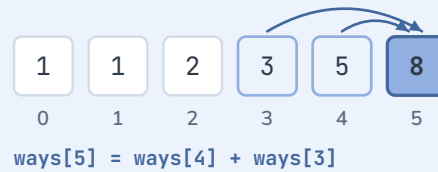


Dynamic programming I

Chapter 3.4 · Overall level: Intermediate



A staircase has five steps and you climb one or two steps at a time. *In how many different orders can you reach the top?* You can list them by hand. For a staircase of a million steps you cannot, but a short table can.

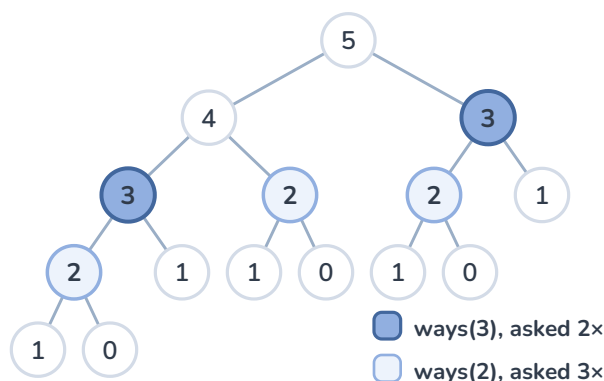
Dynamic programming, or DP, names the smaller questions a big question depends on, answers each of them once and stores the answers in a table. This chapter builds the habit on four small problems.

3.4.1 Recursion that repeats itself

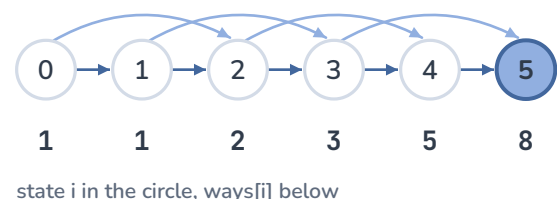
Let $\text{ways}(i)$ be the number of ordered sequences of 1-steps and 2-steps whose total is exactly i . Think about the **last** move of a sequence. If it is a 1-step, the moves before it total $i - 1$; if it is a 2-step, they total $i - 2$. So

$$\text{ways}(i) = \text{ways}(i - 1) + \text{ways}(i - 2)$$

A recursive function that follows this rule is correct, and it is short. It is also very slow. The left half of Figure 3.4.1 shows every call made while computing $\text{ways}(5)$: 15 calls, although there are only six different questions, $\text{ways}(0)$ to $\text{ways}(5)$.



Call tree of $\text{ways}(5)$: 15 calls



Six distinct states, each answered once

Figure 3.4.1 — Different recursive paths can ask the same question; DP answers each distinct state once. Each node is a call $\text{ways}(i)$, labelled with i . On the right, an arrow points from a state to the state that uses it.

The repetition grows quickly. Computing `ways(30)` this way makes 2,692,537 calls, and `ways(40)` makes 331,160,281, while the number of distinct questions is only 31 or 41. The call count itself grows like the answers do, by roughly 1.6 times per extra step.

3.4.2 Memoize, then tabulate

The cure is to remember. **Memoization** keeps a table `memo` beside the recursive function: before working on `ways(i)`, look in `memo[i]`; if the answer is there, return it at once; otherwise compute it and store it. Every state is now computed once, so `ways(n)` costs $O(n)$ work. The tree on the left collapses into the six-node graph on the right.

Tabulation goes one step further and removes the recursion altogether. The graph shows that state i only needs smaller states, so filling the table in increasing order of i guarantees that every answer is ready before it is needed:

```
ways[0] ← 1
ways[1] ← 1
for i from 2 to n:
    ways[i] ← ways[i - 1] + ways[i - 2]
```

Tabulation is usually the better choice in contest code. It has no recursion depth to worry about (a recursive version of `ways(1000000)` would nest a million calls deep, far beyond Python's default limit of about 1000) and its loops show the evaluation order plainly. Memoization is still useful when only a few of the possible states are ever reached.

3.4.3 The four questions of a DP

Every dynamic program in this chapter is designed by answering the same four questions, in this order.

Question	What to write down	For the staircase
State	A complete sentence saying what one table entry means	<code>ways[i]</code> = number of ordered sequences of 1s and 2s with total exactly i
Transition	How an entry is built from smaller entries, and why every case is covered once	last move is 1 or 2: <code>ways[i - 1] + ways[i - 2]</code>
Base	Entries that are known without the transition	<code>ways[0] = 1</code> (the empty sequence), <code>ways[1] = 1</code>
Order	An order in which every dependency is computed first	increasing i

The sentence for the state is the most important line. If you cannot say in words what $\text{dp}[i]$ means, you cannot check whether the transition is right. “ways[i] is the answer for i” is not a sentence; it names nothing that can be counted.

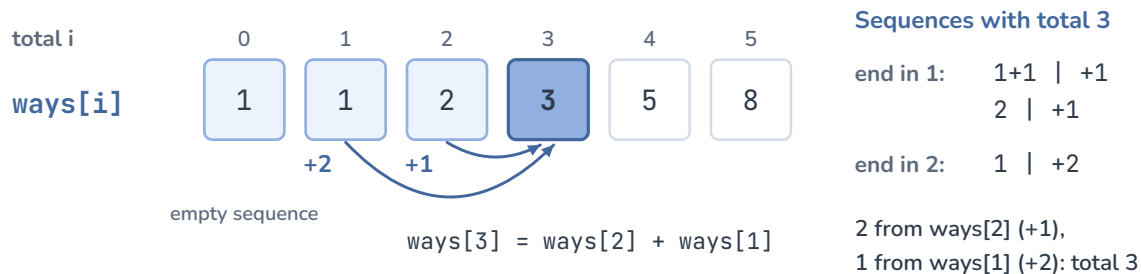


Figure 3.4.2 — The table for the staircase, with the total i above each entry. The three sequences with total 3 split by their last move: two end with a 1-step (one for each sequence counted by $\text{ways}[2]$) and one ends with a 2-step (counted by $\text{ways}[1]$).

Why the transition is right: cases on the last move

Take any total $i \geq 2$. Every non-empty sequence ends with exactly one move, a 1 or a 2, so the sequences fall into two groups that do not overlap. Removing the last 1-step from a sequence in the first group leaves a sequence with total $i - 1$, and adding a 1-step to any sequence with total $i - 1$ gives one back. This matching is one-to-one, so the first group has exactly $\text{ways}[i - 1]$ members. The same argument gives $\text{ways}[i - 2]$ for the second group. Adding the two groups counts every sequence once.

This “cases on the last choice” argument is the standard way to prove a DP transition. The cases must cover every object and must not overlap; otherwise the sum counts something twice or misses something.

Why ways[0] is 1

It is tempting to say that a total of 0 can be reached in 0 ways. But there is exactly one sequence with total 0: the empty sequence, which makes no moves. The recurrence agrees: $\text{ways}[2]$ must be 2 (the sequences 1+1 and 2), and $\text{ways}[1] + \text{ways}[0]$ gives 2 only if $\text{ways}[0] = 1$. The 2-step sequence “2” is the empty sequence followed by one 2-step. Bases are not guesses; they follow from the state sentence applied to the smallest case.

3.4.4 Huge counts and arithmetic modulo 1,000,000,007

Counts grow fast. `ways[100]` already has 21 digits, and `ways[1000000]` has 208,988 digits. No 64-bit integer can hold such numbers, so counting problems usually ask for the answer **modulo** 1,000,000,007.

$a \bmod M$ is the remainder when a is divided by M ; it is always between 0 and $M - 1$. For example, $17 \bmod 7 = 3$, because $17 = 2 \times 7 + 3$. The key fact is that remainders can be added before or after reducing:

$$(a + b) \bmod M = ((a \bmod M) + (b \bmod M)) \bmod M$$

Why: write $a = qM + r$ and $b = sM + t$. Then $a + b = (q + s)M + (r + t)$, and the multiple of M does not change the remainder. So a program may keep only the remainder of every table entry and reduce after each addition. The numbers it stores never exceed $M - 1$, yet the final remainder is exactly the remainder of the true count.

i	0	1	2	3	4	5	6	7	8
<code>ways[i]</code>	1	1	2	3	5	8	13	21	34
<code>ways[i] mod 7</code>	1	1	2	3	5	1	6	0	6
two remainders added			2	3	5	8	6	7	6
that sum mod 7			2	3	5	1	6	0	6

The table uses the small modulus 7 so that you can check it by hand. The last row only ever adds two remainders and reduces, yet it agrees with the true counts reduced (second row).

Why 1,000,000,007

It is a prime number a little above 10^9 . Each stored remainder is below 1,000,000,007, so the sum of two remainders is at most 2,000,000,012, which still fits in a signed 32-bit integer (limit 2,147,483,647). Products of two remainders do not fit (they reach about 10^{18}), which is why the programs below use 64-bit integers throughout; that habit costs nothing and protects later problems that multiply.

Two warnings. First, reduce **after every addition**, not only at the end: by the end the true number has long since overflowed. Second, modulo belongs to counting only. The remainder of a minimum or maximum means nothing, because $1,000,000,008 \bmod M = 1$ is smaller than $2 \bmod M$.

Pseudocode

Step sequences: $O(n)$ time, $O(1)$ extra memory

```
M ← 1000000007
read n
prev ← 1                (ways[0])
cur ← 1                 (ways[1])
repeat for i = 2 to n:
    (prev, cur) ← (cur, (cur + prev) mod M)
output cur
```

Each entry needs only the two before it, so the program keeps two variables instead of the whole table. For $n = 0$ and $n = 1$ the loop does not run and the answer is 1, as the bases say.

3.4.5 Code in four languages

Each program solves the first practice problem, **Step sequences**. The input is one integer n ($0 \leq n \leq 1,000,000$). The output is the number of ordered sequences of moves of length 1 or 2 with total n , modulo 1,000,000,007.

Sample input	Sample output
5	8

Python 3

```
import sys

MOD = 1_000_000_007

def main():
    n = int(sys.stdin.readline())
    prev, cur = 1, 1          # ways[0] and ways[1]
    for _ in range(2, n + 1):
        prev, cur = cur, (cur + prev) % MOD
    print(cur)

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    const long long MOD = 1000000007;
    int n;
    cin >> n;
    long long prev = 1, cur = 1;    // ways[0] and ways[1]
    for (int i = 2; i <= n; i++) {
        long long next = (cur + prev) % MOD;
        prev = cur;
        cur = next;
    }
    cout << cur << '\n';
    return 0;
}
```

Java 17

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        final long MOD = 1_000_000_007L;
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(in.readLine().trim());
        long prev = 1, cur = 1;    // ways[0] and ways[1]
        for (int i = 2; i <= n; i++) {
            long next = (cur + prev) % MOD;
            prev = cur;
            cur = next;
        }
        System.out.println(cur);
    }
}
```

JavaScript (Node)

```
const MOD = 1000000007;
const n = Number(require("fs").readFileSync(0, "utf8").trim());
let prev = 1, cur = 1;    // ways[0] and ways[1]
for (let i = 2; i <= n; i++) {
    const next = (cur + prev) % MOD;
    prev = cur;
    cur = next;
}
console.log(cur);
```

In JavaScript, ordinary numbers are exact below $2^{53} \approx 9 \times 10^{15}$, and the sum of two remainders stays below 2.1×10^9 , so `Number` is safe here. A product of two remainders would not be; it would need `BigInt`.

3.4.6 Fewest coins: a minimum and an unreachable state

You have unlimited coins of values 1, 3 and 4 and want to pay exactly 6 with as few coins as possible. The natural rule “always take the largest coin that fits” pays $4 + 1 + 1$, three coins. But $3 + 3$ uses only two. A greedy rule is not a proof, so let a table decide.

State. $\text{best}[s]$ is the fewest coins whose values add up to exactly s , or ∞ if no choice of coins does.

Base. $\text{best}[0] = 0$: the empty choice pays 0. **Transition.** The last coin paid has some value $c \leq s$, and the coins before it pay $s - c$. **Order.** Increasing s .

$$\text{best}[s] = 1 + \min(\text{best}[s - c]) \quad \text{over coins } c \leq s$$

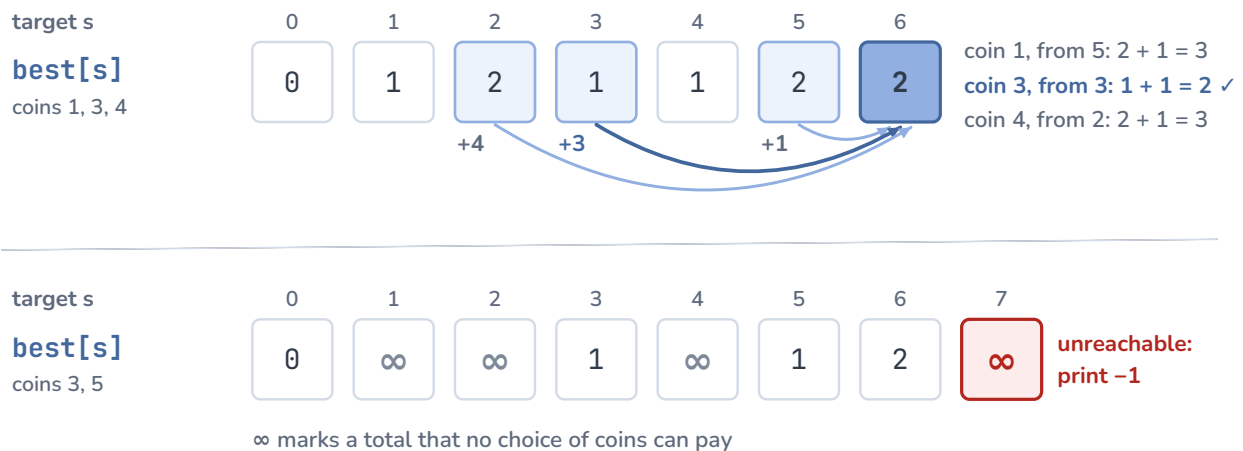


Figure 3.4.3 — Top: each entry looks back by every coin value. For $s = 6$ the candidates come from $\text{best}[5]$, $\text{best}[3]$ and $\text{best}[2]$; the minimum, through coin 3, is 2. Bottom: with coins 3 and 5, some totals cannot be paid at all, and their entries keep the sentinel ∞ .

Why the minimum is right. Take an optimal way to pay $s > 0$ and look at its last coin c . The other coins pay $s - c$, and they must be an optimal way to pay it: if fewer coins paid $s - c$, adding coin c to them would beat our optimum. So the optimum is $1 + \text{best}[s - c]$ for some coin c , and taking the minimum over all coins finds it. The last coin can be any coin, so every coin must be tried.

The unreachable sentinel

With coins 3 and 5, a total of 7 is impossible. The table must say so, and the value it uses for “impossible” is called a **sentinel**. Zero is a bad sentinel: zero coins is a real answer, but only for $s = 0$. If $\text{best}[4]$ were 0, then $\text{best}[7]$ would become $1 + \text{best}[4] = 1$, a “one-coin” payment that does not exist. Use a value larger than any real answer instead, such as $x + 1$ when the target is x (no answer uses more than x coins). Then $1 + \infty$ never wins a minimum against a real answer, and an entry that is still ∞ at the end means “impossible”, printed as -1 in the practice problem. Because this is a minimum, entries combine with $\min$ and are never reduced modulo anything.

3.4.7 Counting routes in a grid

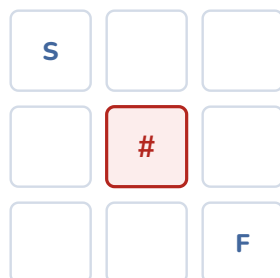
A grid has free cells (.) and blocked cells (#). A route starts in the top-left cell, ends in the bottom-right cell and moves one cell right or one cell down at a time. How many routes avoid every blocked cell?

State. $dp[r][c]$ is the number of routes from the start to cell (r, c) . **Transition.** The last move into a free cell came either from the cell above, $(r - 1, c)$, or from the cell to the left, $(r, c - 1)$. These are two different last moves, so the two groups of routes do not overlap, and

$$\begin{aligned} dp[r][c] &= dp[r - 1][c] + dp[r][c - 1] && \text{for a free cell} \\ dp[r][c] &= 0 && \text{for a blocked cell} \end{aligned}$$

Base and borders. The start cell has $dp[1][1] = 1$ if it is free (the empty route) and 0 if it is blocked. Cells on the top row have no cell above, and cells in the left column have none to their left. Instead of special cases, give the table an extra row 0 and column 0 filled with zeros: “outside the grid” then simply contributes no routes. **Order.** Row by row, left to right, so the cell above and the cell to the left are already done.

The grid



S start, F finish, # blocked;
moves go right or down

Route counts $dp[r][c]$

	c = 0	1	2	3	
r = 0	0	0	0	0	row 0, column 0: outside the grid, zero routes
1	0	1	1	1	red: blocked, 0
2	0	1	0	1	
3	0	1	1	2	from above: 1 from the left: 1 total 1 + 1 = 2

Figure 3.4.4 — Every path to a free cell arrives from above or left, so their counts add. The blocked centre holds 0, so no route passes through it, and the grey row 0 and column 0 stand for “outside the grid”.

The answer for the grid in the figure is 2: around the top and right edges, or around the left and bottom edges. For a 1000×1000 grid the count is astronomically large, so it is reported modulo 1,000,000,007 and every addition is reduced, as in Section 3.4.4. Grid routes are also counted in a different way in some textbooks, with a binomial coefficient, but that formula breaks as soon as a single cell is blocked; the table does not care.

The table needs $n \times m$ entries, but each row only reads the row above it, so two rows (or even one, updated left to right) are enough. Time is $O(nm)$.

3.4.8 Take or skip: a one-dimensional optimum

Now choose some positions of an array so that no two chosen positions are neighbours, and make the chosen values add up to as much as possible. Take $a = [6, 7, 6, 2, 5]$. Grabbing the largest value first takes 7, which rules out both 6s; then 5 is the best remaining choice, for 12 in total. But $6 + 6 + 5 = 17$ is better. Again, a table settles it.

State. $\text{best}[i]$ is the largest total of a legal selection that uses only positions 1 to i . Choosing nothing is legal, so the empty prefix has $\text{best}[0] = 0$, and $\text{best}[1] = \max(0, a[1])$.

Transition: cases on position i . Either position i is skipped, and the rest is a legal selection from positions 1 to $i - 1$; or it is taken, which forbids position $i - 1$ only, so the rest is a legal selection from positions 1 to $i - 2$.

$$\text{best}[i] = \max(\text{best}[i - 1], \text{best}[i - 2] + a[i])$$

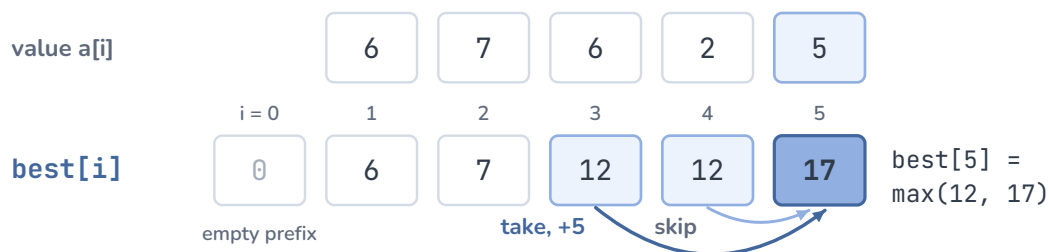


Figure 3.4.5 — Each entry compares skipping position i (copy $\text{best}[i - 1]$) with taking it ($\text{best}[i - 2] + a[i]$). The grey entry is the empty prefix.

A tiny exhaustive check

For five positions there are only 13 legal selections, so we can compare the table with every one of them:

Selection (positions)	none	1	2	3	4	5	1,3	1,4	1,5	2,4	2,5	3,5	1,3,5
Total	0	6	7	6	2	5	12	8	11	9	12	11	17

The largest is 17, exactly $\text{best}[5]$. Such a brute-force comparison on small inputs is the best way to test a new recurrence: it catches a wrong base or a missing case at once. Note that the count of legal selections, 13, is itself $\text{ways}[6]$ from the staircase; the same take-or-skip split counts them.

Why the take-or-skip cases are complete

An optimal selection from positions 1 to i either contains i or does not. If it does not, it is a legal selection from the first $i - 1$ positions, so it is worth at most $\text{best}[i - 1]$. If it does, its other positions avoid $i - 1$, so they form a legal selection from the first $i - 2$ positions, worth at most $\text{best}[i - 2]$. Both values are also achievable, so the maximum of the two cases is exactly the optimum.

3.4.9 Three kinds of state

The four problems in this chapter used the same four questions, but they combine their cases differently. Recognising the kind of question tells you the combining rule, the base and the meaning of “unreachable”.

Kind of question	Combine cases with	Typical base	Unreachable state	Large values
How many? (counting)	addition	the empty object counts 1	0	reduce modulo M
How few / how much? (optimum)	min or max	the empty choice costs 0	∞ or $-\infty$ sentinel	64-bit integers; never modulo
Is it possible?	or (any case works)	the start is possible	false	none

For example, “can coins 3 and 5 pay 7 exactly?” uses `can[s] = can[s - 3] or can[s - 5]` with `can[0] = true`. It is the minimum-coins table with every number replaced by “yes” and every ∞ by “no”.

Cost of the four tables

Problem	States	Work per state	Time	Memory with rolling
Step sequences	$n + 1$	1 addition	$O(n)$	$O(1)$: two values
Fewest coins (k coin values)	$x + 1$	k candidates	$O(kx)$	$O(x)$: no rolling
Grid routes	$n \times m$	1 addition	$O(nm)$	$O(m)$: one row
Take or skip	$n + 1$	1 comparison	$O(n)$	$O(1)$: two values

A take-or-skip total can reach $200,000 \times 10^6 = 2 \times 10^{11}$, beyond 32-bit integers: use `long` in C++ and `Long` in Java.

3.4.10 Where this method stops

DP needs the states to form a graph **without cycles**: there must be an order in which every state comes after the states it uses. If a grid route may move in all four directions and you want the fewest moves, the “state” of a cell depends on its neighbours, which depend on it; there is no valid order. That is a shortest-path question: use breadth-first search (Chapter 3.1) or Dijkstra’s algorithm (Chapter 3.3).

DP also needs a **manageable number of states**. Paying a target of 10^{12} with coins needs a table of 10^{12} entries, which is far too large. Chapter 3.5 uses richer states (items and capacity, two string prefixes, a set of used jobs) and shows how to count memory first.

Greedy rules are sometimes right: for coins 1, 5, 10 and 25, largest-first is optimal. That needs its own proof (Chapter 2.5); coins 1, 3, 4 show it can fail.

3.4.11 Common mistakes

- **A recurrence with no state meaning.** Writing $dp[i] = dp[i - 1] + dp[i - 2]$ because it “looks like” the problem is guessing. Write the state sentence first, then derive the transition from cases on the last choice.
- **Wrong base for the empty prefix.** $ways[0] = 0$ makes every later count 0 or wrong; $best[1] = a[1]$ is wrong when $a[1]$ is negative, because choosing nothing is allowed. Apply the state sentence to the empty case and see what it says.
- **Double counting the same object.** The staircase counts $1 + 2$ and $2 + 1$ as different sequences, which is right for ordered moves. If a problem asks for *combinations* of coins, where order does not matter, the same recurrence counts the combination $\{1, 2\}$ twice. Each object must have exactly one “last choice” under your description.
- **Zero as the sentinel, or modulo on a minimum.** Both turn impossible or huge values into small, believable answers. Use ∞ for “unreachable” and keep min/max values exact.
- **Reducing only at the end.** The unreduced count overflows long before the last line. Reduce after every addition.
- **Filling in the wrong order.** If the loop computes $best[s]$ for decreasing s , the entries it reads are still empty. Check that every arrow in the dependency graph points forward in your loop order.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● Why is $ways[0] = 1$ when counting sequences of steps?
2. ● Why is zero a bad sentinel for an unreachable state in the fewest-coins table?
3. ● Extend the staircase table to $ways[6]$ and $ways[7]$. What is $ways[7] \bmod 7$?
4. ●● Coins have values 1, 5 and 6. Fill $best[0..10]$ for the fewest coins. How many coins does “largest coin first” use for 10, and how many does the table give?
5. ●● Now moves of 1, 2 or 3 steps are allowed. Write the state, transition and bases, and find the number of sequences with total 5.
6. ●● A grid has 2 rows and 3 columns, all free. Fill dp with its zero row and column and give the number of routes. Then block the middle cell of the top row: what changes?

Answers and explanations

Read these after attempting the exercises.

1. There is exactly one sequence with total 0: the empty sequence, which makes no moves. The recurrence needs it too: $\text{ways}[2] = \text{ways}[1] + \text{ways}[0]$ must be 2.
2. Zero coins is a real optimum, but only for target 0. A zero stored for an unreachable total would make $1 + 0 = 1$ look like a valid one-coin payment for a larger total. Use a value larger than any real answer.
3. $\text{ways}[6] = 8 + 5 = 13$ and $\text{ways}[7] = 13 + 8 = 21$. Since $21 = 3 \times 7$, $\text{ways}[7] \bmod 7 = 0$; the table in Section 3.4.4 gets the same from $(6 + 1) \bmod 7$.
4. $\text{best}[0..10] = 0, 1, 2, 3, 4, 1, 1, 2, 3, 4, 2$. Largest coin first pays $6 + 1 + 1 + 1 + 1$, **five** coins; the table finds **two** ($5 + 5$). $\text{best}[10]$ comes from $\text{best}[5] + 1$ through the coin 5.
5. $\text{ways}[i]$ = number of ordered sequences of 1s, 2s and 3s with total exactly i ; $\text{ways}[i] = \text{ways}[i - 1] + \text{ways}[i - 2] + \text{ways}[i - 3]$ for $i \geq 3$, with $\text{ways}[0] = 1$, $\text{ways}[1] = 1$, $\text{ways}[2] = 2$. Then 1, 1, 2, 4, 7, **13**.
6. With the zero row and column, rows 1 and 2 are 1 1 1 and 1 2 3, so there are **3** routes. If the top-middle cell is blocked, row 1 becomes 1 0 0 and row 2 becomes 1 1 1: only **1** route remains (down first, then right twice).

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Step sequences

Apply

Count ordered sequences of moves of length 1 or 2 that sum to n . The empty sequence counts once when $n=0$. Print the count modulo 1000000007.

Fewest coins

Connect

You have unlimited copies of n distinct positive coin denominations. Find the minimum coins needed to sum to x exactly, or -1 if impossible.

Grid route count

Reason

Count routes from top-left to bottom-right of a grid, moving only one cell right or down. Blocked cells may not be used. Print the count modulo 1000000007.

Separated selections

Stretch

Choose any positions of an array, with no two chosen positions adjacent. Maximize their value sum. Choosing no position is allowed.