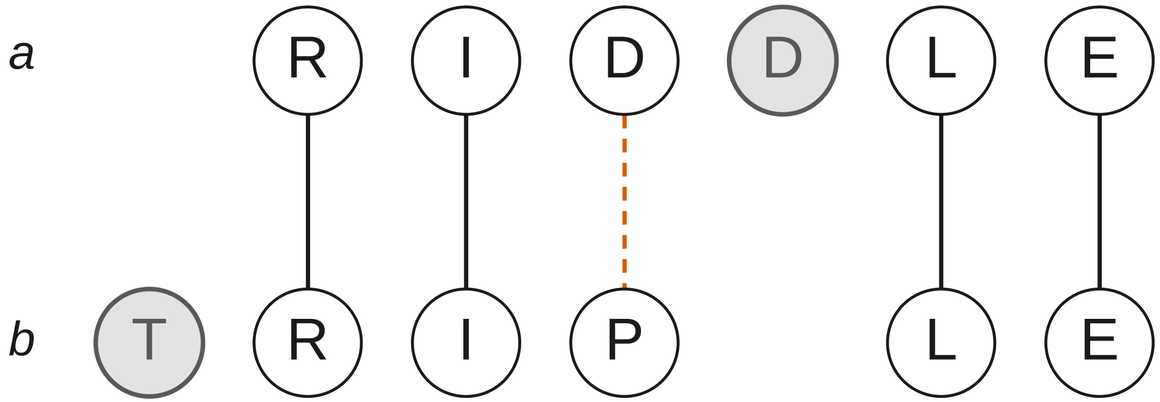
M	**match**	*same character*
R	**replace**	*a substitution*
I	**insert into a**	*a has the gap*
D	**delete from a**	*b has the gap*

- Reading the columns left to right gives an **edit transcript**: `IMMRDMM` turns RIDDLE into TRIPLE.
- Its cost is the number of non-M letters — here 3, the edit distance. Every alignment is a transcript and every transcript an alignment.

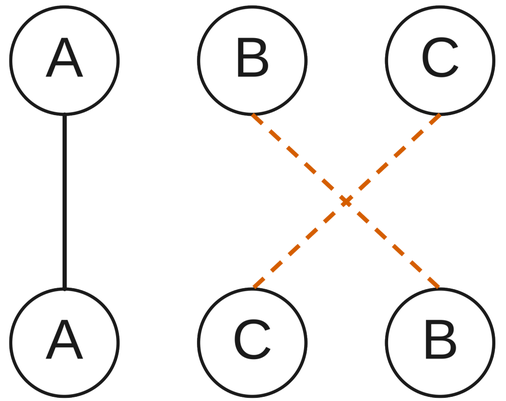
Another view: an alignment is a matching

Draw each string as a row of nodes; each column with two characters becomes an edge. Unmatched nodes are the gaps.

(a) An alignment of RIDDLE and TRIPLE



(b) Crossing edges



not an alignment

- **Edges may not cross** – an alignment keeps both strings in order. (Swapping two characters is not one of our operations.)
- Cost of a matching = *gap* × #unmatched nodes + Σ *cost*(a_i, b_j) over the edges. Solid edges are free matches; dashed orange edges cost a mismatch.

Edit transcripts, on a longer example

An "optimal editor" walks left to right and turns *x* into *y*. `I` = insert into *x* (gap in *x*), `D` = delete from *x* (gap in *y*).

```
x  G C G T A T G C G G C T A - A C G C
   |  |   | | | | | | | | | |   | | | |
y  G C - T A T G C G G C T A T A C G C
   M M D M M M M M M M M M M I M M M M
```

- The G that *x* has and *y* lacks is a **deletion** from *x* → `D`; the T that *y* has and *x* lacks is an **insertion** → `I`.
- Two edits: the edit distance between these 17-character strings is 2.
- Read the other way – turning *y* into *x* – *every I becomes a D and vice versa. Insertion and deletion are the same operation seen from the two ends.*

example after Ben Langmead

Practice: read the transcript off the alignment

(a)

```
prin-ciple
|||| ||| ××
prinncipal
MMMMIMMMRR
```

1 gap, 2 mm

(b)

```
prin-cip-le
|||| ||| |
prinncipal-
MMMMIMMMIMD
```

3 gaps, 0 mm

(c)

```
misspell
||| ||||
mis-pell
MMMDMMMM
```

1 gap

(d)

```
prehistoric
   |||||||||
---historic
DDDMMMMMMMMM
```

3 gaps

(e)

```
aa-bb-ccaabb
|× || | | |
ababbbc-a-b-
MRIMMIMDMDMD
```

5 gaps, 1 mm

(f)

```
al-go-rithm-
|| ×× || × |
alKhwariz-mi
MMIRRIMMRDMI
```

4 gaps, 3 mm

Bold grey = gaps (I, D), orange = substitutions (R): their count is this alignment's cost.

Scoring an alignment

- **gap** – the cost of a column with "–" in it (an insertion or a deletion).
- **cost**(*x*, *y*) – the cost of a column holding *x* over *y*. In the simplest case cost(*x*, *x*) = 0 and cost(*x*, *y*) = *mismatch*.
- The cost of an alignment is the sum over its columns. We want the **lowest-cost** alignment. (Equivalently, the highest-scoring one, with scores = −costs.)

```
- R I D D L E
  |  |  ×     |  |
T R I P - L E
3 0 0 1 3 0 0
```

gap = 3, mismatch = 1

cost = 3 + 0 + 0 + 1 + 3 + 0 + 0 = 7

The costs choose the alignment

Two alignments of the same strings. Which is “best” depends on what gaps cost relative to mismatches.

(a) 2 gaps + 1 mismatch

```
-RIDDLE
  ||×  ||
TRIP-LE
```

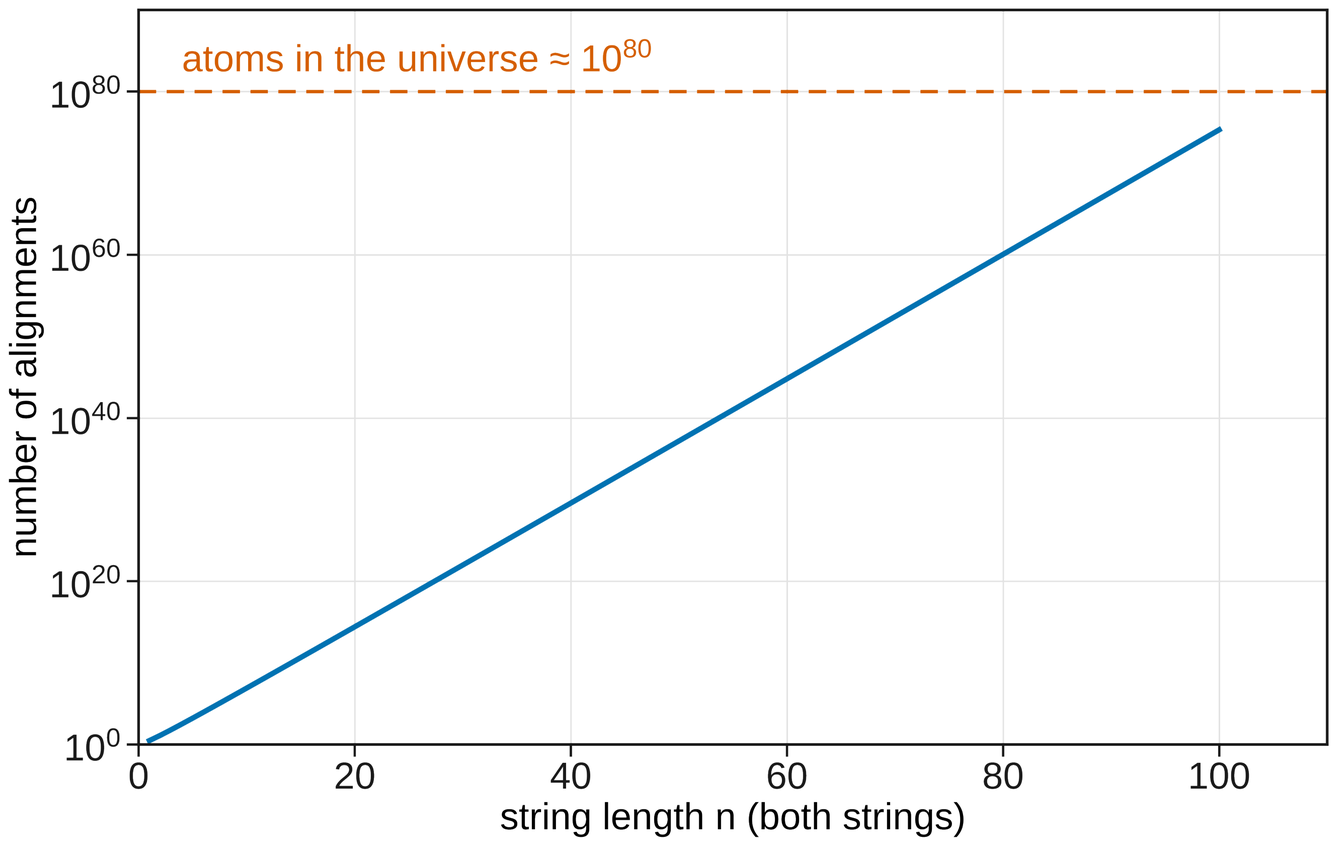
(b) 4 mismatches

```
RIDDLE
××××||
TRIPLE
```

	-RIDDLE / TRIP-LE	RIDDLE / TRIPLE
unit costs (edit distance)	**3** best	4
gap = 3, mismatch = 1	7	**4** best

Edit distance is the special case where every operation costs 1. The algorithm we build works for any costs.

How many alignments are there?



$$f(m, n) = \Sigma_k \; 2^k \cdot C(m, k) \cdot C(n, k)$$

- Why: the last column is one of three things, so $f(m, n) = f(m-1, n-1) + f(m-1, n) + f(m, n-1)$ – the Delannoy numbers.
- Two strings of length 3: **63**. Our 12 × 9 DNA example: **14,218,905**.
- Two of length 100: a **76-digit** number. Enumeration is hopeless.

Formula: Andrade, Area, Nieto & Torres, “The number of reduced alignments between two DNA sequences,” BMC Bioinformatics 15:94 (2014)

What an alignment looks like in practice

```
>gb|AC115706.7| Mus musculus chromosome 8, clone RP23-382B3, complete sequence

Query  1650   gtgtgtgtgggtgcacatttgtgtgtgtgtgcgcctgtgtgtgtgggtgcctgtgtgtgt  1709
              ||||||||||  |    ||  | |||||||||| | ||||||||    ||| ||  |||||
Sbjct  56838  GTGTGTGTGGAAGTGAGTTCATCTGTGTGTGCACATGTGTGTGCA--TGCATGCATGTGT  56895

Query  1710   gtg-gggcacatttgtgtgtgtgtgtgcctgtgtgtgggtgcacatttgtgtgtgtgc  1768
              ||  |||||      ||  ||| ||||||| |||||||| |||  |||  ||||| || |
Sbjct  56896  GTCCGGGCA------TGCATGTCTGTGTGCATGTGTGTGTGTGTGCAT--GTGTGAGTAC  56947

Query  1769   ctgtgtgtgtgtgcctgtgtgtgggggtgcacatttgtgtgtgtgtgtgcctgtgtgtgg  1828
              |||||||||| ||| ||| |||| | |||  ||| |||||  ||||||   |||||  |
Sbjct  56948  CTGTGTGTGTATGCTTGTATGTGTGTGTGTGCATGTGTGTAGGTGTGTATATGTGTAAGT  57007

Query  1829   gggtgcacatttgtgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgtgt  1888
                     ||| ||||||| ||||||  |||| | |||  ||||    |||||||||| ||
Sbjct  57008  T------CATCTGTGTGTATGTGTG--TGTGAGAGTGCATGCA----TGTGTGTGTGAGT  57055

Query  1889   gcctgtgtgt--gtgggtgcacatttgtgtgtgtgtgcctgtg--tgtgt--gggtgcac  1942
               | | |||||  |||  |||  ||  || |   | |  |||||  |||||  | |||  |
Sbjct  57056  TCATCTGTGTCAGTGTATGCTTATGGGTATAACT-TAACTGTGCATGTGTAAGTGTGTTC  57114

Query  1943   atttgtgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgcctgtgtgtgg  2002
              || ||||| ||||||||  ||||||  ||  || ||    | ||||||||   |||||||
Sbjct  57115  ATCTGTGTATGTGTGTG--TGTGTGAGTTAGTTCA----TCTGTGTGTGAGAGTGTGTGA  57168

Query  2003   gtgcacatttgtgtgtgtgtgcctgtgtgtgtgtgcctgtgtgtgtgggtgcacatttgt  2062
              |  | ||| |||||||| ||  | | ||| || |||  ||||| |||| ||| |||  ||
Sbjct  57169  G--CTCATCTGTGTGTGAGTTCATCTGTATGAGTG--TGTGTATGTGTGTGTACAAATGA  57224

Query  2063   gtgtgtgtgtgcctgtgtgtgtgggtgcacatttgtgtgtgtgtgtgtgcctgtgtgtgt  2122
              ||   | ||||| ||||||||||        |||  ||||||   | ||  |||| ||||
Sbjct  57225  GTTCATCTGTGCATGTGTGTGTG--------TTTAAGTGTGTTCATCTG--TGTGCGTGT  57274
```

- NCBI BLAST output: a query aligned against a mouse genomic clone.
- Same three-row layout — **|** for a match, “–” for a gap.
- *BLAST is a heuristic, seed-and-extend search over a whole database; the alignment it reports is a DP like the one we are about to build.*

Dynamic programming

Two warm-ups before the real thing

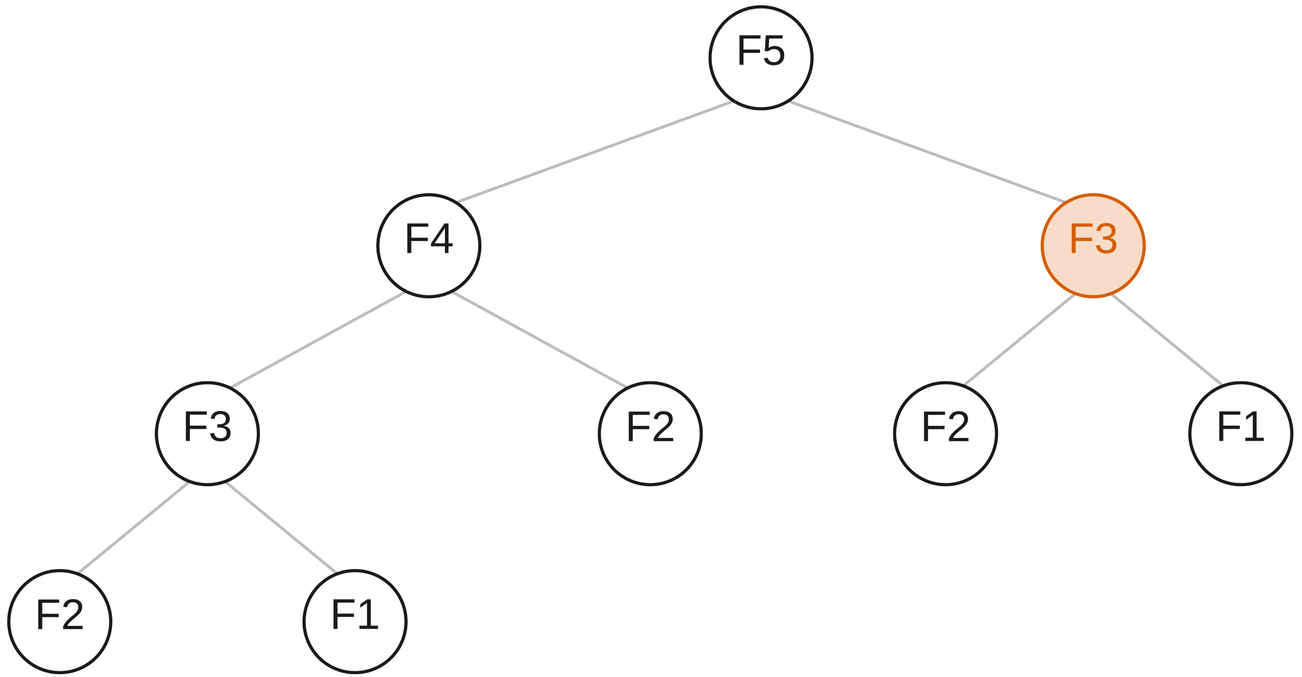
Why compare – Strings & edits – **Dynamic programming** – The alignment DP – Traceback & graphs

Dynamic programming

- A general and powerful *algorithm design* technique. “Programming” in the old sense of filling in a table – nothing to do with code.
- It applies when a problem has
- **optimal substructure** – an optimal solution is built from optimal solutions to smaller subproblems, and
- **overlapping subproblems** – the same subproblem is needed again and again, so its answer is worth keeping.
- The recipe: write the answer as a recurrence over subproblems, then solve the subproblems in an order where each one's ingredients are already in the table.

Example 1: Fibonacci, recursively

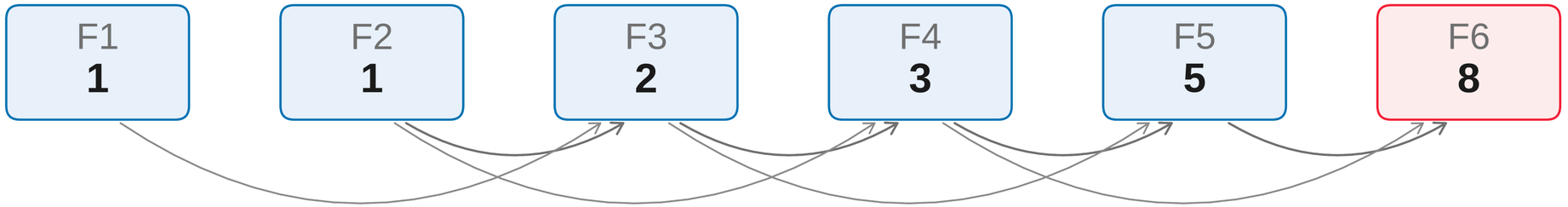
$$F_n = F_{n-1} + F_{n-2} \quad \text{with} \quad F_1 = F_2 = 1$$




```
def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)
```

- F_3 is computed twice (the repeat is shaded), F_2 three times. The waste compounds: `fib(30)` makes 1,664,079 calls to compute 832,040.
- Running time $\Theta(\varphi^n)$, $\varphi = (1+\sqrt{5})/2 \approx 1.618$ – exponential.

Example 1: Fibonacci, in the right order

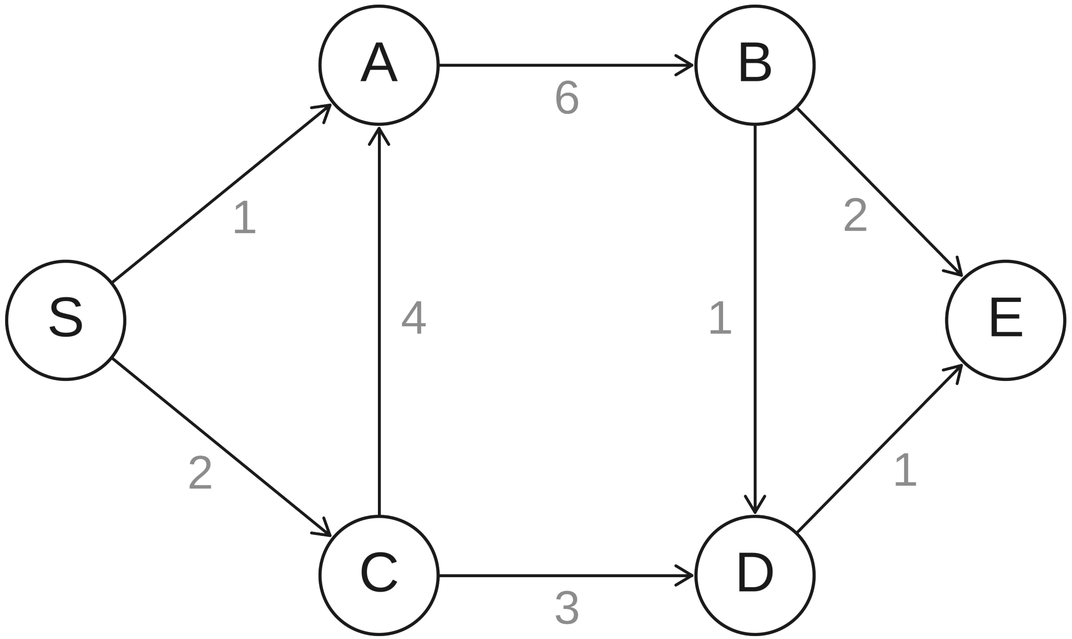



```
def fib(n):
    prev, cur = 1, 1          # F(1), F(2)
    for _ in range(n - 2):
        prev, cur = cur, prev + cur
    return cur
```

- Compute F_1, F_2, F_3, … in order: each needs only the two before it, which are ready.
- **n – 2 additions: O(n).** Same recurrence, exponentially less work.
- *(Treating addition as O(1) – not true once the numbers outgrow a machine word.)*

Example 2: shortest paths in a DAG

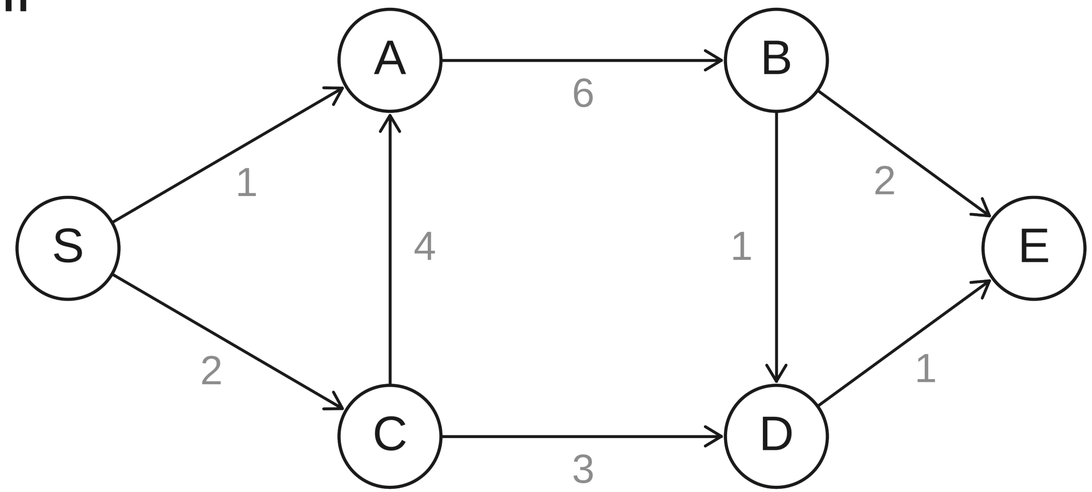
- A **directed acyclic graph** (DAG): edges have a direction, and no path returns to where it started.
- So the vertices can be put in a **topological order** – a numbering in which every edge points from a smaller number to a larger one.
- Edge weights $\ell(u, v)$ are lengths. Question: the shortest distance d(S, X) from S to every X.



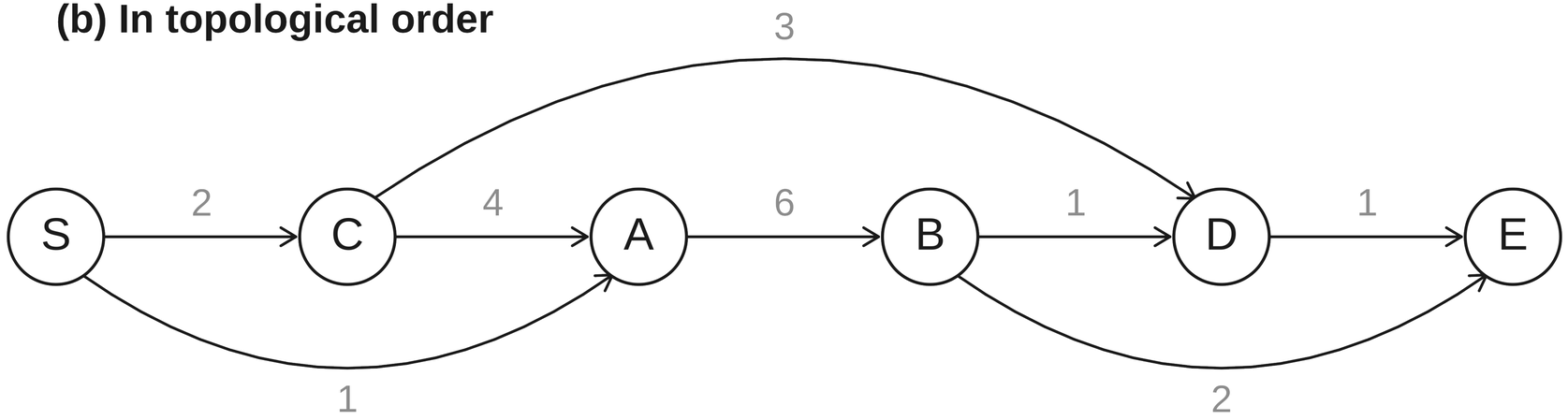
after Dasgupta, Papadimitriou & Vazirani, Algorithms (2006), Fig. 6.1

Every DAG can be linearised

(a) As drawn



(b) In topological order



the same graph, in topological order: every edge points right

after Dasgupta, Papadimitriou & Vazirani, Algorithms (2006), Fig. 6.1

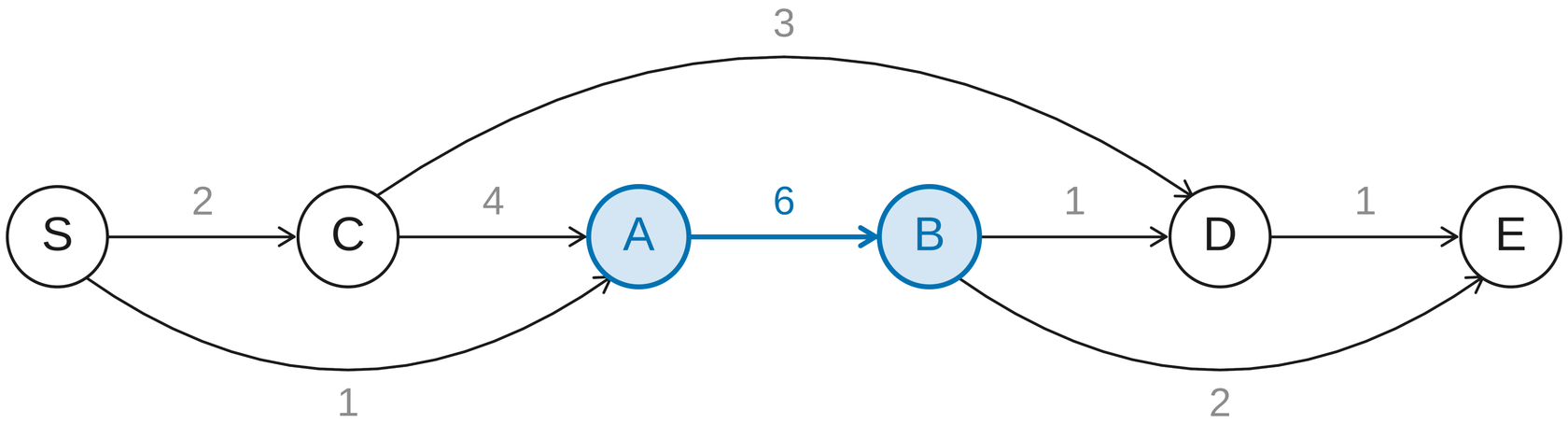
Finding a topological order: Kahn's algorithm

```
L ← empty list                    # the order, built left to right
Q ← all nodes with no incoming edge
while Q is not empty:
    remove a node u from Q; append u to L
    for each edge e = (u, w):
        remove e from the graph
        if w has no other incoming edge:
            insert w into Q
if the graph still has edges: error (it has a cycle)
else: return L
```

- Repeatedly take a vertex nothing points into.
- **O(|V| + |E|)**: each vertex enters Q once, each edge is removed once.
- On our DAG: `S C A B D E`.

en.wikipedia.org/wiki/Topological_sorting

Example 2: what is d(S, B)?



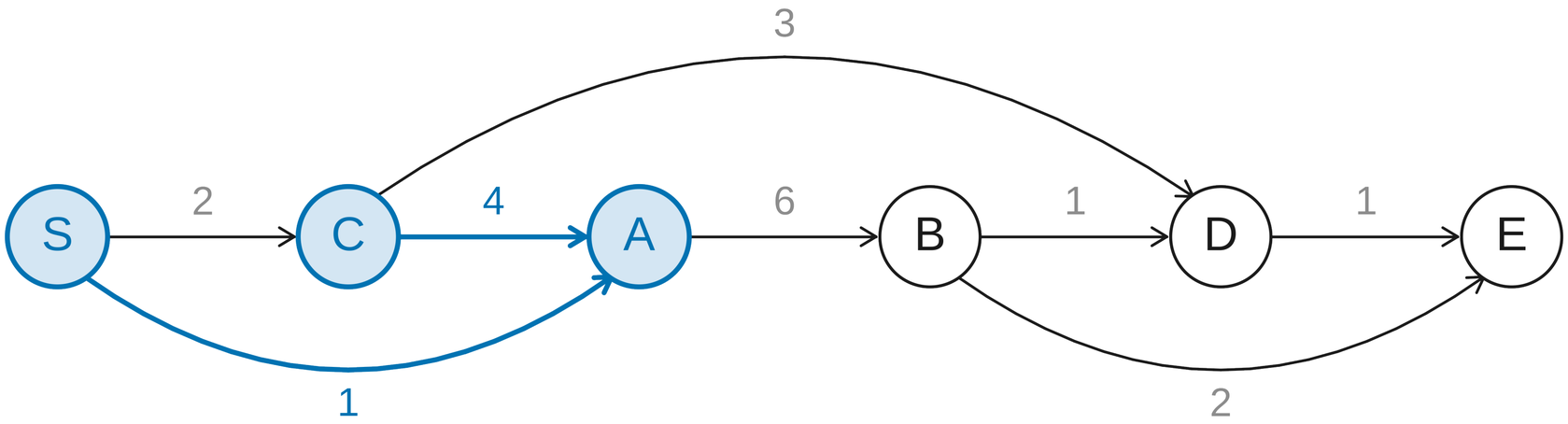
- Only one edge enters B – from A. So every path to B ends A → B, and

$$d(S, B) = d(S, A) + 6$$

- The question about B became a question about A.

after Dasgupta, Papadimitriou & Vazirani, Algorithms (2006), Fig. 6.1

Example 2: what is d(S, B)?

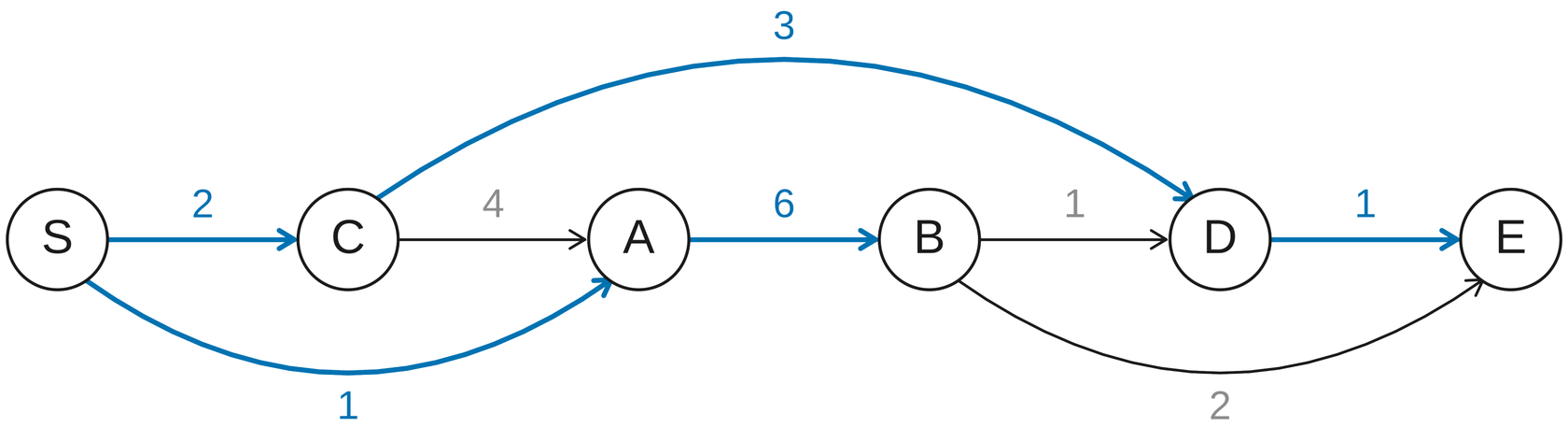


- Two edges enter A – from S and from C. Take the cheaper:
- `d(S, A) = min( d(S, S) + 1, d(S, C) + 4 ) = min(1, 6) = 1`
- so `d(S, B) = 1 + 6 = 7`.

after Dasgupta, Papadimitriou & Vazirani, Algorithms (2006), Fig. 6.1

The DAG recurrence

$$d(S, X) = \min_{(Y,X)\in E} \{\, d(S, Y) + \ell(Y, X) \,\}$$



X	S	C	A	B	D	E
d(S, X)	**0**	**2**	**1**	**7**	**5**	**6**
via	–	S	S	A	C	D

Fill the distances in topological order: when we reach X, every Y with an edge into X is already done. One pass, O(|V| + |E|). Blue edges: the shortest-path tree.

after Dasgupta, Papadimitriou & Vazirani, Algorithms (2006), Fig. 6.1

The alignment DP

Edit distance as a recurrence

Why compare – Strings & edits – Dynamic programming – **The alignment DP** – Traceback & graphs

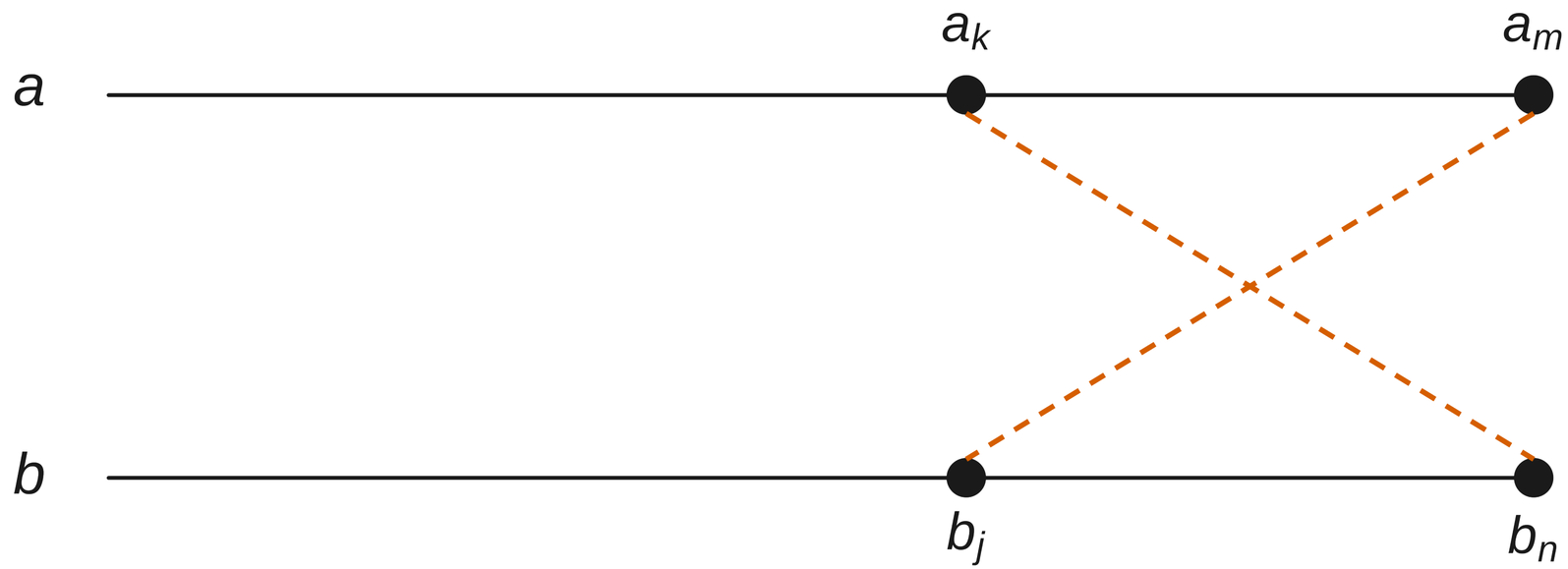
Think about the last column

Take an optimal alignment of $a = a_1 \ldots a_m$ and $b = b_1 \ldots b_n$, and look at what happens to the last characters a_m and b_n. One of these must hold:

1. a_m and b_n are aligned to each other
2. a_m is not aligned to anything (it faces a gap)
3. b_n is not aligned to anything
4. a_m is aligned to some b_j with j ≠ n, *and* b_n to some a_k with k ≠ m

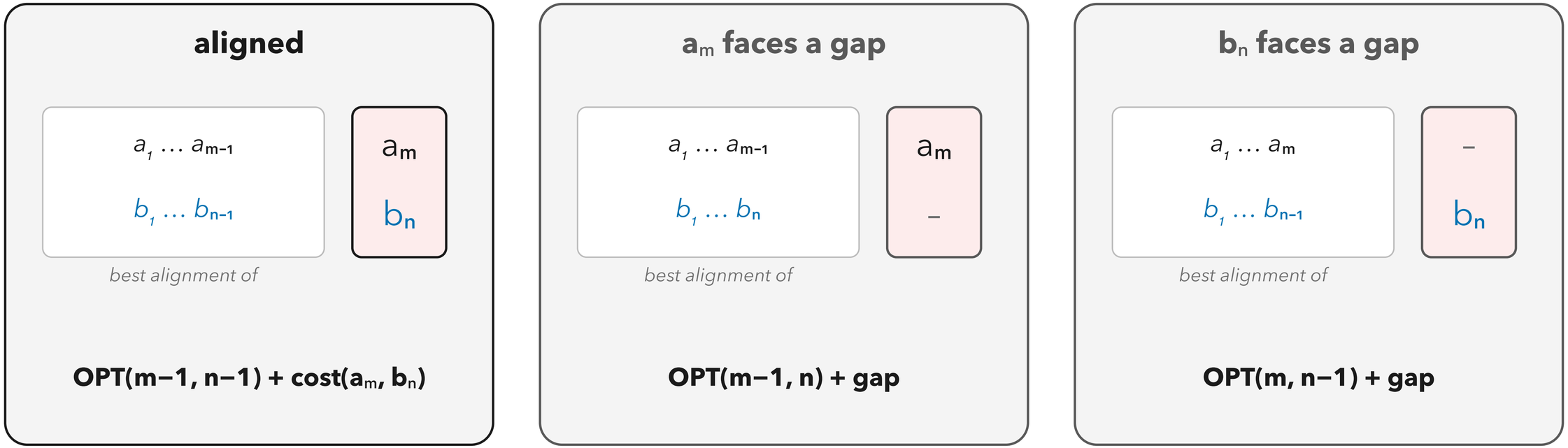
Case 4 cannot happen. Why?

The no-crossing rule forbids case 4



- If a_m pairs with an earlier b_j and b_n with an earlier a_k, the two edges cross — and an alignment never crosses.
- So the last column is **one of exactly three things**: a_m over b_n; a_m over –; – over b_n.

Three cases, three smaller problems



- Whichever case holds, the part before the last column must itself be an optimal alignment of the shorter strings — otherwise swap in a better one and the whole thing improves. That is **optimal substructure**.
- We do not know which case holds, so we try all three and take the cheapest.

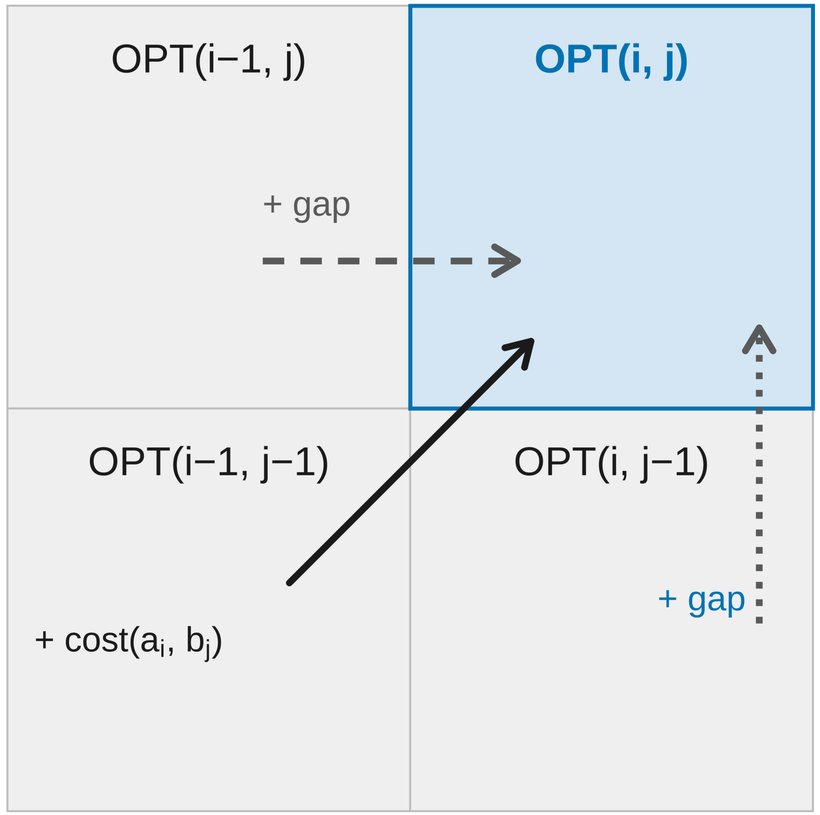
The recurrence

Let **OPT(i, j)** be the cost of the best alignment of the prefixes $a_1 \ldots a_i$ and $b_1 \ldots b_j$. Then for i, j ≥ 1:

$$\text{OPT}(i, j) = \min \begin{cases} \text{cost}(a_i, b_j) + \text{OPT}(i-1, j-1) & \textit{a_i over b_j} \\ \text{gap} + \text{OPT}(i-1, j) & \textit{a_i over a gap} \\ \text{gap} + \text{OPT}(i, j-1) & \textit{b_j over a gap} \end{cases}$$

- **Base cases**: OPT(i, 0) = i · gap and OPT(0, j) = j · gap – aligning i characters against nothing takes i gaps. OPT(0, 0) = 0.
- The answer is **OPT(m, n)** – all of *a* against all of *b*.
- *With gap = 1 and mismatch = 1 this is the edit distance.*

The table: what one cell sees



- String *a* runs along the bottom (column i), *b* up the side (row j); (0, 0) is bottom-left and the answer OPT(m, n) is top-right.
- **diagonal** → adds the column a_i over b_j; cost(a_i, b_j)
- **from the left** → adds a_i over –; gap
- **from below** → adds – over b_j; gap

In what order do we fill the table?

A cell needs its left, lower and lower-left neighbours. Any order that respects that works — here are three, on a 4 × 3 table. Grey = base cases, filled first.

(a) Row by row

	9	10	11	12
	5	6	7	8
	1	2	3	4

(b) Column by column

	3	6	9	12
	2	5	8	11
	1	4	7	10

(c) Anti-diagonals

	4	7	10	12
	2	5	8	11
	1	3	6	9

The pseudocode on the next slide uses column by column (i outermost). Anti-diagonals are the order to use in parallel: every cell on one diagonal is independent of the others.

The algorithm

```
EditDistance(a[1..m], b[1..n]):
    for i = 0..m:  OPT[i, 0] = i * gap
    for j = 0..n:  OPT[0, j] = j * gap
    for i = 1..m:
        for j = 1..n:
            OPT[i, j] = min( cost(a[i], b[j]) + OPT[i-1, j-1],
                             gap + OPT[i-1, j],
                             gap + OPT[i, j-1] )
    return OPT[m, n]
```

- **Time O(mn)**: (m+1)(n+1) cells, O(1) work each.
- **Space O(mn)** for the whole table – needed to trace the alignment back.
- *The cost alone needs only the previous column: O(min(m, n)) space, looping over the longer string outermost. Hirschberg's algorithm recovers the alignment in linear space too.*

Why the corner holds the answer

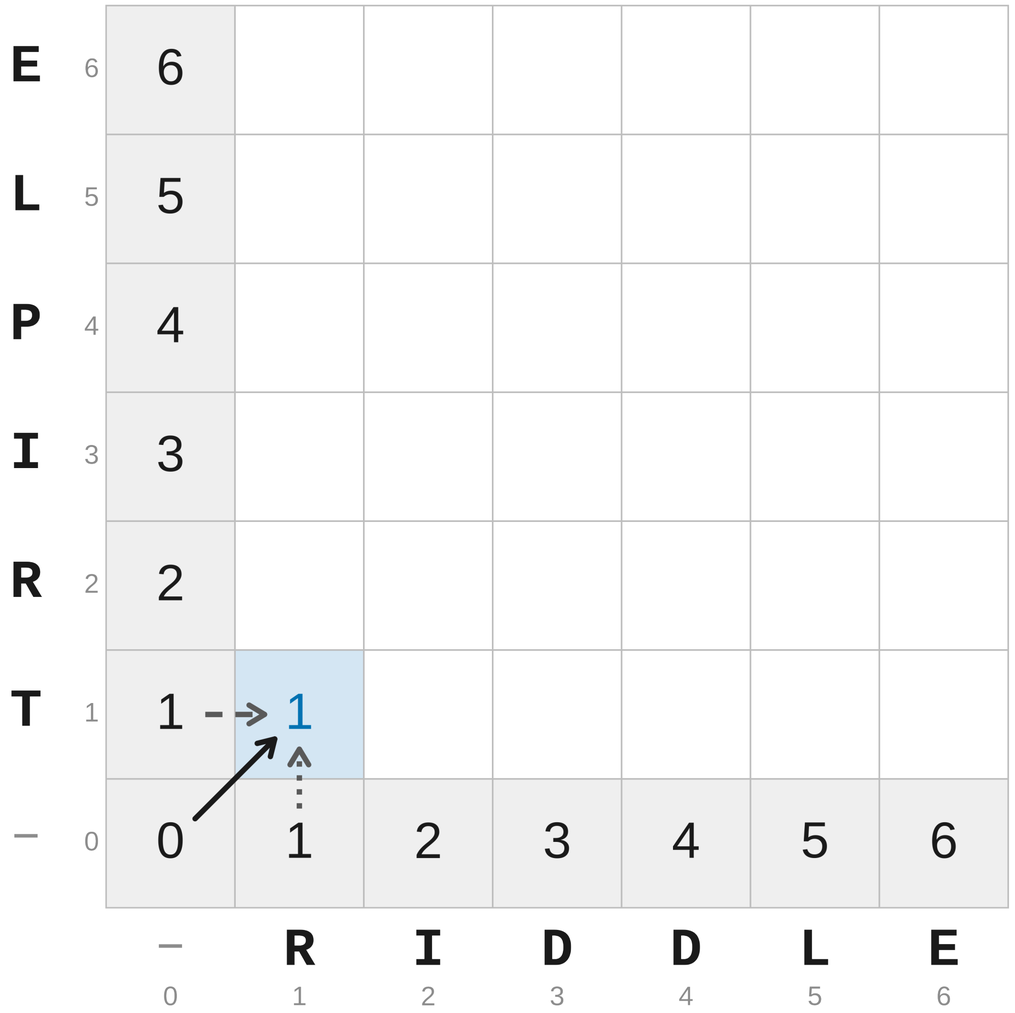
- Claim: **every** cell OPT(i, j) holds the optimal cost of aligning $a_1 \ldots a_i$ with $b_1 \ldots b_j$.
- Base cases: true — the only alignment against an empty string is all gaps.
- Step: when OPT(i, j) is filled, its three neighbours are already right (by the fill order). The last column of any optimal alignment is one of the three cases, and each case's cost is a neighbour plus that column. The min picks the best.
- So OPT(m, n) is the optimal cost of aligning all of *a* with all of *b*. (Note: OPT(m, n), *not* OPT(n, m) — i runs over *a*, which has length m.)

Example: RIDDLE vs TRIPLE, unit costs

		–	R	I	D	D	L	E
		0	1	2	3	4	5	6
E	6	6						
L	5	5						
P	4	4						
I	3	3						
R	2	2						
T	1	1						
–	0	0	1	2	3	4	5	6

- *a* = `RIDDLE` along the bottom, *b* = `TRIPLE` up the side.
- gap = 1, mismatch = 1: this computes the **edit distance**.
- First the base cases: row 0 and column 0 are 0, 1, 2, … – aligning a prefix against nothing.
- Then fill column by column, bottom to top.

Example: RIDDLE vs TRIPLE, unit costs

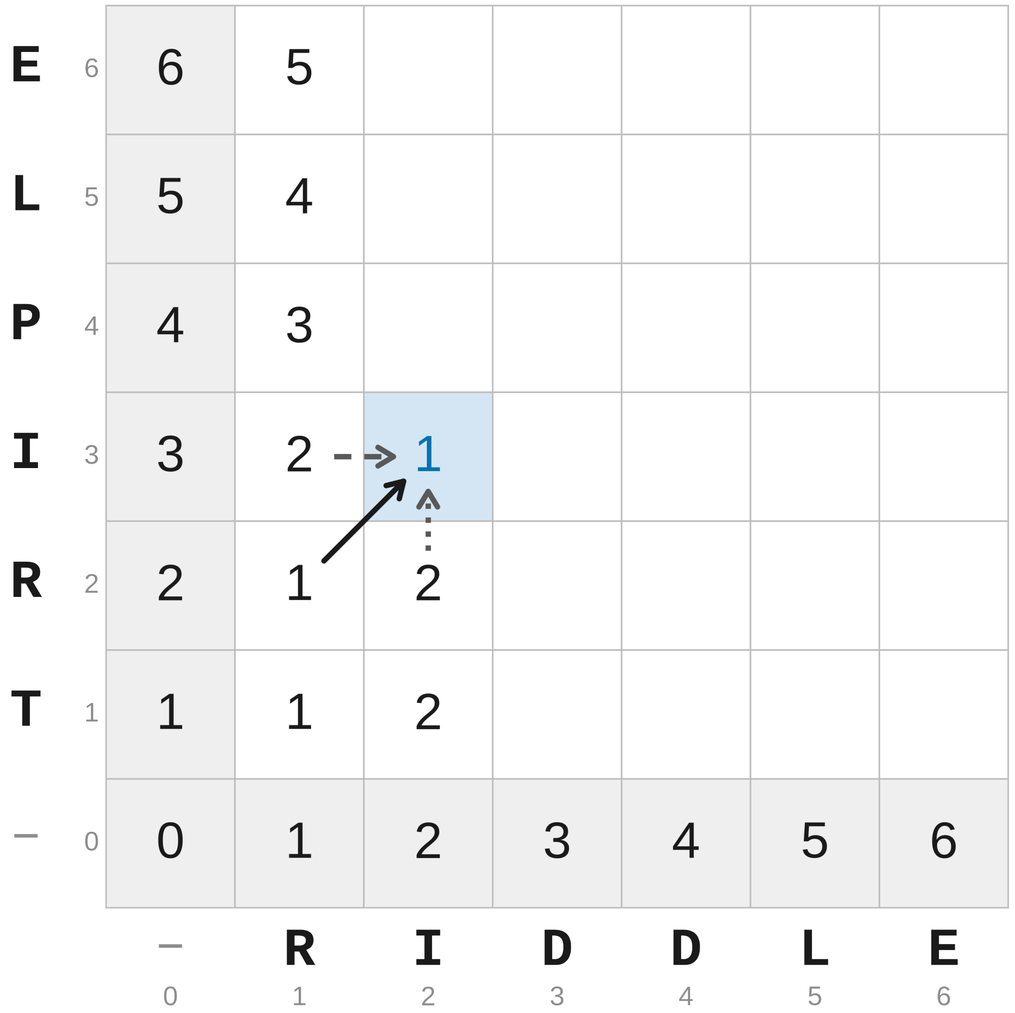


Cell (1, 1): R against T – a mismatch.

```
min( 1 + 0,
     1 + 1,
     1 + 1 ) = 1
```

diagonal (solid arrow), from the left (dashed), from below (dotted). The diagonal wins: put R over T.

Example: RIDDLE vs TRIPLE, unit costs



Column 1 is done; in column 2 we reach cell (2, 3):

`I` against `I` – a **match**, cost 0.

```
min( 0 + 1,
     1 + 2,
     1 + 2 ) = 1
```

“RI” vs “TRI” costs 1: one insertion of T.

Example: RIDDLE vs TRIPLE, unit costs

		–	R	I	D	D	L	E
		0	1	2	3	4	5	6
E	6	6	5	4	4	4	4	3
L	5	5	4	3	3	3	3	4
P	4	4	3	2	2	3	4	5
I	3	3	2	1	2	3	4	5
R	2	2	1	2	3	4	5	6
T	1	1	1	2	3	4	5	6
–	0	0	1	2	3	4	5	6

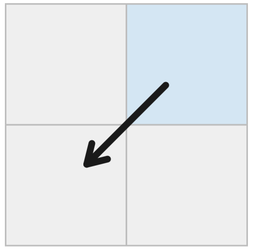
- Every cell filled. The answer is in the top-right corner: **OPT(6, 6) = 3**.
- So three edits really is the best – no two-edit route from RIDDLE to TRIPLE exists.
- *Every other cell answers a smaller question too: e.g. OPT(3, 4) = 2 is the edit distance between "RID" and "TRIP".*

Traceback & graphs

From the number to the alignment

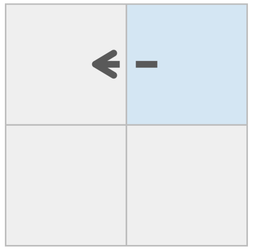
Why compare – Strings & edits – Dynamic programming – The alignment DP – **Traceback & graphs**

Reading the alignment back out



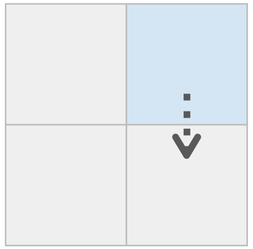
diagonal

emits a_i over b_j



left

emits a_i over –

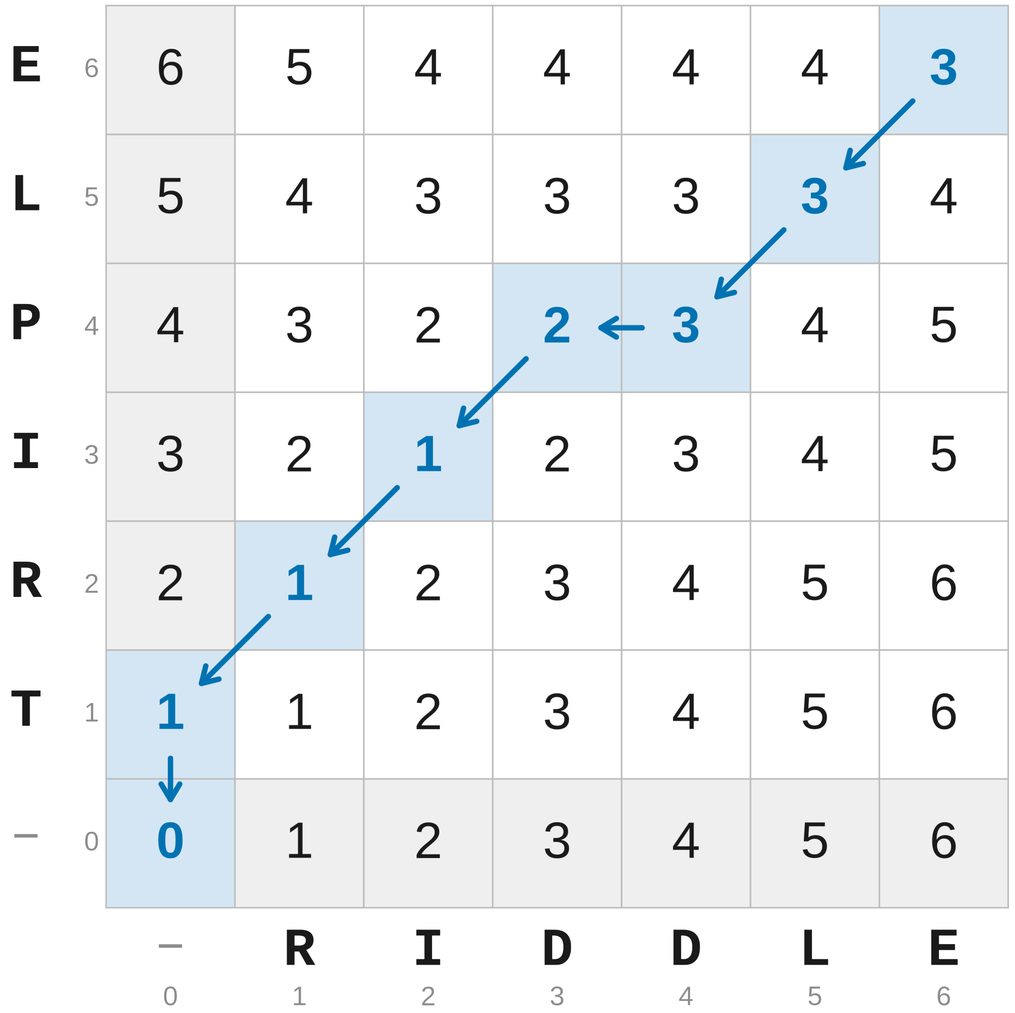


down

emits – over b_j

- Start at OPT(m, n). At each cell, find a neighbour that *produced* its value — its value plus the cost of that move equals this cell — step there, and write down the column.
- Stop at (0, 0). The columns came out **right to left**, so reverse them.
- Either store a pointer per cell while filling, or recompute the three candidates on the way back: O(m + n) steps either way.
- *If two neighbours both work, either choice gives an optimal alignment.*

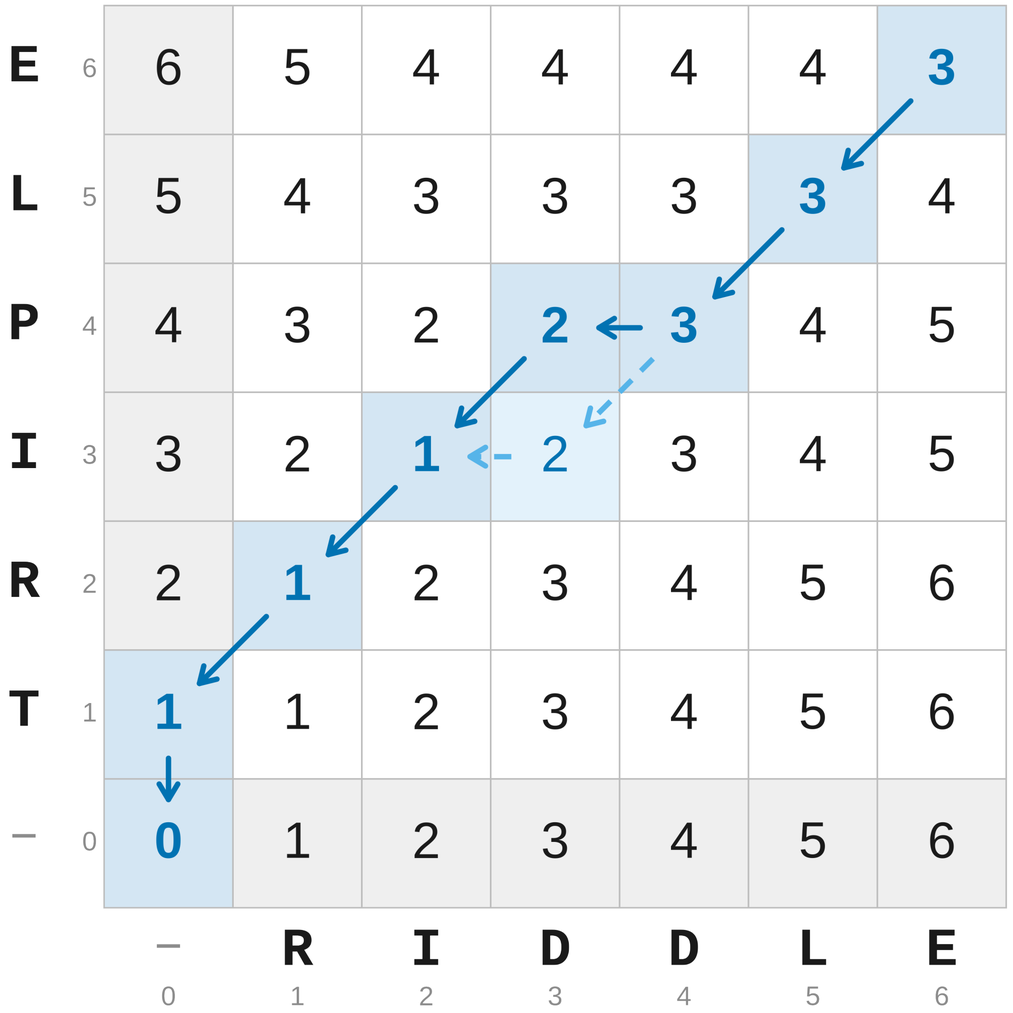
Tracing back the alignment



- Start at the corner and ask: which neighbour did my value come from? Step there, and repeat until (0, 0).
- Each step back emits one column, so the alignment comes out **right to left**.

```
-RIDDLE
  ||×  ||
TRIP-LE
IMMRDMM
```

Ties: more than one optimal alignment



At cell (4, 4) two moves tie, so two paths – and two alignments – cost 3.

(a) Solid path

```
-RIDDLE
 ||×  ||
TRIP-LE
IMMRDMM
```

(b) Dashed path

```
-RIDDLE
 ||  ×||
TRI-PLE
IMMDRMM
```

Same strings, different costs: gap = 3, mismatch = 1

		–	R	I	D	D	L	E
		0	1	2	3	4	5	6
E	6	18	15	12	10	8	6	**4**
L	5	15	12	9	7	5	**4**	7
P	4	12	9	6	4	**4**	7	10
I	3	9	6	3	**3**	6	9	12
R	2	6	3	**2**	5	8	11	14
T	1	3	**1**	4	7	10	13	16
–	0	**0**	3	6	9	12	15	18

- Gaps are now three times as expensive as mismatches, so the best alignment uses none: the path is the straight diagonal.
- **OPT(6, 6) = 4** – four substitutions – beating the 7 that the gapped alignment would cost.

```
RIDDLE
× × × × | |
TRIPLE
R R R R M M
```

A bigger example: gap = 3, mismatch = 1

C	9	27											
A	8	24											
G	7	21											
T	6	18											
T	5	15											
G	4	12											
C	3	9											
A	2	6											
A	1	3											
–	0	0	3	6	9	12	15	18	21	24	27	30	33	36
		–	**A**	**A**	**G**	**G**	**T**	**A**	**T**	**G**	**A**	**A**	**T**	**C**
		0	1	2	3	4	5	6	7	8	9	10	11	12

- *a* = AAGGTATGAATC (m = 12)
- *b* = AACGTTGAC (n = 9)
- gap = 3, so the base row and column count up in threes.
- 108 cells to fill by the recurrence (plus 22 base cases) – against 14,218,905 possible alignments.

A bigger example: the filled table

		–	A	A	G	G	T	A	T	G	A	A	T	C
		0	1	2	3	4	5	6	7	8	9	10	11	12
C	9	27	24	21	18	15	13	10	8	7	5	5	8	10
A	8	24	21	18	15	12	10	7	6	4	4	7	10	13
G	7	21	18	15	12	9	7	5	3	4	7	10	13	16
T	6	18	15	12	9	7	4	2	4	7	10	13	16	19
T	5	15	12	9	6	4	1	4	7	10	13	16	19	22
G	4	12	9	6	3	1	4	7	10	13	16	19	22	25
C	3	9	6	3	1	4	7	10	13	16	19	22	25	27
A	2	6	3	0	3	6	9	12	15	18	21	24	27	30
A	1	3	0	3	6	9	12	15	18	21	24	27	30	33
–	0	0	3	6	9	12	15	18	21	24	27	30	33	36

- The answer: **OPT(12, 9) = 10**.
- Check one cell yourself: (3, 3), `G` vs `C` – `min(1 + 0, 3 + 3, 3 + 3) = 1`
- *Matches (cost 0) pull the cheap values along diagonals; the far corners, where one string has run out, are expensive.*

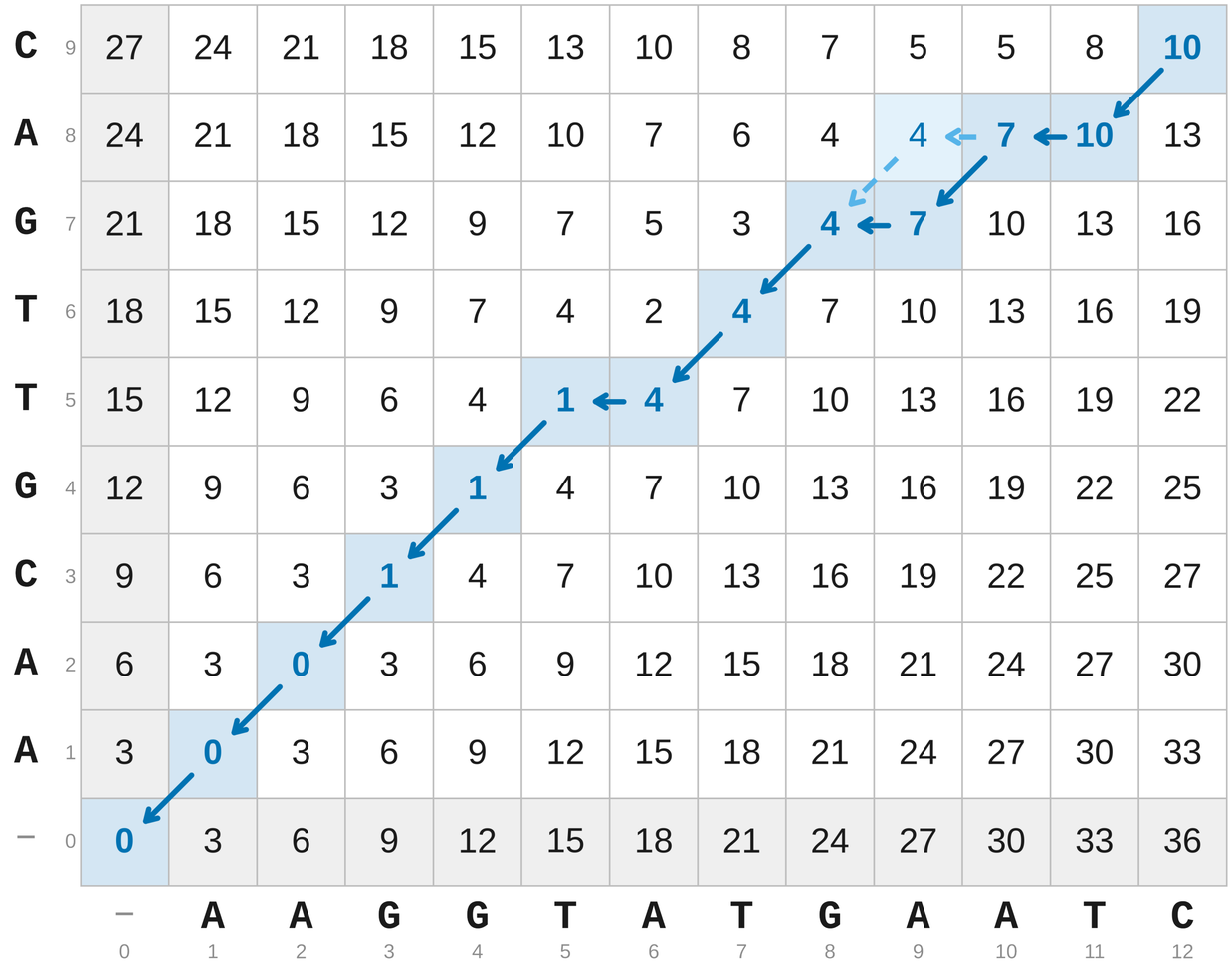
A bigger example: the traceback

		–	A	A	G	G	T	A	T	G	A	A	T	C
		0	1	2	3	4	5	6	7	8	9	10	11	12
C	9	27	24	21	18	15	13	10	8	7	5	5	8	**10**
A	8	24	21	18	15	12	10	7	6	4	4	**7**	**10**	13
G	7	21	18	15	12	9	7	5	3	**4**	**7**	10	13	16
T	6	18	15	12	9	7	4	2	**4**	7	10	13	16	19
T	5	15	12	9	6	4	**1**	**4**	7	10	13	16	19	22
G	4	12	9	6	3	**1**	4	7	10	13	16	19	22	25
C	3	9	6	3	**1**	4	7	10	13	16	19	22	25	27
A	2	6	3	**0**	3	6	9	12	15	18	21	24	27	30
A	1	3	**0**	3	6	9	12	15	18	21	24	27	30	33
–	0	**0**	3	6	9	12	15	18	21	24	27	30	33	36

```
AAGGTATGAATC
|| x || || | |
AACGT-TG-A-C
MMRMMDMMDMDM
0010030030300
```

- 1 mismatch + 3 gaps = 1 + 9 = **10**, the corner value.
- Note what the costs did: with gaps three times dearer than mismatches, the alignment uses a gap only where the strings' lengths force one – 12 – 9 = 3.

A bigger example: two optimal alignments



		–	A	A	G	G	T	A	T	G	A	A	T	C
		0	1	2	3	4	5	6	7	8	9	10	11	12
C	9	27	24	21	18	15	13	10	8	7	5	5	8	10
A	8	24	21	18	15	12	10	7	6	4	4	7	10	13
G	7	21	18	15	12	9	7	5	3	4	7	10	13	16
T	6	18	15	12	9	7	4	2	4	7	10	13	16	19
T	5	15	12	9	6	4	1	4	7	10	13	16	19	22
G	4	12	9	6	3	1	4	7	10	13	16	19	22	25
C	3	9	6	3	1	4	7	10	13	16	19	22	25	27
A	2	6	3	0	3	6	9	12	15	18	21	24	27	30
A	1	3	0	3	6	9	12	15	18	21	24	27	30	33
–	0	0	3	6	9	12	15	18	21	24	27	30	33	36

(a) Solid path

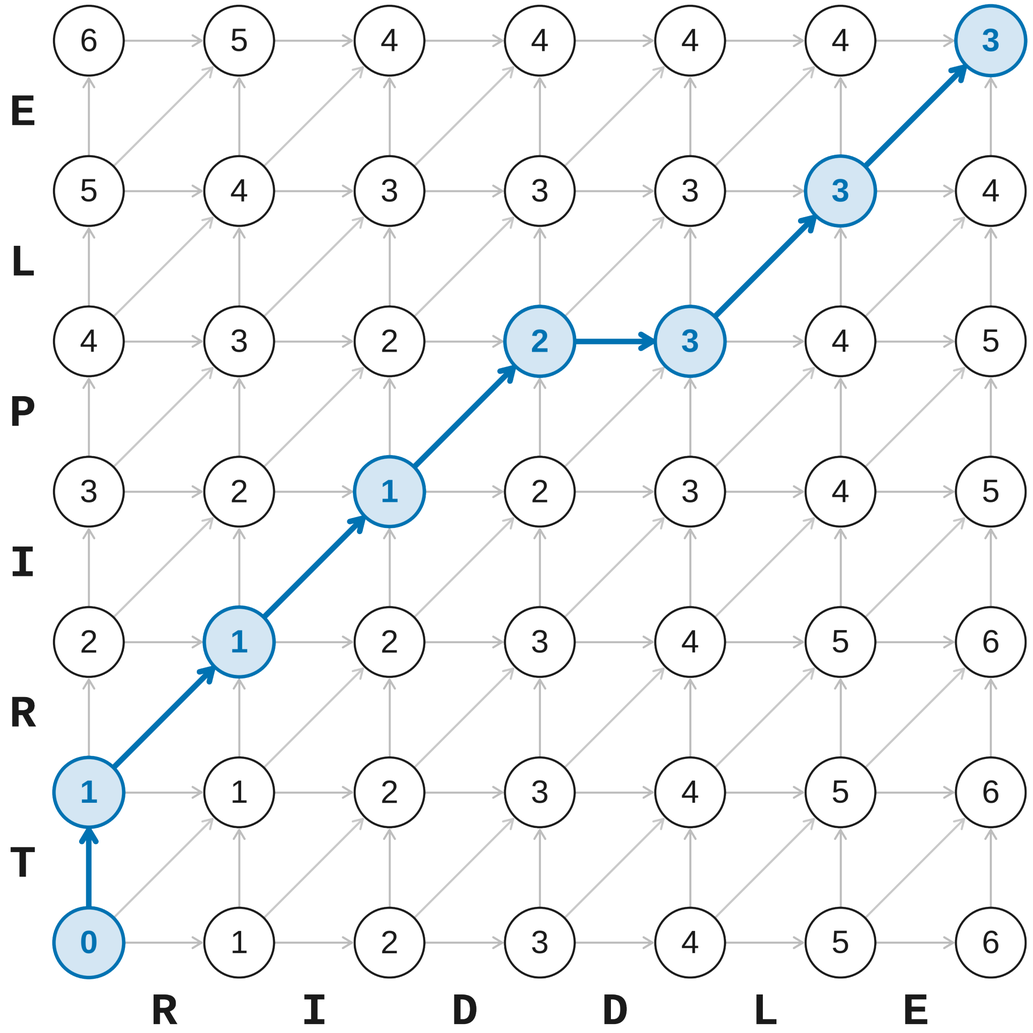
```
A A G G T A T G A A T C
| | × | |   | |   |   |
A A C G T - T G - A - C
```

(b) Dashed path

```
A A G G T A T G A A T C
| | × | |   | | |     |
A A C G T - T G A - - C
```

- At (10, 8) both the diagonal and “from the left” give the minimum. Both alignments cost 10: same score, gaps in different places.
- *Real aligners need a tie-breaking rule — and ties are common.*

Another view: the edit graph



- One node per table cell; an edge for each move: → and ↑ cost **gap**, ↗ costs **cost**(a_i, b_j), which is 0 for a match. This path uses all four match diagonals: `R/R` `I/I` `L/L` `E/E`.
- An alignment is a path from (0, 0) to (m, n). Its cost is the path's length.
- So the best alignment is a **shortest path** – and the number in each node is its distance from (0, 0).

The table is the DAG algorithm

Shortest path in a DAG

d(X) = min over edges (Y, X) of

d(Y) + ℓ(Y, X)

fill in topological order

Alignment

OPT(i, j) = min over its 3 in-edges of

OPT(pred) + edge cost

fill column by column

- The edit graph is a DAG: every edge goes right, up, or up-right, so no path can return.
- Column-by-column (or row-by-row, or anti-diagonal) order is a **topological order** of it – exactly the condition the DAG algorithm needs.
- So the alignment DP is the DAG shortest-path DP, run on a graph with (m+1)(n+1) nodes and ≈ 3mn edges: O(mn), as before.

Recap: when dynamic programming applies

requirement	Fibonacci	DAG shortest path	alignment
the answer is built from answers to smaller subproblems	$F(n-1)$, $F(n-2)$	$d(Y)$ for edges $Y \to X$	OPT of 3 neighbours
only polynomially many subproblems	n	$\|V\|$	$(m+1)(n+1)$
a natural order: smaller ones first	1, 2, …, n	topological	column by column

Main idea: solve the subproblems in an order such that whenever you need an answer, it is already in the table.

Where this goes

- **Local alignment** (Smith–Waterman): the best-matching pair of substrings — one small change to today's recurrence.
- **Affine gaps**: one long gap is more likely than many short ones, so charge open + extend.
- **Banded alignment**: if at most k edits are allowed, only cells within k of the diagonal can matter — O(kn), the version read aligners extend seeds with.
- **Linear space** (Hirschberg): the alignment, not just its cost, in O(m + n) memory.

Next: local alignment and its variants.