

Dynamic programming notes.

One pattern at a time.

Dynamic programming notes by Aditya Verma. Work through examples, test your reasoning with quizzes, and use flashcards to make it stick.

13 sections · 26 quiz questions · 39 flashcards

Inside this notebook

- 01. Start with the state
- 02. Linear DP: grow one prefix
- 03. One target, many previous choices
- 04. Grid DP: arrive from a neighbor
- 05. Two strings: compare two prefixes
- 06. Subset and knapsack: take or leave an item
- 07. Interval DP: solve shorter spans first
- 08. Layered DP: keep the phase in the state
- 09. Bitmask DP: remember an exact small set
- 10. Digit DP: build a bounded number
- 11. Tree DP: combine child subtrees
- 12. Largest X bordered square: certify all four edges
- 13. Count of AP Subsequences: extend a matching difference

Each section includes an explanation, an example, practice questions, flashcards, and a quiz answer key. Cover the flashcard answers before recalling them. Check quiz answers after attempting both questions.

Includes 78 worked problems from the 78-problem catalogue. Compiled October 2026.

Start with the state

Dynamic programming reuses answers to repeated subproblems. Before writing a table, describe exactly what one stored answer means. A state must retain everything that can affect the choices still available.

The idea to remember

- Ask four questions: What does a state mean? Which smaller states determine it? What are the base cases? In what order can dependencies be solved?
- Write $dp[i]$ as a sentence before writing a formula. For stairs, $dp[i]$ counts ordered routes to step i using jumps of 1 or 2.
- A recurrence combines smaller answers. For stairs, remove the last jump: $dp[i] = dp[i - 1] + dp[i - 2]$. These two groups cannot overlap because their last jumps differ.
- A base case is an answer known without the recurrence. $dp[0] = 1$ counts the empty route; $dp[1] = 1$ counts the single +1 route.
- Memoization starts at the question, recursively solves needed states, and caches each answer. Tabulation starts at the bases and fills states in dependency order. Both can use the same recurrence.
- Estimate time as the number of states times the work per state. Reduce memory only after identifying which old states future calculations still need. A zero answer is valid, so use a separate marker for an uncomputed state.

Work through one example

Reach step 4 with jumps of 1 or 2.

Start with $dp[0] = 1$ and $dp[1] = 1$. Then $dp[2] = 1 + 1 = 2$, $dp[3] = 2 + 1 = 3$, and $dp[4] = 3 + 2 = 5$.

5 routes

Check your understanding

1. If $dp[0]$ were 0 but $dp[1]$ stayed 1, what would the stairs recurrence incorrectly give for $dp[2]$?

- A. 0
- B. 1
- C. 2
- D. 3

2. Tabulation already knows $dp[2] = 2$ and $dp[3] = 3$. How many routes reach step 4?

- A. 4
- B. 5
- C. 6
- D. 8

Flashcards: cover the answer to recall it

What four questions should come before a DP implementation?

Define the state, derive its dependencies, identify base cases, and choose an order that solves every dependency first.

How do memoization and tabulation differ?

Memoization discovers needed states through cached recursion. Tabulation explicitly visits states from known bases in dependency order.

How do you estimate DP time and space?

Time is states times transition work per state. Space includes stored states and, for recursion, the call stack. Keep every state a future transition still needs.

Quiz answer key

1. B. It would give $1 + 0 = 1$ and lose the direct $+2$ route. The empty route at step 0 is what that final jump extends.

2. B. A final $+1$ extends 3 routes to step 3. A final $+2$ extends 2 routes to step 2. Add the disjoint groups: $3 + 2 = 5$.

Linear DP: grow one prefix

Use a position or prefix length when the remaining decision depends on a small amount of earlier work. Counting stairs, choosing nonadjacent houses, and prefix optimizations share this shape, but their state meanings and combining operations differ.

The idea to remember

- For house values a , let $dp[i]$ mean the largest sum obtainable from the first i houses without taking adjacent houses. Here i is a count, so the newest value is $a[i - 1]$.
- Skip the newest house to keep $dp[i - 1]$, or take it and add $a[i - 1]$ to $dp[i - 2]$. Choose the maximum: $dp[i] = \max(dp[i - 1], dp[i - 2] + a[i - 1])$.
- For nonnegative values, $dp[0] = 0$ and $dp[1] = a[0]$. Fill i from 2 upward because both dependencies use smaller prefixes.
- Counting alternatives uses addition; optimizing alternatives uses min or max. Do not copy the combining operation from a different problem.
- Only the previous two prefix answers are needed for the next answer, so two variables can replace the table. Preserve both old values until the update is finished.

Work through one example

House values $[2, 7, 9, 3]$.

The prefix answers start 0, 2. Then $\max(2, 0 + 7) = 7$; $\max(7, 2 + 9) = 11$; $\max(11, 7 + 3) = 11$.

11 maximum value

Worked problem notes (23)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Fibonacci Numbers

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the i th Fibonacci number: the value of the sequence at position i , and nothing more. One number names the state, so the table is one row.

THE FINAL DECISION

Ask what the last term added. Position i is defined as the sum of the two positions before it, so there is exactly one way to reach it and both predecessors contribute. Two buckets, added rather than compared, because there is no choice to optimize.

RECURRENCE

$$dp[i] = dp[i - 1] + dp[i - 2]$$

BASE CASES AND ORDER

$dp[0] = 0$ and $dp[1] = 1$ are given by definition; they cannot be derived. Every other cell depends only on smaller indices, so filling left to right guarantees both dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Indexing off by one. Plenty of sources start the sequence 1, 1 rather than 0, 1, so $dp[7]$ is 13 under one convention and 21 under the other. Fix the two base cases first and state them out loud before writing the loop.

► Tribonacci Numbers

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the i th Tribonacci number: the value of the sequence at position i , and nothing more. One number still names the state, so the table is still one row.

THE FINAL DECISION

Ask what the last term added, same as Fibonacci. Position i is defined as the sum of the three positions before it, so there is exactly one way to reach it and all three predecessors contribute. Three buckets, added, and the extra bucket is the only thing that changed.

RECURRENCE

$$dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]$$

BASE CASES AND ORDER

$dp[0] = 0$, $dp[1] = 1$, and $dp[2] = 1$ are given by definition; a third base case is needed because the recurrence now reaches three cells back. Every other cell depends only on smaller indices, so filling left to right still guarantees all three dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

The third base case. Some sources start Tribonacci 0, 0, 1 rather than 0, 1, 1, which shifts every later term. Pin down all three seed values from the problem statement before writing the loop; one wrong seed silently poisons the whole table.

► Lucas Numbers

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the i th Lucas number: the value of this Fibonacci-like sequence at position i , and nothing more. One number names the state, so the table is one row, same as Fibonacci.

THE FINAL DECISION

Ask what the last term added, the same question Fibonacci asks. Position i is defined as the sum of the two positions before it, so there is exactly one way to reach it and both predecessors contribute. The recurrence is identical to Fibonacci's; only the two seeds that start it differ, which is why every Lucas number can also be written as a sum of two Fibonacci numbers.

RECURRENCE

$$dp[i] = dp[i - 1] + dp[i - 2]$$

BASE CASES AND ORDER

$dp[0] = 2$ and $dp[1] = 1$ are given by definition, and they are the only difference from Fibonacci; they cannot be derived. Every other cell depends only on smaller indices, so filling left to right guarantees both dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Reusing Fibonacci's seeds by habit. Starting this recurrence at 0, 1 instead of 2, 1 produces the Fibonacci sequence itself, term for term, since the two sequences share a recurrence but not a starting point. The seeds are the only thing this problem asks you to get right.

► Climbing Stairs

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of distinct ways to reach stair i : the count of routes that land exactly there, and nothing more. One number names the state, so the table is one row.

THE FINAL DECISION

The full derivation (why the last move splits every route into exactly two cases) lives in the flagship lesson at </courses/dynamic-programming/climbing-stairs>. This page assumes it and goes straight to the table.

RECURRENCE

$$dp[i] = dp[i - 1] + dp[i - 2]$$

BASE CASES AND ORDER

$dp[0] = 1$ and $dp[1] = 1$ are given; see the flagship lesson for why standing still counts as one way. Every other cell depends only on smaller indices, so filling left to right guarantees both dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Setting $dp[0] = 0$ because standing on the ground feels like doing nothing. But $dp[0]$ means "one way to be already at the top with zero steps left", one way to stand still, and without it $dp[2]$ comes out wrong: $1 + 1 = 2$ only balances if $dp[0]$ contributes.

► Climbing Stairs with 3 Moves

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of distinct ways to reach stair i under this three-move rule: the count of routes that land exactly there, and nothing more. One number still names the state, so the table is still one row.

THE FINAL DECISION

Ask what the last move could have been, the same question the flagship lesson asks. A move of 1, 2, or 3 steps is now legal, so the last move into stair i came from exactly one of three places: stair $i-1$, $i-2$, or $i-3$. Three buckets, added rather than compared, because there is no choice to optimize, only routes to count.

RECURRENCE

$$dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]$$

BASE CASES AND ORDER

$dp[0] = dp[1] = 1$ and $dp[2] = 2$ are given directly; a third base case is needed because the recurrence now reaches three cells back, and $dp[2]$ would be wrong if it were derived from a two-term rule instead of stated outright (see the pitfall). Every other cell depends only on smaller indices, so filling left to right still guarantees all three dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Treating $dp[2]$ as derivable from $dp[1] + dp[0]$, the way the flagship two-move staircase would. Under three moves $dp[2]$ is still 2 (a single move of 2, or two moves of 1), but the three-term recurrence does not apply until $i = 3$; stating $dp[0]$, $dp[1]$, and $dp[2]$ outright avoids folding a rule into a cell it was never meant to reach.

► Weighted Climbing Stairs

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the cheapest total cost paid to land on stair i , where landing on a stair, including stair i itself, charges its own cost the instant you arrive: one running total, and nothing about which stairs came before it.

THE FINAL DECISION

Ask what the last move into stair i could have been: a step of 1 from stair $i - 1$, or a step of 2 from stair $i - 2$. Either way you pay $cost[i - 1]$ the instant you land on stair i , on top of whichever predecessor already cost less, so the recurrence takes a min over the two arrival routes and then adds that one landing cost once.

RECURRENCE

$$dp[i] = \min(dp[i - 1], dp[i - 2]) + cost[i - 1]$$

BASE CASES AND ORDER

$dp[0] = 0$ and $dp[1] = cost[0]$ are given directly, and they pin the cost convention this table uses: cost is charged for LANDING on a stair, never for leaving one. Standing on the ground before any step is free, since nothing has been landed on yet, while $dp[1]$ already carries $cost[0]$, the price of arriving at stair 1. Every other cell depends only on smaller indices, so filling left to right guarantees both predecessors already hold a real value before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Assuming cost is paid on leaving a stair instead of on landing on it. Sources disagree here, and the two conventions give different totals from the same cost array: under a leaving-cost reading, the top stair's own cost would never be charged, since there is nothing left to leave from, while under this table's landing-cost reading $dp[0]$ carries no charge even though every stair after it does, because nothing was ever landed on at the ground floor. This table pays on landing: $dp[0] = 0$, and every $dp[i]$ for $i \geq 1$ includes $cost[i - 1]$. Compare against a source using the other convention and every value beyond $dp[1]$ will look shifted.

► Maximum Segments

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the maximum number of segments a length of i can be split into using only the allowed lengths, or a marker that no split exists at all: a count of pieces, and nothing about which lengths were chosen. That marker is reported as -1 once the answer reaches the outside world, but inside the table every unreachable cell carries its own version of it, not just the final one: a length that cannot be built at all needs to look impossible everywhere it shows up, not only when it happens to be the one being asked about, since a lone -1 sitting among ordinary segment counts could pass for a tiny real count instead of the warning it is meant to be.

THE FINAL DECISION

Ask what the very last segment cut off could have been. It has to be one of the three allowed lengths, so it leaves a remainder of i minus that length behind; whichever remainder already holds the most segments should be the one to extend, since the final cut always contributes exactly one more segment on top. That is a max over candidates, not a sum, because only one of the three final cuts actually happened.

RECURRENCE

```
dp[i] = 1 + max(dp[i - a], dp[i - b], dp[i - c]) over whichever of those
three predecessors is a valid, non-negative index
```

BASE CASES AND ORDER

$dp[0] = 0$ is the only base: an empty remainder needs no segments at all, and it is a genuine answer, not a placeholder. Every cell from $i = 1$ up depends only on smaller indices ($i - a$, $i - b$, $i - c$, whichever are non-negative), so filling left to right guarantees each candidate already holds a real value, whether that value is a segment count or the impossible marker, before it is read. Lengths smaller than every allowed length ($i = 1$, $i = 2$ here) have no candidate at all: no length fits inside them, so they are impossible before the recurrence even runs.

Time: $O(n)$

Space: $O(n)$, reducible to a rolling window covering the largest allowed length

Watch out

Treating an unreachable length as zero segments instead of impossible. Lengths 1, 2, and 4 cannot be built from 3, 5, and 7 at all, so $dp[1]$, $dp[2]$, and $dp[4]$ are not zero, they are undefined states that must propagate as impossible so no later cell mistakes them for a valid, empty split. Folding an impossible predecessor to 0 would let a later cell wrongly count a phantom extra segment through a remainder that was never actually reachable.

► Minimum Perfect Squares

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the minimum number of perfect squares that sum to exactly i : a count of squares, and nothing about which squares were chosen. One number names the state, so the table is one row.

THE FINAL DECISION

Ask what the last square added could have been. It is some $j*j$ no larger than i , leaving a remainder of $i - j*j$ behind, and whichever remainder already takes the fewest squares should be the one to extend, since the final square always adds exactly one more to the count. That is a min over candidates, not a sum, because only one final square actually got added.

RECURRENCE

```
dp[i] = 1 + min over every j with j*j <= i of dp[i - j*j]
```

BASE CASES AND ORDER

$dp[0] = 0$ is the only base: making zero takes no squares at all, and it is a real answer, not a placeholder. Every cell from $i = 1$ up depends only on smaller indices ($i - j*j$ for every j with $j*j \leq i$), so filling left to right guarantees every candidate already holds a real value before it is read. At $i = 1$ there is only one candidate, $j = 1$, so this cell is filled by a genuine minimum with nothing else to compare it against.

Time: $O(n \sqrt{n})$

Space: $O(n)$

Watch out

Assuming a greedy, largest-square-first strategy works. It does not: 12 is $4 + 4 + 4$, three squares, not $9 + 1 + 1 + 1$, four squares, even though 9 is the largest square not exceeding 12. Every candidate remainder has to be compared through the full table; there is no shortcut that skips checking the smaller squares too.

► House Robber

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the most money that can be collected robbing only among houses 0 through i , with no two adjacent houses both chosen: one running total, and nothing about which specific houses were picked.

THE FINAL DECISION

The full derivation, why the last house is either skipped or robbed with $nums[i]$ added to $dp[i - 2]$, lives in the flagship lesson at </courses/dynamic-programming/house-robber>. This page assumes it and goes straight to the table.

RECURRENCE

```
dp[i] = max(dp[i - 1], nums[i] + dp[i - 2])
```

BASE CASES AND ORDER

$dp[0] = nums[0]$ and $dp[1] = \max(nums[0], nums[1])$ are given; see the flagship lesson for why a single house is always taken and two adjacent houses reduce to picking the richer one. Every other cell depends only on smaller indices, so filling left to right guarantees both predecessors already hold a real value before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Assuming a fixed alternate-houses pattern, rob every other house starting from house 0 or house 1, always wins. It does not: among the first six houses of this street (2, 7, 9, 3, 1, 8), robbing every even house gives $2 + 9 + 1 = 12$ and robbing every odd house gives $7 + 3 + 8 = 18$, but the true optimum is 19, robbing houses 0, 2, and 5 for $2 + 9 + 8$, a gap pattern no fixed parity would ever try.

► Decode Ways

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of ways to decode the first i digits of the string into letters: a count of valid splits, and nothing about which split. One number names the state, so the table is one row.

THE FINAL DECISION

Ask what the last decoded letter could have been. It is either the single digit at position i , valid whenever that digit is not '0', or the two digits ending at position i read together, valid whenever they form a number from 10 through 26. Both readings can be legal at once, in which case both prefixes contribute; either can fail on its own, in which case only the other contributes. The chosen string always leaves at least one reading legal at every position, so no cell is ever left with nothing to read at all.

RECURRENCE

$dp[i] = dp[i - 1]$ (if the single digit at i is valid) plus $dp[i - 2]$ (if the two digits ending at i are valid)

BASE CASES AND ORDER

$dp[0] = 1$: an empty prefix has exactly one, vacuous, decoding. Every cell from $i = 1$ on depends only on $dp[i - 1]$ and/or $dp[i - 2]$, whichever readings are valid there, so filling left to right guarantees any dependency a cell needs already holds a real value before it is read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Treating a lone '0' as a normal digit that can stand on its own. No letter maps to '0', so wherever the digit itself is '0' (position 5 in this string), the single-digit reading is invalid and only the two-digit pair ending in that '0' can consume it. Skipping this check lets $dp[5]$ wrongly add a phantom single-digit decoding of a digit with no letter at all.

► Coin Change – Minimum Coins to Make Sum

STATE: WHAT ONE ENTRY MEANS

$dp[a]$ is the minimum number of coins from $\{1, 3, 4\}$ that sum to exactly a : a count of coins, and nothing about which coins were chosen. One number names the state, so the table is one row.

THE FINAL DECISION

The full derivation, why the last coin spent leaves behind a remainder whose own fewest-coins count is already known, lives in the flagship lesson at </courses/dynamic-programming/coin-change>. This page assumes it and goes straight to the table.

RECURRENCE

$dp[a] = 1 + \min \text{ over every coin } c \leq a \text{ of } dp[a - c]$

BASE CASES AND ORDER

$dp[0] = 0$ is given; see the flagship lesson for why making zero costs no coins. Every cell from $a = 1$ up depends only on smaller amounts ($a - c$ for every coin $c \leq a$), so filling left to right guarantees every candidate already holds a real value before it is read. Unlike maximum-segments, nothing here is ever unreachable: coin 1 is always available, so every amount can always be padded up to, only more or less cheaply.

Time: $O(\text{amount} \times \text{coins.length})$

Space: $O(\text{amount})$

Watch out

Assuming the largest coin that still fits should always be spent first. It should not: with coins $\{1, 3, 4\}$, making 6 by greedily spending a 4 first leaves 2, which then costs two more 1s, for $4 + 1 + 1$, three coins total, while the true optimum is $3 + 3$, two coins. Every candidate remainder has to be compared through the full table; there is no shortcut that skips checking the smaller coins too.

► Painting Fence Algorithm

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of ways to paint the first i fence posts with k colors so that no three consecutive posts share a color: a count of valid prefixes, and nothing about which specific colors were used. One number names the state, so the table is one row.

THE FINAL DECISION

Ask what the last two posts could look like, not just the last one. Every valid painting of the first i posts either repeats post $i-1$'s color once more, stretching a run to exactly two (there are $dp[i-1]$ valid prefixes to extend this way, each extendable in $k-1$ further colors so the run never reaches three), or changes color entirely at post i (there are $dp[i-2]$ valid prefixes of length $i-2$ to extend, again in $k-1$ ways, regardless of what closed each one off). Both cases scale by the same $k-1$, so they factor out as one multiplication over a sum rather than two separate additions.

RECURRENCE

$$dp[i] = (k - 1) \times (dp[i - 1] + dp[i - 2])$$

BASE CASES AND ORDER

$dp[0] = 1$ is the vacuous case: an empty prefix has exactly one arrangement, and it is a real value, not a placeholder. But unlike $dp[1]$ and $dp[2]$, $dp[0]$ is never read by any later cell: with three bases in place the first compute cell is $i = 3$, and it reads only $dp[2]$ and $dp[1]$. $dp[0]$ fills the table's leftmost slot because every index needs a value, not because the recurrence reaches back that far. $dp[1] = k$ and $dp[2] = k^2$ earn their place for a genuinely combinatorial reason instead: the general transition only starts holding once a run of three is even possible, and at $i = 2$ every pair of colors is still allowed, so $dp[2]$ is the full k^2 rather than the $(k-1)$ -scaled value the formula would wrongly give (for $k = 3$, that would be $(k-1)(dp[1]+dp[0]) = 2 \times 4 = 8$, not the true 9). Every cell from $i = 3$ on depends only on smaller indices, so filling left to right guarantees both real dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Reading the constraint as 'no two consecutive posts share a color' instead of the real rule: only three consecutive posts of the same color are forbidden, so up to two in a row are fine. Post i is allowed to match post $i-1$'s color; it just cannot also match post $i-2$'s, or a run of three forms. Solving the stricter no-two-alike version gives a different, smaller count from post 3 onward.

► Jump Game

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the fewest jumps needed to land on index i , starting from index 0. It is a shortest path length measured in jumps, not a distance travelled, and it is defined only for indices you can actually reach; the rest are genuinely unreachable rather than free.

THE FINAL DECISION

Ask where the jump that landed on i came from. It came from some earlier index j , and the jump was legal only if j 's own reach, $j + \text{nums}[j]$, stretches to i or beyond. Whatever that j was, getting to it optimally is the same problem on a shorter prefix, so the cost is $dp[j]$ plus the one jump you just took. Take the cheapest j among the legal ones. If no j reaches i at all, nothing lands there, ever.

RECURRENCE

$$dp[i] = 1 + \min_{\text{over } j < i \text{ with } j + \text{nums}[j] \geq i} dp[j]$$

BASE CASES AND ORDER

$dp[0] = 0$: you start on index 0, so it takes no jumps to be there. Fill left to right, because every j the recurrence reads is strictly smaller than i . Some cells have no legal j whatsoever, and those are unreachable, written as an infinity rather than a zero. In this array index 9 holds a 0, a dead end, and nothing before it reaches past index 9, so indices 10 and beyond can never be landed on no matter how large the table grows.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Treating an unreachable index as costing zero jumps. An empty minimum is not 0, it is infinity, and if you seed the cell with 0 the whole table downstream reads as reachable in one cheap hop. The second trap is reading $\text{nums}[i]$ as an exact jump length rather than a maximum reach. Every index strictly between j and $j + \text{nums}[j]$ is a legal landing spot, which is precisely why one large value can flatten several cells to the same count.

► Max A's using Special Keyboard

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the largest number of A's reachable using exactly i key presses in total, counting every press of every kind: one number for a budget, and nothing about which keys produced it.

THE FINAL DECISION

Ask where the last stretch of typing ended. Say the screen last grew by typing at press j . Everything after that is one select-all, one copy, and then nothing but pastes, because once you have copied there is no reason to type again: a paste is worth at least as much as a keystroke and usually far more. That leaves $i - j$ presses, two of which are select-all and copy, so $i - j - 2$ are pastes. Each paste adds another copy of the buffer to a screen that already holds one, so the screen ends up multiplied by $i - j - 1$. Every choice of breakpoint j gives one candidate, and the best of them wins.

RECURRENCE

$$dp[i] = \max \text{ over } j \text{ of } dp[j] * (i - j - 1)$$

BASE CASES AND ORDER

Everything up to six presses is a base case, and for a reason worth seeing rather than memorising: a copy-and-paste cycle costs three presses before it pays anything back, so on a small budget plain typing simply wins. $dp[i] = i$ for i up to 6, and only from seven presses on does a breakpoint beat typing. Each computed cell reads a whole range of earlier cells, from $dp[1]$ up to $dp[i-3]$, so filling left to right guarantees all of them exist. Note the table's first cells are read constantly by later ones, unlike most tables here where each cell looks back only a step or two.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Assuming more copy-and-paste cycles always beat typing, and starting them as early as possible. A cycle costs three presses (select-all, copy, one paste) before it has gained anything at all, so on a short budget it loses outright, and even on a long one the best breakpoint is later than instinct suggests. The opposite mistake is just as common: assuming one cycle is always enough. It is not, for large n , and the recurrence quietly handles that because $dp[j]$ may itself have been built by an earlier cycle.

► Longest Increasing Subsequence

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the length of the longest strictly increasing subsequence that ends AT index i specifically, using $nums[i]$ as its last chosen value: one count, tied to ending here, and nothing about which earlier elements were chosen.

THE FINAL DECISION

Ask what came immediately before index i in the best subsequence that ends there. Either nothing does, so index i starts a chain of its own, or some earlier index j with a strictly smaller value is the previous chosen element, in which case $dp[i]$ extends whichever qualifying j already holds the longest chain. Every qualifying j is a candidate to extend, compared rather than summed, since only one of them actually precedes index i in the chain that ends here.

RECURRENCE

$dp[i] = 1 + \max \text{ over every } j < i \text{ with } nums[j] < nums[i] \text{ of } dp[j], \text{ or } 1 \text{ if no such } j \text{ exists}$

BASE CASES AND ORDER

$dp[0] = 1$ is the only declared base: with no index before 0, the first element is trivially a subsequence of length 1 all by itself, and it turns out to be read often, since most later values in this array are larger than $nums[0] = 5$. Every other index is a genuine compute cell, even the ones with no qualifying predecessor at all: $dp[1]$ is exactly that case, since $nums[0] = 5$ is not smaller than $nums[1] = 2$, so $dp[1]$ resolves to 1 on the strength of the same plus-one for standing alone, with no `cell_read` emitted, because there is nothing smaller before it to consult. Every other cell depends only on smaller indices, so filling left to right guarantees every predecessor that does qualify already holds a real value before it is read.

Because the answer is whichever cell ends up largest, not necessarily the last one, the table also has to track that `argmax` as it fills, rather than simply reading off $dp[8]$ at the end.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Reading the last cell, $dp[n - 1]$, as the answer. The longest increasing subsequence can end anywhere in the array, so here it peaks at $dp[6] = 4$ while $dp[8]$ only reaches 3; the real answer is the maximum over the whole table. A second, unrelated mistake is assuming the subsequence must be contiguous: it only has to preserve relative order, so 2, 3, 6, 9 counts even though 8 and 6 sit in between 2 and 3 in the array.

► Largest Divisible Subset

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the size of the largest divisible chain that ends AT index i specifically, using $nums[i]$ as its largest member: one count, tied to ending here, and nothing about which earlier numbers were chosen.

THE FINAL DECISION

Ask what the next-smaller number in the chain that ends at index i must have been. Since the array is sorted ascending, any earlier number that divides $nums[i]$ evenly is a legal link one step down, so $dp[i]$ extends whichever qualifying earlier index already holds the longest such chain. Checking only that $nums[i]$ divides by a single earlier member is enough, because a chain built this way is automatically consistent all the way down: if a divides b and b divides c , then a divides c too, so the transition never has to re-check any link further back than the one directly below it.

RECURRENCE

$dp[i] = 1 + \max \text{ over every } j < i \text{ with } nums[i] \% nums[j] == 0 \text{ of } dp[j], \text{ or } 1 \text{ if no such } j \text{ exists}$

BASE CASES AND ORDER

$dp[0] = 1$ is the only declared base: the smallest number is trivially a divisible chain of one all by itself, and because sorting places 1 there for this fixed array, it also ends up divisible by every later number, so it is read constantly ($nums[0] = 1$ divides everything). That happens to mean every later index always has at least one qualifying predecessor here, $dp[0]$ itself, so this particular input never produces a zero-dependency compute cell beyond index 0; a differently chosen array could still have one, since the shape depends entirely on whether some earlier number divides the current one. Every other cell depends only on smaller indices, so filling left to right guarantees every candidate that does qualify already holds a real value before it is read. Because the answer is whichever cell ends up largest, not necessarily the last one, the table also has to track that argmax as it fills.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Forgetting that sorting the array first is what makes a single divisibility check enough. Without sorting, checking only $nums[i] \% nums[j] == 0$ for $j < i$ would miss chains where a smaller-valued divisor happens to sit later in the input order; sorting ascending guarantees every legal chain reads as a run of increasing values from left to right, so the recurrence never needs to look both directions.

► Weighted Job Scheduling

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the most profit obtainable from the first $i + 1$ jobs once they are sorted by end time, whether or not job i itself is taken: a running best over a prefix, not the profit of a schedule that must include job i .

THE FINAL DECISION

Ask what happened to the last job in the sorted prefix. Either you did not take it, in which case the best is whatever the previous prefix already achieved, or you did, in which case you collect its profit and add the best schedule that finishes before it starts. Those two cases cover every possibility with no overlap, so a max over them is the answer. The second case is the interesting one: $p(i)$ is the latest job whose end time is at or before job i 's start, which is almost never $i - 2$ and sometimes does not exist at all.

RECURRENCE

$$dp[i] = \max(dp[i - 1], \text{profit}[i] + dp[p(i)])$$

BASE CASES AND ORDER

The jobs must be sorted by END time before anything else happens, and that sort is what makes the recurrence valid: it guarantees every job compatible with job i sits somewhere to its left, so $p(i)$ is always an index already filled. $dp[0]$ is the first job's profit, since with one job the only choice is to take it. Filling left to right then works because both $dp[i - 1]$ and $dp[p(i)]$ are smaller indices.

Time: $O(n^2)$ as shown, $O(n \log n)$ with binary search

Space: $O(n)$

Watch out

Sorting by start time, or by profit, instead of by end time. Sorting by profit is the greedy instinct and it fails immediately: one fat job spanning the whole day beats nothing else, but two thinner jobs beside each other often beat it. Sorting by start time is subtler and looks fine until you notice it gives no guarantee that a compatible predecessor sits to the left, which is the single property the whole recurrence rests on.

► Word Break Problem

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is a yes-or-no flag, rendered here as 1 or 0: can the first i characters be split entirely into dictionary words? It is not a count of splits and not a length. The cell answers one question about a prefix, and the only thing later cells ever need from it is that answer.

THE FINAL DECISION

Ask where the last word of the split begins. If the first i characters split legally, some word ends exactly at i , and that word starts at some j . Then $s[j..i)$ has to be in the dictionary and everything before j has to split legally on its own, which is $dp[j]$. Try every j whose suffix is a dictionary word and take an or over them. Only j whose $s[j..i)$ is actually a word are worth reading at all, and for many i there are none.

RECURRENCE

$dp[i] = \text{true}$ if some $j < i$ has $dp[j]$ true and $s[j..i)$ in the dictionary

BASE CASES AND ORDER

$dp[0] = \text{true}$, the empty prefix, which splits vacuously into no words at all. That base is what lets any single dictionary word starting at position 0 turn its cell on. Fill left to right, since every j is smaller than i . Several cells here read nothing whatsoever, because no suffix ending at them is a dictionary word; those are 0 by default rather than by comparison.

Time: $O(n^2)$ lookups

Space: $O(n)$

Watch out

Stopping at the first prefix that splits and calling the whole string splittable. On this input the first seven characters split two different ways and the string still fails. The greedy cousin of the same mistake is committing to the longest dictionary word that matches at each position: take "cats" here and you strand "andog"; only backing up to "cat" gets you as far as index 7, and the table tries both without you having to notice.

► Box-Stacking Problem

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the tallest stack achievable with rotation i sitting on top: one running height, tied to finishing with this particular rotation, and nothing about which other rotations are stacked beneath it.

THE FINAL DECISION

Ask what sits directly beneath this rotation in the tallest stack that finishes with it on top. It has to be an earlier rotation, sorted before it because its base area is at least as large, whose base strictly contains this rotation's base in both dimensions, and which comes from a different original box, since a box cannot rest on a different orientation of itself. Whichever qualifying rotation already carries the tallest stack beneath it is the one to build on, compared rather than summed, since only one rotation actually sits directly beneath this one.

RECURRENCE

```
dp[i] = height[i] + max over every earlier rotation j whose base strictly
contains rotation i's base in both dimensions, from a different box, of
dp[j], or height[i] alone if no such j exists
```

BASE CASES AND ORDER

$dp[0] = 5$ is the only declared base: sorted by decreasing base area, the very first rotation has the largest base of any in the table, so no earlier, bigger base exists for it to rest on, and a stack of just this rotation alone stands as tall as its own height. That base, 9 by 8, only exists by rotating the fourth box onto its length as the new height, not the orientation it was given, and the cell does get read often afterward, by every later rotation whose base fits strictly inside 9 by 8 from a different box. Some compute cells still resolve with no read at all, the same zero-dependency shape as the other three problems in this batch: $dp[3]$ is exactly that case, since the only rotation whose base numerically contains 8 by 5 is $dp[0]$, and $dp[0]$ comes from the same original box, so the box-identity check rules it out and $dp[3]$ is left standing alone on its own height. That specific cell then turns out to anchor the entire optimal stack once later rotations from other boxes are free to build on it. Every other cell depends only on smaller indices, so filling left to right guarantees every candidate that does qualify already holds a real value before it is read. Because the answer is whichever cell ends up tallest, not necessarily the last one, the table also has to track that argmax as it fills.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Only considering each box in the orientation it was given, instead of every box's three rotations. The largest base anywhere in this table, 9 by 8, belongs to a rotation of the fourth box that is not its given order: it comes from using that box's length as the new height, a genuine rotation away from [5, 8, 9]. Restricting every box to its given order would shrink the candidate pool before the recurrence even runs and could shrink the answer along with it. The other half of the same trap runs the opposite way: once a box is expanded, its rotations are still the same box, so no rotation may rest on another rotation of itself even when the bases fit. $dp[3]$ is the visible proof, sitting alone with no read despite an 8 by 5 base that fits neatly inside the 9 by 8 above it.

► Count Derangements

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of derangements of i distinct objects: permutations in which no object lands back in its own original position, and nothing about which specific permutation achieves it. One number names the state, so the table is one row.

THE FINAL DECISION

Ask what happens to the object that ends up in position 1: it cannot be object 1 itself, so there are $i-1$ choices for it. Say object k lands there. Now ask what object k 's own position does: either object 1 moves into it, closing a 2-cycle and leaving a derangement of the remaining $i-2$ objects ($dp[i-2]$ ways), or object 1 goes anywhere else that is not its own spot, which is exactly a derangement of the remaining $i-1$ objects with object k 's slot relabeled as object 1's ($dp[i-1]$ ways). Both cases are multiplied by the same $i-1$ choices for the first step, so they factor out as one multiplication over a sum.

RECURRENCE

$$dp[i] = (i - 1) \times (dp[i - 1] + dp[i - 2])$$

BASE CASES AND ORDER

$dp[0] = 1$ and $dp[1] = 0$ are given directly: an empty arrangement is vacuously deranged, and the one permutation of a single object necessarily fixes it, so zero derangements exist. Every other cell, including $dp[2]$, depends only on smaller indices and falls out of the very same transition (see the pitfall); filling left to right guarantees both dependencies exist before they are read.

Time: $O(n)$

Space: $O(n)$, reducible to $O(1)$

Watch out

Assuming $dp[1] = 0$ means the recurrence has broken, or that $dp[2]$ needs a hand-written special case the way painting-fence's $dp[2]$ does. Neither is true here: $dp[1] = 0$ is the honest count of derangements of one object (none exist, since the lone object has nowhere else to go), and feeding $dp[0] = 1$ and $dp[1] = 0$ through that very same transition at $i = 2$ already produces the correct $dp[2] = 1$. No third base case is needed once the two seeds are right.

► Longest Subsequence with 1 adjacent difference

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the length of the longest subsequence, with every consecutive pair differing by exactly 1, that ends AT index i specifically, using $nums[i]$ as its last chosen value: one count, tied to ending here, and nothing about which earlier elements were chosen.

THE FINAL DECISION

Ask what value sat immediately before this one in the best run that ends at index i . It has to be some earlier index whose value is exactly one away, either one higher or one lower, since that is the run's only rule; whichever qualifying earlier index already holds the longest run is the one to extend. Every qualifying j is a candidate to extend, compared rather than summed, since only one of them actually precedes index i in the run that ends here.

RECURRENCE

```
dp[i] = 1 + max over every j < i with abs(nums[i] - nums[j]) == 1 of dp[j],  
or 1 if no such j exists
```

BASE CASES AND ORDER

$dp[0] = 1$ is the only declared base: with no index before 0, the first element trivially starts a run of length 1 by itself, and it does get read later, at index 2, since $nums[2] = 2$ is exactly 1 away from $nums[0] = 1$. Every other index is a genuine compute cell, even the ones with no qualifying predecessor at all: $dp[1]$ is exactly that case, since $nums[0] = 1$ is 99 away from $nums[1] = 100$, nowhere near a difference of exactly 1, so $dp[1]$ resolves to 1 on the strength of the same plus-one for standing alone, with no `cell_read` emitted, because there is nothing one-away before it to consult. Every other cell depends only on smaller indices, so filling left to right guarantees every predecessor that does qualify already holds a real value before it is read. Because the answer is whichever cell ends up largest, not necessarily the last one, the table also has to track that `argmax` as it fills.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Assuming the chosen values must sit next to each other in the array. They do not: here the two runs that actually differ by 1 (1, 2, 3, 4, 5, 6 and 100, 99, 98, 97, 96, 95) are interleaved one element apart the whole way through, so the longest subsequence has to skip every other element to stay on one run. A second version of the same mistake is reading the last cell as the answer instead of the largest cell in the table; both runs finish with a table value of 6, but the ascending run's tail reaches it one index earlier.

► Word Wrap Problem

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the smallest total penalty for laying out the first i words, with a line break placed right after word $i - 1$. The index counts words consumed, not words indexed, so $dp[0]$ is the empty prefix and the table has one more cell than there are words.

THE FINAL DECISION

Ask where the last line of the prefix begins. Whatever word starts it, call that j , the words j through $i - 1$ sit on that line together and everything before j is a smaller version of the same problem. So the cost splits cleanly into $dp[j]$ plus the penalty for that one line, and you take the best j . The only j worth trying are the ones where words j through $i - 1$ actually fit inside the width, which is a short contiguous run ending at $i - 1$.

RECURRENCE

$$dp[i] = \min \text{ over } j < i \text{ of } dp[j] + \text{cost}(j, i)$$

BASE CASES AND ORDER

$dp[0] = 0$, the empty prefix, laid out for free. Fill left to right, because every j the recurrence reads is strictly smaller than i . $\text{cost}(j, i)$ is the squared leftover space on a line holding words j through $i - 1$, counting one space between neighbours, or infinity if they do not fit. The one exception is the final cell: the last line of the whole text is not penalized, so $\text{cost}(j, n)$ is 0 for every j whose words fit. That exception is what makes the last cell drop below its neighbour instead of climbing past it.

Time: $O(n^2)$ worst case, $O(n * \text{words per line})$ in practice

Space: $O(n)$

Watch out

Filling each line as full as it will go. It feels obviously right and it is wrong, because a line that fits one more word cheaply now can force a nearly empty line later, and the penalty is squared, so one badly stranded word outweighs several slightly loose lines. On these 8 words greedy pays 52 against an optimum of 28, and the entire difference is one line holding a single 3. The second trap is forgetting that the last line is free. Penalizing it is a different problem with a different answer, and the recurrence looks identical, so the mistake survives every test you would think to write.

► Program for Bridge and Torch problem

STATE: WHAT ONE ENTRY MEANS

Sort the crossing times ascending first. $dp[i]$ is the minimum total time to get the i fastest people across with the torch ending on the far side. One number names the state only because of the sort: without it you would have to say which people are across, which is a subset, not a count.

THE FINAL DECISION

Ask how the slowest person still on this side gets over. Either they cross paired with the fastest, who then walks the torch back, which costs $t[i-1]$ for the crossing and $t[0]$ for the return on top of solving the smaller group, or the two slowest cross together and the two fastest handle the torch around them: the fastest pair goes over, the fastest returns, the slow pair crosses at $t[i-1]$, and the second fastest returns. That second option counts only one of the two slow times, which is what makes it win once the slow people are far apart from the fast ones.

RECURRENCE

$$dp[i] = \min(dp[i-1] + t[0] + t[i-1], dp[i-2] + t[0] + 2*t[1] + t[i-1])$$

BASE CASES AND ORDER

$dp[0] = 0$, nobody to move. $dp[1] = t[0]$, one person walks across alone. $dp[2] = t[1]$, two people cross together at the slower one's pace, and nobody has to come back. Fill upward from $i = 3$, since both options read strictly smaller counts. The sort is not a convenience here, it is what makes $t[0]$ and $t[1]$ mean the fastest and second fastest at every step, which both options depend on.

Time: $O(n \log n)$ for the sort, $O(n)$ after

Space: $O(n)$, reducible to $O(1)$

Watch out

Assuming the fastest person should always carry the torch back. It is the obvious rule and it is wrong exactly when the two slowest are much slower than the two fastest, because pairing the slow people makes one of their times disappear entirely, and that saving outweighs the extra return trip. $dp[3] = 8$ and $dp[4] = 15$ in this table are the two options trading places. The other trap is forgetting to sort, which quietly breaks $t[0]$ and $t[1]$ and therefore both options at once.

Check your understanding

1. For houses [2, 7, 9, 3], what is the maximum total without taking adjacent houses?

- A. 10
- B. 11
- C. 12
- D. 21

2. Append a house worth 1 to [2, 7, 9, 3]. What is the new best total?

- A. 11
- B. 12
- C. 14
- D. 22

Flashcards: cover the answer to recall it

What does $dp[i]$ mean in the prefix version of house robber?

The best nonadjacent sum among the first i houses. The current house has array index $i - 1$.

Why does taking house $i - 1$ use $dp[i - 2]$?

The adjacent house $i - 2$ must be skipped. The first $i - 2$ houses remain available, and their best answer is $dp[i - 2]$.

When is a two-variable optimization safe?

When every transition needs only the previous two states and both old values survive until the new state has been calculated.

Quiz answer key

1. **B.** Taking values 2 and 9 gives 11. Taking 7 and 3 gives 10. The recurrence keeps 11 when the final house 3 is considered.

2. **B.** Skipping keeps 11. Taking the new house uses the first three houses' answer, 11, plus 1. The houses worth 2, 9, and 1 give 12.

One target, many previous choices

Some one-dimensional states inspect many smaller states. The meaning of a counted object matters: choosing the final element counts ordered sequences, while processing each denomination once counts unordered coin combinations.

The idea to remember

- For ordered jumps from a set S , let $\text{ways}[t]$ count sequences with total t . Choosing the last jump gives $\text{ways}[t] = \text{sum of } \text{ways}[t - s] \text{ over usable } s \text{ in } S$.
- Set $\text{ways}[0] = 1$ for the empty sequence and treat negative totals as 0. Fill target totals upward so every smaller total is ready.
- For unordered coin combinations, initialize $\text{ways}[0] = 1$ and other totals to 0. Loop over each coin outside, then totals from the coin upward. This gives each combination one denomination order.
- For minimum coins, use $\text{best}[0] = 0$ and an unreachable sentinel such as Infinity for positive totals. The transition takes $\min(\text{best}[t - \text{coin}] + 1)$, not a sum of counts.
- A transition scanning k choices at each of n targets usually costs $O(nk)$. Verify whether a choice can be reused and whether order distinguishes two answers before choosing loop order.

Work through one example

Count unordered combinations for total 4 using unlimited coins [1, 2].

Start [1, 0, 0, 0, 0]. After coin 1: [1, 1, 1, 1, 1]. After coin 2, update totals 2, 3, 4 to 2, 2, 3. The combinations are 1+1+1+1, 1+1+2, and 2+2.

3 combinations

Worked problem notes (4)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► nth Catalan Number

STATE: WHAT ONE ENTRY MEANS

$\text{dp}[i]$ is the i th Catalan number. To have something concrete to split, read it as the number of ways to fully bracket a product of $i + 1$ factors, which takes i multiplications: $\text{dp}[2] = 2$ because $a \cdot b \cdot c$ is either $(a \cdot b) \cdot c$ or $a \cdot (b \cdot c)$. One number names the state, so the table is one row.

THE FINAL DECISION

Ask which multiplication happens last. In a fully bracketed product exactly one multiplication sits outermost, and it joins a left block to a right block. If the left block uses k multiplications, the right block uses the other $i - 1 - k$. The two blocks are bracketed independently, so for one value of k the counts multiply: $dp[k] \times dp[i - 1 - k]$. Different values of k put the outermost multiplication in different places, so no bracketing is counted twice, and the values of k are added. Every split is added and none is compared, because this is a count and every split contributes. Matrix chain multiplication, in the interval family, runs this same loop over the last multiplication but keeps only the cheapest split, so it takes a min instead of a sum.

RECURRENCE

$dp[i] = \text{sum over } k \text{ from } 0 \text{ to } i - 1 \text{ of } dp[k] \times dp[i - 1 - k]$

BASE CASES AND ORDER

$dp[0] = 1$: a single factor needs no multiplication, and there is exactly one way to leave it alone, so it is 1 (it is a factor in a product, never a term on its own). $dp[1]$ is not a base: it falls out of the recurrence as $dp[0] \times dp[0] = 1$. Every cell reads only smaller indices, so filling left to right means everything it needs is already written.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Storing the table in 32-bit integers. $C(19) = 1,767,263,190$ still fits, $C(20) = 6,564,120,420$ does not, and in Java or C++ the overflow is silent: the loop keeps filling cells with wrong numbers and nothing fails. JavaScript numbers stay exact up to $C(30)$, which hides the bug until the same code is ported or n grows.

► Count Unique BSTs

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of structurally different BSTs that hold i distinct keys. Which keys they are does not matter, only how many: any i keys in sorted order behave exactly like 1 through i . One number names the state, so the table is one row.

THE FINAL DECISION

Ask which key is the root. Once root r is chosen, the BST property decides where every other key goes: the $r - 1$ smaller keys form the left subtree and the $i - r$ larger keys form the right. Each subtree is a BST on fewer keys, and the left shape places no limit on the right one, so for this root the counts multiply: $dp[r - 1] \times dp[i - r]$. Different roots give different trees, so nothing is counted twice, and the roots are added. The roots are summed, never compared, because every root yields valid trees and all of them count. Optimal BST, a classic interval problem outside this course's list, runs this same loop over the root but keeps the cheapest, a min instead of a sum; matrix chain multiplication in the interval family has the same shape.

RECURRENCE

$dp[i] = \text{sum over roots } r \text{ from } 1 \text{ to } i \text{ of } dp[r - 1] \times dp[i - r]$

BASE CASES AND ORDER

$dp[0] = 1$, and it is the cell people get wrong: zero keys make exactly one tree, the empty one. It has to be 1 because it is multiplied; the smallest key as root always has an empty left subtree, and a 0 there would erase every tree rooted at it. $dp[1]$ then falls out as $dp[0] \times dp[0] = 1$, a single node with two empty subtrees. Every cell reads only smaller key counts, so filling left to right works.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Seeding $dp[0] = 0$ on the grounds that no keys means no tree. The recurrence multiplies, so the zero spreads: with $dp[0] = 0$ and $dp[1]$ set to 1 by hand, $dp[2] = 0 \times 1 + 1 \times 0 = 0$, although two keys plainly make two trees, and the row goes on 0, 1, 0, 1, 0, 2, 0, 5, zeroing every even count of keys.

► Count Valid Parenthesis

STATE: WHAT ONE ENTRY MEANS

$dp[i]$ is the number of balanced strings that use exactly i pairs of parentheses. Only the number of pairs matters, not where the string sits inside a longer one, so one number names the state and the table is one row.

THE FINAL DECISION

Ask where the first opening bracket closes. Every non-empty balanced string starts with (, and that bracket has exactly one partner). Between them sits a balanced string of some k pairs; after the partner sits a balanced string of the other $i - 1 - k$ pairs. The inside and the outside are chosen independently, so for one k the counts multiply: $dp[k] \times dp[i - 1 - k]$. Each string has one partner for its first bracket, so it falls under exactly one k , and the values of k are added. No split beats another; they are all added, not compared. The same loop over a split point returns in the interval family, where a split is chosen by min rather than summed.

RECURRENCE

$dp[i] = \text{sum over } k \text{ from } 0 \text{ to } i - 1 \text{ of } dp[k] \times dp[i - 1 - k]$

BASE CASES AND ORDER

$dp[0] = 1$: zero pairs make exactly one balanced string, the empty one. It is what fills the inside of () and the space after the last pair. Set it to 0 and $dp[1] = 0 \times 0 = 0$, and since every string bottoms out in an empty (), the whole row collapses to 0. Every cell reads only smaller pair counts, so filling left to right works.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Splitting a string wherever its prefix balances, instead of only at the partner of its first bracket. $()()()$ balances after 2, 4 and 6 characters, so pairing a balanced prefix with a balanced remainder at every such point counts it three times. Across the five strings with 3 pairs that rule finds 9 splits where the answer is 5. The first bracket's partner is unique, which is what gives every string exactly one split.

► Ways to Triangulate a Polygon

STATE: WHAT ONE ENTRY MEANS

$dp[v]$ is the number of ways to triangulate a convex polygon with v corners. Convexity means only the corner count matters, never the exact shape, so one number names the state. The table starts at $v = 2$, a lone edge, and each column is labelled by its corner count: the cell under $i=5$ is the pentagon.

THE FINAL DECISION

Fix one edge, from corner 1 to corner v . In any triangulation that edge belongs to exactly one triangle, and its third corner, the apex k , is one of corners 2 to $v - 1$. That triangle cuts the rest into the polygon on corners 1 to k (k corners) and the polygon on corners k to v ($v - k + 1$ corners), triangulated independently, so for one apex the counts multiply. Every triangulation has exactly one triangle on the fixed edge, so each is counted under exactly one apex, and the apexes are added. It is a sum over the apex, never a pick of the best one. Matrix chain multiplication, in the interval family, loops over a middle split the same way and takes a min instead.

RECURRENCE

$$dp[v] = \text{sum over apex } k \text{ from } 2 \text{ to } v - 1 \text{ of } dp[k] \times dp[v - k + 1]$$

BASE CASES AND ORDER

$dp[2] = 1$ is the base, and it is the odd one: a polygon with two corners is a single edge, and there is exactly one way to triangulate it, by doing nothing. It is there so that an apex next to the fixed edge ($k = 2$ or $k = v - 1$), which leaves only an edge on one side, multiplies by 1 instead of needing a special case. $dp[3] = dp[2] \times dp[2] = 1$ then falls out of the recurrence: a triangle is already triangulated. Every cell reads only smaller corner counts, so left to right works.

Time: $O(n^2)$

Space: $O(n)$

Watch out

Indexing off by two. A polygon with v corners has $C(v - 2)$ triangulations, not $C(v)$, so reusing a Catalan table indexed by corner count reports 132 for the hexagon instead of 14. The shift comes from the base: this recurrence bottoms out at a two-corner edge, not at a zero-corner polygon.

Check your understanding

1. How many unordered combinations make 4 using coins [1, 2]?

- A. 2
- B. 3
- C. 4
- D. 5

2. If order now matters, how many sequences of 1s and 2s total 4?

- A. 3
- B. 4
- C. 5
- D. 6

Flashcards: cover the answer to recall it

Why is $\text{ways}[0] = 1$ in counting DP?

There is one empty construction. Extending it lets a single usable choice create a valid nonempty answer.

Which loop order counts unlimited coin combinations?

Process coins outside and totals in ascending order inside. The fixed denomination order avoids counting different arrangements separately.

Why should minimum-coin DP not initialize every state to zero?

Zero would falsely claim that every amount is reachable for free. Use 0 only at amount 0 and an unreachable sentinel elsewhere.

Quiz answer key

1. **B.** There are three possibilities: four 1s; two 1s and one 2; or two 2s. Reordering the same coins does not create a new combination.

2. **C.** A final 1 follows one of 3 sequences totaling 3. A final 2 follows one of 2 sequences totaling 2. The total is $3 + 2 = 5$.

Grid DP: arrive from a neighbor

When moves follow a fixed direction, each cell can summarize all routes that arrive there. A count and a minimum path cost may use the same neighbors, but they store different quantities.

The idea to remember

- For right/down moves, $ways[r][c]$ counts routes from the top-left cell to row r , column c . Each route arrives from above or from the left, so add those two counts.
- Set the unblocked start to 1. A blocked cell has 0 routes. A missing neighbor outside the grid contributes 0; this handles the first row and first column without inventing extra paths.
- Fill rows top to bottom and columns left to right. Both dependencies have already been computed. The answer is the bottom-right cell.
- For minimum path sum, store the smallest cost including the current cell. Add its cost to the minimum valid predecessor; use Infinity for unavailable predecessors.
- A rolling row works because the old value at column c is from above and the updated value at $c - 1$ is from the left. Reset blocked cells to 0 so old paths cannot leak through obstacles.

Work through one example

A 3 by 3 grid with only the center cell blocked. Move right or down.

Route-count rows become $[1, 1, 1]$, then $[1, 0, 1]$, then $[1, 1, 2]$. The bottom-right count adds 1 route from above and 1 from the left.

2 routes

Worked problem notes (15)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Min Sum in a Triangle

STATE: WHAT ONE ENTRY MEANS

$dp[r][c]$ is the smallest total of any descent that starts at the apex, steps down one row at a time onto one of the two numbers immediately below, and finishes ON the number at row r , column c , that number itself included. Two numbers name the state, the row and the column, so the table is a grid even though the input is a triangle. The cell holds a total and nothing about which columns the descent passed through; if you want the path itself you walk the table back up afterwards.

THE FINAL DECISION

Ask which number the descent was standing on one row up. To arrive at (r, c) it can only have come from the number up and to the left, $(r - 1, c - 1)$, or the number directly above, $(r - 1, c)$, because those are the only two positions whose own two downward steps include this one. Whichever it was, the part of the descent above that point has to be a cheapest descent to that neighbour, which is the same question on a shorter triangle, and $\text{tri}[r][c]$ is paid once either way. So take the smaller of the two and add this cell's own number. It is a min rather than a sum because exactly one of the two steps actually happened, the same reason min-cost-path takes a min where count-all-paths-in-a-grid adds. The ragged shape bites at the edges: column 0 has nothing up and to its left, and the last number of a row, column r , has nothing directly above it, so those cells have one candidate instead of two.

RECURRENCE

```
dp[r][c] = tri[r][c] + min(dp[r - 1][c - 1], dp[r - 1][c])
```

BASE CASES AND ORDER

$\text{dp}[0][0] = \text{tri}[0][0] = 4$: a descent that has only reached row 0 is standing on the apex and has paid for it, so the apex's own number is the whole of the cell. Every other cell of the triangle is computed. THE DOTS ARE NOT ZEROS, and this is the one thing on this page a learner can misread into a different answer. A grid has to be a rectangle, so the table is declared size by size, but only the cells with column $\leq$ row belong to the triangle; every position above the diagonal renders as a dot because it DOES NOT EXIST. If those dots were zeros they would be free numbers to stand on, and the cheapest descent would wander out into the empty half of the table and come back with a total far below the truth. Nothing is ever written to a dot and no cell ever depends on one, which is exactly why the edge cells have one genuine candidate rather than one real candidate plus one convenient zero. Filling row-major, left to right within each row, means both of the numbers a cell can descend from are final before it reads them, which is the only ordering property the recurrence needs.

Time: $O(n^2)$ for n rows

Space: $O(n^2)$, reducible to $O(n)$ with one rolling row

Watch out

Stepping onto the smaller of the two numbers below at every move. Greedy descent is the natural first instinct and it is wrong at every size from 3 rows up: on 3 rows it takes 4, 5, 5 for 14 while the true minimum is 4, 6, 2 for 12, and on the full 9 rows it pays 36 where the table pays 32. A small number is worth nothing if it is fenced in by large ones below it, which is precisely what a one-step-ahead rule cannot see. The second trap is reading the last cell of the bottom row as the answer; only the SMALLEST cell of the bottom row is, and here that is column 5 while the last cell holds 41. The third is treating the dots above the diagonal as zeros. They are positions that do not exist, and a zero there is a free number the descent would gladly cheat its way through.

► Ways to Partition a Set

STATE: WHAT ONE ENTRY MEANS

$dp[n][k]$ is the number of ways to split n distinct elements into exactly k non-empty, unordered subsets, so the cell holds a COUNT and nothing about which element went where. UNORDERED is the load-bearing word. The k subsets carry no labels and no order among themselves, so splitting three elements a, b, c into $\{a\}$ and $\{b, c\}$ is ONE partition however you write the two pieces down, not two. Two counts name the state, how many elements there are and how many subsets they have to fill, so the table is a grid even though there is no input array anywhere on this page. The nearest table in this course is binomial-coefficient's, and it is the same table SHAPE cell for cell: an $(n + 1)$ by $(n + 1)$ rectangle with everything above the diagonal skipped, the same two predecessors at every computed cell, the same row-major fill order, and the same base POSITIONS row for row, one per row down the left edge and one per row past the first on the diagonal, which is 17 of them at $n = 8$. Read that page first if you have not. TWO things then differ, and they are worth ranking rather than lumping together as one. The difference that matters to the algorithm is a single multiplication at each computed cell, and that multiplication is what this page is for. The difference that matters to anyone retyping the table is smaller and easier to get wrong: the positions match, but eight of the seventeen base VALUES at $n = 8$ do not, because past row 0 the left edge holds 0 here where it holds 1 there. $dp[0][0]$ is 1 on both pages, and the diagonal is 1 on both; it is the eight left-edge cells below row 0 that disagree. basesAndOrder is where that edge is both explained and priced. pitfall prices the other mistake, the missing multiplication.

THE FINAL DECISION

Single out one particular element, say the last of the n , and ask what became of it. Either it ended up alone in a subset of its own, or it ended up sharing a subset with other elements. No partition is in both groups and none is outside them, so the two cases are disjoint and exhaustive and their counts add. TAKE THE ALONE CASE FIRST, because it is the one with no multiplier. If the singled-out element is by itself, the other $n - 1$ elements have to carry all of the remaining $k - 1$ subsets between them, and there are $dp[n - 1][k - 1]$ ways for them to do that. Each of those arrangements yields exactly ONE partition of the n elements: bolt the lone element on as its own subset and you are finished, there is nothing left to choose. So this case contributes $dp[n - 1][k - 1]$ as it stands. NOW THE OTHER CASE, WHICH IS WHERE THE COEFFICIENT COMES FROM. If the singled-out element is not alone, then pulling it out leaves every subset still non-empty, so the other $n - 1$ elements are already split into exactly k subsets, and there are $dp[n - 1][k]$ ways for them to be. But that is NOT the number of partitions this case contributes, and assuming that it is is the mistake this page exists to prevent. Each of those k subsets is a home the element could have been dropped into. That is k choices, and each choice produces a DIFFERENT partition of the n elements, because the subsets are told apart by their contents and the element sits in a different one each time. Nothing is double counted either: given any partition in which the element is not alone, delete the element and you recover both the arrangement of the other $n - 1$ and which subset it came out of, uniquely. So the correspondence is k partitions of n to one arrangement of $n - 1$, in that direction, and the case contributes $k \times dp[n - 1][k]$. WHY EXACTLY k , and not $k!$ and not 2^k : the element goes into ONE subset, so it is a single choice among k , not a permutation of the subsets and not a selection of several of them. And the multiplier is the COLUMN index, the number of subsets, not the row index n . Both of those numbers are printed on the gutters right next to the cell, which is exactly why reaching for the wrong one is easy. It is a SUM between the two cases and a PRODUCT inside one of them because this is a counting problem: both cases really happen, across different partitions, and we want the total of them rather than the better of them. That is the same distinction binomial-coefficient draws against min-cost-path and it does not change here. What changes is that one of the two branches is not one outcome per arrangement but k of them.

RECURRENCE

$$dp[n][k] = k * dp[n - 1][k] + dp[n - 1][k - 1]$$

BASE CASES AND ORDER

BOTH outer edges are given, as on binomial-coefficient's triangle, and one of the two holds a DIFFERENT NUMBER here. That difference is worth stopping on rather than skimming. $dp[0][0] = 1$. There is exactly one way to split the empty set into zero subsets: take no subsets at all. It is 1 rather than 0 because an empty collection of subsets is a legitimate arrangement of nothing, and it is the seed the whole triangle is built from. $dp[n][0] = 0$ for every row past the first, and binomial-coefficient's matching edge is 1. BOTH ARE RIGHT, and the reason is that the two cells count different things. There, the cell counts selections, and choosing NOTHING out of n items is a real selection, the empty one, which always exists and there is exactly one of it. Here the cell counts partitions of a non-empty set into zero non-empty subsets, and zero subsets hold no elements between them, so there is nowhere for the n elements to go and no such arrangement exists at all. One way to choose nothing from anything; no way to split something into nothing. That 0 is not decorative either. Every computed cell in column 1 reads it, which is what pins the whole of column 1 to 1 all the way down: $dp[n][1] = 1 \times dp[n - 1][1] + 0$. Copy that edge across as a 1, which is the likeliest slip for a reader arriving from the other page, and column 3 comes out 7, 32, 122, 423, 1389 across $n = 4$ to 8 in place of the true 6, 25, 90, 301, 966. $dp[n][n] = 1$ for every row: there is exactly one way to split n elements into n non-empty subsets, every element alone in its own. Both edges have to be GIVEN rather than summed, and that reason is arithmetic rather than convention. $dp[n][0]$ would reach for $dp[n - 1][-1]$, column -1, which is off the table altogether, and $dp[n][n]$ would reach for $dp[n - 1][n]$, which sits above the diagonal and does not exist. Every cell with $1 \leq k \leq n - 1$ is computed, and each of those has exactly two live predecessors, never one. At $n = 8$ the rectangle declares 81 cells, 45 of them live: 17 given, one per row down the left edge and one per row past the first on the diagonal, and 28 summed. THE DOTS ARE NOT ZEROS, in the sense that nothing is ever written to them and no cell ever reads one. A position with column $>$ row would be asking to fill more non-empty subsets than there are elements to fill them, which is not a hard question, it is not a question. Nothing could read one by accident either: a computed cell at (n, k) has $k \leq n - 1$, so the two cells it reads, $(n - 1, k)$ and $(n - 1, k - 1)$, both have column at most $n - 1$ and therefore sit on or below row $n - 1$'s own diagonal. Zero is even the mathematically correct value for those positions, since there are no such partitions, so unlike min-sum-in-a-triangle nothing here would be corrupted by a stray zero above the diagonal. They are dots because they are not part of the question. Filling row-major, left to right within each row, means both cells a computed cell reads are final before it reads them, which is the only ordering property the recurrence needs.

Time: $O(n^2)$ to fill the triangle down to row n , or $O(nk)$ if the sweep stops at column k

Space: $O(n^2)$, reducible to $O(k)$ with one rolling row that stops at column k

Watch out

Dropping the coefficient. Write $dp[n][k] = dp[n-1][k] + dp[n-1][k-1]$, which is binomial-coefficient's recurrence, and everything about the picture stays right: same triangle, same two arrows, same bases, nothing thrown, whole numbers throughout. What you get is Pascal's triangle SHIFTED one row down and one column right, not Pascal's triangle in place, because this page's left edge is 0 where Pascal's is 1. Measured, the broken table holds exactly $C(n-1, k-1)$ at every cell with $1 \leq k \leq n \leq 10$. Three things let it survive a glance, and the numbers are what make the point. FIRST, it is not wrong everywhere. Of the 45 live cells at $n = 8$ it gets 24 right, and those 24 are precisely column 0, column 1 and the diagonal, the cells whose value is forced to a 0 or a 1 anyway. All 21 cells with $2 \leq k \leq n-1$ are wrong. The first cell it gets wrong is $dp[3][2]$, where it says 2 and the truth is 3, so it has already failed AT $n = 3$, not somewhere past it. SECOND, the wrong answers are a recognisable sequence rather than obvious garbage: column 3 comes out 3, 6, 10, 15, 21 across $n = 4$ to 8, which are the triangular numbers, against the true 6, 25, 90, 301, 966. THIRD, and worst, one of the wrong answers is a right answer from next door. The broken table returns 6 at $n = 5$, and 6 is the correct value of $S(4, 3)$, so a spot check at a single size can land on a number it has seen before and be reassured by it. The other two mistakes are about which index goes where, and both are worth measuring rather than waving at. Put the coefficient on the OTHER term, $dp[n][k] = k * dp[n-1][k-1] + dp[n-1][k]$, and column 3 comes out 10, 25, 46, 73, 106 across $n = 4$ to 8: wrong at four of the five sizes and exactly right at $n = 5$, where it returns 25. That is why the independent check behind this page runs every size the slider offers rather than one. Use the ROW index as the multiplier instead of the column and it is wrong everywhere and wildly so, 9, 71, 580, 5104, 48860 over the same range. And reading the two axes the wrong way round puts you in the empty half: $S(8, 3)$ is $dp[8][3]$, the 966, while $dp[3][8]$ is a dot, because splitting 3 elements into 8 non-empty subsets is not a small number, it is not a thing at all.

► Binomial Coefficient

STATE: WHAT ONE ENTRY MEANS

$dp[n][r]$ is the number of ways to choose r items out of n distinct items, order ignored, so the cell holds a COUNT and nothing about which items were picked. Two counts name the state, how many there are to choose from and how many to choose, so the table is a grid even though there is no input array anywhere on this page. Set this next to min-sum-in-a-triangle, which fills a table with the same lower-triangular shape: there, every cell has a number of its own out of a given triangle, so the input contributes a term at every single cell. Here there is no input at all to read. A cell is fully described by the pair of counts labelling it, and its value comes from cells alone.

THE FINAL DECISION

Single out one particular item, say the last of the n , and ask whether a selection contains it. If it does, the remaining $r-1$ picks come from the other $n-1$ items, and there are $dp[n-1][r-1]$ ways to make them. If it does not, all r picks come from the other $n-1$ items, and there are $dp[n-1][r]$ ways. Every selection of r items falls into exactly one of those two groups, because it either holds that item or it does not, so the groups are disjoint and together exhaustive and the two counts simply add. It is a SUM rather than a min or a max because this is a counting problem: both cases really happen, across different selections, and we want the total of them, not the better of them. That is the same distinction as count-all-paths-in-a-grid adding where min-cost-path takes a min.

RECURRENCE

$$dp[n][r] = dp[n - 1][r - 1] + dp[n - 1][r]$$

BASE CASES AND ORDER

BOTH outer edges are given, and the reason is arithmetic rather than convention. $dp[n][0] = 1$ for every row: there is exactly one way to choose nothing, the empty selection, and it always exists. It has to be a base because the cell the recurrence would send it to is $dp[n - 1][-1]$, column -1, which is off the table altogether. $dp[n][n] = 1$ for every row: there is exactly one way to choose all n items, take every one of them. It has to be a base because the cell the recurrence would send it to is $dp[n - 1][n]$, which sits above the diagonal and does not exist. Every cell with $1 \leq r \leq n - 1$ is computed, and each of those has exactly two live predecessors, never one. THE DOTS ARE NOT ZEROS, in the sense that nothing is ever written to them and no cell ever reads one: a grid has to be a rectangle, so the table is declared $n + 1$ by $n + 1$, but only the cells with column $\leq$ row belong to the triangle, and a position with column $>$ row would be asking for more items than the row has to offer. Here that matters less than it does in min-sum-in-a-triangle, and the difference is worth being precise about rather than borrowing that page's alarm. There, a zero above the diagonal would be a free number a descent could cheat through, and the answer would come out below the truth. The two pages depend on the same two directions, up-and-to-the-left and straight up, so that is not the difference. The difference is that this page makes the DIAGONAL a base while the triangle computes it: a computed cell here reads $dp[n - 1][r - 1]$ and $dp[n - 1][r]$, and since $r \leq n - 1$ both of those are on or below the diagonal, so the empty half is unreachable from any live cell. Zero is even the mathematically correct value for those positions, since there are no ways to choose more items than exist. The triangle has no such guarantee, because its diagonal cells are computed rather than given, and the straight-up read from one of those lands in the row above, which is a column shorter; that is the read its spec has to leave out by hand. They are dots because they are not part of the question, not because a zero there would corrupt anything. Filling row-major, left to right within each row, means both cells a computed cell reads are final before it reads them, which is the only ordering property the recurrence needs.

Time: $O(n^2)$ to fill the triangle up to row n , or $O(n r)$ if the sweep stops at column r

Space: $O(n^2)$, reducible to $O(r)$ with one rolling row

Watch out

Computing it as $n! / (r! (n - r)!)$. The formula is correct on paper and wrong in a fixed-width number type, and it fails far earlier than anyone expects: $19!$ is already past the largest integer a double represents exactly, so in JavaScript that route hands back `253.000000000000003` for $C(23, 2)$, whose true value is the small integer 253, and by $C(171, 2)$ the numerator has overflowed to Infinity while the answer itself is only 14535. The additive table never divides and never leaves the integers, which is the whole reason to prefer it at these sizes. The second real mistake is reading the two axes the wrong way round: $C(6, 3)$ is $dp[6][3]$, the 20, while $dp[3][6]$ is a dot, because choosing 6 items out of 3 is not a small number, it is not a thing at all. If a lookup lands above the diagonal, the arguments went in swapped.

► Pascal's Triangle

STATE: WHAT ONE ENTRY MEANS

$dp[r][k]$ is entry k of row r of the triangle, for $0 \leq k \leq r$. That is binomial-coefficient's table, where the same cell is read as $C(r, k)$, the number of ways to choose k items out of r ; this page needs nothing from that reading, but it is why the entries are called binomial coefficients. Two numbers name a cell, its row and its position in the row. What changes from every neighbouring page is not the state but what counts as the answer: here it is EVERY cell of the table, so there is no cell to aim at and nothing to throw away afterwards.

THE FINAL DECISION

The statement hands you this rule, and it can give it outright because the triangle is defined by it. binomial-coefficient derives the same rule from a counting argument (single out one item; a selection either keeps it or drops it), and nothing in that derivation changes here. What is worth noticing on this page is the shape of the dependency rather than its origin: every interior entry needs exactly two entries of the row above, an adjacent pair, so a row can be written as soon as the row above it is finished, and never sooner.

RECURRENCE

$$dp[r][k] = dp[r-1][k-1] + dp[r-1][k], \text{ for } 0 < k < r$$

BASE CASES AND ORDER

$dp[r][0] = 1$ and $dp[r][r] = 1$ on every row: the two outer edges are 1s by definition. They have to be given rather than summed, because the rule would reach for column -1 at the left end and for a position above the diagonal at the right end, and neither exists. Row 0 is the single cell where the two edges meet. Fill row by row, top to bottom; within a row the interior entries do not read each other, so left to right is as good as any order. The positions above the diagonal show a dot: they are not part of the triangle, and no interior cell can reach one, since a cell at column $k < r$ reads columns $k-1$ and k of row $r-1$, both on or below that row's diagonal.

Time: $O(\text{numRows}^2)$, one addition per interior entry

Space: $O(\text{numRows}^2)$, which is the size of the output itself

Watch out

Carrying *nth-row-of-pascals-triangle's* one-array trick over to this problem. There, a single array updated from the right turns each row into the next, and that is correct because only the last row is wanted. Here, pushing that one array into the result after each pass pushes the SAME array every time, so every row of the result is one object, and when the loop finishes they all show its final contents: at $\text{numRows} = 8$ the result is 1, 7, 21, 35, 35, 21, 7, 1 eight times over. Nothing throws and every number in it is a real entry of the triangle, which is what makes it easy to miss in a quick look at the last row. Push a copy each pass, or build a fresh row as the snippet does.

► Nth Row of Pascal Triangle

STATE: WHAT ONE ENTRY MEANS

$dp[n][k]$ is entry k of row n of Pascal's triangle, which is the number of ways to choose k items out of n . That is the SAME TABLE binomial-coefficient fills, cell for cell: same lower-triangular shape, same two given edges, same rule at every interior cell, same row-major order. One thing changed, and it is the question asked of the table. There, the input named a pair of counts and one cell answered it. Here the input names a row, so the answer is a WHOLE ROW: all $n + 1$ cells of the bottom row are output, and the sweep's final pass is the result rather than a step towards it. A reader arriving here first loses nothing by reading it in this order: a cell holds a count and nothing about which items were picked, and the two numbers labelling it, the row and the position within that row, are the whole of the state.

THE FINAL DECISION

Single out one of the n items and ask whether a selection of k contains it: the selections that keep it need $k - 1$ more from the other $n - 1$ items, the selections that drop it need all k from the other $n - 1$, no selection is in both groups and none is outside them, so the two counts add rather than compete. binomial-coefficient makes that argument in full, including why a counting problem sums where an optimization problem takes a min, and not one word of it changes here. What is worth adding is what the same rule says once you are after the whole row: row n is determined by row $n - 1$ ALONE, one adjacent pair of it at a time. The rule never reaches back further than a single row, so every row above that one is dead the moment the row below it has been written. That single observation is what the execution section below spends its whole length on.

RECURRENCE

$$dp[n][k] = dp[n - 1][k - 1] + dp[n - 1][k]$$

BASE CASES AND ORDER

$dp[n][0] = 1$ and $dp[n][n] = 1$ on every row, one way to choose nothing and one way to choose everything, and both must be given rather than summed because the cells the rule would reach for, column -1 at one end and a position above the diagonal at the other, are off the table. Now say what that means HERE, because it is not what it means on binomial-coefficient's page, where the bases are the two places the answer is not. TWO OF THE NUMBERS THIS PAGE RETURNS ARE BASES. The 1 at each end of row 8 was not worked out by the sweep, it was handed to it, and only the 7 interior entries of the answer row are computed at all. Of the 45 live cells at $n = 8$, 17 are given, one per row down the left edge and one per row past the first on the diagonal, and 28 are summed. The positions above the diagonal show a dot, meaning a position that does not exist rather than a zero, and nothing here could read one even by accident: a computed cell in row r at column k has $k \leq r - 1$, so the two cells it reads are columns $k - 1$ and k of row $r - 1$, both on or below that row's own diagonal. The FILL ORDER is that page's order too, and it is what this page's whole objection is about. Row-major, top to bottom and left to right inside each row, is enough for the rule, though it is not the only order that would do (column-major satisfies the same dependencies), and it is exactly the order the statement objects to: reaching row 8 this way means writing all 36 cells of rows 0 through 7, and 21 of those are additions this order performs before the first interior entry of row 8 is reached. None of the 36 is wanted. The order is not wrong. It is the cost, and execution is where that gets settled.

Time: $O(n^2)$ additions to sweep the triangle down to row n , against $O(n)$ multiply-divide steps for the in-row closed form

Space: $O(n^2)$ for the full table, $O(n)$ with one rolling row, which is the size of the answer

Watch out

Updating the single rolling row from the LEFT. It is the one step of the reduction that has a direction, and going the wrong way fails silently: nothing throws, the values stay whole numbers, and they are simply the wrong whole numbers. Writing $\text{row}[k] = \text{row}[k] + \text{row}[k - 1]$ with k running upwards means position $k - 1$ has already been overwritten with the new row's value by the time position k reads it, so the cell adds its NEW left neighbour instead of the old one the rule asked for. At $n = 8$ that returns 1, 8, 35, 110, 275, 572, 1001, 1430, 1430 where the row is 1, 8, 28, 56, 70, 56, 28, 8, 1: too large from column 2 rightwards, no longer symmetric, and no longer even ending in a 1. What makes this survive a glance is that the wrong numbers are not noise. Run the broken loop for $n = 1$ through 8 and its last entry comes out 1, 2, 5, 14, 42, 132, 429, 1430, which are the Catalan numbers, so what comes back is a real and recognisable integer sequence rather than obvious garbage. Run the inner loop downwards from $k = i$ instead and every cell reads the old value it wanted, with no second buffer. A smaller trap worth naming: the 1s at the two ends of the row are given, not computed, so a loop that tries to compute every entry of row n uniformly will ask for column -1 at one end and a position above the diagonal at the other.

► Min Cost Path

STATE: WHAT ONE ENTRY MEANS

$\text{dp}[r][c]$ is the cheapest total cost of any right-and-down route from the top-left corner to cell (r, c) , counting every cell the route stands on. Two numbers name the state, the row and the column, so the table is a grid rather than a row. The cell holds a price and nothing about the route that achieved it; if you want the route itself you walk the table backwards afterwards.

THE FINAL DECISION

Ask which step landed on (r, c) . Only two are legal: down from $(r - 1, c)$, or right from $(r, c - 1)$.

Whichever it was, the rest of that route has to be a cheapest route to that neighbour, which is the same question on a smaller corner of the grid, and the final step is charged $\text{cost}[r][c]$ either way. So take the cheaper of the two neighbours and add this cell's cost once. It is a min rather than a sum because exactly one of the two steps actually happened. `count-all-paths-in-a-grid` asks about the same two neighbours and adds them instead, because there every route counts rather than competing.

RECURRENCE

```
dp[r][c] = cost[r][c] + min(dp[r - 1][c], dp[r][c - 1])
```

BASE CASES AND ORDER

$dp[0][0] = cost[0][0]$, and that is a CONVENTION rather than something the statement forces. The statement says a cell has a cost to enter, and you never enter the cell you begin on, so a reading where the start is free is perfectly defensible; it just gives a different number at every size. This page charges for the start, because that is what almost every published version of the problem does, so a learner checking the answer against another source sees the same total. If you meet the other convention, subtract $cost[0][0]$ from every cell. Row 0 has nothing above it and column 0 has nothing to its left, so those cells have one candidate instead of two and come out as running totals along the two edges. Filling row-major, left to right within each row, means the cell above and the cell to the left are both already final before any cell uses them, which is the only ordering property the recurrence needs.

Time: $O(\text{rows} \times \text{cols})$

Space: $O(\text{rows} \times \text{cols})$, reducible to $O(\text{cols})$ with one rolling row

Watch out

Stepping onto whichever neighbour is cheaper. From the start of this grid the cell below costs 1 and the cell to the right costs 2, so a greedy walk goes down, and that 1 is fenced in by an 8 and a 9: greedy pays 21 where the table pays 17. A cheap cell is worth nothing unless its neighbours are cheap too, which is precisely what a local rule cannot see and what the table prices for you. The second trap is the base. Decide out loud whether $dp[0][0]$ is $cost[0][0]$ or 0 before filling a single cell, because both conventions are in circulation and they disagree at every cell by exactly $cost[0][0]$.

► Count all paths in a Grid

STATE: WHAT ONE ENTRY MEANS

$dp[r][c]$ is the number of distinct right-and-down routes from the top-left corner to cell (r, c) . Two numbers name the state, the row and the column, so the table is a grid rather than a row, and the cell holds a count with nothing in it about which routes were counted.

THE FINAL DECISION

Ask what the last step onto (r, c) was. It was either a step down from $(r - 1, c)$ or a step right from $(r, c - 1)$. Those two sets of routes can never overlap, because they differ in their final step, and between them they are every route that arrives here at all. So the count is the sum of the two, not a choice between them. min-cost-path asks this same question about these same two neighbours and takes a min instead, because there the routes compete on price; here they are all counted.

RECURRENCE

$$dp[r][c] = dp[r - 1][c] + dp[r][c - 1]$$

BASE CASES AND ORDER

$dp[0][0] = 1$: standing on the start is one route, the empty one. Every other cell of row 0 and of column 0 is 1 as well, and those are worth declaring out loud rather than deriving: a cell in the top row has no cell above it to come from, so the only route into it is the single straight run of steps right, and the first column is the same story with steps down. Filling row-major, left to right within each row, means the cell above and the cell to the left already hold their final counts before any cell uses them, which is the only ordering property the recurrence needs. This grid keeps its 6 rows fixed and lets the slider change the number of columns, so you can watch the table get wider one column at a time while its row count, and with it the number of downward steps every route has to take, stays put.

Time: $O(\text{rows} \times \text{cols})$

Space: $O(\text{rows} \times \text{cols})$, reducible to $O(\text{cols})$ with one rolling row, or $O(1)$ via the closed form

Watch out

Counting steps rather than orderings, and then getting the closed form off by one. Every route across a 6 by 4 grid takes exactly $5 + 3 = 8$ steps, so there is nothing here to optimize and no route is shorter than another; the only question is how many orders those 8 steps can come in. And both numbers in the closed form count steps rather than rows and columns: the top is the total number of steps, $(\text{rows} - 1) + (\text{cols} - 1)$, and the bottom is how many of those steps go in one chosen direction. A 6 by 4 grid takes 5 downs and 3 rights, so it is $C(8, 5) = 56$, never $C(10, 6)$ and never $C(5, 3)$. The table is the safer place to start precisely because it never asks you to get that right.

► Paths in a Grid with Obstacles

STATE: WHAT ONE ENTRY MEANS

$dp[r][c]$ is the number of right-and-down routes from the top-left corner to cell (r, c) that never stand on an obstacle. That is the same two-number state as the unobstructed grid: the obstacles change what a cell is allowed to be, not what the cell means. Read a 0 carefully, because two different situations produce one: a blocked cell is 0 because no route may occupy it, and a clear cell is 0 when no route manages to reach it. The table prints the same digit for both, so keeping them apart is on you.

THE FINAL DECISION

For a clear cell, ask exactly what count-all-paths-in-a-grid asks: the last step was either down from above or right from the left, the two sets of routes are disjoint, so add them. The obstacle check comes first and cancels the question entirely. If no route may stand here, there is no last step to ask about, and the count is 0 without either neighbour entering into it. Where the zero comes from is the thing to hold on to: a blocked cell is zero by rule, and a clear cell is zero only when the ways into it sum to zero, one way in along an edge and two everywhere else, which means every way in is itself dead.

RECURRENCE

$dp[r][c] = 0$ if (r, c) is blocked, otherwise $dp[r - 1][c] + dp[r][c - 1]$

BASE CASES AND ORDER

$dp[0][0] = 1$ is the only base, and that single difference from the unobstructed grid is worth dwelling on. There the whole first row and first column can be declared 1 up front; here they cannot, because one obstacle in row 0 makes every cell to its right unreachable, so the edges have to be computed like every other cell, just with one candidate instead of two. Filling row-major keeps the cell above and the cell to the left final before they are used. A blocked cell is the one write in this table that depends on nothing at all: it has no cells to consult, so the walkthrough does not stop and ask you to predict it, because there is no arithmetic there to predict.

Time: $O(\text{rows} \times \text{cols})$

Space: $O(\text{rows} \times \text{cols})$, reducible to $O(\text{cols})$ with one rolling row

Watch out

Reading every 0 in the table as an obstacle. In the 5 by 5 table, row 4 column 1 holds 0 and is perfectly walkable; both of its ways in are blocked, so no route reaches it, and it then hands that 0 on to its own neighbours like any ordinary value. Whole regions of clear cells can go dark like this behind a couple of well-placed obstacles. The other half of the trap is muscle memory on the edges: filling the first row and first column with 1s is right only until an obstacle appears in one of them, and seeding 1s past that obstacle invents routes that walk straight through a wall.

► Permutations with K Inversions

STATE: WHAT ONE ENTRY MEANS

$dp[r][k]$ is the number of permutations of 1 through $r + 1$ that contain exactly k inversions, an inversion being a pair of positions holding values in the wrong order, larger before smaller. THE ROW INDEX IS ONE LESS THAN n , and getting that straight is the single thing most likely to make every number on this page look wrong. Row 0 is $n = 1$, row 1 is $n = 2$, row r is $n = r + 1$, and the table therefore declares size rows rather than size + 1, because $n = 0$ answers nothing anybody asked. It is not the only 2D table in this course that offsets an index, so treat no row gutter as self-evident: read the row heading on whatever page you are on before trusting the numbers down the side. The cost of missing it is concrete: at $n = 7$ the bottom row is row 6, and a reader who takes row 6 for $n = 6$ reads 169 where the answer for $n = 6$ is the 71 one row up. Two numbers name the state, how many values are being permuted and how many inversions they have to hold, so the table is a grid even though there is no input array anywhere on this page; that much it shares with binomial-coefficient and ways-to-partition-a-set. THE SHAPE IS WHERE IT PARTS COMPANY WITH THEM. Those two declare a square and skip the half above the diagonal, as do nth-row-of-pascals-triangle and min-sum-in-a-triangle: measured, four of the eight 2D tables authored before this one are lower-triangular in exactly that way, all four skipping the cells with column greater than row, and a dot on one of them means a position that does not exist. This table is a FULL RECTANGLE with every declared cell live and not a single dot, and its width does not grow with n past 12 columns. The cell holds a count and nothing about which orderings achieved it.

THE FINAL DECISION

Take the largest of the n values and ask where it ended up. Nothing outranks it, so every inversion it belongs to is an inversion with something to its RIGHT, one for each value sitting after it. Put it last and it contributes none. Slide it one position forward and it contributes one. Put it at the very front and all $n - 1$ of the others are to its right, so it contributes $n - 1$. That is n placements contributing $0, 1$, up to $n - 1$ inversions, one value of j apiece. NOW THE PART THAT MAKES IT A RECURRENCE. Delete the largest value and what is left is a permutation of the other $n - 1$, in the same relative order, holding exactly the inversions it held before: none of them involved the largest value, so none of them is disturbed by where it sat. So if the largest contributed j , the rest have to supply the remaining $k - j$ between them, and the row holding permutations of $n - 1$ values already counts the ways to do that. Mind which row that is: this cell lives in row $n - 1$, so the row it reads is row $n - 2$, one above it, and in the table's own coordinates that is simply $dp[r - 1][k - j]$. The correspondence runs both ways with nothing left over. From a permutation of n you read off one j and one permutation of the rest; from a j and a permutation of the rest you rebuild exactly one permutation of n . So the counts add, and they add over a WINDOW rather than a pair, which is the whole difference between this page and the three Pascal-shaped tables it sits near. TWO SEPARATE THINGS CAP THE WINDOW and both are real. j cannot exceed the number of positions the largest value can move forward through, which is $n - 1$, and in table coordinates $n - 1$ is the ROW INDEX r . j also cannot exceed k , because the rest of the permutation cannot supply a negative number of inversions to make up the difference. So j runs 0 to $\min(k, r)$ and the window is $\min(k, r) + 1$ cells wide. It is a SUM rather than a min or a max because this is a counting problem: every placement really happens, across different permutations, and the total of them is what is wanted rather than the best of them. That is the same distinction count-all-paths-in-a-grid, two problems up this file, draws against min-cost-path.

RECURRENCE

$dp[r][k] = \text{sum over } j = 0 \text{ to } \min(k, r) \text{ of } dp[r - 1][k - j]$

BASE CASES AND ORDER

THE WHOLE OF ROW 0 IS GIVEN, all of it, not merely its first cell. Row 0 is $n = 1$ and there is no row above it, so nothing in it could be summed from anywhere. $dp[0][0] = 1$, because the one ordering of a single value has no inversion: an inversion needs two positions to disagree and there is only one position. $dp[0][k] = 0$ for every k from 1 to the right edge, because one value cannot be out of order with anything. At $n = 7$ that is 12 given cells, one per declared column. COLUMN 0 IS GIVEN ON EVERY ROW TOO, value 1, and that is a choice with a reason on each side. The value itself is not in doubt: exactly one permutation of any n has no inversions at all, the sorted one. What is in doubt is whether to compute it, and the arithmetic settles that. The window for column 0 is $\min(0, r) + 1 = 1$ cell wide, so the recurrence would copy $dp[r - 1][0]$ across and call it a step, and a one-term sum restates a value rather than deriving one. Saying the sorted permutation out loud says strictly more. At $n = 7$ that is 6 further given cells, one per row past the first, so 18 of the table's 84 cells are given and the other 66 are summed. THE TABLE IS CUT OFF ON THE RIGHT, AND THE CUT IS A PROOF RATHER THAN A NOTE. A permutation of n values holds at most $n(n - 1)/2$ inversions, which is 21 at $n = 7$, so a complete table would need 22 columns; this one declares 12, numbered 0 through 11. Here is why nothing surviving is damaged. Every cell $dp[r][k]$ reads only cells $dp[r - 1][k - j]$ with j at least 0, so every cell it reads sits in its own column or a column to its LEFT. No cell anywhere in this table ever reads a column to its right. Removing columns 12 and beyond therefore removes cells that no remaining cell could ever have consulted, and every cell left on screen holds exactly the value it would hold in the full 22-column table. The answer cell, column 5, is nowhere near the cut. NOW THE HONEST HALF: the rows themselves are incomplete, visibly so at the bottom. At $n = 7$ five of the seven rows shown are complete, $n = 1$ through $n = 5$, whose largest inversion counts are 0, 1, 3, 6 and 10 and all fit inside column 11. The other two are truncated, $n = 6$ running to 15 and $n = 7$ to 21. What that costs you is a picture rather than a value: every complete row is symmetric about its middle, 1, 4, 9, 15, 20, 22, 20, 15, 9, 4, 1 at $n = 5$, because reversing a permutation turns each of its inversions into a non-inversion and back, so k and $n(n - 1)/2 - k$ always tie. A cut row shows that symmetry only up to where it stops. As it happens neither cut hides its row's peak: $n = 6$'s twin 101s sit at columns 7 and 8 and $n = 7$'s twin 573s at columns 10 and 11, both pairs on screen, and what falls off the right edge is only the descent back to a single 1. THE FILL ORDER is row-major, left to right within each row, and the window makes the ordering requirement weaker rather than stronger. Every cell a computed cell reads lies in the row DIRECTLY above it, never in its own row, so finishing each row before starting the next is the whole of what the recurrence needs; within a row the order is free, because no cell in a row depends on another cell in the same row.

Time: $O(n \times k \times \min(k, n))$ to fill the table, since each of the $(n - 1) \times k$ computed cells sums a window of at most $\min(k, n - 1) + 1$ cells; $O(n^2 k)$ as a loose bound

Space: $O(n \times k)$, reducible to $O(k)$ with one rolling row

Watch out

The window width. There are THREE ways to get the width itself wrong, they fail in completely different registers, and they are worth ranking rather than lumping together; then there is a fourth mistake that is not about the width at all, and it is the one this table protects you from least. TWO OF THE FOUR CAN FAIL SILENTLY, which is the ranking that matters. USING $\min(k, n)$ WHERE THE TRUTH IS $\min(k, n - 1)$. This grants the largest value one more position than it has, and it is loud rather than subtle: measured, it reports 11, 41, 105 and 224 across $n = 4$ to 7 against the true 3, 22, 71 and 169, wrong at every size by a factor of 3.667, 1.864, 1.479 and 1.325, and the first cell it gets wrong is $dp[1][2]$, where it claims one ordering of two values holds two inversions. Nothing about it looks fine at small n ; it has already failed by the second row. USING $\min(k, n - 2)$ IS THE DANGEROUS ONE, and it is dangerous at every size rather than at some of them. It reports 0, 3, 22 and 71 across the same four sizes, and ALL FOUR are the correct answer for the next size down: measured, that variant's bottom row is exactly the true row for $n - 1$, cell for cell, at every one of $n = 4, 5, 6$ and 7. The 0 it reports at $n = 4$ is no exception, because a permutation of 3 values cannot hold 5 inversions, so 0 is the right answer to the question it has quietly substituted. This is not a table that looks broken; it is a table that looks perfect and answers a question one size down. A spot check anywhere lands on a number it has seen before and is reassured by it, which is why the independent check behind this page runs every size the slider offers rather than one. FORGETTING THE $\min$ WITH k ALTOGETHER, so j runs to $n - 1$ and the sum reaches for a negative column. This one is not a wrong-number bug at all, and saying it is would be wrong. Every out-of-range term is a term the true sum did not want, so if reading past the left edge contributes nothing the table is unchanged; in JavaScript $dp[r - 1][-1]$ is undefined instead, so $total += undefined$ poisons that cell to NaN. Measured, the NaN reaches the reported answer at $n = 5, 6$ and 7 but not at $n = 4$, where the poisoned cells happen to sit outside the answer's cone. It is a crash-shaped mistake, not a plausible-numbers one. READING THE TWO AXES THE WRONG WAY ROUND is the fourth mistake, the one that is not about the window at all, and the second of the four that can fail silently. This table gives you less protection against it than the triangular pages do. On binomial-coefficient a swapped lookup lands above the diagonal, on a dot, and tells you so at once. Here every cell of the rectangle is live, so a swap lands on a real number: at $n = 7$ the answer is $dp[6][5] = 169$ and the transposed cell $dp[5][6] = 90$, which is a perfectly good count of something else, the orderings of 6 values with 6 inversions.

► Water Overflow

STATE: WHAT ONE ENTRY MEANS

$dp[r][c]$ is how much water ARRIVES at the glass in row r , position c , measured in units of one over 32 of a glass. Read that word again, because everything on this page turns on it. ARRIVING IS NOT HOLDING. A glass holds at most 32 units and passes the rest on, so what a glass HOLDS is $\min(dp[r][c], 32)$, and that is a different number from $dp[r][c]$ at every glass that has filled. At a pour of 18 the table writes 116 at $dp[4][2]$; that glass holds 32 and has already sent 84 onward. Fifteen of the twenty-one glasses are full at that pour and the table shows the arriving number at every one of them, so this is not a corner case, it is most of the table. WHY THE ANSWER CAN STILL BE A CELL is therefore not a technicality to skip past. The statement asks what the target glass ends up HOLDING; the answer cell reads what ARRIVED there. The two agree only while that glass has not filled, and they agree on this page because the target was CHOSEN so that they would. Glass (5, 1), the bottom row's second from the left, first receives any water at all at a pour of 14, and first reaches its 32-unit capacity at a pour of 22. That is where both ends of the slider come from: it runs 14 to 21 and stops one short of the pour that would break the reading. Drag it to the maximum and the answer is 28 out of 32, close to full and not there. At a pour of 22 the cell would read 32 and be right by accident; at 23 it would read 36, and no glass holds 36 units of anything. The bottom LEFT glass, (5, 0), would have been the obvious target and is a worse one in the other direction: it is 0 at every pour below 32, which is already past the top of any range this page could show, so every answer here would be 0. It is not dry forever, and the honest bound is worth stating rather than the dramatic one: from a pour of 32 it holds exactly pour minus 31 units, and it does not itself fill until a pour of 63, so a slider running 32 to 62 would be perfectly usable. It is simply not usable HERE. If you take one sentence from this page, take this one. A table that models a quantity is not obliged to model the question, and knowing where the two stop lining up is your job and not the table's.

THE FINAL DECISION

THERE IS NO DECISION HERE, and that is the thing this cell has to teach. Ask where the water in this glass came from and the answer is not one of two places, it is both of them, at the same time. The glass up and to the left and the glass up and to the right each spill whatever they could not hold, each splits its spill evenly between the two glasses beneath it, and this glass sits beneath both. So the two arms are ADDED, never compared, `combine` reports op sum, and the timeline on this page shows no compare panel at any cell, because there is no choice for one to display. Hold that against the pages either side of it. min-cost-path takes a MIN over the cell above and the cell to the left, because a path arrives from one of them and you want the cheaper. min-sum-in-a-triangle does the same going downwards. maximum-tip-calculator takes a MAX over its two arms, because exactly one of the two waiters served the last customer. All three ask WHICH of two things happened, and a max or a min plus a winner is what that question costs. This page asks HOW MUCH of two things happened, and the arithmetic follows the question: a plus. The division is the second half of the rule and it is physics rather than bookkeeping. A glass that spills S does not send S to each glass below it, it sends $S/2$ to each. Halving the SUM of the two spills is the same number as adding the two halves, and it is one operation instead of two, which is why the narration on every cell states a single division. THE SUBTRACTION IS THE PART THAT IS EASY TO DROP. A parent's cell value is what arrived at it, so the spill is that value minus the 32 it keeps, floored at zero, and a parent that never filled contributes a real, honest 0 rather than a negative number. Use the parent's raw value in place of its spill and the pyramid stops modelling glasses at all: the water would go on being passed down whole, and the top glass would still be handing 576 units to row 1 after it had already kept 32 of them.

RECURRENCE

$dp[r][c] = (\max(0, dp[r-1][c-1] - 32) + \max(0, dp[r-1][c] - 32)) \div 2$, where a parent that does not exist contributes no term at all rather than a zero

BASE CASES AND ORDER

THERE IS EXACTLY ONE BASE, $dp[0][0]$, and it holds the whole pour: 18 glasses at the drive size, which in units of one over 32 of a glass is 18 times 32, or 576. It is also the only place the amount poured enters the table at all. Every one of the other 20 live cells is computed, and each of them draws from the row above and from nowhere else, which is why dragging the slider never moves a dot. TWO THINGS A READER GETS WRONG HERE, and they are different mistakes. FIRST, THE DOTS. The grid is declared 6 by 6 because row 5 needs six positions, but row r holds only $r + 1$ glasses, so every position with $col > row$ shows a dot. A dot is not a glass holding nothing. It is not a glass. This page is unusual in how much it would GET AWAY WITH if the dots were zeros, and that is worth measuring rather than borrowing another page's alarm. min-sum-in-a-triangle takes a min over sums, and as its own page works out, a zero above its diagonal is a free number a descent would cheat through and its answer would come out below the truth. Here a phantom position would compute $\max(0, 0 - 32) = 0$ and contribute 0 to a sum, which is exactly what a missing arm contributes already. Measured at all eight slider sizes: fill the whole 6 by 6 rectangle, skip nothing, and let every phantom position compute from its neighbours; every one of them comes out 0 and the answer cell does not move, staying at 1, 4, 8, 12, 16, 20, 24, 28. THE MARGIN IS EXACTLY ZERO, at every phantom position and every pour, and that is a stronger statement than a cushion rather than a weaker one, because it is structural instead of numeric: the phantom region is CLOSED under this recurrence, since a position with $col > row$ is fed only by positions with $col > row$, and it starts empty, since row 0 has water in position 0 alone. Zeros there can never fill and so can never spill. Be exact, then, about what the dot is doing on this page, because it is NOT propping up the answer the way it is on min-sum-in-a-triangle. It is keeping the table honest about the pyramid it is a picture of. A zero at row 2, position 4 would assert that a glass stands there, and the next reader would be right to ask what happens when it fills. SECOND, THE EDGE CELLS. Ten of the twenty computed cells have ONE parent rather than two, and the reason is positional again. A cell in column 0 has no glass up and to its left, because that would be row $r - 1$, position -1 , and no row has a position -1 : that one is off the pyramid altogether. A cell on the diagonal, where col equals row , has no glass up and to its right, because that would be row $r - 1$, position r , and row $r - 1$ stops one position short: that one is a dot inside the declared rectangle. Off the table and dot on the table is the only difference between the two cases, and neither is a parent that contributed nothing. The other ten cells have two parents. Both counts hold at every slider position, because the slider moves the POUR and not the pyramid. Measured by driving every authored problem that has a slider, this is the only one whose declared shape does not move across its own range: 6 by 6 at all eight pours. What moves inside that fixed shape was measured too, rather than assumed, and it is not everything. Of the 21 live cells, 19 hold a different value at a pour of 21 than at a pour of 14; $dp[5][0]$ and $dp[5][5]$ are 0 at every one of the eight pours, and $dp[4][0]$ and $dp[4][4]$ are 0 at the first two before they start climbing. ORDER. Row-major, left to right, and it needs one property and has it: both parents of any glass sit in the row ABOVE it, never in its own row. The previous row being finished is the whole requirement, so the order within a row is free.

Time: $O(\text{rows}^2)$, constant work at each of the rows $\times (\text{rows} + 1) / 2$ glasses, which is 21 here and does not change with the pour

Space: $O(\text{rows}^2)$ for the whole pyramid, reducible to $O(\text{rows})$ with one array swept right to left

Watch out

Reporting what ARRIVED at the target glass when the question asked what it HOLDS. This is not a slip at the edge of the problem, it is the problem: a table whose cells mean one thing is being read for an answer that means another, and the two coincide over exactly the range where the target has not filled. On this page they coincide everywhere the slider goes, and that is a fact about the target glass rather than about the method. Push one step past the slider's maximum and it breaks: at a pour of 22 the cell reads 32 and is accidentally right, at 23 it reads 36 and the true answer is still 32, at 30 it reads 64 and the true answer is still 32. The fix is one line, `Math.min(dp[r][c], cap)`, and the cost of not knowing you need it is an answer that is wrong without ever looking wrong. THE SAME CONFUSION IN ITS OPERATIONAL FORM is what actually breaks people's code, and it happens one level down. When a cell reads its two parents, the numbers it gets back are ARRIVALS, not spills. There are two ways to get that wrong and one of them goes quiet at the drive size, which is the part worth knowing. Stand at the gate on `dp[5][1]` at a pour of 18, with 6 and 64 in front of you. DROP THE SUBTRACTION ENTIRELY and you get $(6 + 64) \div 2 = 35$ against a true 16. That one is CAUGHT, and not by you: 35 is more than the 32 units a glass holds, so the capacity bound rules it out on sight. Run the same slip through the whole table and it stays caught, giving 70, 75, 80, 85, 90, 95, 100 and 105 at pours of 14 to 21, over capacity at every one. DO THE SUBTRACTION BUT DROP THE FLOOR AT ZERO and at this pour nothing rules it out. `dp[4][0]` holds 6, so it contributes $6 - 32 = -26$, a glass handing water back up the pyramid, and the cell computes $(-26 + 32) \div 2 = 3$. Three is whole, under capacity and entirely plausible, and the truth is 16. Through the whole table that slip gives -17, -12, -7, -2, 3, 8, 13 and 18 across the slider: obviously broken at the first four pours, where the sign gives it away, and quietly wrong at the last four. The floor is doing more work than the subtraction is, and it is the half that gets left out. THE THIRD TRAP IS TREATING THE TWO ARMS AS ALTERNATIVES, out of habit from the other ragged 2D tables in this course. min-sum-in-a-triangle in the basic tier and maximum-tip-calculator in this one both pick a winner from the row above; this one takes both. If a compare panel would help you here, the recurrence you have written down is not this problem's. THE FOURTH costs nothing at first and then costs everything: solving in glasses instead of in thirty-seconds. Nothing crashes and nothing is even inexact, because every value on this pyramid is a dyadic fraction and a double holds those exactly; the answer just arrives as 0.5 where the table said 16, and `dp[4][0]` as 0.1875 where it said 6. What goes is the ability to read a row at a glance, and the habit of trusting the arithmetic to be exact, which stops being earned the moment the split stops being a half. The uneven-split variation below is where that second bill comes due.

► Tile Stacking Problem

STATE: WHAT ONE ENTRY MEANS

$dp[r][j]$ is the number of ways to colour the first $r + 1$ tiles from the 3 colours so that no run of one colour anywhere among them is longer than 3 AND the run the tiles FINISH on is at most j long. THE ROW INDEX IS ONE LESS THAN THE TILE COUNT: row 0 is 1 tile, row r is $r + 1$ tiles, so the table declares size rows rather than size + 1, because a row of no tiles answers nothing anybody asked. Two numbers name the state, how many tiles have been coloured and how long a run they are allowed to finish on, so the table is a grid even though the input is a single number and there is no array anywhere on this page. AT MOST IS THE LOAD-BEARING PHRASE, AND IT IS NOT A TRICK. The state you would write down first is exactly j , the colourings whose final run has length precisely j , and it is a perfectly good state with a perfectly good recurrence. Its answer, though, is the SUM of the bottom row rather than any one cell of it, because a legal colouring of the whole row ends on a run of 1, 2 or 3 and all three count. This page reports one highlighted cell, so a total across a row is not something it can point at. Making the state cumulative fixes that at no cost: once a cell counts every final run up to j , the cell at $j = 3$ has already done the summing, and $dp[size - 1][3]$ IS the answer outright. Three separate measurements back that up rather than one. The cumulative table equals the running row-total of the exact-run table, cell for cell, at every size the slider offers; the exact-run table's bottom row sums to the same answer as the cumulative table's last cell at every one of those sizes; and the cumulative recurrence was checked against brute-force enumeration of all 3^n colourings at every n from 1 to 9, agreeing exactly. THAT REFORMULATION IS THE TRANSFERABLE PART OF THIS PAGE and it is worth more than the recurrence. Whenever the natural answer is a fold across a row, look for an axis along which at most is meaningful and push the fold into the state. The axis here is a LENGTH, which is ordered, so the move is available. It is not always available: an axis of unordered labels, a colour or a common difference, has no at most to accumulate along, and a page whose answer folds over such an axis cannot be rescued this way. Two smaller things the cell does NOT hold. It holds a count, so nothing here tells you which tile got which colour. And column 1 is NOT no two neighbours ever match: it is every legal colouring whose last two tiles differ, with the tiles before them free to run up to 3. Those are different questions and different numbers, 3888 against 384 at 8 tiles, and running them together is the easiest wrong reading of this axis.

THE FINAL DECISION

Ask what the LAST tile of the row does. It either takes a different colour from the tile before it or it repeats that colour. Every colouring does exactly one of those two things, so the two cases are disjoint and together exhaustive and their counts add. TAKE THE REPEAT CASE FIRST, because it is the one with no multiplier. If the last tile repeats, the run it joins is one longer than the run the shorter row ended on, so that shorter run had to be at most $j - 1$ long for this one to come in at at most j . There are $dp[r - 1][j - 1]$ colourings of the first r tiles that finish that way, and each of them yields exactly ONE colouring of $r + 1$ tiles: the colour is forced, it is the one already there, and there is nothing left to choose. So this case contributes $dp[r - 1][j - 1]$ as it stands. NOW THE CHANGE CASE, WHICH IS WHERE THE COEFFICIENT COMES FROM. If the last tile takes a different colour it starts a fresh run of length 1, and 1 is at most j for every column this table computes, so the new run cannot be the thing that breaks the rule. What came before is therefore under NO constraint beyond being legal in its own right, and legal in its own right is exactly what column k holds: $dp[r - 1][3]$ is every colouring of the first r tiles, whatever run it ends on. Each of those can be followed by any of the $m - 1 = 2$ colours that are not the one on tile r , and each choice is a DIFFERENT colouring, so this case contributes $2 \times dp[r - 1][3]$, not $dp[r - 1][3]$. Nothing is double counted either: given a colouring of $r + 1$ tiles whose last two tiles differ, delete the last tile and you recover both the shorter colouring and which of the two other colours was used, uniquely. WHY $m - 1$ AND NOT m : the last tile is choosing a colour DIFFERENT from its neighbour, so one of the three is ruled out. Use 3 and the answers come out 189, 747, 2952, 11664 and 46089 across 4 to 8 tiles against the true 78, 228, 666, 1944 and 5676, wrong at every size and off by a factor of more than eight at the top. AND WHY THE FACTOR SITS ON THE COLUMN-3 CELL rather than on the column- j one: reach for $dp[r - 1][j]$ instead and you have quietly required the earlier run to be short as well, which the change case does not require. That is the dangerous mistake on this page, because it is RIGHT at 4 tiles, returning 78, and wrong at the other four sizes, returning 216, 576, 1488 and 3744. It is a SUM between the two cases and a PRODUCT inside one of them because this is a counting problem: both cases really happen, across different colourings, and the total of them is wanted rather than the better of them. That is the same distinction count-all-paths-in-a-grid draws against min-cost-path. ways-to-partition-a-set is the other page in this course where one branch contributes several outcomes per shorter arrangement rather than one, and the two coefficients are worth telling apart: there the multiplier is the COLUMN index and changes as you walk right, here it is the constant $m - 1$ and never moves.

RECURRENCE

$dp[r][j] = (m - 1) * dp[r - 1][k] + dp[r - 1][j - 1]$, which at $m = 3$ and $k = 3$ is $2 * dp[r - 1][3] + dp[r - 1][j - 1]$

BASE CASES AND ORDER

THE WHOLE OF ROW 0 IS GIVEN, and so is the whole of column 0. Row 0 is 0, 3, 3, 3. A single tile has 3 colourings, its run is exactly 1 long, so it satisfies at most j for every j from 1 up and none at all at $j = 0$. Row 0 is given rather than summed for the plainest reason there is: there is no row above it to read. Get its scale wrong and the whole table scales with it, because every cell is linear in row 0. Write 0, 1, 1, 1 there, on the theory that one tile is one way to colour a tile, and the answers come out 26, 76, 222, 648 and 1892 across 4 to 8 tiles, which is exactly one third of the truth at every size. Write 3, 3, 3, 3, letting the impossible column 0 carry a 3 as well, and they come out 81, 234, 684, 1998 and 5832, above the truth at every size. $dp[r][0] = 0$ for every row past the first. No colouring of a non-empty row of tiles finishes on a run of length at most 0, because it finishes on a run of at least 1. That 0 has to be a BASE rather than a computed cell for two separate reasons and both matter. Arithmetically, $dp[r][0]$ would reach for $dp[r - 1][-1]$, column -1, which is off the table altogether. Editorially, a cell with only one live term would narrate as a bare restatement of another cell, matching none of the arithmetic forms this course narrates in, so it would need writing by hand anyway. And that column is not decorative: every computed cell in column 1 reads it, which is what makes column 1 read $2 \times dp[r - 1][3] + 0$ and nothing else. The 0 in that sum is a real case that cannot happen, not a term somebody forgot. Getting that edge wrong is quiet at the slider's minimum and loud after it, which is worth measuring rather than guessing at. Copy it across as a 3 and the answers come out 78, 231, 675, 1971 and 5757 across 4 to 8 tiles: EXACTLY RIGHT at 4 tiles and wrong at the other four. Copy it across as a 1 and they come out 78, 229, 669, 1953 and 5703, right at 4 tiles again. The reason is a propagation delay rather than luck: a wrong column-0 cell corrupts column 1 one row down, column 2 the row after that and column 3 only on the third row, so at 4 tiles the damage has not reached the answer cell yet. THE FIRST TERM IS THE SAME CELL FOR EVERY CELL IN THE ROW, which is this table's one genuinely unusual property. $dp[r - 1][3]$ does not merely look like the other cells' first dependency, it IS it: at 8 tiles the whole of row 7 multiplies the same 1944. Recount that by driving the spec at any size and listing each computed cell's first dependency, which is column 3 in every row. Two consequences. A row costs three multiplications by one shared number plus three additions of the neighbour one column to the left, so the multiplication hoists straight out of the inner loop. And the arrows fan out of a single cell rather than tracking each cell they feed, which is what the animation shows. THERE IS NO SKIP AND NOT A SINGLE DOT ON THIS PAGE, unlike the triangular tables in this course that declare a square and skip everything above the diagonal. Every declared cell is live, which the shape makes plain: 4 columns times size rows declared, size + 3 of them given (the 4 in row 0 plus one per row below it in column 0) and 3 times (size - 1) computed, so at the slider's maximum of 8 tiles that is 32 declared, 11 given and 21 computed. Filling row-major, left to right within each row, needs only one ordering property and has it: both cells a computed cell reads sit in the row ABOVE, never in its own row and never below, so the previous row being finished is the whole requirement and the left-to-right order within a row is free.

Time: $O(n \times k)$ to fill the table, one multiply and one add per cell, so $3 \times (n - 1)$ computed cells at $k = 3$

Space: $O(n \times k)$, reducible to $O(k)$ with one rolling row swept right to left

Watch out

Using the exact-run state and then forgetting to sum the bottom row. Build $dp[r][j]$ as the final run is EXACTLY j long instead of at most j and you still have a correct table answering a real question, with the same shape, the same row-major fill order and whole numbers throughout. THE ARROWS ARE THE ONE THING THAT DOES CHANGE, and they are the tell. Column 1 has to read the WHOLE of the row above rather than one cell of it, because a fresh run of length 1 can follow any legal colouring whatever run it ended on, and the exact-run state has no single cell holding that total the way this page's column 3 does; columns 2 and 3 then read exactly ONE cell each, since a run of exactly j extends a run of exactly $j - 1$ and nothing else. Measured, that is a fan of 3, 1, 1 across the row against this page's uniform 2, so the dependency picture is neither two arrows nor even the same count from one cell to the next. The fold this page pushed into its state has to happen somewhere, and in the exact-run table it happens in that column-1 fan. Do not try to keep both halves either. Hold on to this page's two arrows while switching to exact-run bases and what you get is not a different correct answer, it is nonsense: measured down five tiles it reads 0 3 0 0, then 0 0 3 0, then 0 0 0 3, then 0 6 6 6, then 0 12 18 18, against the true exact table's 0 3 0 0, then 0 6 3 0, then 0 18 6 3, then 0 54 18 6, then 0 156 54 18. So the arithmetic is where this mistake hides and the arrows are where you catch it. Now its cost. Read the answer off the bottom right cell of the genuine exact-run table the way this page does, and you have reported the colourings that end in a run of exactly 3 rather than all of them. Measured, that returns 6, 18, 54, 156 and 456 across 4 to 8 tiles against the true 78, 228, 666, 1944 and 5676, which is between 7.6 and 8.2 per cent of the truth at every one of those five sizes. Three things let it survive a glance. FIRST, the wrong answers are a recognisable sequence rather than obvious garbage, and they are not small enough to look broken. SECOND, and worse, every one of them is a number that appears on the CORRECT table: at all five sizes the wrong answer sits in column 1 exactly two rows above the bottom, and it is also exactly twice the true answer three sizes down, so a reader checking it against the picture or against a smaller run can find it and be reassured. THIRD, the sibling mistake of reading column 1 of the exact table is invisible for a stronger reason still: the exact and the cumulative tables AGREE throughout column 1, because at most 1 and exactly 1 are the same thing once column 0 is 0, so that read lands on a genuine cell of this page's own table, the 3888 at 8 tiles. The fix is not to sum more carefully. It is to notice that a fold across a row can be pushed into the state, which is what at most does. The second mistake is which cell the coefficient multiplies, and it is dangerous because it is right at the smallest size the slider offers. Write $2 \times dp[r - 1][j] + dp[r - 1][j - 1]$, reading straight up instead of across to column 3, and the answers come out 78, 216, 576, 1488 and 3744: exactly right at 4 tiles and wrong at the other four. That is why the check behind this page runs every size the slider offers rather than one. The other three index and constant mistakes are loud rather than quiet, and they are worth separating from the quiet ones for exactly that reason. Swap the two terms so the diagonal neighbour is the one multiplied and it reads 57, 147, 369, 891 and 2217, wrong everywhere and below the truth. Drop the coefficient altogether and it reads 21, 39, 72, 132 and 243, wrong everywhere and further below. Use m instead of $m - 1$ and it reads 189, 747, 2952, 11664 and 46089, wrong everywhere and above. The last mistake is reading the two axes the wrong way round, and this table protects you against it less than the triangular pages do, because every declared cell here is live and a swapped lookup lands on a real number rather than on a dot. It does not always land at all: at 8 tiles the answer is $dp[7][3]$ and the transposed $dp[3][7]$ is off the table, since the table is only 4 columns wide however many rows it has. Inside the top left 4 by 4 corner the swap is silent, though. $dp[1][3]$ is 9 and $dp[3][1]$ is 54, and both are perfectly good counts of something else.

► Maximum size square sub-matrix with all 1s

STATE: WHAT ONE ENTRY MEANS

$dp[r][c]$ is the side length of the largest square made up entirely of 1s whose BOTTOM-RIGHT CORNER sits exactly at row r , column c . Not the largest square found anywhere so far, and not the largest square that happens to touch this cell somewhere: the corner is pinned here, and the square extends up and to the left from it. $dp[r][c] = 3$ therefore means a 3 by 3 block of 1s occupies rows $r - 2$ through r and columns $c - 2$ through c , and nothing else. Pinning the corner is the entire insight of this page, and reading the cell as 'the biggest square seen up to here' is the misreading that makes the recurrence look impossible, because a biggest-so-far number has no local rule that produces it. Since every cell answers about the square ending at itself, no cell you can name in advance holds the problem's answer; some cell does hold it, but which one depends on the input, so the answer is the largest number the finished table contains.

THE FINAL DECISION

Ask how large a square can end at (r, c) . If the matrix holds a 0 there, none can, at any size, because every square ending at that corner has to stand on that cell. Otherwise the side is at least 1, and the question becomes what stops it being larger. A square of side k cornered at (r, c) contains three squares of side $k - 1$ inside it: one cornered at $(r - 1, c)$ directly above, one cornered at $(r, c - 1)$ to the left, and one cornered at $(r - 1, c - 1)$ diagonally up and to the left. All three have to exist, so $k - 1$ cannot exceed the smallest of those three answers; and going the other way, if all three do reach $k - 1$ then together with this cell's own 1 they tile the whole k by k block with nothing left uncovered. So the side is one more than the SMALLEST of the three. This is the first three-way comparison in the course, and the third candidate is not decoration: the diagonal neighbour is the only one of the three that can see a hole sitting up and to the left, and dropping it is the standard way to get this problem wrong.

RECURRENCE

```
dp[r][c] = 0 where the matrix holds a 0, else 1 + min(dp[r - 1][c], dp[r][c - 1], dp[r - 1][c - 1])
```

BASE CASES AND ORDER

Row 0 and column 0 are declared rather than derived, for a reason that is the recurrence in miniature: a square cornered in the top row has no room above it and a square cornered in the left column has no room to its left, so neither can have a side above 1. Each of those cells is simply a copy of the input, 1 where the matrix holds a 1 and 0 where it holds a 0. Every interior cell is computed, including the ones standing on a 0: those resolve to 0 without consulting a single neighbour, because no neighbour value could rescue a square that has to stand on a 0, so nothing is read and there is nothing to predict. Filling row-major, left to right within each row, means all three neighbours are final before they are used, which is the only ordering property the recurrence needs, and notice that the diagonal one costs nothing extra: a row-major sweep finished the whole previous row before it started this one.

Time: $O(\text{rows} \times \text{cols})$

Space: $O(\text{rows} \times \text{cols})$, reducible to $O(\text{cols})$ with one rolling row plus one saved diagonal

Watch out

Reading a cell as 'the largest square anywhere in the board so far'. It is not: it is the largest square whose bottom-right corner is exactly that cell, which is why a cell can fall back to 0 or 1 while a far bigger square already sits elsewhere in the table, and why the answer has to be a scan over the whole table rather than the value in the last cell. Here the answer 4 sits at row 6, column 7 while the last cell holds 1. The second trap is taking the min of only two neighbours, above and left, and forgetting the diagonal, and it is not a matter of style: at row 1, column 2 of this matrix the cell above holds 1 and the cell to the left holds 1, so a two-way min writes 2 and claims a 2 by 2 block of 1s at rows 0 to 1, columns 1 to 2. There is a 0 at row 0, column 1. The diagonal neighbour, holding 0, is the only one of the three that sees it, and it is what pulls the cell back to the correct 1.

► Maximum Tip Calculator

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the largest total tip collectable from the FIRST i customers when exactly j of them went to the first waiter. Two numbers name the state, how many customers have been served and how many of those the first waiter took, so the table is a grid even though the two inputs are flat lists. READ THE COLUMN CAREFULLY, because this is the one thing on this page a reader can misread into being unable to read the table at all. THE COLUMN COUNTS ONE WAITER, NOT BOTH. j is the FIRST waiter's share, and the second waiter's share is $i - j$, which appears nowhere on the screen. Row 6, column 2, which at 9 customers holds 31, is not two customers, it is six: two served by the first waiter and four by the second, and that four is the number you have to subtract to see. WHY THE SUBTRACTION IS LEGAL is worth separating into its two halves, because they come from different parts of the statement and only one of them is about the caps. The half that makes $i - j$ the second waiter's count is simply that EVERY one of the first i customers is served by somebody, so the two waiters' counts add to i ; nothing about x or y is needed for that, and it survives making the caps slack. The half that $x + y = n$ buys is narrower and just as useful: with the caps exactly partitioning the customers, row n admits only the split x and y , so the bottom row holds exactly ONE live cell and the answer is a cell rather than a fold across a row. The first two variations below take those two halves away one at a time and they do different damage. The cell holds a TOTAL and nothing about who served whom; if you want the assignment itself you walk the table backwards afterwards, and at 9 customers that walk hands customers 3, 4, 5, 7 and 9 to the first waiter. One more thing the cell does not hold: $dp[i][j]$ is not the best you can do with j slots and i customers to choose from, it is the best you can do having served ALL i of them. Every customer in the first i is assigned to somebody in every arrangement this cell scores, which is why the two arms of the recurrence are a choice between waiters and never a choice to skip.

THE FINAL DECISION

Ask who served CUSTOMER i , the last one in the row. Exactly one of the two waiters did, so the two cases are disjoint and between them exhaustive. If the FIRST waiter served customer i , the other $i - 1$ customers left $j - 1$ of them to that waiter, and the tip collected on this step is $A[i]$: that is $dp[i - 1][j - 1]$, the cell up and to the LEFT, plus $A[i]$. If the SECOND waiter served customer i , the other $i - 1$ customers left j to the first waiter, unchanged, and the tip is $B[i]$: that is $dp[i - 1][j]$ directly ABOVE, plus $B[i]$. Both sub-problems are the same question on a shorter list, so both are final in the table by the time this cell is filled. It is a max rather than a sum because only one of the two things happened and the better of them is wanted, the same reason min-cost-path takes a min over its two neighbours where count-all-paths-in-a-grid adds them. WHICH TIP GOES WITH WHICH CELL IS THE THING TO GET RIGHT, and the rule is short: the step that moves LEFT is the step that grows the first waiter's count, so it carries the first waiter's tip. The diagonal is the first waiter, the vertical is the second. Note also that every candidate here is ITSELF a sum, a cell plus a tip, which is why a cell with only one live arm still comes out as a two-term addition rather than as a bare copy of its neighbour. Both ends of this table have cells with one arm, at column 0 and on the diagonal, and none of them reads differently from the cells with two. Array indexing is worth a word, because the formula above counts customers from 1 while the snippet indexes the tip arrays from 0: row i is CUSTOMER i , so the tip added on that row is $a[i - 1]$, not $a[i]$.

RECURRENCE

```
dp[i][j] = max(dp[i - 1][j - 1] + A[i], dp[i - 1][j] + B[i]), where A[i] and B[i] are what customer i tips the first and the second waiter
```

BASE CASES AND ORDER

THERE IS EXACTLY ONE BASE, $dp[0][0] = 0$. No customers served, nobody assigned to either waiter, no tips collected. Every other live cell is computed. TWO SEPARATE SKIP CONDITIONS CARVE THE SHAPE, and both are IMPOSSIBILITY rather than zero. A position with $col > row$ would have the first waiter holding more customers than have been served between the two of them, which is not a hard arrangement to score, it is not an arrangement. A position with $row - col > y$ would leave the second waiter more than their cap of y customers, which the statement forbids outright. A zero in either place would look like a legal assignment worth nothing, when what is actually there is no assignment at all. Nothing is ever written to either kind of position and no live cell reads one, which is why a cell in COLUMN 0 or on the DIAGONAL has ONE genuine candidate rather than one real candidate plus one convenient zero. Those two positions are where an arm is missing, and they are NOT the same thing as the two ends of a row: a row's ends are $\max(0, i - y)$ and $\min(i, x)$, which coincide with column 0 and the diagonal only while i is at most y . Measured at 9 customers: of the seventeen row-end cells that get computed, nine have one arm and eight have two. Row 5 starts at column 1 rather than column 0, so its left end $dp[5][1] = 26$ is a row end with both arms, reading $dp[4][0] = 15$ and $dp[4][1] = 22$; and $dp[9][5] = 56$ is both ends of row 9 at once and still reads $dp[8][4] = 47$ and $dp[8][5] = 49$. At 4 customers it is four one-armed against three two-armed. Where each condition bites: $col > row$ shaves the TOP RIGHT, so row i stops at column $\min(i, x)$; $row - col > y$ shaves the BOTTOM LEFT, so row i starts at column $\max(0, i - y)$. What is left is in exact bijection with the pairs of counts inside the two caps, one live cell per (first waiter's count, second waiter's count) with both inside their own cap, so the live cells number $(x + 1)$ times $(y + 1)$ inside the $(n + 1)$ times $(x + 1)$ rectangle the grid declares. Recount that by driving the spec at a slider size and counting its writes against the declared shape rather than trusting the formula. THE DOTS ARE NOT ZEROS, and this page is unusual in how much it would GET AWAY WITH if they were, which is worth measuring rather than borrowing another page's alarm. min-sum-in-a-triangle takes a min over sums, so a zero above its diagonal is a free cheap number the descent gladly cheats through and its answer comes out below the truth. This page takes a MAX over sums of positive tips, so a zero is the least attractive thing a phantom position could offer: it can only win where the honest candidate is smaller, and every honest candidate here is a non-negative running total plus a positive tip. Measured at every one of the six slider sizes: fill the whole rectangle with no skip at all, let each phantom position compute from its neighbours, and the answer cell is unchanged, 26, 34, 35, 43, 47 and 56. The only phantom positions a live cell could even read are the ones just above the diagonal, at $(i - 1, i)$, read by the diagonal cell (i, i) ; the bottom left region is unreachable outright, because a position at (r, c) with $r - c > y$ is read only by $(r + 1, c)$ and $(r + 1, c + 1)$, and both of those are further past the same cap. NOW STATE THE MARGIN, because the margin is what makes that a fact about these numbers rather than a guarantee. At 9 customers the diagonal cells $dp[1][1]$, $dp[2][2]$, $dp[3][3]$, $dp[4][4]$ and $dp[5][5]$ beat their phantom zero by 2, 9, 15, 26 and 32. The 2 at $dp[1][1]$ is the whole of the safety margin at the top of the table, and it is exactly $A[1] - B[1]$, nine against seven; had customer 1 tipped the second waiter more than the first, the phantom zero would have won that cell outright. So the correctness of this table rests on the position not existing, NOT on the operator being forgiving. Filling row-major, left to right within each row, needs one ordering property and has it: both cells a computed cell reads sit in the row ABOVE, never in its own row, so the previous row being finished is the whole requirement and the order within a row is free.

Time: $O(n \times x)$, one max of at most two candidates per live cell, with $(x + 1) \times (y + 1)$ live cells inside the $(n + 1) \times (x + 1)$ rectangle the grid declares

Space: $O(n \times x)$, reducible to $O(x)$ with one rolling row swept right to left

Watch out

Giving each customer to whoever tips them more, TIES TO THE FIRST WAITER, then handing the rest to the other waiter once one of them hits their cap. It is the natural first instinct and it loses at every size this page can show: 17 against 26, 25 against 34, 26 against 35, 34 against 43, 38 against 47 and 44 against 56, a gap of 9 at the first five sizes and 12 at the sixth. The tie-break has to be named rather than assumed, because customer 2 is worth exactly 2 to each waiter, so "whoever tips them more" decides nothing there and those six figures are the tie-to-first reading. Send the tie to the second waiter instead and every figure moves, to 21, 32, 33, 38, 42 and 45; the rule still loses at all six sizes, which is the part this page rests on. WHY IT LOSES IS NOT THAT GREED IS WRONG HERE, and getting that right is worth more than the pitfall itself. A greedy rule does solve this problem exactly, as the execution note works out; the instinct just reaches for the WRONG greedy rule. Asking whether $A[i]$ beats $B[i]$ uses only the SIGN of the margin $A[i] - B[i]$ and throws away its SIZE, and the sizes are what matter, because the first waiter's slots are scarce and should go to the customers the first waiter is worth the most EXTRA on. At 9 customers the margins run 2, 0, 4, 7, 4, 1, 7, -3, 7 across customers 1 to 9. The five largest sit at customers 3, 4, 5, 7 and 9, which is exactly the set the table picks. The sign rule instead walks left to right and fills the cap with customers 1 to 5, two of whom, customers 1 and 2, have margins of 2 and 0; by the time customers 7 and 9 arrive, each worth 7 more to the first waiter, there is no slot left. That is 36 on the first waiter and 8 on the second for 44, against the table's 42 and 14 for 56. The two assignments differ by a single two-for-two swap: move customers 1 and 2 out of the first waiter's five and customers 7 and 9 in, and that side gains 6; the second waiter then takes 1 and 2 in place of 7 and 9 and gains 6 as well. Twice 6 is the whole of the 12. THE SECOND TRAP IS READING THE COLUMN AS A CUSTOMER COUNT. It is one WAITER'S customer count. $dp[6][2] = 31$ at 9 customers is six customers served, two by the first waiter and four by the second; it is not two customers, and it is not four either. Every cell's value totals the tips of all i customers in its row, never just the j in its column. The third trap is treating the shape as the lower triangle those other 2D pages use. It is a triangle shaved at BOTH ends, one condition per end, and the two ends are shaved for different reasons: $col > row$ is arithmetic nonsense, more customers with one waiter than have been served at all, while $row - col > y$ is a real assignment that a stated cap forbids. Only the second one moves as the caps move. This shape does protect you against the classic index swap better than a full rectangle would, and the reason is exact rather than lucky: a live cell off the diagonal has $col < row$, so the swapped position has $col > row$ and is never a live cell, at any slider size. WHERE it lands depends on the row rather than on the slider: while the row is at most x the swap is a dot inside the table, and past that the column index runs off the declared width altogether. Measured at 9 customers, of the 24 live off-diagonal cells 14 transpose to a dot and 10, every one of them in rows 6 to 9, transpose off a table that has only 6 columns; at 4 customers it is 3 and 3. The answer cell $dp[9][5]$ is one of those ten, its transpose $dp[5][9]$ landing outside the declared columns, but nothing there is special to the answer cell.

► The painter's partition problem

STATE: WHAT ONE ENTRY MEANS

$dp[p - 1][i - 1]$ is the least time the SLOWEST of p painters can be held to when the first i boards, in the order they are given, are shared out among them and every painter takes one contiguous, unsplit block. Two numbers name the state, how many painters and how many boards, so the table is a grid rather than a row. The cell holds one time and nothing about where the cuts fell; if you want the blocks themselves you walk the table backwards afterwards. Row index is $p - 1$ and column index is $i - 1$, so the bottom right cell is the whole crew on the whole job, and every cell above or to the left of it is a smaller crew, a shorter job, or both. Watch the indices, because this page uses more than one convention. The recurrence below is written the way it reads best, $dp[p][i]$ for p painters over the first i boards, so each of its indices sits one place later than the grid coordinate it points at: the $dp[3][10]$ of the recurrence is the cell the grid labels $dp[2][9]$. Read every $dp[\dots]$ here against the words next to it. The recurrence and the derivation under it count painters and boards from 1; every reference to a cell of the grid itself, in this panel and in the step-by-step narration and the gates, uses the zero-based row and column. The code further down is a THIRD labelling, on purpose, because it is written the way you would actually type it: painters zero-based like the grid, boards one-based over a padding column like neither. Its $dp[k - 1][n]$, the grid's $dp[2][9]$ and the recurrence's $dp[3][10]$ are all the same cell, and its $dp[p][i]$ is $p + 1$ painters where the recurrence's $dp[p][i]$ is p , which is worth holding onto because the highlighted line of code and the narrated cell move together.

THE FINAL DECISION

Ask where the LAST painter's block BEGINS. If it begins after board j , that painter's time is the load of boards $j + 1$ through i , a number you add up from the board lengths and from nothing else. The other $p - 1$ painters then have to cover the first j boards as evenly as they can, which is the same question on a shorter row and a smaller crew, so $dp[p - 1][j]$ already answers it. A crew waits for its slowest member, so this cut costs the crew the LARGER of those two numbers. Every j from $p - 1$ up to $i - 1$ is a legal place for that block to begin, exactly one of them is what the best assignment did, and you want the j that makes the larger of the two numbers as small as it can be. Hence a min over maxes. Notice what the max is doing, because it is the part that reads oddly at first: it is not comparing two ways of arriving here, the way min-cost-path compares a step down against a step right. It scores ONE cut by its own worst block. The min is what compares the cuts.

RECURRENCE

```
dp[p][i] = min over j of max(dp[p - 1][j], load(j + 1, i)), for p painters  
over the first i boards
```

BASE CASES AND ORDER

Row 0 is one painter, and one painter has no decision to make: $p = 1$ means every board from 1 to i is theirs, so $dp[0][i - 1]$ is the plain total of the first i boards. That is why the whole of row 0 is a base rather than a computed cell, and why it reads as a running total, 10, 20, 25, 29 and on up to 61 at ten boards. It is also the only row that needs no comparison, which makes it the right place for a sweep to start. THE DOTS ARE NOT ZEROS, and it is worth being exact about what that does and does not buy you here. A grid has to be a rectangle, so the table is declared 3 rows by however many boards, but a cell with $i < p$ asks three painters to share two boards while every painter must paint at least one contiguous block. That has no answer rather than a cheap one, so those positions render as a dot because they DO NOT EXIST. Nothing is written to them and no cell depends on one: the split range, j from $p - 1$ up to $i - 1$, is exactly the range that stays clear of them. Now the part that is easy to get wrong in the other direction, and this page is the exception rather than the rule. On a min-over-SUM table, a zero sitting where a dot belongs is a free move and the min takes it every time: min-sum-in-a-triangle says exactly that about its own dots, and it is right to, because a descent that could stand on the empty half of that table would come back under the truth. It does not transfer here. There is exactly one place the mistake could even bite, since row 0 has no dots at all: widen the range and row 2's leftmost candidate becomes the dot at $dp[1][0]$, which the shipped range never reads. So fill every dot with a zero AND widen the split range to allow every j , and all seven answers come out unchanged, 10, 11, 14, 15, 19, 20 and 25, with not one live cell moving either. The max is what protects this table: a zero prior gets paired with the LARGEST remaining block, so $\max(0, \text{that block})$ is just the block. At $dp[1][0]$ that block is boards 2 through 10, a SUFFIX and not one of row 0's prefix totals, and it comes to 51 against a true 25 at ten boards, so the min never takes it. So the min over maxes makes the ragged half harmless, which is a property worth noticing rather than a hazard to fear, and the reason to keep the dots is that a position which does not exist is not a value, not that a zero would hand you the wrong number. The fill order needs one property and has it. Every candidate a cell reads lies in the row ABOVE and at a column strictly to the LEFT of this one, never in this row and never further right, so a row-major sweep left to right within each row has all of them final before the cell asks for them.

Time: $O(k \times n^2)$ for k painters and n boards, so $3 \times n^2$ here

Space: $O(k \times n)$, reducible to $O(n)$ with one rolling row

Watch out

Splitting by COUNT instead of by LOAD. Three painters, so give each a third of the boards: it is the first thing everyone tries, and on ONE natural way of pinning down what a third of the boards means it loses at every one of the seven slider sizes. Cutting into three blocks as equal in count as possible, with any remainder going to the earliest blocks, loses by 10, 9, 6, 10, 6, 5 and 4 from 4 boards up to 10. At 7 boards that rule puts boards 1, 2 and 3 together, 10 and 10 and 5, for 25, against a true optimum of 15. Be careful how much you conclude from that, because the remainder convention is doing some of the work. Send the remainder to the LATEST blocks instead and the same rule ties the optimum at 4 boards; take the best of EVERY cut whose three counts differ by at most one and it ties the optimum at 4 boards and at 10, while still losing by 4, 6, 5, 1 and 5 at the other five sizes. So counting boards is not a good rule with a bad tie-break. It is a rule with no reason to be right that sometimes gets lucky, and the reason is right there in the input: a board's count is always 1, while its length here runs from 2 to 10. The second trap is assuming the heaviest block sits somewhere predictable, usually that the last painter is left the smallest one. It does not. Following the cuts this table picks, the heaviest block is the third at 5, 6 and 10 boards, the second at 7 and 8, the first at 9, and tied across the first two at 4. At the slider's top the last painter carries 25 against 20 and 16. Worse for anyone hoping to name THE optimal cut: at 10 boards four different cuts all score 25, and two of them do leave the last painter the smallest block, so which cut you get is a property of your tie-break and not of the input. The third trap is reading the answer off the wrong axis. Row 2 is three painters and the last column is all the boards, and the answer is the cell where both are true. Further left in row 2 is three painters on a SHORTER job, which answers a different question and is never a LARGER number: strictly smaller from 5 boards up, and at 4 boards the single cell to the left of the answer holds 10 exactly as the answer does. Higher up in the last column is FEWER painters on this job, which is strictly larger at every slider size.

► Maximum sum rectangle

STATE: WHAT ONE ENTRY MEANS

TWO KINDS OF ROW, and reading the table means knowing which one you are standing on. A K cell is the one that holds a subproblem answer: $K[t-b][j]$ is THE LARGEST RECTANGLE SUM WHOSE ROW BAND IS EXACTLY ROWS t TO b AND WHOSE RIGHT EDGE IS EXACTLY COLUMN j . Both exactlys are doing work. The band is pinned, so a rectangle using only row 0 is not a candidate for the band 0-1 cell even though it fits inside those rows; that rectangle has its own cell, in the band 0-0 rows. The right edge is pinned, so a rectangle stopping at column $j - 1$ is not a candidate for column j either. What is NOT pinned is the left edge, and that is the only freedom a K cell has: it ranges over every start column from 0 to j . A C cell is a different animal and is not a subproblem answer at all. $C[t-b][j]$ is the total of column j over exactly rows t to b , which is arithmetic on the input rather than the answer to any question the statement asks. It exists because Kadane's scan needs one number per column, and collapsing the band is how a rectangle problem becomes a one-dimensional one. THE ANSWER IS THE LARGEST K CELL ANYWHERE IN THE TABLE, not a corner and not the end of a row, and that is structural rather than a quirk of this matrix: every rectangle in the matrix has exactly one row band and exactly one right edge, so it is counted at exactly one K cell, and the largest rectangle is therefore the largest K cell. No C cell can beat them, since $K[t-b][0]$ equals $C[t-b][0]$ and every later K cell is a max whose first candidate is its own C cell. WHICH CELL WINS MOVES, and the slider is there to show it. Across the eight column counts from 5 to 12 the answer cell is $K[0-1]$ at columns 2, 5, 5, 7, 8, 9, 10 and 10, and the answer reads 5, 7, 7, 11, 14, 21, 24 and 24. maximum-square-submatrix-of-ones says the same thing about itself and for the same reason: when the answer is an argmax rather than a corner, the table has to be searched once it is full, and a page that always pointed at the same cell would be teaching a coincidence.

THE FINAL DECISION

Stand at column j of one band and ask about the best rectangle whose right edge is column j . Its left edge is either column j itself, or something further left. If it is column j itself, the rectangle is that one column of the band and it is worth $C[t-b][j]$. If it is further left, then chopping column j off it leaves a rectangle in the same band ending at column $j - 1$, and the best such rectangle is exactly what $K[t-b][j-1]$ holds, so the whole thing is worth $K[t-b][j-1] + C[t-b][j]$. Those two cases are exhaustive and the cell takes the larger. START FRESH OR EXTEND, and the $C[t-b][j]$ term appears in both, which is what makes the decision so cheap to see: the two candidates differ by $K[t-b][j-1]$ and nothing else. So a NEGATIVE RUNNING TOTAL IS ALWAYS WORTH DISCARDING, and that is a proof rather than an instinct. If $K[t-b][j-1]$ is negative then $K[t-b][j-1] + C[t-b][j]$ is smaller than $C[t-b][j]$ whatever $C[t-b][j]$ happens to be, so the fresh start wins by exactly the amount the carried prefix was under water. It has nothing to do with the sign of the current column. Column 6 of the two-row band is the case that makes this concrete and it points the other way: the column total is -11, which is as negative as anything on the table, and the cell still EXTENDS, because the 7 to its left is positive and $7 + -11 = -4$ is better than -11. Column 7 restarts, on the same band, one column later, because the -4 it would carry is negative. The current column's sign never enters either decision. THE OTHER RECURRENCE ON THIS TABLE IS THE ROW COLLAPSE and it is not a decision at all. The column totals of the band rows 0 to 1 are the column totals of row 0 plus the column totals of row 1, added straight down, which is why the C row of the two-row band is computed here rather than handed over: the sum of a band is the sum of the bands it is made of, and the general form of that is $C[t-b][j] = C[t-(b-1)][j] + \text{matrix}[b][j]$, one row added at a time.

RECURRENCE

$K[t-b][j] = \max(C[t-b][j], K[t-b][j-1] + C[t-b][j])$ for $j > 0$, where $C[t-b][j]$ is column j totalled over rows t to b ; and for the two-row band, $C[0-1][j] = C[0-0][j] + C[1-1][j]$

BASE CASES AND ORDER

TWO ROWS AND THREE CELLS ARE BASES, 27 of the 72 cells at the full 12 columns, and each of the two kinds is a base for a different reason. THE FIRST KIND IS THE TWO SINGLE-ROW C ROWS. The column totals of a band that is one row deep are that row of the matrix, unchanged: there is nothing to add up. So $C[0-0]$ IS the matrix's first row and $C[1-1]$ IS its second, and the whole input is visible on the table before a single cell is computed. THE SECOND KIND IS THE FIRST COLUMN OF EACH K ROW, three cells in all. Kadane genuinely starts there. The only rectangle in a band whose right edge is column 0 is column 0 itself, since there is nothing to its left to extend, so $K[t-b][0] = C[t-b][0]$ with no decision to make and no cell to compare against. That is the same reason longest-increasing-subsequence declares $dp[0] = 1$ rather than computing it. WHAT IS NOT A BASE IS THE POINT OF THE PAGE. The C row of the two-row band, 12 of the 45 computed cells, is COMPUTED, from the two single-row C rows one column at a time. It would have been easy to hand that row over as data, since the matrix is right there, and the page would then have started from three collapsed bands and only ever shown Kadane. Computing it puts the row collapse the problem is named for inside the trace, with its own reads and its own gate, instead of in the setup. ORDER is row-major and it needs exactly two properties, both of which the layout was arranged to give it. A K cell at column j reads its own band's C cell in the row directly above and its own K cell one column to the left, so it needs the row above finished and the current row filled left to right. The C cell of the two-row band reads the two single-row C cells in the same column, which sit in table rows 0 and 2, so it needs the two single-row bands laid out FIRST. That is why the band order is 0-0, 1-1, 0-1 and not, say, 0-0, 0-1, 1-1: the enumeration order of the bands is free in the algorithm, and here it is spent on making the collapse a computed row.

Time: $O(m^2 \times n)$ for an m by n matrix: $m(m+1)/2$ row bands, one $O(n)$ Kadane pass each, which is 3 bands and 12 columns here

Space: $O(m^2 \times n)$ to show the whole table, reducible to $O(n)$ with one array of column totals and one running value

Watch out

Reading the end of the last K row as the answer. The answer is the largest K cell ANYWHERE, and on this table it is not the last one: at 12 columns K[0-1][11] holds 21 while the answer, 24, sits one column to its left. That is not a near miss engineered for the page either. The last cell of a K row is the best rectangle whose right edge is the last column, and there is no reason for the best rectangle in the matrix to end there; whenever the columns after the true right edge total negative, as column 11 totals -3 here, the last cell is strictly smaller. The same mistake in the code is initialising `best` after the loops instead of updating it inside them. THE SECOND MISTAKE IS SKIPPING THE SINGLE-ROW BANDS, and it is the one the picture encourages. The problem says rectangle, the method says collapse a band of rows, and it is easy to hear both as meaning the rectangle must be more than one row tall. It need not be: a one-row rectangle is a rectangle, and the bands with top equal to bottom are exactly the ones that find them. Drop them here and the answer on this matrix does not move, because the two-row band wins at every slider size, which is precisely why the failure is dangerous rather than obvious. Change the matrix and it bites, though it takes more than a single-row optimum to make it bite, and the near miss is worth seeing first. On $[[1, -3, 4, 2, -1, 5], [-2, 6, -4, 3, 1, -3]]$ the best rectangle IS a single row, columns 2 to 5 of row 0 summing to 10, and a solver skipping the single-row bands still returns 10, because that 10 is TIED: columns 1 to 5 of both rows reach it too. A tie rescues the broken solver. On $[[-5, 3, 1, -2, 6, -6], [-5, -4, 1, 1, -4, -1]]$ nothing rescues it. The best rectangle there is columns 1 to 4 of row 0, summing to 8, it is the unique rectangle reaching 8, and the best rectangle using both rows is only 3, so a solver that skipped the single-row bands returns 3 against a true 8. Check that by brute force over the four corners rather than taking it from here, and check the uniqueness too, since it is the half that makes the example work. THE THIRD IS SEEDING THE RUNNING TOTAL WITH ZERO, which is the same bug wearing a rectangle costume. Write ``running = Math.max(0, running + colTotal[j])`` and you have allowed the EMPTY rectangle, worth 0, as a candidate. On a matrix with a positive answer nothing goes wrong and the bug ships. On an all-negative matrix every answer comes back 0, which is not the sum of any rectangle at all. On this very matrix, take the leftmost 2 columns: every rectangle inside them is negative, the true answer is -2, the single cell in row 1 column 1, and the zero-seeded version returns 0. The slider starts at 5 columns so the page never shows that case; the bug is still there at 5 columns, waiting for a different matrix. THE FOURTH IS COLLAPSING THE BAND FROM SCRATCH FOR EVERY BAND, which costs correctness nothing and time a whole factor of m. Adding the band rows up afresh for each of the $m(m+1)/2$ bands is $O(m)$ per column per band; carrying the previous band's totals and adding one matrix row is $O(1)$. This table does the cheap version in plain sight: $C[0-1]$ is $C[0-0]$ plus $C[1-1]$, read off the table, not re-added from the matrix.

Check your understanding

1. How many right/down routes cross a 3 by 3 grid when the center cell is blocked?

- A. 0
- B. 2
- C. 4
- D. 6

2. Remove that obstacle. How many right/down routes cross the same 3 by 3 grid?

- A. 3
- B. 4
- C. 6
- D. 9

Flashcards: cover the answer to recall it

What does `ways[r][c]` count?

The number of valid routes from the start to cell (r, c) , using the allowed directed moves.

What value should a blocked route-count cell store?

Zero. No valid route enters that cell, including when a rolling array still contains an earlier row's value.

Why does top-to-bottom, left-to-right order work for right/down moves?

Every cell depends only on the cell above and the cell to its left, both of which are already solved.

Quiz answer key

1. B. The center contributes 0. The final cell receives 1 route from above and 1 from the left, giving 2.

2. C. The bottom-right cell adds its upper count 3 and its left count 3. There are 6 routes.

Two strings: compare two prefixes

A pair of prefix lengths records how much of each string has been considered. The same rectangular shape can represent a longest common subsequence or an edit distance; the state sentence determines the recurrence.

The idea to remember

- For longest common subsequence, $L[i][j]$ is the longest subsequence shared by the first i characters of A and the first j of B . A subsequence preserves order but may skip characters.
- If $A[i - 1]$ equals $B[j - 1]$, use $1 + L[i - 1][j - 1]$. Otherwise use $\max(L[i - 1][j], L[i][j - 1])$ to skip one unmatched last character.
- Set row 0 and column 0 to 0 because an empty string shares no nonempty subsequence. Fill prefix lengths upward so the upper, left, and diagonal cells are ready.
- For edit distance, $E[i][j]$ counts the fewest insertions, deletions, and replacements. Its bases are $E[i][0] = i$ and $E[0][j] = j$, not all zeros.
- Do not confuse a subsequence with a substring, which must be contiguous. When compressing to one row, preserve the previous diagonal before overwriting it.

Work through one example

Find the longest common subsequence of "ab" and "acb".

Including the empty prefixes, L has rows $[0, 0, 0, 0]$, $[0, 1, 1, 1]$, and $[0, 1, 1, 2]$. Matching the final b extends the earlier a match from length 1 to length 2.

2 characters

Worked problem notes (7)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Longest Common Substring

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the length of the longest common substring that ENDS exactly at A's i -th letter and B's j -th letter: the longest run of letters that A's first i letters and B's first j letters both finish with. Here $m = 7$ is A's length and n is how many letters of B the slider takes, 10 at the drive size. Row i 's own letter is $A[i - 1]$ and column j 's is $B[j - 1]$, which is why row 5's gutter reads 5 D. Pinning the run's END at both i and j is the whole insight. The cell on the longest-common-subsequence page holds the best answer found anywhere within the two prefixes, so it can lean on a neighbour that found a good answer earlier. This cell cannot, because a run that ends here is either glued to the run ending one letter back on both strings or it does not exist. $dp[5][8] = 5$ says that the five letters ending at A's D and at B's D agree, XABCD in both, and it says nothing about any run that ends somewhere else.

THE FINAL DECISION

A common substring that ends at this pair has to use both last letters, $A[i - 1]$ and $B[j - 1]$, since it ends exactly on them. So there is no choice to make here, only a test. If the two letters differ, no run ends on this pair at any length, and the cell is 0. If they match, the run is this pair plus whatever run ended one letter earlier in both strings, $dp[i - 1][j - 1]$, and nothing else. The cell above ends its run on the same letter of B that this pair uses, and the cell to the left ends on the same letter of A, so this pair cannot extend either of them; only the diagonal stops one letter short in both. Hold that against LCS. A subsequence may skip letters, so LCS's mismatch branch drops the last letter of one string and keeps $\max(dp[i - 1][j], dp[i][j - 1])$. A substring may not skip anything, so the same mismatch here resets the cell to 0.

RECURRENCE

$$dp[i][j] = dp[i - 1][j - 1] + 1 \text{ if } A[i - 1] = B[j - 1], \text{ else } 0$$

BASE CASES AND ORDER

Row 0 and column 0 are given as 0, because no run can end on a pair where one of the two prefixes is empty: that side has no letter to pair with. At the drive size that is 18 bases: 11 across row 0 for columns 0 to 10, and 7 more down column 0 for rows 1 to 7. A computed cell reads at most one cell, the diagonal up and to the left, and a row-major sweep, left to right within each row, has always finished it, since the sweep completed the whole previous row first. A mismatch reads nothing at all. THE RESET IS WHY THE ANSWER IS A SCAN. On the LCS page no cell is ever smaller than the cell above it or the cell to its left, so the last cell holds the answer. Here a mismatch drops a cell to 0 however long the run beside it, so the last cell reports only on the run that ends at both strings' final letters: $dp[7][10] = 1$, the shared Z, against the answer 5 at $dp[5][8]$. At lengths 6 to 9 the last cell holds 0. The answer is the largest cell anywhere in the table, so a running best has to travel with the sweep.

Time: $O(m \times n)$

Space: $O(m \times n)$, reducible to $O(n)$ with one row swept right to left, or $O(1)$ by walking each diagonal with a counter

Watch out

Carrying LCS's mismatch rule over: writing $\max(dp[i-1][j], dp[i][j-1])$ where this page writes 0. The table that rule builds is the LCS table, and on this pair it hides well. Read as its last cell or as its largest cell, which is the same number in that table, it returns 3, 4, 5, 5 at lengths 6 to 9, right at every one, because up to length 9 the longest common subsequence of these strings happens to be contiguous. At length 10 it returns 6, for XABCDZ, which is not a substring of either string: in A an E sits between the D and the Z, and in B an A does. One slider position in five catches it, so test on an input where the subsequence and the substring differ. The second trap is LCS's other habit, returning $dp[m][n]$; here that gives 0, 0, 0, 0, 1 across the five lengths, wrong at every one.

► Longest Common Subsequence

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the length of the longest common subsequence of A's first i letters and B's first j letters. Two numbers name the state, one prefix length per string, so with $m = |A|$ and $n = |B|$ the table is an $(m+1) \times (n+1)$ grid, one extra row and one extra column for the empty prefixes. Row i 's own letter is $A[i-1]$ and column j 's is $B[j-1]$, which is why row 3's gutter reads 3 C: the 3 is the dp index and the C is the letter that row adds. The cell holds a length and nothing about which letters achieved it.

THE FINAL DECISION

The full derivation, why a matching last pair can be taken together at no risk and why a mismatch has to drop the last letter of one string or the other, lives in the flagship lesson at </courses/dynamic-programming/longest-common-subsequence>. This page assumes it and goes straight to the table.

RECURRENCE

```
dp[i][j] = dp[i-1][j-1] + 1 if A[i-1] = B[j-1], else max(dp[i-1][j], dp[i][j-1])
```

BASE CASES AND ORDER

Row 0 and column 0 are given as 0, because an empty prefix shares nothing with any prefix of the other string. At the drive size that is 14 bases: 7 across row 0 for columns 0 to 6, and 7 more down column 0 for rows 1 to 7. Every interior cell reads only cells above it, to its left, or diagonally above-left, so a row-major sweep, left to right within each row, has finished all three before any of them is needed. The diagonal costs nothing extra: the sweep completed the whole previous row before it started this one.

Time: $O(m \times n)$ for $|A| = m$ and $|B| = n$

Space: $O(m \times n)$, reducible to $O(\min(m, n))$ with two rolling rows

Watch out

Adding 1 to the wrong cell on a match. The match branch reads the DIAGONAL, $dp[i - 1][j - 1]$, because pairing the two letters uses both of them up. Writing $\max(dp[i - 1][j], dp[i][j - 1]) + 1$ instead looks like a harmless generalisation, and on this pair it first goes wrong at $dp[1][6]$. That cell pairs A's first letter, an A, with B's sixth letter, also an A. Its left neighbour $dp[1][5] = 1$ already spends that same A, matched against B's fourth letter, so $\max(0, 1) + 1$ writes 2 and claims a two-letter common subsequence from a one-letter prefix of A. The diagonal $dp[0][5] = 0$ is the honest starting point and gives 1. The error compounds from there: the wrong rule finishes this table at 5 instead of 4.

► Edit Distance

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the fewest single-letter edits (insert, delete, replace) that turn the first i letters of $A = \text{SUNDAY}$ into the first j letters of B , the target. Here $m = 6$ is A's length and n is how many letters of SATURDAY the slider takes, 8 at the drive size. Row i 's own letter is $A[i - 1]$ and column j 's is $B[j - 1]$, which is why row 4's gutter reads 4 D and column 3's reads 3 T. The edits run one way, A into B, so a delete removes a letter of A and an insert adds a letter of B. Swap the two words and the number stays the same, but deletes and inserts trade names, and so do the table's two edges.

THE FINAL DECISION

Line the two prefixes up the way a cheapest script treats them, each surviving letter of A above the letter of B it becomes, and look at the last column of that alignment. It has one of three shapes. A's last letter sits over B's last letter: kept for free if they are equal, replaced for 1 if not, and the rest turns A's first $i - 1$ letters into B's first $j - 1$, so the diagonal, $dp[i - 1][j - 1] + \text{cost}$. A's last letter sits over nothing: it is deleted, for 1, and the rest turns A's first $i - 1$ letters into all j of B's, so the cell above, $dp[i - 1][j] + 1$. B's last letter sits under nothing: it is inserted, for 1, and the rest turns all i of A's letters into B's first $j - 1$, so the cell to the left, $dp[i][j - 1] + 1$. The cell is the smallest of the three, read in that order: diagonal, up, left, which is keep or replace, delete, insert. On a match the diagonal cannot lose. One extra letter on either word moves the distance by at most 1, since one more delete or insert covers it, so $dp[i - 1][j - 1]$ is at most $dp[i - 1][j] + 1$ and at most $dp[i][j - 1] + 1$. This page still reads all three cells at a match and lets the min show it.

RECURRENCE

```
dp[i][j] = min(dp[i - 1][j - 1] + cost, dp[i - 1][j] + 1, dp[i][j - 1] + 1),
where cost = 0 if A[i - 1] = B[j - 1], else 1
```

BASE CASES AND ORDER

Row 0 and column 0 are the cost against an empty word, and they are given rather than computed. $dp[0][j] = j$, because turning A's empty prefix into B's first j letters takes j inserts: A has no letter to keep or replace, and each edit adds at most one letter. $dp[i][0] = i$, because turning A's first i letters into nothing takes i deletes. At the drive size that is 15 bases: 9 across row 0 for columns 0 to 8, and 6 more down column 0 for rows 1 to 6. These are not LCS's zeros, and borrowing LCS's zeros is the second trap in the pitfall below. The code panel highlights line 4, the row-0 loop, for every base step, so the bases down column 0, which line 5 writes, light up the line above their own: 6 of the 15 base steps at the drive size, and 6 at every slider length, since SUNDAY keeps its 6 letters. A page names one code line per kind of step, and the two loops stay apart because that is the order the trace writes the bases in, row 0 first and then column 0. Each interior cell reads the cell diagonally up and to the left, the cell above and the cell to its left, so a row-major sweep, left to right within each row, has finished all three before any of them is needed.

Time: $O(m \times n)$

Space: $O(m \times n)$, reducible to $O(\min(m, n))$ with two rolling rows

Watch out

Dropping the + 1 from the diagonal on a mismatch, so the diagonal is $dp[i - 1][j - 1]$ whatever the letters and every replacement is free. At length 6 the result is plainly absurd: SUNDAY and SATURD both have six letters, the free diagonal runs from $dp[0][0]$ to $dp[6][6]$ without paying once, and the table answers 0, as if the two words were already equal. The real answer there is 5. Across the slider the broken rule gives 3, 2, 1, 0, 1, 2 against the real 4, 5, 5, 5, 4, 3, wrong at every length, including 2 against 3 at the drive size. The second trap is borrowing LCS's bases. A row 0 and column 0 of zeros say an empty word grows into SATURDAY for free, and the table built on them gives 2, 3, 4, 4, 3, 2, also wrong at every length.

► Longest Common Increasing Subsequence

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$, defined only where $A[i] = B[j]$, is the length of the longest common increasing subsequence that ENDS on that value, taken at position i of A and position j of B . Positions count from 0, and unlike longest-common-subsequence there is no empty-prefix row or column: $dp[i][j]$ speaks about $A[i]$ and $B[j]$ themselves, which is why row 3's gutter reads 3 4, position 3 of A holding the value 4. Pinning the end is what makes a recurrence possible, and it forces the match. The last value of a common subsequence sits at some position in A and some position in B , and it is the same value in both, so a chain can only end on a pair where $A[i] = B[j]$. Every other position is a dot, and a dot is not a 0: it is a pair of positions on which no common increasing subsequence can end at all. This is longest-increasing-subsequence lifted to pairs, with longest-common-subsequence's matching condition added: LIS's $dp[i]$ is the longest increasing chain ending at index i of one array, and here the chain has to end at a position in each array at once. A cell holds a length, not the chain, and since the best chain can end on any matched pair, the answer is the largest cell anywhere.

THE FINAL DECISION

Ask what came immediately before $A[i]$ in the best chain that ends here. Either nothing did, and this pair starts a chain of its own, of length 1, or some earlier matched pair (p, q) did. That pair has to come strictly before this one in BOTH arrays, $p < i$ and $q < j$, because a subsequence keeps each array's order, and its value has to be strictly smaller, $A[p] < A[i]$, because the chain increases. Any chain ending at (p, q) can be extended by this pair, and the longest such chain is what $dp[p][q]$ already holds, so each qualifying pair is a candidate worth $dp[p][q] + 1$, and the cell keeps the largest. Each candidate is one cell, the same one-cell-per-candidate shape longest-increasing-subsequence uses, with the scan running over a rectangle above and to the left instead of a stretch of one array.

RECURRENCE

$dp[i][j] = 1 + \max(dp[p][q])$ over matched pairs with $p < i$, $q < j$ and $A[p] < A[i]$, or 1 if there are none (defined only where $A[i] = B[j]$)

BASE CASES AND ORDER

No bases. Every live cell is computed, including the ones with no candidate, which compute 1 from nothing and consult no other cell: at the drive size those are $dp[0][0]$, A's first item, with nothing before it in A, and $dp[4][3]$, the 1, with no smaller value anywhere above and to the left of it. A row-major sweep, left to right within each row, is enough, because every candidate sits in a strictly earlier row and the sweep finished that row before starting this one. Sitting strictly to the left as well is what the problem demands, since B's order has to be kept; the sweep does not need it. The flip side is that a cell in the last row is never a candidate for anything, since no row comes after it: $dp[6][1] = 2$, the chain 2, 7, is a real state that no other cell uses. The answer is the largest cell anywhere, so the sweep keeps the largest value written so far. It is not the last cell, which is a dot at four of the five slider lengths.

Time: $O(m^2 \times n^2)$ as drawn, $O(m \times n)$ with a running best per row of A

Space: $O(m \times n)$, or $O(n)$ for the running-best version

Watch out

Dropping the increasing condition: counting every matched pair above and to the left as a candidate, whatever its value. What that computes is a longest common subsequence, built over matched pairs, and on this input it first goes wrong at $dp[4][3]$. $A[4] = B[3] = 1$, and $dp[2][2] = 2$, the chain 2, 6, sits above and to the left, so the cell writes 3 and claims 2, 6, 1: common to both arrays, but not increasing. The correct value is 1. $dp[5][4]$ follows it to 4, for 2, 6, 1, 5, against the correct 2. At length 7 the wrong rule returns 4 against the correct 3. At lengths 8 to 11 it returns 4, 4, 5, 5, the right numbers, because from length 8 on one of the longest common subsequences happens to be increasing (2, 3, 4, 5, then 2, 3, 4, 5, 7). So the answer exposes the bug at one slider length out of five, and at the drive size the answer is right while those two cells are wrong. The second trap is longest-common-subsequence's habit of returning the last cell: here that position is a dot at lengths 7, 8, 9 and 11.

► Wildcard Pattern Matching

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is 1 if the first i characters of the pattern p match the first j letters of the text t entirely, both prefixes used up, and 0 if not. Here $m = 5$ is the pattern's length and n is how many letters of `abycbxcbzca` the slider takes, 10 at the drive size. Row i 's own character is $p[i - 1]$ and column j 's letter is $t[j - 1]$, which is why row 2's gutter reads `2 *` and column 4's reads `4 c`. The answer is yes or no, so the page writes each cell as 1 or 0. It is a whole-prefix match, pinned at both ends, rather than "the pattern occurs somewhere in the text", and that is what lets a cell be built from smaller ones. Peel the last character off the pattern, and what remains has to account for every letter before the ones that character took, no more and no fewer. A somewhere-answer cannot promise that: `b?c` occurs in `abycb`, at letters 2 to 4, yet `a*b?c` does not match `abycb`, because the pattern has to end where the text does and nothing after its `c` can take the final `b`.

THE FINAL DECISION

Look at the pattern's last character, $p[i - 1]$, and ask which text letters it takes. A letter or a `?` takes exactly one, and because it is last, the one it takes is the text's last letter, $t[j - 1]$. That works if the two agree, which is what `ok` records (a `?` agrees with any letter), and if the shorter prefixes, the pattern's first $i - 1$ characters and the text's first $j - 1$ letters, match entirely, which is the diagonal. Both have to hold. A star takes a run of letters at the end of the text, possibly an empty run. Either it matches nothing more, and the pattern's first $i - 1$ characters must match all j letters: the cell above, $dp[i - 1][j]$. Or it swallows at least one letter, and because the star is last, its run ends with $t[j - 1]$; set that letter aside, and the same star, still the pattern's last character, takes whatever is left of its run, possibly nothing. So the first i characters must match the first $j - 1$ letters: the cell to the left, $dp[i][j - 1]$, in the SAME row, because swallowing a letter does not use the star up. That left cell has already weighed the star taking two, three or more letters, which is why two reads cover a run of any length. Either case is enough. On 1s and 0s, both-have-to-hold (AND) is the smaller of the two, min, and either-is-enough (OR) is the larger, max. The page writes them that way, so each computed cell's narration ends in a min or a max, and a star cell lists the cell above first and the cell to the left second, in its gates and compare labels too.

RECURRENCE

```
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) if p[i - 1] is *, else min(ok,
dp[i - 1][j - 1]), where ok = 1 if p[i - 1] is ? or equals t[j - 1], else 0
```

BASE CASES AND ORDER

Row 0 and column 0 are given rather than computed. $dp[0][0] = 1$: the empty pattern matches the empty text. $dp[0][j] = 0$ for j from 1 up: the empty pattern takes no letters, so a text with a letter in it is out of reach. Column 0 is the pattern against the empty text, and a star is the one character that can match nothing, so $dp[i][0] = 1$ exactly when the pattern's first i characters are all stars. That is a closed form of the pattern alone, not a recurrence, and the snippet writes it that way. On `a*b?c` rows 1 to 5 of column 0 are 0, row 2's `a*` included, because the `a` needs a letter. At the drive size that is 16 bases: 11 across row 0 for columns 0 to 10, and 5 more down column 0 for rows 1 to 5. A star cell reads the cell above and the cell to its left in its own row; a letter or `?` cell reads the diagonal, up and to the left. A row-major sweep, left to right within each row, has finished each of those before it is needed, and the left-to-right part is what the star depends on: its left neighbour was written one step earlier in the same row.

Time: $O(m \times n)$

Space: $O(m \times n)$, reducible to $O(n)$ with two rolling rows

Watch out

Letting the star read the diagonal instead of its left neighbour: $\max(dp[i - 1][j], dp[i - 1][j - 1])$. It looks like the letter and ? rows, which read the diagonal too, but the diagonal moves the pattern on past the star, so the star can take nothing or exactly one letter and is then spent; it has turned into a ? that may be skipped. On this input row 2 then holds 1 at columns 1 and 2 and 0 from column 3 on, where the correct row 2 is 1 at each column from 1 up. The slider gives 1, 0, 0, 0, 0, 0, 0, 0 instead of 1, 0, 0, 1, 0, 0, 1, 0: right at length 4, where the star takes nothing, and wrong at length 7 and at the drive size 10, where it has to take three letters and six. The left read stays in the star's own row so that the star can keep taking letters.

► Regular Expression Matching

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is 1 if the first i characters of the pattern p match the first j letters of the text t entirely, both prefixes used up, and 0 if not: the same yes-or-no state as wildcard pattern matching, written as 1 or 0 so that both-must-hold is a min and either-will-do is a max. Here $m = 8$ is the pattern's length and n is how many letters of aabxbcyd the slider takes, 8 at the drive size. Row i 's own character is $p[i - 1]$ and column j 's letter is $t[j - 1]$, which is why row 4's gutter reads 4 * and column 6's reads 6 c. The twist is that * has no meaning of its own. It belongs to the character in front of it, and the unit the pattern is made of is x^* , zero or more copies of x . So $c^*a^*b^*c$ is five units, c^* , a^* , b^* , $.$ and c , over eight rows, and a row that ends on a letter whose star sits on the next row, such as row 3, c^*a , is an honest state of its own: a pattern whose a has not got its star yet. As in wildcard matching, the match is pinned at both ends rather than found somewhere. The text's first 7 letters, aabxbcy, contain a match of the whole pattern, aabxbc, and still $dp[8][7] = 0$, because the y is left over and no unit takes it. Being pinned is what lets a cell lean on smaller ones: peel the last unit off the pattern, and what remains has to account for exactly the letters in front of the ones that unit took.

THE FINAL DECISION

Look at the pattern's last unit and ask which text letters it takes. If $p[i - 1]$ is a letter or a $.$, the unit is that one character, and it takes exactly one letter, the text's last, $t[j - 1]$. That works if the two fit, which is what fits records ($a.$ fits any letter), and if the shorter prefixes, the first $i - 1$ characters and the first $j - 1$ letters, match entirely, which is the diagonal. Both have to hold: $\min(\text{fits}, dp[i - 1][j - 1])$. If $p[i - 1]$ is a $*$, the unit is x^* , two characters, with $x = p[i - 2]$, and one of two things happens. Either it matches zero copies of x , so the whole unit drops out, letter and star together, and the first $i - 2$ characters must match all j letters: the cell TWO rows up, $dp[i - 2][j]$. Or it takes $t[j - 1]$ as one more copy of x , which needs $t[j - 1]$ to fit x , and stays available for any copies in front of that one, so the first i characters, star included, must match the first $j - 1$ letters: the cell to the left in the SAME row, $dp[i][j - 1]$. Either is enough, so the cell is the max of the two, and the second is itself a min, $\min(\text{fits}(x, t[j - 1]), dp[i][j - 1])$ against the left cell. This is where it parts from wildcard matching. Wildcard's $*$ stands alone and takes any letters, so stopping it drops one character and reads ONE row up, and swallowing a letter needs no check. Here the $*$ is the back half of x^* , so dropping it drops its letter too and reads TWO rows up, and taking another letter has to check that letter against x . A star cell lists the cell two rows up first and the cell to its left second, in its gates and compare labels too.

RECURRENCE

```
dp[i][j] = max(dp[i - 2][j], min(fits(p[i - 2], t[j - 1]), dp[i][j - 1])) if  
p[i - 1] is *, else min(fits(p[i - 1], t[j - 1]), dp[i - 1][j - 1]), where  
fits(x, y) = 1 if x is . or equals y, else 0
```

BASE CASES AND ORDER

Row 0 and column 0 are given rather than computed. $dp[0][0] = 1$: the empty pattern matches the empty text. $dp[0][j] = 0$ for j from 1 up: the empty pattern takes no letters. Column 0 is the pattern against the empty text, and an x^* unit is the one thing that can match nothing, so $dp[i][0] = 1$ exactly when the first i characters cut into x^* units throughout: i is even and the 2nd, 4th and every later even-position character up to i is a $*$. That is a closed form of the pattern alone, and the snippet writes it that way. On $c^*a^*b^*.c$ it is 1 at rows 2 and 4 (c^* and c^*a^*) and 0 at rows 1, 3, 5, 6, 7 and 8: rows 1 and 3 end on a letter whose star is not in the prefix yet, and from row 5 down the bare b needs a letter. At the drive size that is 17 bases: 9 across row 0 for columns 0 to 8, and 8 more down column 0. A star cell reads two rows up and to its left in its own row; a letter or $.$ cell reads the diagonal. A row-major sweep, left to right within each row, has written each of those before it is needed. The two-row jump has a consequence you can see in the table: rows 1, 3 and 6, the c , the a and the $.$ that a star follows, are filled like the rest, yet no later cell reads any of them. The star row below each one jumps over it to the row above, and checks its letter against the text directly rather than through a cell. Their 1s, such as $dp[3][1]$ (c^*a matches a), are true and go unused; taking exactly one copy of x is covered by the star row instead, through its left read.

Time: $O(m \times n)$

Space: $O(m \times n)$, reducible to $O(n)$ with three rolling rows

Watch out

Treating this $*$ like wildcard matching's: stopping the star reads $dp[i - 1][j]$, one row up, and column 0 is 1 only under a run of stars. Row $i - 1$ ends on the starred letter x itself, so zero copies has quietly become exactly one, and with the left read adding more, every x^* now means one or more copies. On this input the pattern turns into $c+a+b.+c$, which needs the text to open with a c ; this one opens with a , so at the drive size every cell below row 0 is 0, and the slider gives 0 at all six lengths where the right answers are 0, 0, 1, 0, 1, 0. Half the mistake can hide. Keep column 0's correct closed form and change only the star's upward read, and at the drive size it changes one cell, $dp[7][3]$ ($*$ taking nothing after aab), from 1 to 0, and no answer on the slider moves. Dropping a star means dropping its letter with it, which is why the read goes two rows up.

► Longest repeating and non-overlapping substrings

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$, for $i < j$, is the length of the longest run of letters that ends exactly at letter i AND ends exactly at letter j , capped at $j - i$ so that the copy ending at letter i finishes before the copy ending at letter j begins. Put another way, it is the longest common suffix of s 's first i letters and its first j letters, clipped to the gap between their ends. The two axes are the same string: s is the first n letters of `abababab`, $n = 8$ at the drive size, the row index says where the earlier copy ends and the column index says where the later copy ends. Row i 's own letter is $s[i - 1]$ and column j 's is $s[j - 1]$, which is why row 4's gutter reads `4 b`. A pair with $i > j$ names the same two copies as (j, i) with their roles swapped, so it is not a separate state, and those cells are left out of the table. $dp[4][8] = 4$ says that the four letters ending at letter 4 and the four ending at letter 8 agree, `abab` both times, and that the two copies, letters 1 to 4 and 5 to 8, do not overlap. It says nothing about a run that ends anywhere else.

THE FINAL DECISION

A repeated run that ends at this pair has to use both last letters, $s[i - 1]$ and $s[j - 1]$. If they differ, no run ends on this pair and the cell is 0. If they match, the run is this pair glued to the run that ended one letter earlier at both places, $dp[i - 1][j - 1] + 1$, the same step longest-common-substring takes. What is new is the cap. A copy of length k ending at letter j covers letters $j - k + 1$ to j , and the earlier copy ends at letter i , so the two do not overlap exactly when $j - k + 1 > i$, which is $k \leq j - i$. So the cell keeps the smaller of the extended run and the gap: $\min(dp[i - 1][j - 1] + 1, j - i)$. Capping, rather than throwing the run away, loses nothing: if the k letters ending at letter i and the k ending at letter j agree, so do the last $j - i$ of each, and those two copies fit. The cap also passes down a diagonal intact, because $j - i$ is the same number at each cell of one diagonal; once a run reaches its diagonal's gap, the cell stays at the gap for as long as the letters keep matching. The gap is read off the cell's own coordinates, not out of another cell.

RECURRENCE

```
dp[i][j] = min(dp[i - 1][j - 1] + 1, j - i) if s[i - 1] = s[j - 1], else 0
(for i < j)
```

BASE CASES AND ORDER

Two kinds of base, both 0. Row 0, across columns 0 to n : the earlier prefix is empty, so it has no letter to end a run on. The main diagonal, $dp[i][i]$ for i from 1 to n : a letter set against itself, where the gap $j - i$ is 0, so the cap leaves no room for a run of even one letter. At the drive size that is 9 bases in row 0 and 8 on the diagonal, 17 in all. The cells below the diagonal are left out, which is why column 0 holds only $dp[0][0]$ and row n holds only $dp[n][n]$. No computed cell reads a diagonal base: a cell (i, j) with $j > i$ reads at most $dp[i - 1][j - 1]$, whose column is still ahead of its row, so that cell sits in row 0 or above the diagonal. The diagonal bases are in the table to say what the pair (i, i) is, not to feed anything. Row 0 is read in only a few places: $dp[0][j]$ is read only by $dp[1][j + 1]$, and only when letter 1 matches letter $j + 1$, which at the drive size means $dp[0][2]$, $dp[0][4]$ and $dp[0][6]$. A row-major sweep, left to right within each row, finishes the previous row before it starts the next, so the diagonal neighbour is final whenever a match needs it, and a mismatch needs nothing. As on longest-common-substring, the answer is the largest cell anywhere, not the last one; here the last cell, $dp[n][n]$, is a diagonal base and holds 0 at every size.

Time: $O(n^2)$

Space: $O(n^2)$, reducible to $O(n)$ with one row swept right to left, or $O(1)$ by walking each diagonal with a counter capped at that diagonal's gap

Watch out

Dropping the cap: writing $dp[i - 1][j - 1] + 1$ on a match, as longest-common-substring does. That table measures the longest substring that occurs twice with overlap allowed, and on this string overlapping repeats are common: it returns 2, 3, 4, 5, 6 across lengths 4 to 8 against the correct 2, 2, 2, 3, 4, right only at length 4. At length 8 it returns 6, from $dp[6][8]$, for ababab at letters 1 to 6 and again at letters 3 to 8, two copies that share letters 3 to 6. The cap and the $j > i$ bound each keep a letter from counting as a repeat of itself, and either one alone is enough: with the cap, $dp[i][i] = \min(dp[i - 1][i - 1] + 1, 0) = 0$, and with the bound the main diagonal is never computed. Drop both, letting j start at i rather than $i + 1$ with no cap, and the main diagonal climbs $dp[i][i] = i$, reporting the whole string, 8 letters at length 8, as its own repeat. The third trap is longest-common-subsequence's habit of returning the last cell; here that is $dp[n][n]$, a diagonal base, which gives 0 at every length.

Check your understanding

1. What is the LCS length of "ab" and "acb"?

- A. 0
- B. 1
- C. 2
- D. 3

2. With unit-cost edits, how many insertions transform an empty string into "cat"?

- A. 0
- B. 1
- C. 2
- D. 3

Flashcards: cover the answer to recall it

Which characters does $L[i][j]$ compare at the current boundary?

$A[i - 1]$ and $B[j - 1]$. The state indices are prefix lengths, while string indices start at zero.

How is a subsequence different from a substring?

A subsequence can skip characters while preserving their order. A substring must occupy consecutive positions.

Why are the edit-distance empty-prefix bases not zero?

Transforming a prefix of length i into empty needs i deletions. Transforming empty into a prefix of length j needs j insertions.

Quiz answer key

1. **C.** Both strings contain a followed by b. That gives length 2, and the first string has only 2 characters, so no longer answer is possible.
2. **D.** Insert c, then a, then t. Three characters require 3 insertions, which explains the edit-distance base $E[0][3] = 3$.

Subset and knapsack: take or leave an item

Track which items have been considered and how much capacity remains or has been used. The key distinction is whether an item can appear once or can be reused.

The idea to remember

- For 0/1 knapsack, $\text{best}[i][c]$ is the greatest value using the first i items with total weight at most capacity c . Each item may be chosen at most once.
- For weight w and value v of item $i - 1$, compare skipping, $\text{best}[i - 1][c]$, with taking, $\text{best}[i - 1][c - w] + v$, if w fits.
- With positive weights, nonnegative values, and an at-most capacity, zero items or zero capacity have value 0. An exact-fill problem needs a different base: only weight 0 is initially reachable.
- In a one-dimensional 0/1 table, visit capacities downward. This keeps $\text{best}[c - w]$ from the previous item. Upward iteration can reuse the same item and instead models an unbounded choice.
- Subset sum stores reachability rather than value: $\text{reachable}[0] = \text{true}$. A false state means no considered subset reaches that exact total; it does not mean the best value is zero.

Work through one example

0/1 items (weight, value): (2, 3), (3, 4). Capacity 5.

After the first item, capacities 0 through 5 hold [0, 0, 3, 3, 3, 3]. For the second item at capacity 5, compare skipping value 3 with taking value $4 + \text{best}[2] = 4 + 3 = 7$.

7 maximum value

Worked problem notes (8)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Subset Sum Problem

STATE: WHAT ONE ENTRY MEANS

$\text{dp}[i][s]$ is 1 if some subset of the first i numbers adds up to exactly s , and 0 if none does. Two numbers name the state, how many of the numbers you may use and which total you are asking about, so the table is a grid: row 0 for no numbers, then one row per number, and one column per total from 0 up to the target. A cell is a yes or a no. It is not a count, and it does not record which numbers made the total.

THE FINAL DECISION

Ask about the last number allowed, a_i . Every subset of the first i numbers either leaves it out or puts it in, and there is no third case. Leave it out, and the rest is a subset of the first $i - 1$ numbers adding to s : that is the cell straight above. Put it in, and the rest is a subset of the first $i - 1$ numbers adding to $s - a_i$: that is the cell a_i columns to the left in the row above. If either one says yes, s is reachable. On a table of 1s and 0s, either one being 1 is the same thing as the larger of the two being 1, so the animation writes the or as a max. When a_i is bigger than s , putting it in is impossible and the cell copies the one above. The flagship coin-change lesson teaches the same habit of asking what the last decision had to be; there the decision was which coin came last, here it is whether the last number is in.

RECURRENCE

$dp[i][s] = dp[i - 1][s] \text{ or } dp[i - 1][s - a_i]$, the second only when $a_i \leq s$

BASE CASES AND ORDER

Row 0 is the only base row. With no numbers allowed, the empty subset is the only subset, so $dp[0][0] = 1$ and the rest of row 0 is 0. Column 0 is not a base: it computes, and comes out 1 all the way down, because leaving every number out still makes 0. Fill row by row from the top. Both cells a cell reads sit in the row above, so they are final before anything reads them. It has to be the row ABOVE for the put-it-in case, not the current row: the current row's cell at $s - a_i$ may already have used a_i , and reading it would let one number appear twice.

Time: $O(n \times \text{target})$

Space: $O(n \times \text{target})$, reducible to $O(\text{target})$ with one row swept right to left

Watch out

Letting a number be used twice. The put-it-in case must read $dp[i - 1][s - a_i]$, from the row above. Read $dp[i][s - a_i]$ from the current row instead and the 3 can join a subset that already holds the 3: on this set that answers yes at target 6, from $3 + 3$, and yes at target 9, from $3 + 3 + 3$, where the true answer at both is no. That one changed row index is the entire difference between this problem and its use-any-number-again version, the shape coin-change-count-ways uses, so the bug produces a correct program for the wrong problem and nothing crashes. In the single-row version the same bug is sweeping s from left to right.

► Coin change problem – Count Ways

STATE: WHAT ONE ENTRY MEANS

$dp[i][s]$ is the number of combinations of the first i coin denominations, each usable as many times as you like, that add up to exactly s . A combination is how many of each coin, never the order they are laid down in. Two numbers name the state, how many denominations are allowed and the amount, so the table is a grid: row 0 for no coins, then one row per coin, and one column per amount.

THE FINAL DECISION

Split the combinations that make s by whether they use the last allowed coin, c_i , at all. Those that use none of it are combinations of the first $i - 1$ coins making s : the cell above. Those that use at least one: take one c_i away and what is left is a combination making $s - c_i$ in which c_i is STILL allowed, because there may be more of it. That is the cell c_i columns to the left in the same row. Every combination lands in exactly one of the two groups, so the cell is their sum. The flagship coin-change lesson splits on a different question, which coin was spent last. That split is right for the fewest coins, because the order coins are spent in does not change how many there are. For counting it is wrong: asking which coin came last counts $3 + 1 + 1$, $1 + 3 + 1$ and $1 + 1 + 3$ as three answers. Deciding coin by coin whether it is used at all is what makes the order disappear.

RECURRENCE

$dp[i][s] = dp[i - 1][s] + dp[i][s - c_i]$, the second only when $c_i \leq s$

BASE CASES AND ORDER

Row 0 is the only base row. With no coins allowed, the only combination is the empty one, so $dp[0][0] = 1$ and every other cell in row 0 is 0. Column 0 computes, and is 1 all the way down for the same reason: taking no coins makes 0 whichever coins are allowed. Fill row by row and left to right within a row. The use-at-least-one case reads a cell to the LEFT in the SAME row, so that cell has to be final first, which a left-to-right sweep guarantees. Apart from adding where subset-sum takes the larger, that same-row read is what separates this table from subset-sum's, which reads the row above because each number there may be used only once.

Time: $O(\text{coins} \times \text{amount})$

Space: $O(\text{coins} \times \text{amount})$, reducible to $O(\text{amount})$ with one row and coins as the outer loop

Watch out

Counting orders as different combinations. The natural first attempt carries the flagship lesson's one-row recurrence across with a sum in place of $1 + \min$ and $\text{ways}[0] = 1$, $\text{ways}[s] =$ the sum over coins $c \leq s$ of $\text{ways}[s - c]$, and that counts sequences, not combinations. At amount 5 with coins $\{1, 3, 4\}$ it gives 6, counting $3 + 1 + 1$, $1 + 3 + 1$ and $1 + 1 + 3$ separately and $4 + 1$ and $1 + 4$ separately, where the answer is 3. The gap grows quickly: at amount 11 it is 104 against 9. Dividing out the duplicates afterwards has no clean formula. The fix is to decide one coin at a time, which is what the rows of this table do.

► Cutting a Rod

STATE: WHAT ONE ENTRY MEANS

$dp[i][L]$ is the best total price for a rod of length L when the only piece lengths you may cut are 1 through i . Row i widens the menu of piece lengths by one, and column L is the length of rod being sold, so the bottom-right cell, every length allowed and the whole rod, is the answer. Leaving the rod whole is not a special case: it is cutting one piece of length n , which only the last row allows. Throughout, $\text{price}[i]$ means the price of a piece of length i , which the snippet's 0-indexed array stores as $\text{price}[i - 1]$.

THE FINAL DECISION

Ask about the longest piece length this row allows, i . Either the best cut uses no piece of length i , and then it is the best cut with lengths up to $i - 1$: the cell above. Or it uses at least one: sell that piece for $\text{price}[i]$, and cut the remaining $L - i$ as well as possible with lengths up to i STILL allowed, because another piece of length i may come out of it too. That is the cell i columns to the left in the same row. Take the better of the two. This is unbounded-knapsack exactly, with piece lengths as the items, lengths as weights and prices as values, which is why this site files it with the knapsacks rather than with the one-row version most textbooks print.

RECURRENCE

```
dp[i][L] = max(dp[i - 1][L], price[i] + dp[i][L - i]), the second only when i
<= L
```

BASE CASES AND ORDER

Row 0 is the only base row, and it is 0 everywhere. At column 0 that is exact: a rod of length 0 sells for nothing. At a positive length it is a stand-in: with no piece lengths allowed no cut exists, and the exact value would be minus infinity. 0 is safe here because the only cells that read row 0 are row 1's no-piece options, and row 1's other option is $L \times \text{price}[1]$, which is above 0 for every $L > 0$ because $\text{price}[1]$ is positive, so the stand-in never wins at a positive length. Rows 2 and later never read row 0 at all. Fill row by row, left to right within a row: the with-a-piece case reads its own row further left, so that cell must already be final. The slider here moves both axes at once, because a longer rod needs one more column for its length and one more row for the new piece length it allows.

Time: $O(n^2)$

Space: $O(n^2)$, reducible to $O(n)$ with one row

Watch out

Trusting the best-looking piece. At length 4 the whole rod sells for 9, and the length with the best price per unit is 3 (8 for 3, about 2.67 a unit, against 2.5 for length 2), so cutting the densest piece first gives $8 + 1 = 9$. Both lose to two pieces of length 2 for 10, which neither rule ever looks at. The second trap is reading the with-a-piece case from the row above: that lets each length be cut at most once, and at length 4 it answers 9, because $5 + 5$ needs the length-2 piece twice.

► 0-1 Knapsack Problem

STATE: WHAT ONE ENTRY MEANS

$\text{dp}[i][c]$ is the best total value you can carry using only the first i items, each at most once, in a knapsack of capacity c . Two numbers name the state, so the table is a grid: row 0 for no items, then one row per item, and one column per capacity from 0 up to the knapsack's. A cell holds a value, not the list of items behind it; recovering the list is what printing-items-in-zero-one-knapsack is about.

THE FINAL DECISION

Ask about the last item allowed, item i , which weighs w_i and is worth v_i . The best load from the first i items either leaves it out or puts it in. Out: the best load is the best from the first $i - 1$ items in the same capacity, the cell above. In: it is worth v_i and uses w_i of the capacity, and whatever else is in the knapsack must be the best load of the first $i - 1$ items in the $c - w_i$ that remains, which is the cell w_i columns to the left in the row above. It is the row above because item i is already in and cannot go in twice. Take the larger. When w_i is more than c , the item cannot go in at all and the cell copies the one above.

RECURRENCE

```
dp[i][c] = max(dp[i - 1][c], v_i + dp[i - 1][c - w_i]), the second only when  
w_i <= c
```

BASE CASES AND ORDER

Row 0 is the only base row: with no items allowed the knapsack stays empty, so every cell in row 0 is 0. Column 0 computes, and is 0 all the way down, because every item here weighs at least 1. Fill row by row; both cells a cell reads are in the row above, so they are final before anything reads them. In drive mode only the two-option cells in rows 2 to 4 are gated, because row 1's without-it option is always the empty knapsack's 0, so its comparison is a formality.

Time: $O(n \times \text{capacity})$

Space: $O(n \times \text{capacity})$, reducible to $O(\text{capacity})$ with one row swept from high to low

Watch out

Packing greedily by value per unit of weight. (5, 7) has the best ratio, 1.4, so greedy packs it first; (3, 4) and (4, 5) no longer fit beside it, (1, 1) does, and greedy stops at 8 with one unit of capacity unused. The table's 9 comes from (3, 4) and (4, 5), a pair greedy never considers once (5, 7) is in. Greedy by ratio is correct for the fractional knapsack, where items can be cut; with whole items it is not, and that difference is the reason this problem needs a table.

► Printing Items in 0/1 Knapsack

STATE: WHAT ONE ENTRY MEANS

$dp[i][c]$ is the best total value using only the first i items, each at most once, within capacity c , exactly as on zero-one-knapsack: row 0 for no items, then one row per item, one column per capacity. The table stores VALUES only. Which items produced a value is in no cell; it is recovered afterwards from how each cell compares with the one above it.

THE FINAL DECISION

The fill's last decision is zero-one-knapsack's: item i is either out (the cell above) or in (v_i plus the cell w_i columns to the left in the row above). The walk asks the same question in reverse, about a cell that is already filled. If $dp[i][c]$ equals $dp[i - 1][c]$, the value here is reachable without item i , so leave it out and step up. If they differ, the only way this cell got bigger than the one above is by taking item i , so record it and step to $dp[i - 1][c - w_i]$, the cell the take option read. Each step settles one item and climbs one row, so the walk takes n steps.

RECURRENCE

fill: $dp[i][c] = \max(dp[i - 1][c], v_i + dp[i - 1][c - w_i])$; walk: item i is in exactly when $dp[i][c] \neq dp[i - 1][c]$

BASE CASES AND ORDER

Row 0 is all 0s and the fill runs row by row from the top, as on zero-one-knapsack. The walk then runs the other way: it starts at the answer cell, $dp[n][capacity]$, climbs one row per step to row 0, and moves left only when it records an item. It reads cells and writes none, which is why the animation stops at the answer: every animated step reads cells in order to write one, and the walk writes nothing. In drive mode only the last row's comparisons are gated, because the walk starts in the last row and the fill itself is zero-one-knapsack's lesson.

Time: $O(n \times capacity)$ to fill, plus $O(n)$ to walk back

Space: $O(n \times capacity)$; the walk needs every row's comparison, kept either as the full table or as one bit per cell

Watch out

Walking a space-optimised table. Fill with the single row zero-one-knapsack's execution describes and the rows the walk compares no longer exist, so there is nothing to compare $dp[i][c]$ against; the walk needs the full grid, or one stored bit per cell. The other mistake is deciding an item is in because it FITS in the capacity still left. At capacity 7 that rule takes item 4 first, since it weighs 5, then item 1 in the 2 that remain, which reaches 9 here only because items 1 and 4 happen to be the second optimal selection. At capacity 6 the same rule takes item 4, finds nothing else fits in the 1 that remains, and reports a value of 6 against the table's 8. The walk must compare each cell with the one above it, not with the item's weight.

► Unbounded Knapsack

STATE: WHAT ONE ENTRY MEANS

$dp[i][c]$ is the best total value from the first i item types within capacity c , where any type may be taken as many times as you like. Two numbers name the state: row 0 for no types, then one row per type, and one column per capacity from 0 up to the knapsack's.

THE FINAL DECISION

Ask whether the best load uses type i at all. If not, it is the best from the first $i - 1$ types, the cell above. If it does, take one copy out: that copy was worth v_i , and what remains is a best load in $c - w_i$ in which type i is STILL available, because there may be more copies of it. That is the cell w_i columns to the left in the SAME row, not the row above. That row index is the only thing that differs from zero-one-knapsack's recurrence, whose take option reads the row above because an item cannot go in twice. cutting-a-rod is this exact recurrence with piece lengths as the types.

RECURRENCE

$dp[i][c] = \max(dp[i - 1][c], v_i + dp[i][c - w_i])$, the second only when $w_i \leq c$

BASE CASES AND ORDER

Row 0 is the only base row: with no types allowed nothing is carried, so it is 0 at every capacity.

Columns 0 and 1 compute and stay 0, since every type here weighs at least 2. Fill row by row and left to right within a row. The take option reads its own row further left, so that cell has to be final first, and a left-to-right sweep is what makes it so.

Time: $O(\text{types} \times \text{capacity})$

Space: $O(\text{types} \times \text{capacity})$, reducible to $O(\text{capacity})$ with one row swept upward

Watch out

Reading the take option from the row above. That allows each type once and returns zero-one-knapsack's answer: 9 at capacity 6, from the (5, 9) alone, against the true 10 from two (3, 5)s. In the one-row form the same mistake is sweeping capacity downward. Both versions run and return a plausible number, and they agree with the right answer wherever the optimum happens to use each type at most once (capacity 5 here, where both say 9), so a test at the wrong capacity passes.

► Partition Problem

STATE: WHAT ONE ENTRY MEANS

$dp[i][s]$ is 1 if some subset of the first i numbers adds up to exactly s , and 0 if none does, for s from 0 up to half the total. That is subset-sum's state with the target fixed by the input rather than given: row 0 for no numbers, then one row per number, and one column per sum up to half the total.

THE FINAL DECISION

The insight is the reduction. Splitting the numbers into two groups with equal totals is the same as choosing ONE group that adds to $\text{total} / 2$, because whatever is left over then adds to $\text{total} / 2$ as well. Once that is seen, the last decision is subset-sum's: the last number is either in the chosen group (read $s - a_i$ in the row above) or not (read s in the row above), and either route reaching s is enough, which on 1s and 0s is a max. So the table never needs a column past $\text{total} / 2$.

RECURRENCE

```
dp[i][s] = dp[i - 1][s] or dp[i - 1][s - a_i], the second only when a_i <= s;  
target total / 2, and an odd total is no
```

BASE CASES AND ORDER

Before any table: if the total is odd, the answer is no, because two equal whole-number totals add up to an even number. This page's numbers are chosen so every slider size has an even total, and its spec refuses an odd one rather than build a table for a question that is already answered. Row 0 is the only base row: $dp[0][0] = 1$ for the empty group, and every other row-0 cell is 0. Column 0 computes and is 1 all the way down. Fill row by row; both cells a cell reads are in the row above.

Time: $O(n \times \text{total})$

Space: $O(n \times \text{total})$, reducible to $O(\text{total})$ with one row swept from high to low

Watch out

Dropping the odd-total check and letting integer division pick the target. Take [2, 3, 4]: the total is 9, $\text{Math.floor}(9 / 2)$ is 4, the table finds {4}, and the program answers yes for numbers that cannot be split into equal halves at all, since 4 against 5 is not equal. The check costs one line and has to come before the table, because once the target is rounded the table is answering a different question correctly.

► All distinct subset (or subsequence) sums

STATE: WHAT ONE ENTRY MEANS

$\text{dp}[i][s]$ is 1 if some subset of the first i numbers adds up to exactly s , and 0 if none does. That is subset-sum's state, with one change of reach: there the columns stopped at a target, and here they run all the way to the total of every number, because the question is about every total at once and no subset can make more than that. What is different is the answer. subset-sum reads one cell. This problem reads the WHOLE bottom row: every column whose last-row cell is 1 is a total some subset makes, and the list of those columns is the output. A cell is a yes or a no, so a total made two different ways is still a single 1, and that is where the word distinct is taken care of.

THE FINAL DECISION

subset-sum's decision, unchanged: every subset of the first i numbers either leaves the last one, a_i , out, and then the rest makes s from the first $i - 1$ numbers (the cell above), or puts it in, and then the rest makes $s - a_i$ from the first $i - 1$ numbers (a_i columns to the left in the row above). Either route reaching s is enough, and on 1s and 0s that is the larger of the two, which is why the animation writes it as a max. Nothing about the recurrence knows that the whole row will be read at the end; it fills every column the same way subset-sum fills the ones left of its target.

RECURRENCE

$\text{dp}[i][s] = \text{dp}[i - 1][s] \text{ or } \text{dp}[i - 1][s - a_i]$, the second only when $a_i \leq s$

BASE CASES AND ORDER

Row 0 is the only base row: with no numbers allowed the only subset is the empty one, so $\text{dp}[0][0] = 1$ and every other cell of row 0 is 0. Column 0 computes, and is 1 all the way down, since leaving everything out always makes 0; that is the empty subset's total the statement asks you to include. Fill row by row. Both cells a cell reads are in the row above, and the put-it-in read has to come from the row above rather than the current one, or a number can be used twice.

Time: $O(n \times \text{total})$

Space: $O(n \times \text{total})$, reducible to $O(\text{total})$ with one row swept right to left

Watch out

Letting a number be used twice. Read the put-it-in cell from the current row, $dp[i][s - a_i]$, instead of the row above, and a number can join a subset that already holds it. On $[2, 3, 5]$ the bottom row then claims 4 ($2 + 2$), 6 ($3 + 3$) and 9 ($2 + 2 + 5$) as well, ten totals where the answer has seven: every total from 0 to 10 except 1. The output is still a sorted list of plausible-looking totals, so nothing looks broken; it is the right answer to a different problem, the one where each number may be reused. In the single-row version the same bug is sweeping s upwards.

Check your understanding

1. What is the best value at capacity 5 with one item (2, 3) and one item (3, 4)?

- A. 3
- B. 4
- C. 6
- D. 7

2. There is only one 0/1 item of weight 2 and value 3. At capacity 4, what is the valid best value?

- A. 0
- B. 3
- C. 4
- D. 6

Flashcards: cover the answer to recall it

Why do compressed 0/1 knapsack capacities run downward?

The source capacity must still describe the previous set of items. Descending updates stop the current item from feeding another update in the same pass.

How do exact-fill bases differ from at-most capacity bases?

For exact fill, only total weight 0 is initially reachable. Other weights need an unreachable marker until a valid selection produces them.

What does `reachable[s]` mean in subset sum?

Some subset of the considered items has total exactly s . It is a yes/no state, not a maximum-value state.

Quiz answer key

1. **D.** Weights $2 + 3$ use capacity 5 exactly, and their values add to $3 + 4 = 7$. Each item is taken once.

2. **B.** The item may be used only once, giving value 3. A result of 6 would reuse it, which an ascending in-place loop could accidentally allow.

Interval DP: solve shorter spans first

When a decision splits a contiguous range into smaller ranges, store answers by both endpoints. The important choice may be the final operation on a range, because its boundaries are then known.

The idea to remember

- For matrix-chain multiplication, $\text{cost}[l][r]$ is the fewest scalar multiplications needed to multiply matrices l through r , inclusive. Matrix i has dimensions $p[i]$ by $p[i + 1]$.
- Split after k : $\text{cost}[l][k] + \text{cost}[k + 1][r] + p[l] * p[k + 1] * p[r + 1]$. Minimize this quantity over $l \leq k < r$.
- A single matrix needs no multiplication, so $\text{cost}[i][i] = 0$. Fill intervals by increasing length; both subintervals in every split are shorter.
- Define whether endpoints are inclusive before writing loops. Different interval problems may use empty spans or fixed outside boundaries, so their bases are not interchangeable.
- There are $O(n^2)$ intervals and up to $O(n)$ splits per interval, giving $O(n^3)$ time and $O(n^2)$ table space for this recurrence.

Work through one example

Matrices A: 2 by 3, B: 3 by 4, C: 4 by 2.

$(AB)C$ costs $2*3*4 + 2*4*2 = 24 + 16 = 40$. $A(BC)$ costs $3*4*2 + 2*3*2 = 24 + 12 = 36$. Compare both possible final splits.

36 multiplications

Worked problem notes (8)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Longest Palindromic Subsequence

STATE: WHAT ONE ENTRY MEANS

$\text{dp}[i][j]$ is the length of the longest palindromic subsequence of the stretch $s[i..j]$, both ends included. Two numbers name the state, where the stretch starts and where it ends, so the table is a square in which only the upper triangle, $i \leq j$, means anything; the cells below the diagonal are not stretches and show a dot. The answer is the whole string, $\text{dp}[0][n - 1]$, in the top-right corner.

THE FINAL DECISION

Ask what happens to the two end characters, $s[i]$ and $s[j]$. This is the LCS decision, and its derivation lives in the LCS lesson at </courses/dynamic-programming/longest-common-subsequence>; only the setting changes, the two ends of one stretch instead of the last characters of two strings. If the ends match, using them as the outer pair is never worse: a best palindrome that skipped one of them could swap its own outer pair for $s[i]$ and $s[j]$ without getting shorter. So wrap the best palindrome strictly inside, $dp[i][j] = 2 + dp[i + 1][j - 1]$, which is LCS's diagonal step. If the ends differ, no palindrome in the stretch can use both, because its first and last characters would have to be $s[i]$ and $s[j]$. So at least one end is out: drop the left ($dp[i + 1][j]$) or drop the right ($dp[i][j - 1]$) and keep the longer, which is LCS's max of up and left. It is a max because one dropped end is enough and you want the better of the two, and nothing is added because no character joined the palindrome.

RECURRENCE

```
dp[i][j] = 2 + dp[i + 1][j - 1] if s[i] = s[j] (just 2 when j = i + 1),  
otherwise max(dp[i + 1][j], dp[i][j - 1])
```

BASE CASES AND ORDER

$dp[i][i] = 1$ on the diagonal: a single character is a palindrome. A length-2 stretch with matching ends, like 'bb' at $dp[0][1]$, is set to 2 without reading anything, because its inside, $dp[i + 1][i]$, would sit below the diagonal and is not a stretch at all. Every other cell reads only shorter stretches inside its own: the one below it (left end dropped), the one to its left (right end dropped), or the one diagonally below-left (both ends kept). Filling by length, every length-2 stretch, then every length-3 stretch, and so on, guarantees each of those is final before it is read. On the grid that is a sweep along the diagonals, from the main one out to the top-right corner.

Time: $O(n^2)$

Space: $O(n^2)$, reducible to $O(n)$

Watch out

Filling the triangle row by row from the top, i rising and then j rising, the way a grid fills. $dp[0][j]$ then reads $dp[1][...]$ before row 1 has been computed, and in a zero-filled array that read quietly returns 0 instead of failing. Measured on this page's string, that loop returns 2 at every slider size, which is right only at 2 and 3 characters; at 6 it says 2 where the answer is 5. Fill by length, or run i downwards from $n - 1$. The other classic slip is answering the substring question: the longest run of ADJACENT characters in 'bbabcb' that is a palindrome is 'bab' or 'bcb', length 3, not 5.

► Minimum insertions for palindrome

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the fewest characters that must be inserted into the stretch $s[i..j]$ to make it a palindrome. As on longest-palindromic-subsequence, two numbers name the state, where the stretch starts and where it ends, only the upper triangle $i \leq j$ exists, and the answer for the whole string is $dp[0][n - 1]$ in the top-right corner.

THE FINAL DECISION

Look at the two ends, as longest-palindromic-subsequence and the LCS lesson (/courses/dynamic-programming/longest-common-subsequence) do. If $s[i] = s[j]$, they can stand as the outer pair of the finished palindrome exactly as they are, so they cost nothing and the whole cost is the inside's: $dp[i + 1][j - 1]$, copied across unchanged. If they differ, they cannot face each other in the result, so one of them needs a new partner on the far side. A copy of $s[i]$ inserted just after $s[j]$ pairs with $s[i]$ and leaves $s[i + 1..j]$ to fix; a copy of $s[j]$ inserted just before $s[i]$ leaves $s[i..j - 1]$. Either way exactly one insertion is paid, so take the cheaper remainder and add 1. It is a min because only one of the two ends needs the new partner, and you want the cheaper one.

RECURRENCE

```
dp[i][j] = dp[i + 1][j - 1] if s[i] = s[j] (0 when j = i + 1), otherwise 1 + min(dp[i + 1][j], dp[i][j - 1])
```

BASE CASES AND ORDER

$dp[i][i] = 0$: one character is already a palindrome. A length-2 stretch with matching ends is 0 without reading anything, since its inside would sit below the diagonal; this page's string never puts two equal letters side by side, so that case never fires at any slider size, while longest-palindromic-subsequence's 'bb' does. Every other cell reads shorter stretches inside its own (below, to the left, or diagonally below-left), so filling by length makes each of them final before it is read.

Time: $O(n^2)$

Space: $O(n^2)$, reducible to $O(n)$

Watch out

Counting the mismatched mirror pairs, $s[0]$ against $s[n - 1]$, $s[1]$ against $s[n - 2]$ and so on, and calling that the answer. That count answers a different question, the fewest characters to REPLACE, and on this page's string it is wrong for insertions in both directions. At 6 characters, 'abdcdb', all three mirror pairs mismatch and the count says 3, yet one inserted 'a' finishes the job. At 7 characters, 'abdcdbx', only 'a' against 'x' mismatches and the count says 1, but the answer is 2. An insertion shifts everything after it, so the pairs a fixed mirror compares are not the pairs the finished palindrome matches. A greedy two-pointer walk that mirrors the left end at every mismatch fails too: it happens to get 1 at 6 characters, and at 7 it pays 6, one for every character before the 'x', where the table pays 2.

► Egg Dropping Problem

STATE: WHAT ONE ENTRY MEANS

$dp[e][f]$ is the fewest drops that are GUARANTEED to find the critical floor among f floors when you hold e eggs, however the floors turn out. Two numbers name the state, eggs and floors, so the table is a rectangle, not a triangle. Watch the index convention, because the two counts start in different places: e counts eggs from 1 and f counts floors from 0. So $dp[e][f]$ lives in grid row $e - 1$, the row labelled with e eggs, and in column f . $dp[2][5]$ is the row labelled 2 eggs, column 5. Every $dp[...]$ on this page, in the narration, the gates and the recurrence, uses that $dp[\text{eggs}][\text{floors}]$ form, and so does the code below, which allocates a 0-egg row it never uses so that its indices match. The answer, all the eggs on all the floors, is the bottom-right cell. The cell holds a number of drops, not the plan that achieves it.

THE FINAL DECISION

Ask where the FIRST drop happens: floor k , for some k from 1 to f . Two things can happen, and you do not get to choose which. If the egg breaks, the critical floor is k or below, you have $e - 1$ eggs left, and only the $k - 1$ floors below k still need checking: $dp[e - 1][k - 1]$ more drops. If it survives, the critical floor is above k , you still have all e eggs, and the $f - k$ floors above k are left: $dp[e][f - k]$. Only how many floors are left matters, not which ones, so floors $k + 1$ to f are the same problem as floors 1 to $f - k$. A guarantee has to hold in whichever case happens, so a first drop from floor k costs $1 + \text{the LARGER of the two}$: a max, because the building picks the outcome. Then you pick k , so take the smallest such cost over every k : a min, because the floor is your choice. The min over choices is the move coin change makes, derived in its lesson at </courses/dynamic-programming/coin-change>; what is new is that each choice is scored by its worse outcome before the min sees it. It is also the same loop over split points the convolution1d pages run, nth-catalan-number among them: there every split is summed, and here the floor k splits the building into below and above and only the cheapest split is kept.

RECURRENCE

```
dp[e][f] = min over k from 1 to f of 1 + max(dp[e - 1][k - 1], dp[e][f - k]);  
dp[e][0] = 0, dp[1][f] = f
```

BASE CASES AND ORDER

$dp[e][0] = 0$ down column 0: with no floors there is nothing to find. $dp[1][f] = f$ across the 1-egg row: with a single egg you cannot risk a break that leaves a floor untested, so you drop from floor 1, then 2, and on up, and in the worst case every one of the f floors gets a drop. Every other cell reads $2f$ cells: the row above at columns 0 to $f - 1$, one per floor k for the break, and its own row at columns $f - 1$ down to 0, one per floor k for the survival. All of them sit in the row above or to the left in the same row, so a row-major sweep, left to right along each row, has every read final before the cell needs it. As k rises the break side, $dp[e - 1][k - 1]$, can only grow and the survive side, $dp[e][f - k]$, can only shrink, which is why the best k sits where the two cross.

Time: $O(e \times f^2)$ for e eggs and f floors

Space: $O(e \times f)$, reducible to $O(f)$ with two rows

Watch out

Scoring a drop by its BETTER outcome: $1 + \min(dp[e - 1][k - 1], dp[e][f - k])$ where the max belongs. It reads naturally, min everywhere for a minimisation problem, and it is badly wrong: a first drop from floor 1 always has a break side of $dp[e - 1][0] = 0$, so every computed cell comes out 1 and the answer reads 1 at every slider size, as if one drop could settle any building. The worst case is the whole point of the question, so the inner operation has to be the max. The other classic slip is reaching for binary search: drop from the lower middle floor, floor $f / 2$ rounded up, while you have more than one egg, then go floor by floor. On the 2-egg row it is right up to 8 floors and then wrong at 9, 10 and 11, needing 5, 5 and 6 drops where the table finds 4, 4 and 5, because halving spends an egg too early and leaves one egg facing a long climb. Round to the upper middle floor instead and the 2-egg row already goes wrong at 6 floors, 4 drops against 3. On the 3-egg row, the one the answer comes from, it happens to match at every slider size, which is exactly how it survives a few hand tests.

► Palindrome Partitioning

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the fewest cuts that split the piece $s[i..j]$, both ends included, into pieces that are all palindromes. The table counts cuts rather than pieces because a palindrome then costs 0 and the arithmetic stays a plain sum; the number of pieces is always the cuts plus 1, so the question as asked is answered by $dp[0][n - 1] + 1$. Two numbers name the state, where the piece starts and where it ends, so only the upper triangle, $i \leq j$, means anything, and the answer for the whole string sits in the top-right corner.

THE FINAL DECISION

First ask whether $s[i..j]$ is already a palindrome. If it is, no cut can improve on leaving it whole, so the cell is 0, and that check is made on the string itself, comparing characters inward from both ends, not on the table. If it is not, at least one cut goes inside it, so ask where ONE of the cuts falls: after position k , for some k from i to $j - 1$. Everything left of that cut and everything right of it are then cut independently, and each side should use its own fewest cuts, because a side cut any worse could be swapped for its best without touching the other side. That costs $dp[i][k] + 1 + dp[k + 1][j]$, and since you do not know which k the best cut set uses, try them all and keep the smallest. Taking the minimum over the choices for one decision is the move coin change makes, derived in its lesson at </courses/dynamic-programming/coin-change>. It is also the same loop over split points the convolution1d pages run, nth-catalan-number among them: there every split is summed, because each one builds different trees, and here only the cheapest is kept.

RECURRENCE

$dp[i][j] = 0$ if $s[i..j]$ is a palindrome, otherwise $\min$ over k from i to $j - 1$ of $dp[i][k] + 1 + dp[k + 1][j]$

BASE CASES AND ORDER

$dp[i][i] = 0$ on the diagonal: one character is a palindrome. Any longer piece that is a palindrome, like 'aba' at $dp[0][2]$ or 'baab' at $dp[1][4]$, is also set to 0 without reading anything. Every other cell reads pairs: for a cut after position k , the piece to its left, $dp[i][k]$, in its own row, and the piece to its right, $dp[k + 1][j]$, in its own column, both strictly shorter than $s[i..j]$. Filling by length, every length-2 piece, then every length-3 piece, and so on, guarantees all of them are final first. On the grid that is a sweep along the diagonals, out to the top-right corner.

Time: $O(n^3)$

Space: $O(n^2)$

Watch out

Cutting greedily: take the longest palindrome the string starts with, cut there, and repeat on the rest. It looks safe because a long palindrome swallows many characters at once, and on this page's string it is right at seven of the eight slider sizes, which is how it survives a few hand tests. At 5 characters it fails: on 'abaab' it takes 'aba', is left with 'ab', and ends at 'aba' | 'a' | 'b', 2 cuts, when 'a' | 'baab' needs 1. Taking a short first piece can open up a long palindrome behind it, and only trying every cut position sees that. The other slip is dropping the palindrome check and always minimising over cuts: then no piece longer than one character can ever cost 0, every cell comes out as its length minus 1, and at 6 characters the answer reads 5 instead of 2.

► Palindromic Substring Count

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is 1 if the stretch $s[i..j]$, both ends included, is a palindrome and 0 if it is not. Two numbers name the stretch, so the table is the upper triangle, $i \leq j$, of a square. That flag is not the answer: the question counts palindromes over EVERY stretch, so the answer is the number of 1s in the whole triangle. $total[i]$, in the extra last column, is the count of palindromes that start at position i or later, and $total[0]$ is the answer.

THE FINAL DECISION

Look at the two end characters, $s[i]$ and $s[j]$, the move the LCS lesson derives at </courses/dynamic-programming/longest-common-subsequence> and longest-palindromic-subsequence applies to one string. Here there is no choice to make. A palindrome's first and last characters are equal, so differing ends settle it: 0. If they match, they mirror each other and can never be what spoils the stretch, so it is a palindrome exactly when the stretch strictly inside is one, and the cell copies $dp[i + 1][j - 1]$. There is no max, as there is on the subsequence page, because a substring cannot drop an end: the stretch is fixed, and the only question is yes or no. The count is not a decision at any one cell. It is a sum over the whole triangle, and the total column is where that sum is built: row i 's flags are the palindromes that start at i , and $total[i + 1]$ already counts the ones that start later.

RECURRENCE

$dp[i][j] = 0$ if $s[i] \neq s[j]$, 1 if $s[i] = s[j]$ and $j = i + 1$, otherwise $dp[i + 1][j - 1]$; $total[i] = dp[i][i] + \dots + dp[i][n - 1] + total[i + 1]$, and the answer is $total[0]$

BASE CASES AND ORDER

$dp[i][i] = 1$ on the diagonal: one character is a palindrome. Two kinds of cell are set without reading anything: a stretch whose ends differ is 0, and a length-2 stretch whose ends match, like 'aa' at $dp[2][3]$, is 1, because its inside, $dp[i + 1][i]$, would sit below the diagonal and is not a stretch at all. A longer stretch with matching ends reads the one cell diagonally below-left of it, a shorter stretch, so filling by length makes it final in time. The total column exists because the count is a sum over the table, and a sum needs a cell to land in; no flag cell can hold it. Strictly, $total[i]$ needs only its own row, which is final once the stretches of length $n - i$ are done, plus $total[i + 1]$; filling the column after the whole triangle, from the bottom up, is simply the order that is always safe: $total[n - 1]$ is the last row's lone flag, a base, and each $total[i]$ then reads its own row and the total just below it.

Time: $O(n^2)$

Space: $O(n^2)$

Watch out

Writing the matched-ends case as one line, $dp[i][j] = dp[i + 1][j - 1]$, with no special case for length 2. For a pair of neighbours, $dp[i + 1][j - 1]$ is $dp[i + 1][i]$, below the diagonal; a zero-filled array returns 0 there instead of failing, so every pair of equal neighbours is marked NOT a palindrome. The error then spreads outward: on 'abaab' it loses 'aa' and with it 'baab', which wraps 'aa', and counts 6 instead of 8. Measured on this page's string, it is wrong at every slider size from 4 characters up, 13 instead of 15 at 9, and right only at 2 and 3, where the string has no equal neighbours to lose. Either set a length-2 stretch with matching ends to 1 directly, as this page does, or initialise the cells below the diagonal to 1, the empty stretch counting as a palindrome.

► Optimal Strategy for a Game

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the most the player ABOUT TO MOVE can collect from coins $i..j$, both ends included, when those are the only coins left and both players play their best from there. It is not tied to the first player: whoever is on move faces exactly this question about exactly this stretch, so one table serves both of you. Two numbers name the state, where the remaining row starts and where it ends, so only the upper triangle, $i \leq j$, means anything, and the answer, the first player moving on the whole row, sits in the top-right corner at $dp[0][n - 1]$.

THE FINAL DECISION

Your move is one of two things: take the left coin or take the right coin. Two options, keep the better one: the max over two choices that the house robber lesson derives at </courses/dynamic-programming/house-robber>, rob this house or skip it. What is new here is what happens after you choose. You take an end, and your opponent is now the player about to move on the stretch that is left, and they play it as well as you would, so they collect exactly dp of that stretch, the quantity this table stores. Every coin on the stretch ends up with one of you, so you get everything they do not: $S - dp[i + 1][j]$ if you took the left coin, $S - dp[i][j - 1]$ if you took the right. The coin you took is inside S , which is why it never appears on its own. Notice where the opponent's play went: you never write a min over their replies, because their best total on the rest is already your worst case, and subtracting it from S states that directly.

RECURRENCE

```
dp[i][j] = max(S - dp[i + 1][j], S - dp[i][j - 1]), where S = coins[i] + ... + coins[j]; dp[i][i] = coins[i]
```

BASE CASES AND ORDER

$dp[i][i] = coins[i]$ on the diagonal: with one coin left, the player to move takes it and the game ends. Every other cell reads exactly two cells, each one coin shorter: $dp[i + 1][j]$, the stretch without its left coin, directly below, and $dp[i][j - 1]$, the stretch without its right coin, directly to the left. S is added up from the coins themselves, not read from the table. Filling by length, every pair of coins, then every run of three, and so on, has both reads final before the cell needs them, which on the grid is a sweep along the diagonals out to the top-right corner.

Time: $O(n^2)$

Space: $O(n^2)$, reducible to $O(n)$

Watch out

Taking the larger end coin every turn. It feels safe, a bigger coin now is a bigger coin, and on this page's coins it is right at 2, 3, 5 and 7 coins. At 4 it fails at once: on 8, 15, 3, 7 it takes the 8, because 8 beats 7, and that hands the 15 to your opponent; you finish with $8 + 7 = 15$, where taking the 7 first guarantees 22. Played against an opponent who answers well, greedy also falls short at 6, 8 and 9 coins: 21 against 24, 25 against 30, 27 against 34. The coin you take matters less than the coin you expose. The other common slip forgets the opponent altogether: $dp[i][j] = \max(\text{coins}[i] + dp[i + 1][j], \text{coins}[j] + dp[i][j - 1])$ treats the rest of the row as yours too, so every cell comes out as the sum of its coins, 33 at 4 coins, as if you were playing alone.

► Matrix Chain Multiplication

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the fewest scalar multiplications needed to multiply matrices i through j , both included, down to one matrix. Matrix m is $\text{dims}[m] \times \text{dims}[m + 1]$, and the matrices are lettered $A, B, C, \dots$ from 0, so $dp[1][3]$ is the chain BCD . Whatever the grouping, the product of matrices $i..j$ is always $\text{dims}[i] \times \text{dims}[j + 1]$; only the cost of getting there varies, which is why a chain's best cost can be worked out once and reused by every longer chain around it. Two numbers name the state, where the chain starts and where it ends, so only the upper triangle, $i \leq j$, means anything, and the answer for the whole chain sits in the top-right corner, $dp[0][n - 1]$.

THE FINAL DECISION

However the chain $i..j$ is bracketed, it ends with ONE last multiplication, joining the product of $i..k$ on the left with the product of $k + 1..j$ on the right, for some k from i to $j - 1$. Those two products are $\text{dims}[i] \times \text{dims}[k + 1]$ and $\text{dims}[k + 1] \times \text{dims}[j + 1]$ no matter how each was built, so that last multiplication costs $\text{dims}[i] \times \text{dims}[k + 1] \times \text{dims}[j + 1]$, and each side should be built as cheaply as possible on its own: a side built any dearer could be swapped for its cheapest grouping without changing the shapes the last multiplication sees. That is $dp[i][k] + dp[k + 1][j] + \text{dims}[i] \times \text{dims}[k + 1] \times \text{dims}[j + 1]$ for this k , and since you do not know which k the best grouping ends with, try them all and keep the smallest. Taking the minimum over the choices for one decision is the move coin change makes, derived in its lesson at </courses/dynamic-programming/coin-change>. It is also the same loop over split points the convolution1d pages run, nth-catalan-number among them: there every split is summed, because each one builds different bracketings, and here only the cheapest is kept.

RECURRENCE

$dp[i][i] = 0$, and for $i < j$, $dp[i][j] = \min$ over k from i to $j - 1$ of $dp[i][k] + dp[k + 1][j] + \text{dims}[i] \times \text{dims}[k + 1] \times \text{dims}[j + 1]$

BASE CASES AND ORDER

$dp[i][i] = 0$ on the diagonal: one matrix needs no multiplying. Every longer chain reads pairs: for a split after matrix k , the left side $dp[i][k]$, in its own row to the left, and the right side $dp[k + 1][j]$, in its own column below, both strictly shorter than the chain itself. A length-2 chain has one split, between two diagonal 0s, so its value is just the one multiplication. Filling by length, every length-2 chain, then every length-3 chain, and so on, guarantees every side is final before it is read. On the grid that is a sweep along the diagonals, out to the top-right corner.

Time: $O(n^3)$

Space: $O(n^2)$

Watch out

Multiplying the cheapest adjacent pair first, and repeating. It feels safe because every step it takes is small, and on this page it matches the table at 2 and 3 matrices. At 4 it fails: BC is the cheapest pair (6), and after it the chain is A (4×2), BC (2×1) and D (1×3), whose cheapest pair is BC times D (6) rather than A times BC (8). That leaves a 4×2 times 2×3 multiplication, 24, for last: $(A((BC)D))$ costs 36, against 26 for $((A(BC))D)$. A cheap step can leave wide matrices for the final multiplication, and only weighing every split point, sides and join together, sees that; at 8 matrices the greedy order costs 150 against the table's 81. The other common slip is not choosing at all and multiplying straight left to right, as a loop over the array would: 48 at 4 matrices and 268 at 8.

► Printing Matrix Chain Multiplication

STATE: WHAT ONE ENTRY MEANS

$dp[i][j]$ is the fewest scalar multiplications needed to multiply matrices i through j , both included, into one, exactly as on matrix-chain-multiplication: matrix m is $\text{dims}[m] \times \text{dims}[m + 1]$, lettered A, B, C, ... from 0, and the answer sits at $dp[0][n - 1]$. Printing a grouping needs one more fact per cell, which k won it, because the grouping cannot be read off the values alone without redoing the arithmetic: a cell's value says how cheap its chain can be, its winning k says where that chain's outermost brackets go. That k is a byproduct of the min the cell already takes, so it is remembered alongside the value, not computed by a second recurrence.

THE FINAL DECISION

However the chain $i..j$ is bracketed, it ends with ONE last multiplication, joining the product of $i..k$ with the product of $k + 1..j$, and that multiplication costs $\text{dims}[i] \times \text{dims}[k + 1] \times \text{dims}[j + 1]$ whatever happened inside the two sides, so each side should be as cheap as it can be on its own. Try every k and keep the cheapest: $dp[i][k] + dp[k + 1][j] + \text{dims}[i] \times \text{dims}[k + 1] \times \text{dims}[j + 1]$. The k that wins is the reconstruction's whole content, because the last multiplication of a chain is its outermost pair of brackets: the left side goes inside on the left, the right side on the right, and each side is bracketed by its own winning k . Taking the minimum over the choices for one decision is the move coin change makes, derived in its lesson at </courses/dynamic-programming/coin-change>. It is also the same loop over split points the convolution1d pages run, nth-catalan-number among them: there every split is summed, because each one builds different bracketings, and here only the cheapest is kept.

RECURRENCE

```
dp[i][j] = min over k from i to j - 1 of dp[i][k] + dp[k + 1][j] + dims[i] ×
dims[k + 1] × dims[j + 1], with split[i][j] = the first k that reaches it;
then group(i, i) is the letter of matrix i, and group(i, j) = "(" + group(i,
split[i][j]) + group(split[i][j] + 1, j) + ")"
```

BASE CASES AND ORDER

$dp[i][i] = 0$ on the diagonal, and a single matrix prints as its own letter. Every longer chain reads, for each split after matrix k , its left side $dp[i][k]$ in its own row and its right side $dp[k + 1][j]$ in its own column, both strictly shorter, so the table fills by length, shortest first, along the diagonals to the top-right corner. The walk runs the other way, after the table is full: from the corner down, and it only ever visits chains some winning split produced.

Time: $O(n^3)$

Space: $O(n^2)$

Watch out

Choosing each bracket by the cost of its own final multiplication, top down. It sounds like the table's idea run backwards: for ABCDE the multiplication that joins the two sides is cheapest after B (6, against 15, 12 and 18), so split there and recurse. That prints $((AB)(C(DE)))$, which costs $30 + 32 + 6 = 68$, not 57: the cheap join hides the 30 spent building AB, and only the full sum, both sides included, sees it. The other slip is in the walk itself: recovering each split from the finished table as the k with the smallest $dp[i][k] + dp[k + 1][j]$, leaving out the join. On this page that happens to print the right grouping at 5 matrices, but at 4 it prints $((AB)C)D$, which costs 126, instead of $((AB)(CD))$ for 114, and at 8 its grouping costs 183 against 96. Record the k the min actually chose, or put the join back in when you look for it.

Check your understanding

1. For A (2 by 3), B (3 by 4), and C (4 by 2), what is the minimum multiplication cost?

- A. 24
- B. 36
- C. 40
- D. 48

2. What should $cost[2][2]$ be when it represents a chain containing one matrix?

- A. 0
- B. 1
- C. 2
- D. Its row count times its column count

Flashcards: cover the answer to recall it

Why fill interval DP by increasing interval length?

A transition splits the current interval into shorter subintervals. Length order guarantees their answers already exist.

What three costs appear in a matrix-chain split?

The optimal left subchain cost, the optimal right subchain cost, and the cost of multiplying the two resulting matrices.

What boundary convention must be explicit in interval DP?

Whether endpoints are included, what an empty or single-element interval means, and which outside values remain fixed.

Quiz answer key

1. **B.** $(AB)C$ costs $24 + 16 = 40$. $A(BC)$ costs $24 + 12 = 36$. The smaller complete cost is 36.

2. **A.** No multiplication is needed for one matrix, so the base cost is 0. Its dimensions matter only when it is multiplied with another result.

Layered DP: keep the phase in the state

A position alone is insufficient when earlier choices change what actions are legal. Add a small mode, such as holding a stock or holding cash, and move from one day or layer to the next.

The idea to remember

- For unlimited stock transactions with no fee or cooldown, cash is the largest profit after a day while holding no share. hold is the largest balance after a day while holding one share.
- At price p , $\text{nextCash} = \max(\text{cash}, \text{hold} + p)$ and $\text{nextHold} = \max(\text{hold}, \text{cash} - p)$. Each next state uses the previous day's values.
- Before the first day, $\text{cash} = 0$ and $\text{hold} = -\text{Infinity}$ because no share has been bought. After the first price, hold becomes $-\text{price}$.
- Return cash when the objective requires all positions closed. hold can include an unsold purchase and is not realized profit.
- Transaction limits, fees, and cooldowns change the state or transitions. A cooldown needs a distinction between newly sold and ready to buy; a transaction cap needs a remaining-count dimension.

Work through one example

Prices [3, 1, 4], unlimited transactions, no fee or cooldown.

Start (cash, hold) = (0, -Infinity). After price 3: (0, -3). After price 1: (0, -1). After price 4: (3, -1).
Selling the share bought for 1 realizes $4 - 1 = 3$.

3 profit

Worked problem notes (6)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Arrange Balls with adjacent of different types

STATE: WHAT ONE ENTRY MEANS

$dp[t][S]$ is the number of distinct arrangements of exactly the balls in S , any sub-pile of the full one up to all of it, with no two equal neighbours, that END with a ball of type t . S names the columns (every non-empty sub-pile, smallest first, written like AAB) and t names the layers, one per type, with a fourth row adding the three up. A plain count is not enough: a new ball can only go on the end of a row that does not already end in its own type, so the state has to carry the last type. This page draws exactly three types, A, B and C, so the table fits on screen; the recurrence is the same for any number of types (see the first variation).

THE FINAL DECISION

Look at the last ball. If it is a t , the balls before it are S with one t taken out, arranged with no equal neighbours, and their own last ball must be anything but a t . Every such shorter row gives exactly one longer row, so add up the ways the shorter pile ends in each other type. It is the climbing-stairs lesson's count by last step, with the step being which type goes last; that lesson derives the idea and this page assumes it. A sum, not a min or max, because every arrangement counts.

RECURRENCE

```
dp[t][S] = sum over types s ≠ t of dp[s][S minus one t]; all endings[S] =  
dp[A][S] + dp[B][S] + dp[C][S]
```

BASE CASES AND ORDER

A single ball is one arrangement, ending in its own type, so each one-ball column is 1 in its type's row and 0 in the others. Any cell whose pile has no ball of that row's type is 0 too: nothing can end in a type that is not there, which is why a preset without C still shows a C row of zeros. Every other cell reads a column with one ball fewer, and the all-endings cell reads its own column, so the table fills one pile at a time, smallest piles first, and down each column top to bottom.

Time: $O(k^2 \times (a + 1)(b + 1)(c + 1))$ for $k = 3$ types with counts a, b, c ; each extra type adds one (count + 1) factor

Space: $O(k \times (a + 1)(b + 1)(c + 1))$ for $k = 3$ types; each extra type adds one (count + 1) factor

Watch out

Keeping only the all-endings row. Without knowing how a shorter row ended, a step cannot tell whether the new ball lands next to one of its own type, so the rule never gets applied and the count comes out as every arrangement there is: 12 for AABC, against the true 6. The other common slip is treating balls of one type as different from each other. Swap the two A's in ABAC and you get the same row, but a count over labelled balls adds it twice and answers 12 here as well, for a different reason.

► Bellman–Ford Algorithm

STATE: WHAT ONE ENTRY MEANS

$dp[k][v]$ is the cost of the cheapest walk from the source to vertex v that uses at most k edges, or ∞ if no walk that short reaches v . The edge allowance k is the layer: one row per allowance, one column per vertex. The note above is exactly this: relaxation is a DP over "shortest path using at most k edges", and the table makes the layers visible.

THE FINAL DECISION

Take the cheapest walk to v with at most k edges and ask about its last edge. Either it does not need the k -th edge at all, and it is already $dp[k - 1][v]$; or its last edge is some $u \rightarrow v$, and the walk up to u is a walk with at most $k - 1$ edges, so swapping in the cheapest such walk, $dp[k - 1][u]$, can only help. So the cell is the best of staying put and every incoming edge. It is the coin-change lesson's shape: a minimum over the last choice made, a coin there and an incoming edge here, each pointing at a smaller subproblem plus a cost. That lesson derives the pattern; this page assumes it.

RECURRENCE

$$dp[k][v] = \min(dp[k - 1][v], \min \text{ over edges } u \rightarrow v \text{ of } dp[k - 1][u] + w(u, v))$$

BASE CASES AND ORDER

Row 0 allows no edges, so only the source is reachable: 0 there and ∞ everywhere else. Each row reads only the row above, so the table fills one whole row at a time, which is exactly one round of relaxation. On 5 vertices, row 4 is final when there is no negative cycle: a cheapest walk then never needs to repeat a vertex, so it uses at most 4 edges. Row 5 is the check: if any cell there is lower than the one above it, a walk with 5 edges beat every walk with at most 4, which means it repeated a vertex and gained by going round a cycle, a negative one.

Time: $O(V \times E)$

Space: $O(V^2)$ for the table, $O(V)$ with one array

Watch out

Stopping after $V - 1$ rounds and reporting row $V - 1$ without the check. On graph C that reports -6 as the distance to vertex 4, but row 5 is lower again in three of the five columns. The graph has negative cycles, the cheapest being $0 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 0$ at -4, and that one passes through every vertex, so a walk can go round it as often as it likes: no vertex has a shortest distance, not even vertex 4, whose column happens to hold still for one round. The second trap shows up when ∞ is stored as a large number such as $1e9$: $1e9 + (-4)$ is smaller than $1e9$, so an unreachable vertex looks reachable after one round. Skip any edge whose start is still ∞ , which is what JavaScript's Infinity does for free.

► Floyd Warshall Algorithm

STATE: WHAT ONE ENTRY MEANS

$dp[k][i][j]$ is the cost of the cheapest path from i to j whose stops in between all come from the first k vertices, 0 to $k - 1$. Row 0 allows no stops at all, so it is just the edges. Three numbers name the state, so the table unrolls each layer's matrix into one row: one row per k , one column per ordered pair $i \rightarrow j$. The diagonal is left out, since a vertex is 0 away from itself when there is no negative cycle. This is the note's "shortest path using only the first k vertices as intermediates", one layer per phase.

THE FINAL DECISION

Ask whether the cheapest $i \rightarrow j$ path that may stop at vertices 0 to m actually stops at m . If not, it only uses 0 to $m - 1$ and is already $dp[k - 1][i][j]$. If it does, it passes m exactly once, since with no negative cycle going round a loop never helps, and m cuts it into two pieces, $i \rightarrow m$ and $m \rightarrow j$, each stopping only at 0 to $m - 1$: $dp[k - 1][i][m] + dp[k - 1][m][j]$. Two candidates, take the smaller. It is the coin-change lesson's minimum over a last decision, with the decision shrunk to yes or no; that lesson derives the pattern and this page assumes it. The second candidate is a PAIR of cells, which is why the compare lights two cells at once.

RECURRENCE

$dp[k][i][j] = \min(dp[k - 1][i][j], dp[k - 1][i][m] + dp[k - 1][m][j])$, where $m = k - 1$ is the vertex newly allowed

BASE CASES AND ORDER

Row 0 is the edge weights, ∞ where there is no edge. Each row reads only earlier rows, so the table fills one row at a time, which is one phase of the algorithm. A dot is a pair that starts or ends at the vertex m that row adds: $dp[k][i][m] = \min(dp[k - 1][i][m], dp[k - 1][i][m] + dp[k - 1][m][m])$, and $dp[k - 1][m][m] = 0$ (the diagonal left out of the table), so it cannot change, and the same holds for $dp[k][m][j]$ with the 0 first. The row skips it and anything that needs it reads the last row that shows it, which is where the longer arrows come from. That same fact is why the textbook code can overwrite one matrix in place: during phase m , the row and column of m never move.

Time: $O(n^3)$

Space: $O(n^3)$ for the layered table, $O(n^2)$ with one matrix updated in place

Watch out

Putting the stop loop innermost, for i , for j , for m . It looks like the same three loops, but it settles each pair in a single pass, using whatever the other pairs hold at that moment, so a route that needs two or more stops can be missed. On graph B it answers 11 for $1 \rightarrow 0$ instead of 1 : it settles $1 \rightarrow 0$ before it has found $1 \rightarrow 3$ for -1 . The stop loop must be outermost, because each phase relies on every pair already being final for the stops before it. The second trap is running it on a graph that does have a negative cycle and trusting the result; check the diagonal afterwards, where a negative $d[v][v]$ means v lies on a negative cycle, and any pair whose route can pass through v has no shortest distance.

► Stock Buy and Sell – At-Most k Times

STATE: WHAT ONE ENTRY MEANS

$dp[t][d]$ is the most profit you can have made by the end of day d using at most t complete trades, with no share in hand. Two numbers name it, the trade allowance and the day, so the table has one row per allowance from 0 up to k and one column per day. The rows are LAYERS: row t is built entirely out of row $t - 1$ and its own left neighbour, so the k you pick only decides how many layers get stacked.

THE FINAL DECISION

Ask whether anything was sold on day d . If not, day d changed nothing and the answer is $dp[t][d - 1]$. If so, it closed the last of your trades, which was bought on some earlier day j . That trade earned $prices[d] - prices[j]$, and every OTHER trade, at most $t - 1$ of them, had to finish by day j , so the best those can add is $dp[t - 1][j]$. Selling one trade and buying the next on the same day j is allowed; it never helps, since holding through earns the same with one trade fewer. Try every j and keep the best, then compare with not selling at all. It is the wait-or-act choice from the house-robber lesson with a whole row of ways to act; this page assumes that lesson rather than re-deriving it.

RECURRENCE

```
dp[t][d] = max(dp[t][d - 1], max over j < d of dp[t - 1][j] + prices[d] - prices[j])
```

BASE CASES AND ORDER

Row 0 is all zeros: with no trades allowed nothing is earned. Column 0 is all zeros: one day cannot hold a buy and a later sale. Every other cell reads its left neighbour and the row above at earlier days, so the table fills one whole row at a time, $t = 1$ first. That is the order the animation uses, and it is why picking a bigger k just adds rows underneath without changing any row above.

Time: $O(k \times n^2)$ as drawn, $O(k \times n)$ with a running best per row

Space: $O(k \times n)$, reducible to two rows

Watch out

Adding up the k biggest day-to-day rises. It feels right, because a trade is a run of rises, but one trade can span several rises and the dips between them. At all ten days with $k = 2$ the biggest two rises are 4 and 3, so that method answers 7, while the table says 9: buy at 2 and sell at 6, then buy at 0 and hold through the dips to sell at 5. A long trade can be worth more than any single rise inside it, and a table that weighs every buy day, as this one does, sees that. The rises method is guaranteed right only once k is large enough not to bind.

► Stock Buy and Sell – At Most 2 Times

STATE: WHAT ONE ENTRY MEANS

$dp[s][d]$ is the most cash you can be up at the end of day d while standing in position s , counting a purchase as minus its price and a sale as plus its price. There are five positions, one per row: 0, no share bought yet; 1, holding your first share; 2, first share sold and not holding; 3, holding your second share; 4, second share sold. The rows are LAYERS of one walk: you can only move down them, one row at a time, and each move down is a buy or a sell. The cell holds an amount of cash and nothing about which days you traded on.

THE FINAL DECISION

Ask what you did on day d , given that you end it in position s . Either nothing, and you were already in position s at the end of day $d - 1$, with exactly the cash you had then; or you made the one move that leads INTO s , from the row above it, paying today's price if that move is a buy and collecting it if it is a sell. Those are the only two ways to be in s tonight, so take the larger. It is the skip-or-take choice the house-robber lesson derives, with "skip" meaning wait a day; this page assumes that lesson and goes straight to the table. Every cell below row 0 has exactly these two candidates, which is why those four rows share one rule and differ only in the sign of the price.

RECURRENCE

```
dp[s][d] = max(dp[s][d - 1], dp[s - 1][d - 1] - prices[d]) for s = 1, 3 (a buy);  
dp[s][d] = max(dp[s][d - 1], dp[s - 1][d - 1] + prices[d]) for s = 2, 4 (a sell)
```

BASE CASES AND ORDER

Row 0 is 0 on every day: a walk that has bought nothing has spent nothing. Day 0 is given down the column: holding the first share on day 0 means you bought it at day 0's price, so $dp[1][0] = -prices[0]$; sold 1 is 0, since buying and selling on the same day gains nothing; hold 2 is $-prices[0]$ again, the first trade done for nothing and the second share bought the same day; sold 2 is 0. Every other cell reads only the column to its left, so the table fills one whole day at a time, rows 1 to 4 top to bottom. That is the order the animation uses.

Time: $O(n)$, four cells per day

Space: $O(n)$ for the table, $O(1)$ with four rolling variables

Watch out

Taking the single best trade first and then adding the best trade you can fit around it. On prices 1, 5, 2, 8 the best single trade is 1 to 8 for 7, and nothing fits outside it, so that method answers 7; the true best is 1 to 5 and then 2 to 8, for $4 + 6 = 10$. The best pair of trades can split the best single trade in two, so a correct method has to choose both trades together, as this table does, or try every day on which to split them. The second trap is in the bases: start the two holding rows at 0 instead of $-prices[0]$ and day 0 hands you a share for free. On the eight-day example that answers 9 instead of 6, for instance by selling the free share at 5 and then trading 0 to 4.

► Min cost to sort strings using Reversals

STATE: WHAT ONE ENTRY MEANS

$dp[0][i]$ is the cheapest way to make strings 0 to i sorted with string i left as it is, and $dp[1][i]$ the same with string i reversed; ∞ means no sorted arrangement ends that way. Those two rows are the layers. A third row, *best*, holds the smaller of the two: the answer if the array stopped at string i . Two layers rather than one because the only thing about strings 0 to i that string $i + 1$ cares about is how string i ended up spelled.

THE FINAL DECISION

Fix how string i ends up, kept or reversed. Its only link to everything before it is its neighbour: string $i - 1$'s final spelling must not sort after it, and every string before that was already checked against string $i - 1$. So try both spellings of string $i - 1$, throw away a spelling that sorts after this one, take the cheaper survivor, and pay $cost[i]$ on top if string i itself is reversed. This is the take-or-skip choice from the house-robber lesson, with one twist. House robber folds both options into one number and steps back two houses to avoid tracking which was taken; no fixed step back does that here, because whether string $i + 1$ fits depends on how string i is spelled, so the two options live on as two rows.

RECURRENCE

```
dp[r][i] = (r = 1 ? cost[i] : 0) + min over q in {0, 1} with form(i - 1, q)
<= form(i, r) of dp[q][i - 1], or  $\infty$  if neither q fits; best[i] = min(dp[0]
[i], dp[1][i])
```

BASE CASES AND ORDER

String 0 has no neighbour to agree with, so $dp[0][0] = 0$ and $dp[1][0] = cost[0]$, and $best[0]$ is the smaller. A kept or reversed cell reads only the column before it, and a best cell reads only the two cells above it, so the table fills one string at a time: kept, then reversed, then best. A cell is ∞ when no spelling of the string before it both fits and is finite; it stays in the table because the next string may still be able to follow the other row.

Time: $O(n \times L)$, L the length of the longest string

Space: $O(n)$, reducible to $O(1)$ with two rolling pairs

Watch out

Deciding left to right and leaving a string alone whenever you can, because leaving it costs nothing. On this list that keeps "ca" free and then has no move at "ab", since "ab" and "ba" both sort before "ca": the method reports the array impossible, where reversing "ca" for 2 makes room and the true answer for the first two strings is 3. Whether a string should be reversed depends on strings that come after it, which is exactly why both spellings have to stay alive as two rows until the end. The second trap is comparing string i only against string $i - 1$ as originally given, forgetting that it may have been reversed.

Check your understanding

1. With prices [3, 1, 4], no fee, and at most one share held, what is the maximum closed-position profit?

- A. 0
- B. 1
- C. 3
- D. 4

2. Under the same rules, what is the maximum profit for prices [5, 2]?

- A. -3
- B. 0
- C. 2
- D. 3

Flashcards: cover the answer to recall it

Why track both cash and hold states?

The legal next actions depend on ownership: cash can buy, while hold can sell. One profit number loses that distinction.

Why initialize hold to -Infinity before the first stock price?

Holding a share is initially impossible. A finite zero would allow a sale without a preceding purchase.

Why compute next-day states from saved previous-day values?

The recurrence advances one layer. Saved values make that time boundary explicit and avoid accidental same-layer transitions when rules become more restrictive.

Quiz answer key

1. **C.** Buy for 1 and sell for 4. The profit is $4 - 1 = 3$. The initial price 3 can be skipped.

2. **B.** Buying at 5 and selling at 2 loses 3. Keeping cash throughout earns 0, which is better. A state must preserve the option to skip.

Bitmask DP: remember an exact small set

When the identity of selected items affects future choices, a count is not enough. A bitmask records which members of a small universe have been used; an extra coordinate records the endpoint or phase when needed.

The idea to remember

- Bit j is 1 when item j is selected. Test membership with mask $\& (1 \ll j)$, and add an unused item with mask $| (1 \ll j)$. For n items, masks range from 0 through $(1 \ll n) - 1$.
- In assignment DP, $\text{best}[\text{mask}]$ is the lowest cost of assigning the first $\text{popcount}(\text{mask})$ workers to exactly the jobs in mask. popcount is the number of selected bits.
- From worker $i = \text{popcount}(\text{mask})$, try each unused job j and relax $\text{best}[\text{mask} | (1 \ll j)]$ with $\text{best}[\text{mask}] + \text{cost}[i][j]$. Initialize $\text{best}[0] = 0$ and other masks to Infinity.
- Process masks in increasing selected-set size, or increasing numeric value for these add-only transitions. Every transition adds one bit, so no dependency points backward to a smaller set size.
- A route problem may need $\text{best}[\text{mask}][\text{last}]$ because the next travel cost depends on the endpoint. Assignment does not need last because mask already determines the next worker.
- The number of subsets is 2 to the power n . Assignment takes $O(n * 2 \text{ to the power } n)$ time; use bitmask DP only when the item universe is small enough.

Work through one example

Two workers and two jobs. Cost rows: worker 0 [3, 1], worker 1 [2, 4].

Assigning worker 0 to job 0 gives mask 01 at cost 3, then job 1 costs 4, total 7. Assigning worker 0 to job 1 gives mask 10 at cost 1, then job 0 costs 2, total 3.

3 minimum cost

Worked problem notes (3)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Bitmasking and DP

STATE: WHAT ONE ENTRY MEANS

$dp[c][mask]$ is the number of ways to deal out cap types 1 to c , each to at most one person who likes it, so that exactly the people in $mask$ end up wearing a cap, one cap each. The $mask$ is a group of PEOPLE written in binary, bit p for person p . That choice is the insight the whole page is built around: the set that goes into the $mask$ is the SMALL one. In the usual telling of this problem there are at most 10 people and 100 cap types. A $mask$ over people has $2^{10} = 1024$ values; a $mask$ over cap types would have 2^{100} , a number with 31 digits. So the people become the $mask$ and the cap types become an ordinary loop. A loop is all the caps need: they are dealt in order, and once the loop moves past cap c it is never offered again, so nothing has to remember which caps are used. People are not served in any order, since any of them may end up with any cap, so the table has to remember who is already covered. That memory is the $mask$.

THE FINAL DECISION

Ask what happened to cap type c , the last one dealt with. Either nobody wore it, and then the people in $mask$ got their caps from types 1 to $c - 1$: $dp[c - 1][mask]$ ways. Or exactly one person p in $mask$ wore it, which needs p to like cap c , and then everyone else in $mask$ got their caps from types 1 to $c - 1$: $dp[c - 1][mask \text{ without } p]$ ways. No assignment falls under two of these options, because cap c 's fate is different in each, and none falls under no option, so the counts ADD. Distinctness takes care of itself: each cap type is handed to one person at most, and each person leaves the $mask$ the moment they are covered. In code, $mask$ without p is $mask \& \sim(1 \ll p)$.

RECURRENCE

```
dp[c][mask] = dp[c - 1][mask] + sum over persons p in mask who like cap c of
dp[c - 1][mask without p]
```

BASE CASES AND ORDER

Row 0 is given: with no cap types dealt out yet, the only possible group is nobody, so $dp[0][000] = 1$, the empty assignment, and every other $dp[0][mask] = 0$. Every later cell reads only the row directly above it, so filling top to bottom is enough, and within a row the order does not matter. The answer is the bottom-right cell, every cap type considered and every person covered. Cells for partial groups are not wasted: $dp[3][101]$, persons 0 and 2 covered by caps 1 to 3, is exactly what row 4 reads when cap 4 goes to person 1.

Time: $O(C \times 2^n \times n)$, for C cap types and n people, with each like looked up in constant time

Space: $O(C \times 2^n)$, reducible to $O(2^n)$

Watch out

Putting the caps in the $mask$. It is a common first attempt, because the caps are what gets used up, and it works on toy inputs, but a $mask$ over 100 cap types has 2^{100} values and no table will ever hold it. The rule to carry away: the $mask$ goes on whichever set is small, and here that is the people. The other common slip is skipping the table and multiplying each person's options. With caps 1 to 4 that is $2 \times 3 \times 2 = 12$, against the true 5, because the product happily gives cap 1 to persons 0 and 1 at once.

► Traveling Salesman Problem

STATE: WHAT ONE ENTRY MEANS

$dp[mask][j]$ is the cheapest cost of a route that starts at home, has visited exactly the cities in mask, each once, and is standing at city j , which is one of them. The single exception is the starting point, $dp[0000][0]$: standing at home before visiting anything, so home sits outside the empty set. The mask is a set written as a binary number, bit k for city k . Notice what the state forgets: the ORDER the cities were visited in. Two routes through $\{1, 2, 3\}$ that both end at city 3 face exactly the same future, the same cities left and the same place to leave from, so only the cheaper of them is worth remembering. That forgetting is the whole method. It turns $(n - 1)!$ orderings into $2^{(n - 1)}$ sets times n endpoints.

THE FINAL DECISION

Ask where the route stood just before it arrived at j . Call that city k . Before the last leg the route had visited every city in mask except j and was standing at k , and the cheapest way to be in that position is $dp[mask \text{ without } j][k]$, the same question on a set one city smaller. Add the leg $dist[k][j]$ and you have one candidate per possible k . It is a min because exactly one city was the predecessor and you want the cheapest of them. The trip home is not a special case: it is the same rule applied once more, at the full set, standing at home. In code, mask without j is $mask \& \sim(1 \ll j)$, which clears j 's bit.

RECURRENCE

```
dp[mask][j] = min over k in mask without j of dp[mask without j][k] + dist[k][j], and when mask without j is empty the only k is home, read from dp[0000][0]
```

BASE CASES AND ORDER

One base: $dp[0000][0] = 0$, standing at home having visited nothing. Every other cell clears one bit to find its predecessors, so it only ever reads a SMALLER mask, and filling the rows in increasing mask order guarantees every predecessor is final before it is read. Two kinds of cell are missing on purpose, shown as dots. A cell whose city is not in its own set cannot exist, since you cannot stand somewhere you have not been; the base is the one exception, home before the route begins. And the rows skip every set that contains home but not everything, because such a set describes a route that came home before the tour was over. Home's bit appears in exactly one row, the last, where it is the final city visited. That also keeps the table at 9 rows for 4 cities rather than 16.

Time: $O(2^n \times n^2)$

Space: $O(2^n \times n)$

Watch out

Walking to the nearest unvisited city each time. From home the cheapest leg on this map is city 2 at 3, then city 1 at 1, and now the only city left is 3, nine away from city 1, before 4 more to get home: $3 + 1 + 9 + 4 = 17$. Both 12-cost tours open with a dearer leg, 5 to city 1 or 4 to city 3, and neither touches the 9. A cheap step is worth nothing if it strands you, and only something that prices every set of visited cities can see that. The second slip is quieter: stopping at the row of all away cities and reporting its smallest cell, 7 here, which forgets the leg home and is the answer to a different problem.

► Sum over Subsets

STATE: WHAT ONE ENTRY MEANS

$dp[i][mask]$ is the sum of $A[sub]$ over the subsets sub of $mask$ that agree with $mask$ on every bit from i upward. In other words only the lowest i bits are allowed to differ, and only by being switched off. Row 0 allows no bit to vary, so the only subset it counts for each mask is the mask itself, and the row is A . Row n allows every bit to vary, so it counts every subset and holds the answer F . The rows in between are the trick: instead of jumping from one subset to all of them, the table opens up one bit at a time.

THE FINAL DECISION

Look at bit $i - 1$, the newest bit allowed to vary. Every subset counted at $dp[i][mask]$ either matches $mask$ on that bit or does not. Those that match differ from $mask$ only in the lower $i - 1$ bits, and that is exactly $dp[i - 1][mask]$. Those that do not match must have the bit switched OFF, which is only possible if $mask$ has it ON; they then agree with $mask$ -without-that-bit on every higher bit and differ only below, which is $dp[i - 1][mask \wedge \text{bit}]$. If $mask$ has the bit clear, a subset cannot switch it on, so the second group is empty and contributes 0. No subset is in both groups and none is in neither, so the two sums add.

RECURRENCE

```
dp[i][mask] = dp[i - 1][mask] + (bit i - 1 set in mask ? dp[i - 1][mask with  
bit i - 1 cleared] : 0)
```

BASE CASES AND ORDER

Row 0 is given: $dp[0][mask] = A[mask]$. Every later cell reads only the row above it, at its own mask and at most one smaller mask, so filling top to bottom is enough. Each subset is counted exactly once, and the reason is worth seeing: a subset of $mask$ is reached by switching off some set of $mask$'s bits, and the table switches off bit $i - 1$ only in row i , so there is exactly one path down the rows for each subset.

Time: $O(n \times 2^n)$

Space: $O(n \times 2^n)$, reducible to $O(2^n)$ in place

Watch out

Swapping the two loops in the one-array version described under execution, so masks are the outer loop and bits the inner one. In the layered table of the snippet the swap happens to be harmless, because each row keeps its own copy; it is the in-place form people actually write that breaks. There it overcounts: by the time a mask is reached, the smaller masks it borrows from already hold their full subset sums, so a subset arrives once for every order in which its missing bits could be switched off, twice for a subset missing two bits and six times for 000 under 111. On this array $F[111]$ comes out 20 instead of 4, and with every entry of A set to 1, which should count the 8 subsets of 111, it reports 16. In place, the bit has to be the outer loop.

Check your understanding

1. For assignment costs $[[3, 1], [2, 4]]$, what is the minimum total when each job is used once?

- A. 2
- B. 3
- C. 5
- D. 7

2. In assignment DP, mask 101 (binary) means jobs 0 and 2 are used. Which zero-based worker is assigned next?

- A. 0
- B. 1
- C. 2
- D. 3

Flashcards: cover the answer to recall it

How do you test whether item j is in a bitmask?

Check whether $\text{mask} \& (1 \ll j)$ is nonzero. Add an unused item with mask $| (1 \ll j)$.

Why does assignment DP use $\text{popcount}(\text{mask})$ as the next worker?

Each transition assigns exactly one new job to the next worker. The number of used jobs equals the number of completed workers.

When does a subset state also need a last coordinate?

When the cost or legality of the next action depends on the endpoint, as in a shortest route visiting a selected set of cities.

Quiz answer key

1. **B.** Worker 0 takes job 1 for 1, and worker 1 takes job 0 for 2. The total is 3, compared with $3 + 4 = 7$ for the other assignment.

2. **C.** Mask 101 has two set bits, so workers 0 and 1 have been assigned. Worker 2 is next. The largest job index is not the worker count.

Digit DP: build a bounded number

Count valid integers by choosing digits from left to right. Instead of trying every integer up to a large bound, group prefixes that have the same remaining constraints.

The idea to remember

- A typical state is (position, tight, property, started). position is the next digit index; tight says the chosen prefix still equals the bound's prefix; property stores information such as a digit sum or remainder.
- If tight is true, the next digit is at most the bound's current digit. Otherwise it may be 0 through 9. $\text{nextTight} = \text{tight AND (chosen digit equals the bound digit)}$.
- At the end of the digits, return 1 only if the completed number satisfies the property. That acceptance check is the base case, not an automatic count of every prefix.
- Leading zeros pad shorter numbers. A started flag is needed when padding zeros must not count as real digits, such as counting occurrences or banning digit 0. Decide explicitly whether the integer zero is included.
- Memoize all state information that changes future choices. If the cache is reused across different bounds, clear it or account for the bound; tight states can depend on its digits.
- To count a range $[L, R]$, define $F(x)$ as the count from 0 to x , then use $F(R) - F(L - 1)$. Define $F(x) = 0$ for negative x .

Work through one example

Count integers from 0 through 25 whose digit sum is 2. Include zero in the range.

Use two digit positions with leading-zero padding. Valid strings are 02, 11, and 20. Their digit sums are 2. Number 00 has sum 0, so it is not accepted.

3 integers

Worked problem notes (1)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Digit DP

STATE: WHAT ONE ENTRY MEANS

Build the number one digit at a time from the left, and after p digits ask two things: has it already dropped BELOW N , and how many 7s has it used. $dp[s][p]$ is the number of ways to choose the remaining digits p to $L - 1$ so the finished number has exactly one 7, starting from state s . There are three states. TIGHT means every digit so far copied N 's, so the next digit may not exceed N 's next digit. FREE0 and FREE1 mean some earlier digit was smaller than N 's, so every later digit is free, with no 7 or one 7 used so far. TIGHT needs no 7-count of its own, because its digits are N 's digits and N 's count is known.

THE FINAL DECISION

Ask what digit goes at position p , and follow it to the state it leads to. From a free state any of the ten digits is allowed: the nine that are not 7 keep the count where it is, and the 7 raises it by one, so FREE0 gets 9 ways to stay FREE0 plus one way to become FREE1, while FREE1 gets 9 ways to stay FREE1 and a 7 would make two, which can never be undone, so that branch adds nothing. From TIGHT the digit cannot exceed N 's digit d . Each of the d smaller digits drops the number below N for good, into the free state with N 's count so far (on this page none of them is a 7, since N has no 8 or 9), and copying d itself keeps it TIGHT. Different digits give different numbers, so the branches add.

RECURRENCE

```
dp[FREE0][p] = 9 × dp[FREE0][p + 1] + dp[FREE1][p + 1]; dp[FREE1][p] = 9 ×  
dp[FREE1][p + 1]; dp[TIGHT][p] = d × dp[FREE_c][p + 1] + dp[TIGHT][p + 1],  
where d is N's digit at p and c is how many 7s N's first p digits hold
```

BASE CASES AND ORDER

The END column, past the last digit, is given. A free number that finished with exactly one 7 counts once: $dp[FREE1][END] = 1$ and $dp[FREE0][END] = 0$. A TIGHT number that reaches END is N itself, so $dp[TIGHT][END]$ is 1 if N has exactly one 7 and 0 if not. Every other cell reads only the column to its right, so the table fills right to left, the reverse of the order the digits are chosen in. That is the memoized recursion count(position, state) turned into a table: the recursion starts at position 0, but it cannot answer until the calls for the later positions have returned.

Time: $O(L \times S \times 10)$ for L digits and S states, here $O(L)$

Space: $O(L \times S)$, reducible to $O(S)$ with two columns

Watch out

Forgetting the tight flag, and counting every string of L digits with exactly one 7. For $N = 370$ that is $3 \times 81 = 243$, every number from 000 to 999 with one 7, against the true 62; the 243 is sitting in the table as $dp[FREE0][0]$, the count for a number that is already free before its first digit. That state, like $dp[FREE1][0]$ beside it, is one no real number is ever in at position 0; the table fills it anyway, because a free row is computed for every position without asking whether anything can arrive there. The cap only binds while the prefix still equals N , which is exactly what TIGHT tracks. The opposite slip is to keep the cap after the number has dropped below N , so that after a first digit of 2 the next two digits are still held to at most 7 and at most 0. That finds only 4 of the 62, because 2 is already below 3 and everything after it should be free.

Check your understanding

1. How many integers from 0 through 25 have digit sum 2?

- A. 2
- B. 3
- C. 4
- D. 5

2. The upper bound is 25. You choose first digit 1. What is the largest allowed next digit?

- A. 2
- B. 4
- C. 5
- D. 9

Flashcards: cover the answer to recall it

What does tight mean in digit DP?

The chosen prefix exactly matches the bound so far. Only then must the next digit obey the bound's current digit.

Why might digit DP need a started flag?

It distinguishes padding zeros from digits of the number. This matters when zero occurrences, forbidden digits, or number length affect validity.

How do you convert a prefix count into a closed-range count?

If $F(x)$ counts valid integers from 0 through x , the count in $[L, R]$ is $F(R) - F(L - 1)$, with $F(x) = 0$ for $x < 0$.

Quiz answer key

1. **B.** The valid integers are 2, 11, and 20. The padded string 02 represents the single integer 2, not an additional number.

2. **D.** Choosing 1 makes the prefix smaller than the bound, so tight becomes false. Any units digit 0 through 9 is allowed; 19 is still below 25.

Tree DP: combine child subtrees

Root a tree so that each node's descendants form independent subproblems once the node's own decision is fixed. Solve children before their parent, then combine the child answers.

The idea to remember

- For maximum-weight independent set, $\text{take}[u]$ is the best sum in node u 's subtree when u is selected. $\text{skip}[u]$ is the best sum there when u is not selected. Adjacent selected nodes are forbidden.
- If u is selected, every child must be skipped: $\text{take}[u] = \text{weight}[u] + \sum(\text{skip}[\text{child}])$. If u is skipped, each child can choose its better mode: $\text{skip}[u] = \sum(\max(\text{take}[\text{child}], \text{skip}[\text{child}]))$.
- A leaf has $\text{take} = \text{its weight}$ and $\text{skip} = 0$. Process nodes in postorder so all child states exist before the parent's states.
- The final answer is $\max(\text{take}[\text{root}], \text{skip}[\text{root}])$. Keeping both answers is necessary because a parent may forbid selecting its child even if selecting that child looks best locally.
- For an undirected tree, pass the parent or track visited nodes to avoid traversing the same edge back again. This recurrence relies on the absence of cycles and overlapping child subtrees.
- Each node and edge is processed a constant number of times, giving $O(n)$ time. Account for stored states and an $O(\text{height})$ recursion stack, which can reach $O(n)$ on a chain.

Work through one example

A root worth 4 has two leaf children worth 3 each. Select no adjacent nodes.

Each leaf has $(\text{take}, \text{skip}) = (3, 0)$. Taking the root gives $4 + 0 + 0 = 4$. Skipping it gives $\max(3, 0) + \max(3, 0) = 6$.

6 maximum value

Worked problem notes (1)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Max Height of Tree when any Node can be Root

STATE: WHAT ONE ENTRY MEANS

Root the tree at node 0, only to give every edge a direction; the answer is still computed for every root. Each node then holds three numbers. $\text{down}[v]$ is the height of v 's own subtree: the longest path that starts at v and only moves away from node 0. $\text{up}[v]$ is the longest path that starts at v and leaves through its parent first, and it is 0 for node 0, which has no parent. $\text{height}[v]$ is the height of the whole tree when v is the root, the number the problem asks for. Every path out of v starts either down into v 's subtree or up through its parent, never both, so the two directions cover every path and $\text{height}[v]$ is simply the larger.

THE FINAL DECISION

For down, the decision is which child the longest downward path steps into first: one edge to that child, then the best the child can do below it. For up, the first step is forced, to the parent p , and the decision is what happens at p : carry on the way p 's own upward path does, which is $\text{up}[p]$ plus the one edge, or turn down into a sibling's subtree, which is one edge to p , one edge to the sibling s , and $\text{down}[s]$ below that. The path cannot turn back down into v 's own subtree, because a path does not revisit v ; that is why the sibling max leaves v out. That exclusion is the subtle step in rerooting: $\text{up}[v]$ is exactly how tall p 's side of the tree is when you look at it from v .

RECURRENCE

$\text{down}[v] = 1 + \max \text{ over children } c \text{ of } \text{down}[c]$, 0 at a leaf; $\text{up}[v] = \max(1 + \text{up}[p], 2 + \text{down}[s] \text{ for each sibling } s \text{ of } v)$, where p is v 's parent, 0 at the root; $\text{height}[v] = \max(\text{down}[v], \text{up}[v])$

BASE CASES AND ORDER

Bases: every leaf's down is 0, a path that goes nowhere, and $\text{up}[0]$ is 0 because the root has no parent to leave through. down needs every child before its parent, so it fills from the leaves; with every parent numbered before its children, that is simply descending node order. up needs the parent's up before the child's, and the siblings' down, which the first pass has already finished, so it fills from the root out, in ascending order. height reads only its own node's two numbers, so it can go in any order once both passes are done. Two passes in opposite directions is the rerooting pattern: the first sees each subtree, the second sees everything else.

Time: $O(n)$ with each parent's two tallest children kept; the plain sibling loop shown is $O(n^2)$ on a star

Space: $O(n)$

Watch out

Reading the parent's down value for the up step: $\text{up}[v] = 1 + \max(\text{up}[p], \text{down}[p])$. $\text{down}[p]$ is the tallest path below p , and that path can run back through v itself. Here $\text{down}[0] = 3$ is the path 0, 1, 4, 7, so the shortcut makes $\text{up}[1] = 1 + 3 = 4$, describing a 'path' 1, 0, 1, 4, 7 that crosses the edge between nodes 0 and 1 twice. The shortcut then gives $\text{height}[1] = 4$, when the farthest node from node 1 is only 3 edges away. Leaving v out of the max over p 's children is the fix, and it is the reason up reads the siblings rather than $\text{down}[p]$.

Check your understanding

1. A root worth 4 has two leaf children worth 3 each. What is the best nonadjacent total?

- A. 3
- B. 4
- C. 6
- D. 10

2. Change only that root's weight from 4 to 7. What is the new best nonadjacent total?

- A. 6
- B. 7
- C. 10
- D. 13

Flashcards: cover the answer to recall it

Why keep both take[u] and skip[u]?

A parent's selection can forbid taking u. The parent needs the best subtree answer for either condition, not only the locally larger value.

Why does tree DP usually process children before the parent?

The parent's recurrence combines child-subtree answers. Postorder makes those dependencies available first.

Why pass a parent when traversing an undirected tree?

Each edge is listed in both directions. Ignoring the parent edge avoids revisiting it and recursing back indefinitely.

Quiz answer key

1. **C.** Taking the root forces both children out, giving 4. Skipping the root allows both children, giving $3 + 3 = 6$.

2. **B.** Taking the root gives 7 while skipping it still gives 6. Choose 7. Adding the children to the root would violate the adjacency rule.

Largest X bordered square: certify all four edges

Find the largest square whose border cells are all marked. Here X means a cell with value 1, not an X drawn along diagonals. The interior can contain zeros. This chapter returns the square's side length.

The idea to remember

- Store two directional runs: $\text{left}[r][c]$ is the consecutive marked length ending horizontally at a cell; $\text{up}[r][c]$ is the consecutive marked length ending vertically there.
- At a 1, extend left from the previous column and up from the previous row, adding 1. At a 0, both run lengths are 0. Missing predecessors contribute 0.
- For a square of side s ending at (r, c) , $\text{top} = r - s + 1$ and $\text{firstCol} = c - s + 1$. Check $\text{left}[r][c]$, $\text{up}[r][c]$, $\text{left}[\text{top}][c]$, and $\text{up}[r][\text{firstCol}]$ are each at least s . They certify bottom, right, top, and left respectively.
- Read row by row from left to right. Try candidate sides downward and stop after finding a valid side at a corner. Skip sizes that cannot beat the best answer, but keep trying smaller sizes after a large candidate fails.
- A hollow square is valid, so the filled-square recurrence cannot replace the four border checks. The answer is a side length; area is its square.
- Worst-case time is $O(\text{rows} * \text{cols} * \min(\text{rows}, \text{cols}))$; the two run tables use $O(\text{rows} * \text{cols})$ space. Empty and all-zero inputs return 0.

Work through one example

`[[1, 1, 1], [1, 0, 1], [1, 1, 1]]`

For bottom-right $(2, 2)$ and side 3, the bottom, right, top, and left runs all have length 3. The center 0 is not on an edge and does not invalidate the square.

Largest side length: 3. Its area is 9.

Worked problem notes (1)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Largest X Bordered Square

STATE: WHAT ONE ENTRY MEANS

$\text{left}[r][c]$ counts consecutive 1s ending at (r, c) when moving horizontally from left to right. $\text{up}[r][c]$ counts consecutive 1s ending there when moving from top to bottom. These are border lengths, not the size of a filled square. best is the largest valid side length found so far.

THE FINAL DECISION

Choose a bottom-right corner and a candidate side length. A square is valid exactly when all four border runs are long enough. Scan candidate lengths downward, stopping at the first valid one for this corner. Only lengths larger than best can improve the answer; a failed large candidate does not rule out smaller ones.

RECURRENCE

If $\text{matrix}[r][c] = 0$, $\text{left}[r][c] = \text{up}[r][c] = 0$. Otherwise $\text{left}[r][c] = 1 + \text{left}[r][c - 1]$ and $\text{up}[r][c] = 1 + \text{up}[r - 1][c]$, treating an out-of-bounds predecessor as 0. For side s ending at (r, c) , set $\text{top} = r - s + 1$ and $\text{firstCol} = c - s + 1$. Require $\text{left}[r][c] \geq s$ (bottom), $\text{up}[r][c] \geq s$ (right), $\text{left}[\text{top}][c] \geq s$ (top), and $\text{up}[r][\text{firstCol}] \geq s$ (left).

BASE CASES AND ORDER

Initialize both run tables to 0 and best to 0. Read cells row by row, left to right, so the left and upper predecessors are ready. The first row and first column use 0 for the missing predecessor. Earlier top-row cells and earlier cells in the current row are ready when checking candidates. An empty matrix, zero-width matrix, or all-zero matrix returns 0.

Time: $O(\text{rows} * \text{cols} * \min(\text{rows}, \text{cols}))$ worst case: each cell can test up to $\min(\text{rows}, \text{cols})$ sides with four constant-time checks.

Space: $O(\text{rows} * \text{cols})$ for the left and up run tables.

Watch out

Checking only the bottom and right runs can accept a square with a broken top or left border. The filled-square recurrence $1 + \min(\text{top}, \text{left}, \text{diagonal})$ solves a different problem and rejects valid hollow squares. X means a marked cell, not diagonal arms. Return the side length here; multiply by itself only when a variant asks for area.

Check your understanding

1. For $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$, what should this side-length function return?

- A. 1, because the center is 0
- B. 2, because only filled squares count
- C. 3, because all four outer edges are marked
- D. 9, because the square covers nine positions

2. At bottom-right (2, 2) of $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$, $\text{left}[2][2]$ and $\text{up}[2][2]$ are both 3. Why must the candidate of side 3 still be rejected?

- A. Its interior contains a 0
- B. Its top run $\text{left}[0][2]$ is only 1
- C. The bottom row has too many marked cells
- D. Every square must start in row 1

Flashcards: cover the answer to recall it

What do $\text{left}[r][c]$ and $\text{up}[r][c]$ store?

Consecutive 1s ending at (r, c) , horizontally and vertically. At a 1, add 1 to the preceding run; at a 0, reset to 0. Missing predecessors have length 0.

How do you certify side s ending at bottom-right (r, c) ?

Set $\text{top} = r - s + 1$ and $\text{firstCol} = c - s + 1$. Require $\text{left}[r][c] \geq s$ for bottom, $\text{up}[r][c] \geq s$ for right, $\text{left}[\text{top}][c] \geq s$ for top, and $\text{up}[r][\text{firstCol}] \geq s$ for left.

Does a center 0 invalidate an X bordered square? What does the answer measure?

No. X means a marked border cell, and the interior is unrestricted. This function returns side length, so a hollow three-by-three square returns 3. Its area would be 9.

Quiz answer key

1. C. All four edges have length 3 and contain only 1s. The interior may be 0, so side 3 is valid. The answer is 3; 9 would be the area.

2. B. The top row $[1, 0, 1]$ has a gap, giving $\text{left}[0][2] = 1 < 3$. A valid bottom and right edge do not certify the top or left. The smaller side-2 square in the bottom-right is valid, so the overall answer is 2.

Count of AP Subsequences: extend a matching difference

Find every arithmetic subsequence of length at least 3 in [2, 4, 6, 8]. The answers are [2, 4, 6], [4, 6, 8], and [2, 4, 6, 8]. AP means arithmetic progression: consecutive chosen terms have equal differences. Chosen indices must increase, but they need not be adjacent.

The idea to remember

- Store $dp[i][d]$ as the number of subsequences of length at least 2 ending at index i with common difference d . The endpoint and difference determine which future values can extend them.
- For each $j < i$, calculate $d = \text{values}[i] - \text{values}[j]$. If $\text{extend} = dp[j][d]$, then $\text{answer} += \text{extend}$ and $dp[i][d] += \text{extend} + 1$. Missing entries have count 0.
- The extra 1 stores the new pair (j, i) . Only the extended sequences enter the answer because only they now have at least 3 terms. A unique previous index j prevents double counting.
- Visit endpoints from left to right and begin with empty maps. Keep the original order. Equal values at different indices are separate choices; difference 0 and negative differences are allowed.
- Expect $O(n^2)$ map operations and $O(n^2)$ entries in the worst case. Use `BigInt` for counts and convert safe integer inputs to `BigInt` before taking differences. Counts and differences can exceed Number precision; `BigInt` arithmetic has additional bit costs.

Work through one example

values = [2, 4, 6, 8]

At value 6, $dp[2][2]$ stores two sequences: the pair [4, 6] and the triple [2, 4, 6]. Only the triple has entered answer, which is 1. Appending 8 extends both stored sequences, producing [4, 6, 8] and [2, 4, 6, 8]. Add 2 to answer. Store the new pair [6, 8] too, but do not count it yet.

3 arithmetic subsequences of length at least 3

Worked problem notes (1)

Open a problem for its state, decision, recurrence, base cases, and common mistake.

► Count of AP Subsequences

STATE: WHAT ONE ENTRY MEANS

$dp[i][d]$ counts index-distinct arithmetic subsequences of length at least 2 that end at index i and have common difference d . Each endpoint owns a map from differences to counts. Keep pairs in the map because a later value can extend one into a valid length-3 answer. A separate answer counts only lengths at least 3.

THE FINAL DECISION

Choose the previous index j before the new final index i . Every stored sequence ending at j with difference d can append values[i], so extend sequences now qualify for the answer. The extra 1 creates the new pair (j, i) , which belongs in dp but not in answer. Every new sequence has a unique previous index j , so different transitions count disjoint choices of indices.

RECURRENCE

```
For every  $j < i$ :  $d = \text{values}[i] - \text{values}[j]$ ;  $\text{extend} = dp[j][d]$  or 0;  $\text{answer} += \text{extend}$ ;  $dp[i][d] += \text{extend} + 1$ .
```

BASE CASES AND ORDER

Start every endpoint map empty and answer at 0. A missing difference means 0 sequences, not an uncomputed state. Visit i from left to right, then every j from 0 to $i - 1$. Each $dp[j]$ is complete before it is read. Inputs shorter than 3 return 0. Do not sort the input: subsequences must retain the original index order. JavaScript inputs must be safe integer Numbers; convert values to BigInt before subtraction so differences remain exact even when they exceed the safe Number range. Store counts and answer as BigInt too.

Time: $O(n^2)$ expected map operations across $n(n - 1)/2$ index pairs. BigInt arithmetic cost grows with the bit length of the values and counts.

Space: $O(n^2)$ map entries in the worst case, plus $O(n)$ converted values. This entry count excludes the variable bit length of BigInt keys and counts.

Watch out

Do not add $\text{extend} + 1$ to answer: the extra 1 is only a pair. Do not use one shared map for every endpoint or replace accumulated counts with $\text{extend} + 1$; multiple previous indices may contribute to the same endpoint and difference. Zero and negative differences are valid. Preserve index order and count equal-valued choices separately. Number counts can silently lose precision; this implementation returns BigInt, so use `answer.toString()` for display and avoid converting a large result back to Number.

Check your understanding

1. $dp[2][2] = 2$ stores $[4, 6]$ and $[2, 4, 6]$. When 8 is appended, what does this transition add to answer, and what count does it contribute to $dp[3][2]$?

- A. Answer +2; map +3
- B. Answer +3; map +3
- C. Answer +1; map +2
- D. Answer +2; map +2

2. How many arithmetic subsequences of length at least 3 does $[5, 5, 5, 5]$ contain, counting different index choices separately?

- A. 1
- B. 2
- C. 4
- D. 5

Flashcards: cover the answer to recall it

What does $dp[i][d]$ count in arithmetic-subsequence counting?

Index-distinct arithmetic subsequences with at least 2 terms, ending exactly at index i with common difference d . Pairs are stored to seed later triples.

Why does a transition add extend to answer but extend + 1 to $dp[i][d]$?

The extend old sequences gain a term and now have at least 3 terms. The extra 1 starts the pair (j, i) , which can be extended later but is not yet a qualifying answer.

Can the input be sorted or equal values deduplicated before counting?

No. Sorting changes the allowed index order, and deduplication removes distinct index choices. Keep all original positions, including repeated values; zero and negative differences are valid.

Quiz answer key

1. **A.** Add 2 to answer for $[4, 6, 8]$ and $[2, 4, 6, 8]$. Contribute $2 + 1 = 3$ to the map because $[6, 8]$ must also be available for future extensions.

2. **D.** All four triples and the one quadruple have difference 0, giving $4 + 1 = 5$. Identical values do not merge different choices of indices.