Chapter 3: Dynamic Programming

Advanced Algorithms

Overlapping subproblems | Memoization and tabulation | Fibonacci | 0/1 and unbounded knapsack | Subset sum | Longest increasing subsequence | Longest common substring | Longest common subsequence

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_03_Dynamic_Programming.html

Learning Outcomes

Recognition

Spot **overlapping subproblems** and **optimal substructure** in a problem; decide when dynamic programming applies.

Formulation

Define the subproblem (state), write the recurrence and base cases, and fix the order in which the table is filled.

Implementation

Fill the tables for Fibonacci, 0/1 and unbounded knapsack, subset sum, LIS, longest common substring, and LCS by hand, and code them in Java.

Analysis

Compare memoization with tabulation; justify the $O(n \cdot W)$, $O(m \cdot n)$, $O(n^2)$, and $O(n \log n)$ bounds; reconstruct the solution from the table.

What Is Dynamic Programming?

- Complex problems sometimes call for **clever methods**; dynamic programming is one of them.
- It breaks a big problem into **small, recurring subproblems** and solves them step by step.
- Each subproblem's solution is written into a **table**; when the same subproblem comes up again, the answer is **read from the table**, which saves time.
- It is widely used for **optimization problems** (largest value, shortest path, fewest operations).

In short

DP = recursion + **remembering**. Every DP problem can also be solved by plain recursion; DP makes sure the same work is **never done twice**.

The name comes from Richard Bellman (1950s); "programming" here means **planning with a table**, not writing code.

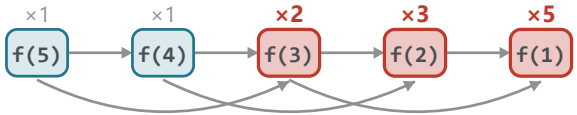
BASIC CONCEPTS

Two Key Properties for DP

1. Overlapping subproblems

The recursive solution calls **the same subproblem again and again**. Example: computing fib(5) computes fib(3) twice and fib(2) three times.

If the subproblems are independent (as in merge sort), a table does not help; that is **divide and conquer**.



2. Optimal substructure

An optimal solution of the problem can be built from **optimal solutions** of its subproblems.

Bellman's principle of optimality: every subsolution contained in an optimal solution is itself optimal for its subproblem.

A Bellman equation example (maximum subarray): if V(i) is the largest sum ending at i, then `V(i) = max(a[i], V(i-1) + a[i])`. The value is computed from the **optimal** value of the previous subproblem.

Steps in Designing a DP Solution

1. Study the **structure of an optimal solution**: which subproblems must be solved?
2. Write the **state and recurrence**: express the solution in terms of smaller subproblems (recurrence + base cases).
3. Fix the **order of computation**: when a cell is computed, the cells it needs must be ready.
4. Fix the **table size**: how many dimensions, how many values each? (e.g., $(n+1) \times (W+1)$)
5. **Fill the table:** bottom-up (tabulation) or top-down (memoization).
6. **Reconstruct the solution:** not just the value; the chosen items are found by reading the table backward.

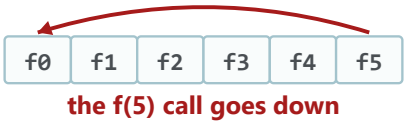
Total time ≈ (number of states) × (time per state). This rule explains every bound in this chapter.

BASIC CONCEPTS

Memoization and Tabulation

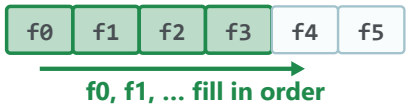
Top-down: memoization

- Write the recursive solution as is; store the results in a **cache** (array / HashMap).
- Before a call, check the table: if the value is there, return it; otherwise compute and store it.
- Only the subproblems **actually needed** are solved.
- Deep recursion risks a stack overflow (StackOverflowError).



Bottom-up: tabulation

- Start from the smallest subproblem and fill the table in order with **loops**.
- No recursion and no call overhead; usually faster.
- The fill order must be worked out correctly in advance.
- If only the previous row is needed, **memory can be reduced** (e.g., O(W)).



Both solve each subproblem **once**; the asymptotic time is the same. They differ in order and style.

BASIC CONCEPTS

DP, Divide and Conquer, and Greedy

	Divide and Conquer	Dynamic Programming	Greedy
Subproblems	independent	overlapping	a single choice per step
Choice	none; divide and combine	try every option, store the best in the table	take the current best, never go back
Optimality	—	guaranteed if optimal substructure holds	only with the greedy-choice property
Example	merge sort, binary search	0/1 knapsack, LCS, Bellman-Ford	fractional knapsack, Dijkstra, Kruskal
Cost	recurrence, master theorem	number of states × work per state	usually O(n log n): sort + one pass
Memory	recursion stack	the table (often O(n · W) or O(m · n))	usually O(1) extra memory

Greedy can be seen as dynamic programming that looks at only one option per step: if that choice can be proved correct, no table is needed.

Applications of Dynamic Programming

Shortest paths
GPS devices and maps: Bellman-Ford, Floyd-Warshall.

String comparison
diff, spell checking: LCS, edit distance.

Bioinformatics
DNA/protein alignment (Needleman-Wunsch, Smith-Waterman), RNA structure prediction.

Resource allocation
Knapsack, budget and investment planning, production-line scheduling.

Compilers and numerics
Matrix-chain multiplication, optimal binary search trees, parsing (CYK).

Artificial intelligence
Chess-playing computers, machine translation, reinforcement learning (the Bellman equation).

Fibonacci: Plain Recursion

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

$$F(0) = 0,\ F(1) = 1$$
$$F(n) = F(n-1) + F(n-2),\ n \geq 2$$

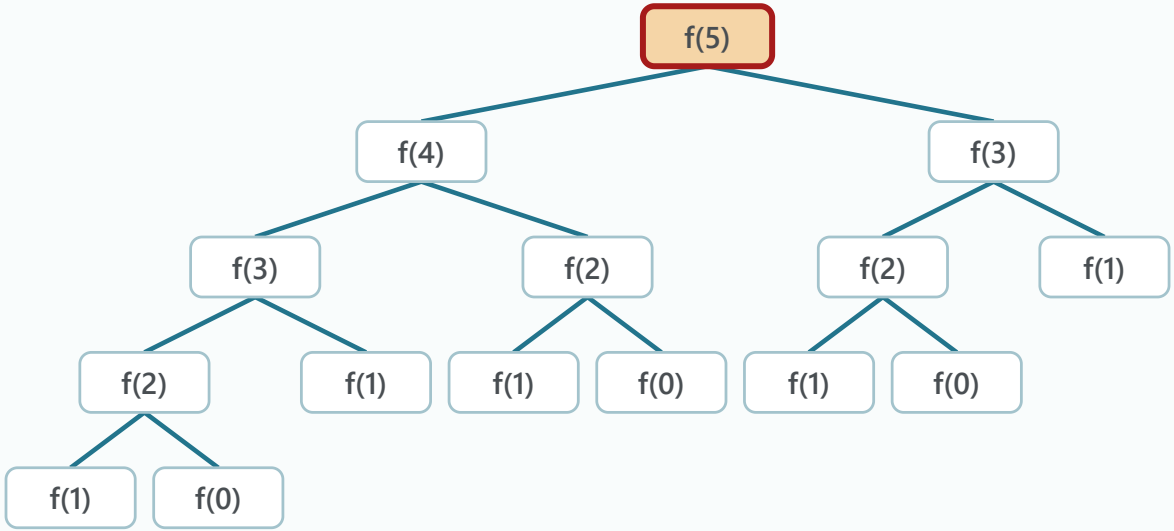
```
static long fib(int n) {
  if (n <= 1)
    return n;          // base cases
  return fib(n - 1) + fib(n - 2);
}
```

- A literal copy of the definition; correct but **very slow**.
- Number of calls: `T(n) = T(n-1) + T(n-2) + 1` → **$\Theta(\varphi^n)$**, $\varphi \approx 1.618$ (the golden ratio).
- fib(40) makes about **331 million** calls; fib(50) takes minutes.
- Why? The same subproblems (fib(3), fib(2), …) are computed **again and again**: overlapping subproblems.

FIBONACCI

The fib(5) Recursion Tree: Call Order

Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).

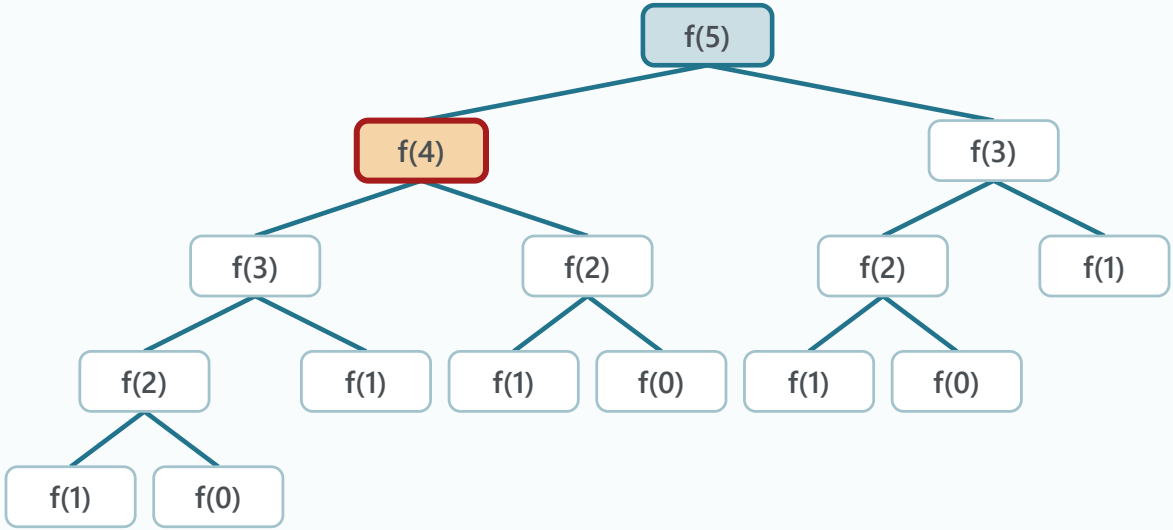


`fib(5)` called. Stack: fib(5)

Step 1 / 16

The fib(5) Recursion Tree: Call Order

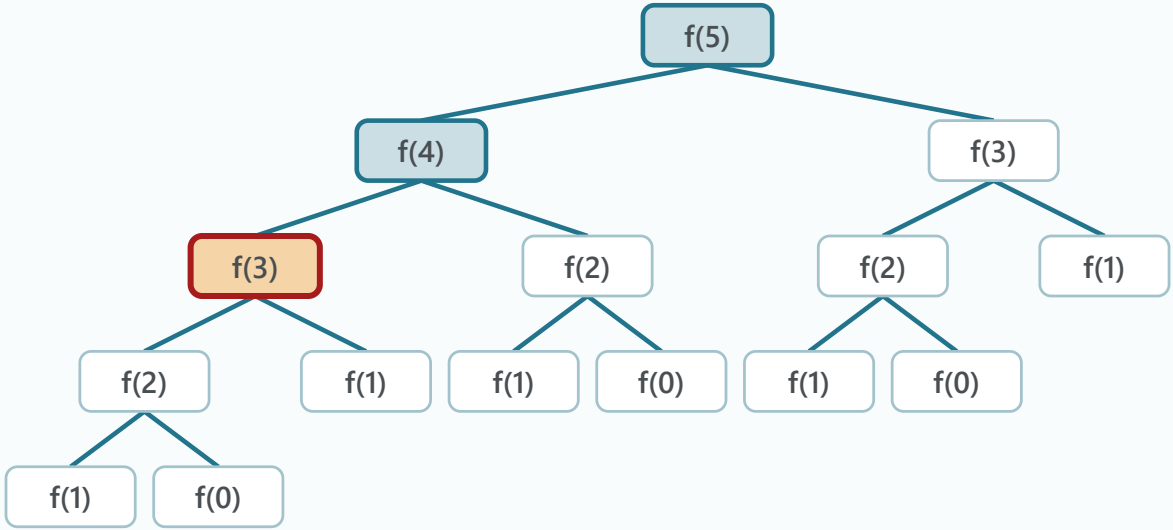
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(4)` called. Stack: fib(5) → fib(4)

The fib(5) Recursion Tree: Call Order

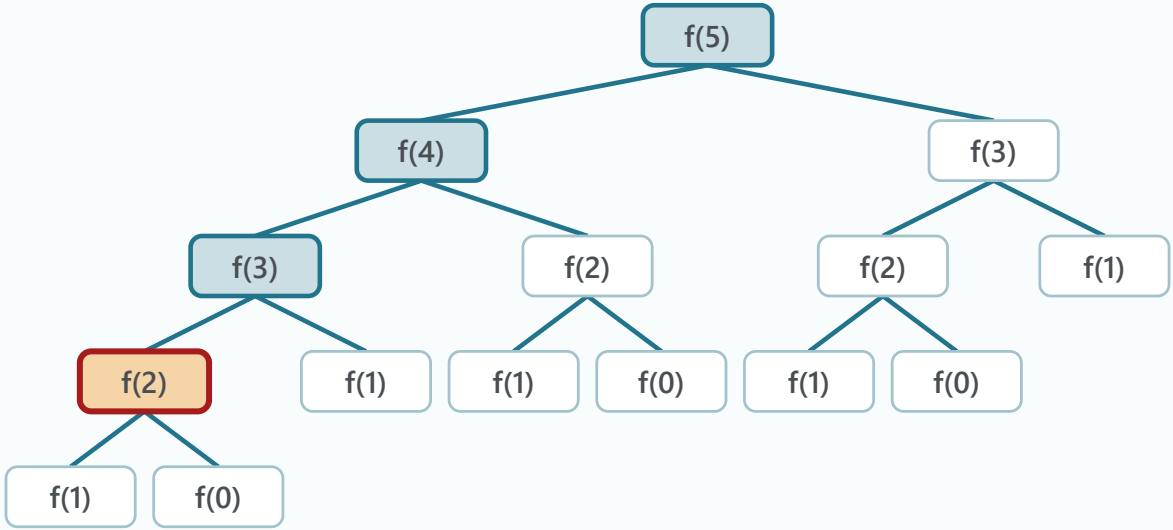
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(3)` called. Stack: fib(5) → fib(4) → fib(3)

The fib(5) Recursion Tree: Call Order

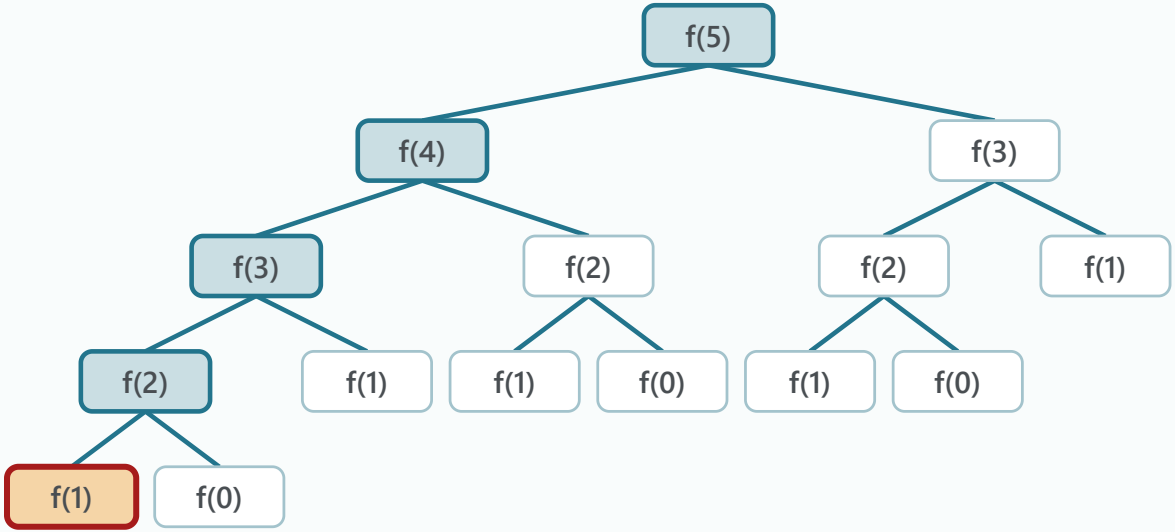
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(2)` called. Stack: fib(5) → fib(4) → fib(3) → fib(2)

The fib(5) Recursion Tree: Call Order

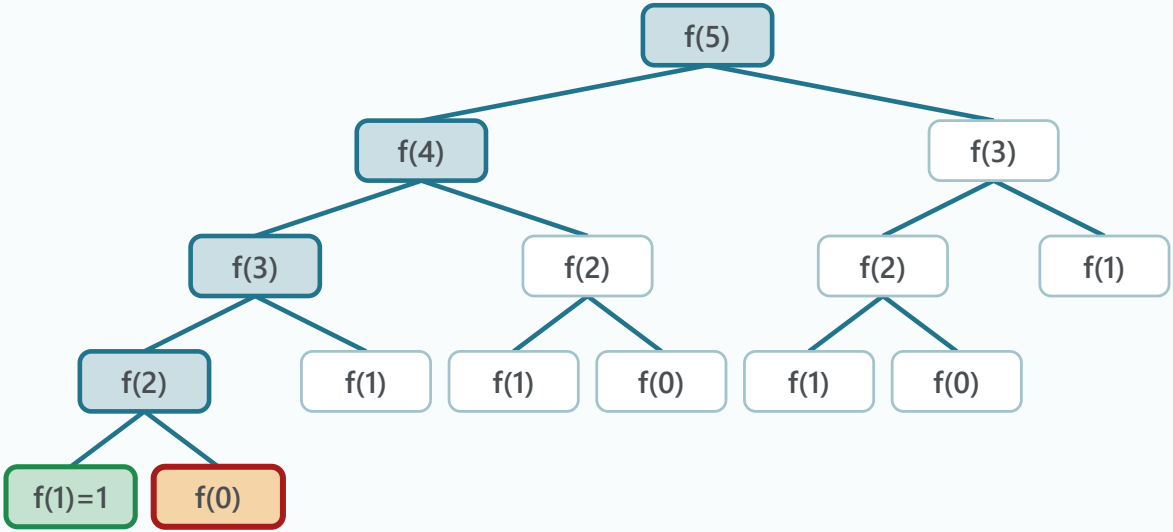
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(1)` base case: returns **1** at once. Stack: fib(5) → fib(4) → fib(3) → fib(2) → fib(1)

The fib(5) Recursion Tree: Call Order

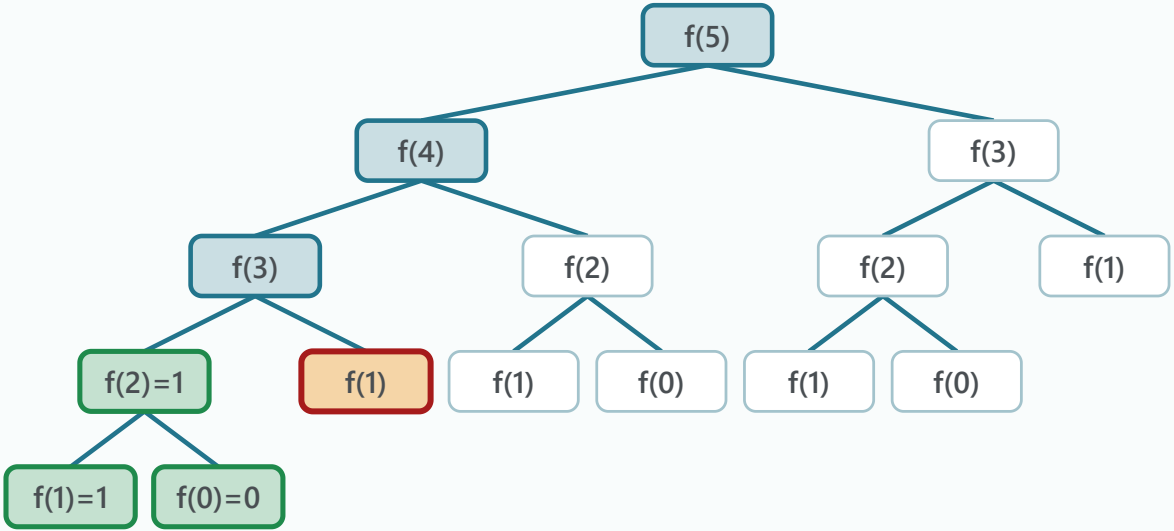
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(0)` base case: returns **0** at once. Stack: fib(5) → fib(4) → fib(3) → fib(2) → fib(0)

The fib(5) Recursion Tree: Call Order

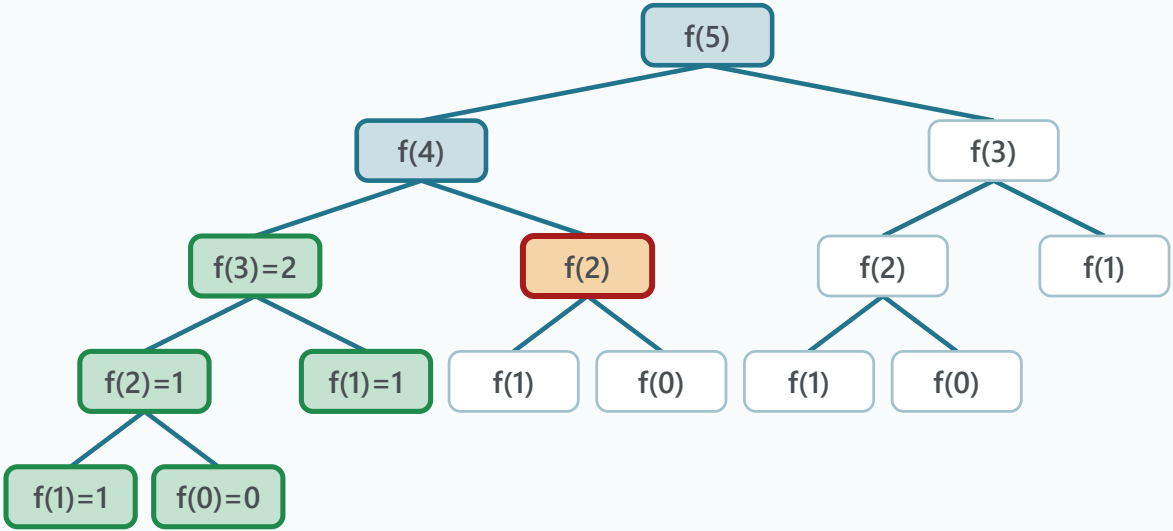
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(1)` base case: returns **1** at once. Stack: fib(5) → fib(4) → fib(3) → fib(1)

The fib(5) Recursion Tree: Call Order

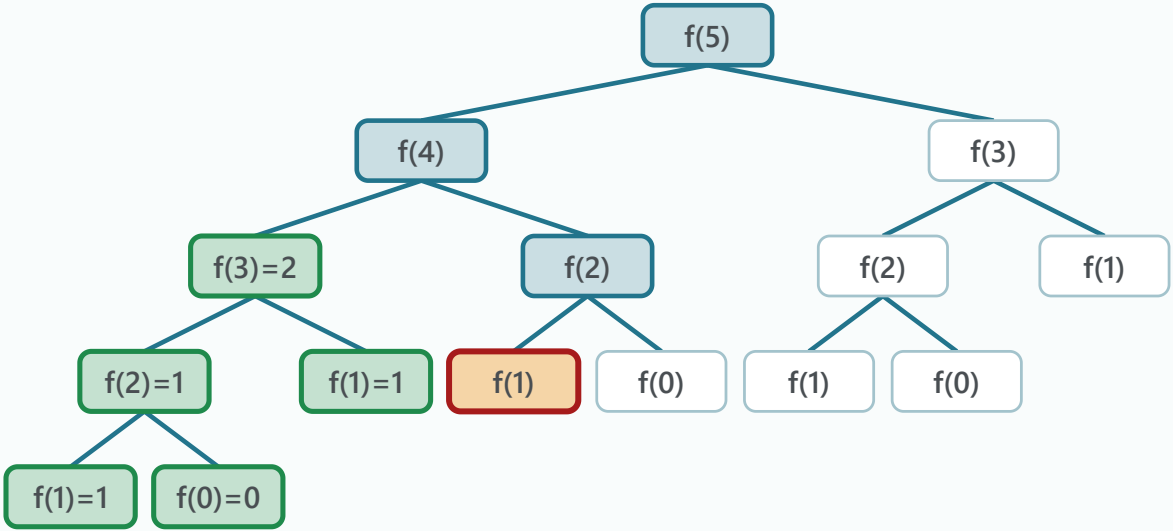
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(2)` called — computed for the **2nd time**! Stack: fib(5) → fib(4) → fib(2)

The fib(5) Recursion Tree: Call Order

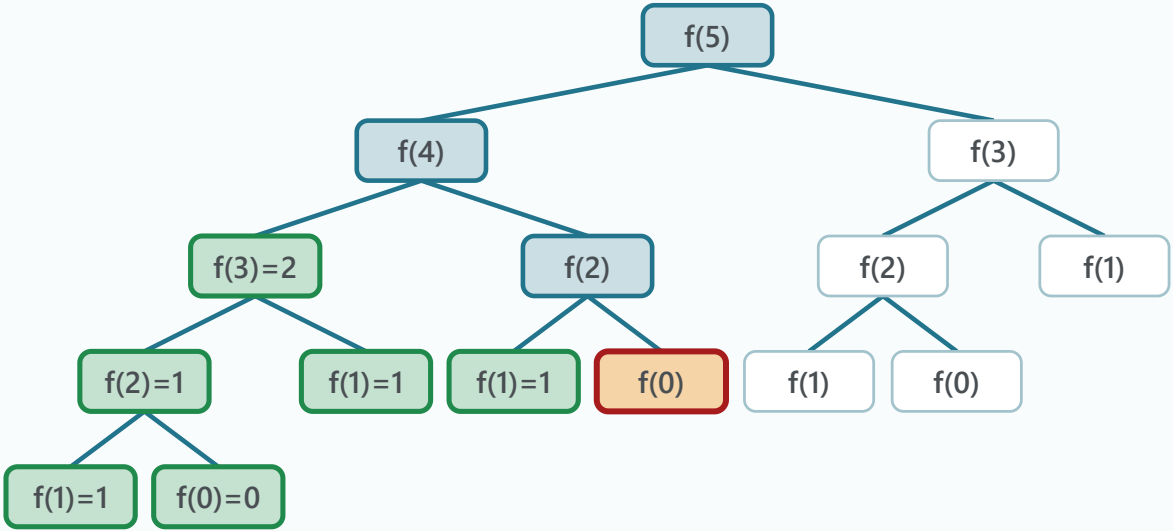
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(1)` base case: returns **1** at once. Stack: fib(5) → fib(4) → fib(2) → fib(1)

The fib(5) Recursion Tree: Call Order

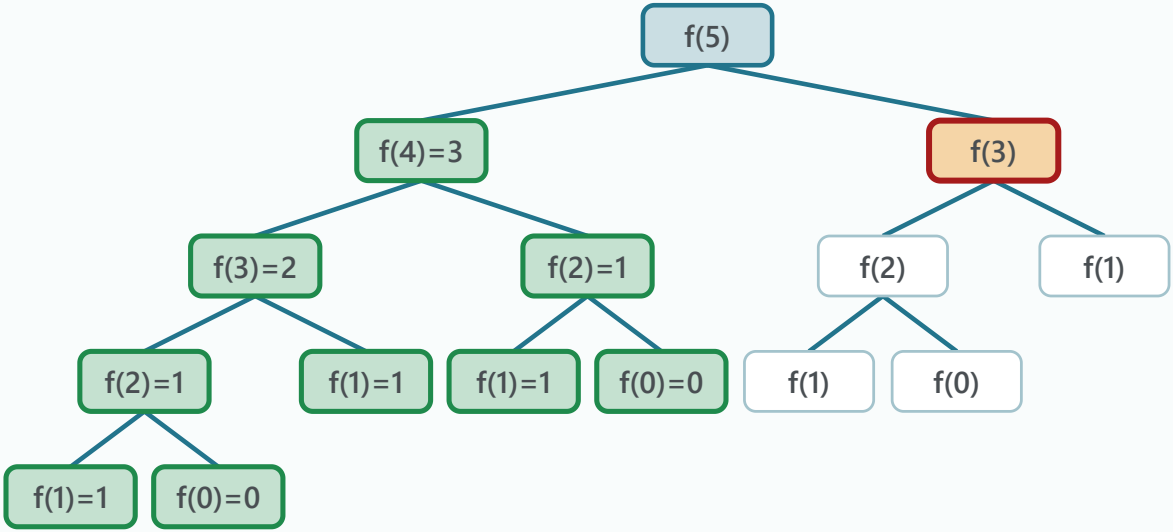
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(0)` base case: returns **0** at once. Stack: fib(5) → fib(4) → fib(2) → fib(0)

The fib(5) Recursion Tree: Call Order

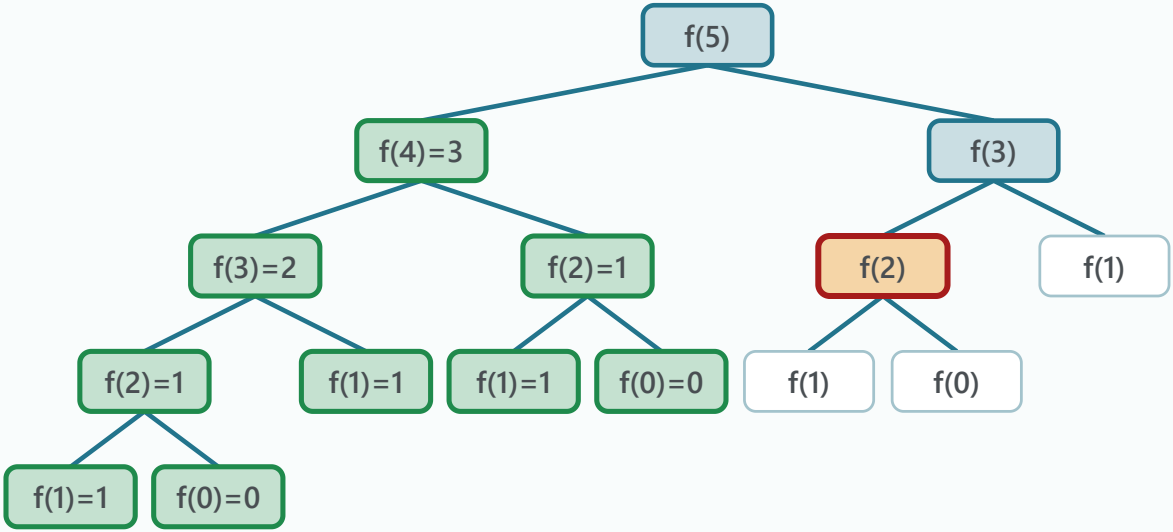
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(3)` called — computed for the **2nd time**! Stack: fib(5) → fib(3)

The fib(5) Recursion Tree: Call Order

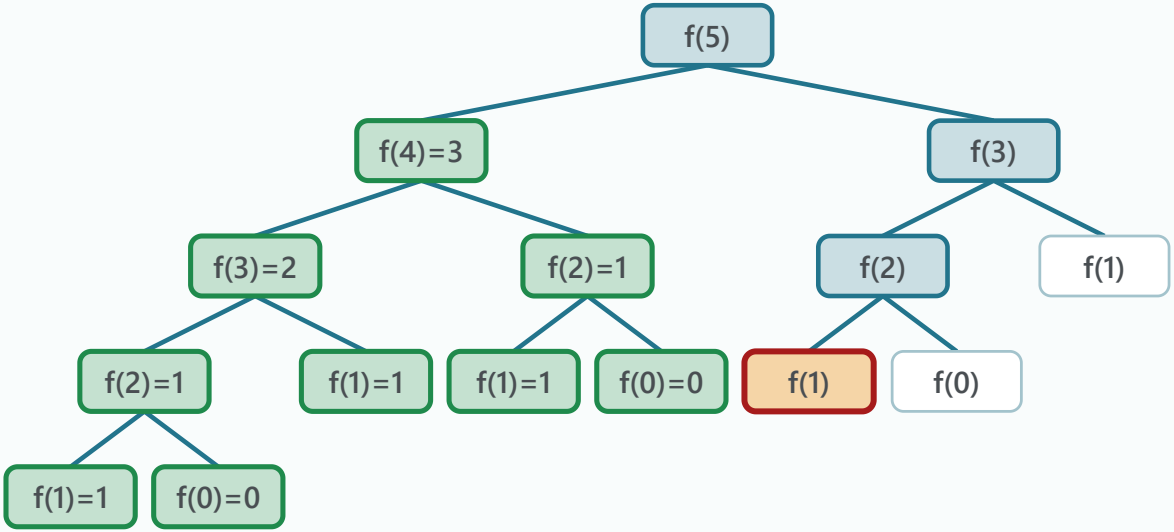
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(2)` called — computed for the **3rd time**! Stack: fib(5) → fib(3) → fib(2)

The fib(5) Recursion Tree: Call Order

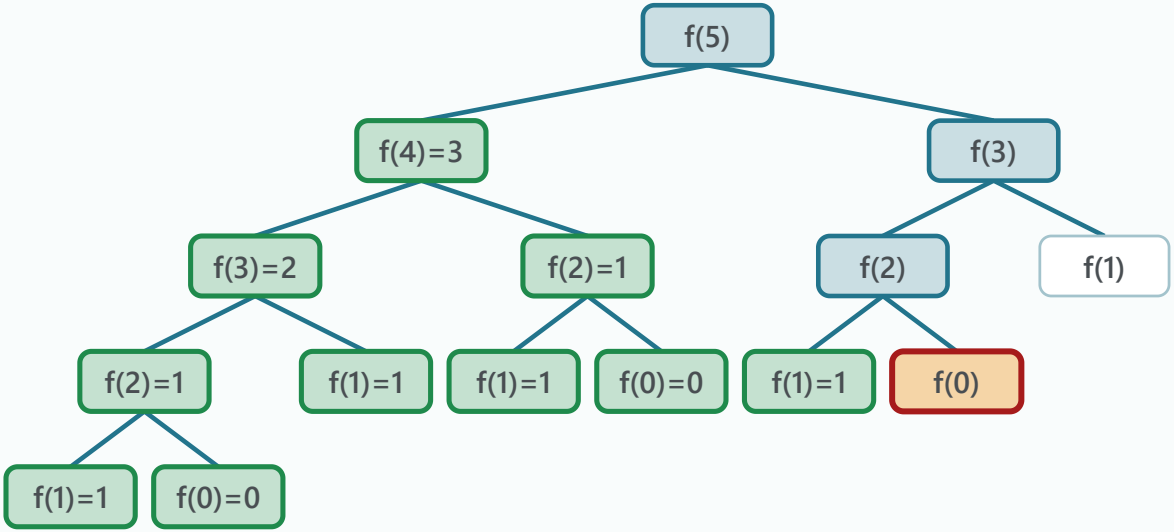
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(1)` base case: returns **1** at once. Stack: fib(5) → fib(3) → fib(2) → fib(1)

The fib(5) Recursion Tree: Call Order

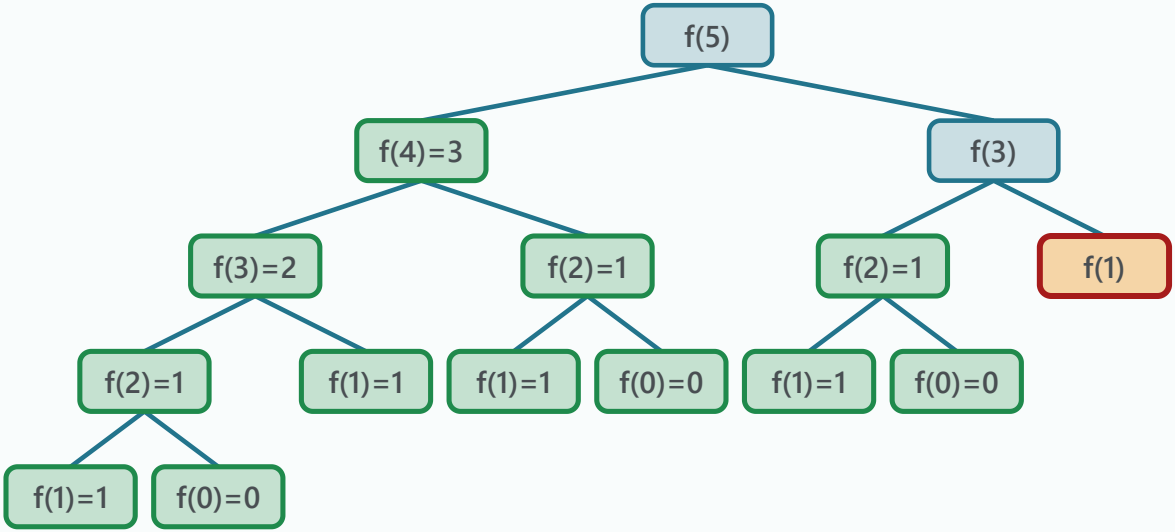
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(0)` base case: returns **0** at once. Stack: fib(5) → fib(3) → fib(2) → fib(0)

The fib(5) Recursion Tree: Call Order

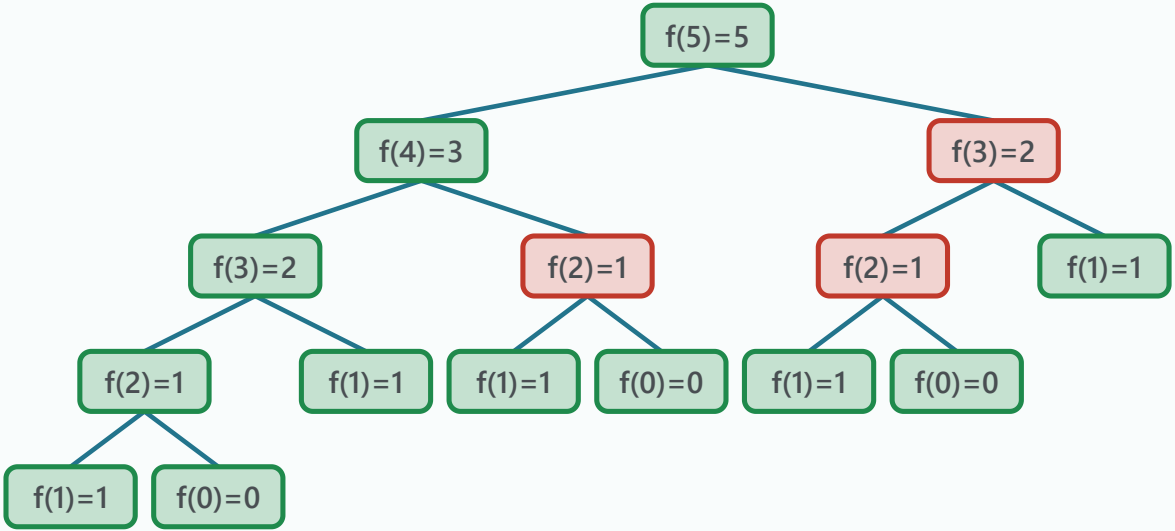
Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



`fib(1)` base case: returns **1** at once. Stack: fib(5) → fib(3) → fib(1)

The fib(5) Recursion Tree: Call Order

Orange: running call · blue: calls waiting on the stack · green: finished call (with its value).



Result fib(5) = **5**, **15 calls** in total. Red boxes are repeats: fib(3) was computed 2 times, fib(2) 3 times, fib(1) 5 times.

The Cost of Repeated Work

n	F(n)	plain recursive calls	memoized calls
5	5	15	9
10	55	177	19
20	6,765	21,891	39
30	832,040	2,692,537	59
40	102,334,155	331,160,281	79

- Plain recursion makes **2·F(n+1) − 1** calls: it grows as fast (exponentially) as the Fibonacci numbers themselves.
- Yet there are only **n+1 distinct** subproblems: fib(0), fib(1), …, fib(n).
- Solving each **once** and storing it cuts the calls to **2n − 1**: from exponential to linear.

The essence of DP: when the **number of distinct subproblems** is small but the recursion **calls them many times**, storing results pays off hugely.

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```

f(5)

memo	—	—	0	0	0	0
	0	1	2	3	4	5

`fib(5)` : not in the table; first call `fib(4)` .

Step 1 / 17

VARIABLES

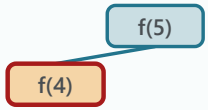
`n = 5` `a = ?` `b = –`

CALL STACK

`fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



memo	—	—	0	0	0	0
	0	1	2	3	4	5

`fib(4)` : not in the table; first call `fib(3)` .

Step 2 / 17

VARIABLES

`n = 4` `a = ?` `b = –`

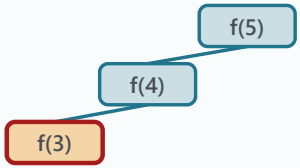
CALL STACK

`fib(4)`

`fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
     long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



memo	—	—	0	0	0	0
	0	1	2	3	4	5

`fib(3)` : not in the table; first call `fib(2)` .

Step 3 / 17

VARIABLES

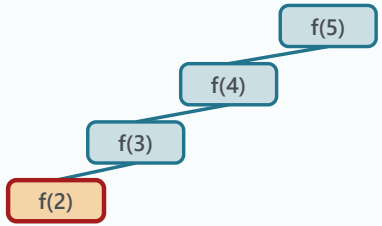
`n = 3` `a = ?` `b = –`

CALL STACK

- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
     long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



memo	—	—	0	0	0	0
	0	1	2	3	4	5

`fib(2)` : not in the table; first call `fib(1)` .

Step 4 / 17

VARIABLES

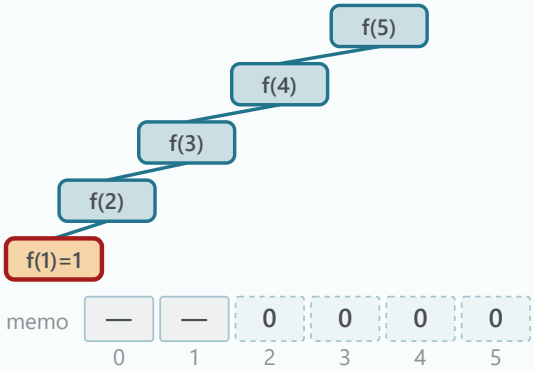
`n = 2` `a = ?` `b = –`

CALL STACK

- `fib(2)`
- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;          // base case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(1)` : base case, returns **1**.

Step 5 / 17

VARIABLES

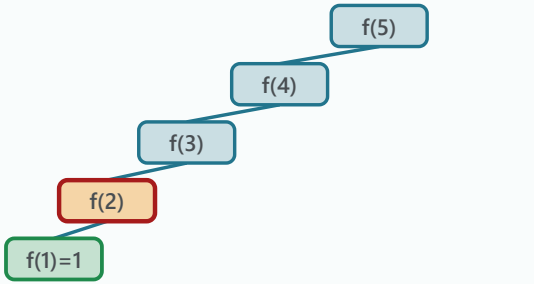
`n = 1` `a = –` `b = –`

CALL STACK

- `fib(1)`
- `fib(2)`
- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
▶    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(1)` returned (a = 1); now call `fib(0)`.

Step 6 / 17

VARIABLES

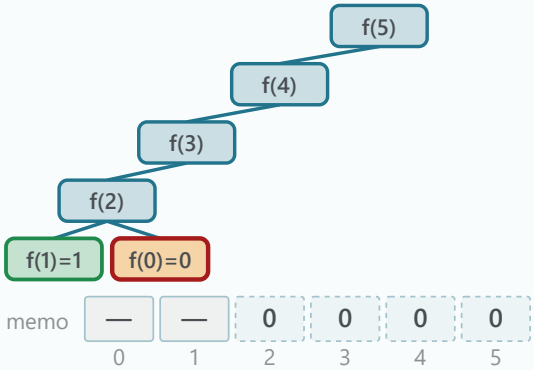
`n = 2` `a = 1` `b = ?`

CALL STACK

- `fib(2)`
- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;          // base case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(0)` : base case, returns **0**.

Step 7 / 17

VARIABLES

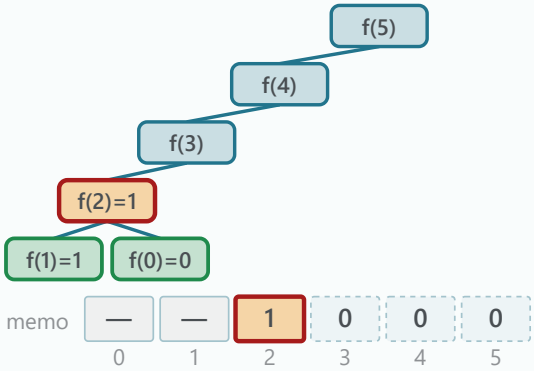
`n = 0` `a = –` `b = –`

CALL STACK

- `fib(0)`
- `fib(2)`
- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
▶    memo[n] = a + b;
▶    return memo[n];
10 }
```



`memo[2] = 1 + 0 = 1` is stored in the table and returned.

Step 8 / 17

VARIABLES

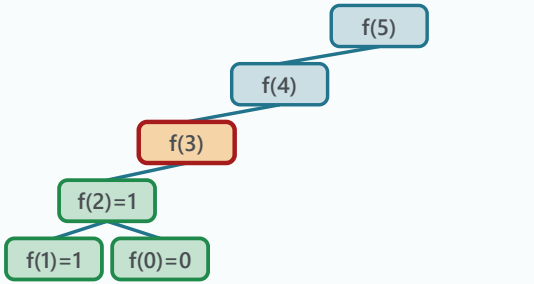
`n = 2` `a = 1` `b = 0`

CALL STACK

- `fib(2)`
- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
▶    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(2)` returned (a = 1); now call `fib(1)`.

Step 9 / 17

VARIABLES

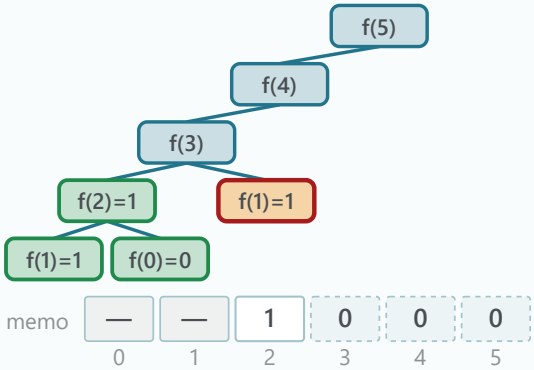
`n = 3` `a = 1` `b = ?`

CALL STACK

- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;          // base case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(1)` : base case, returns **1**.

Step 10 / 17

VARIABLES

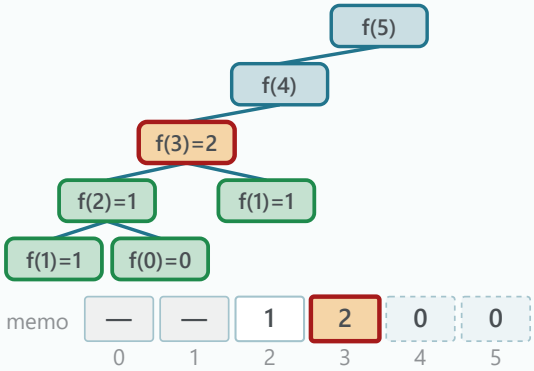
`n = 1` `a = –` `b = –`

CALL STACK

- `fib(1)`
- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
▶    memo[n] = a + b;
▶    return memo[n];
10 }
```



`memo[3] = 1 + 1 = 2` is stored in the table and returned.

Step 11 / 17

VARIABLES

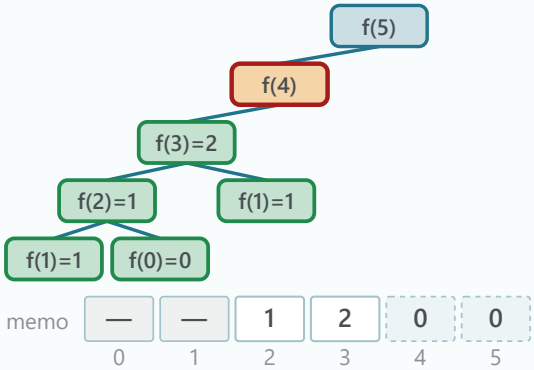
`n = 3` `a = 1` `b = 1`

CALL STACK

- `fib(3)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
▶    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(3)` returned (a = 2); now call `fib(2)`.

Step 12 / 17

VARIABLES

`n = 4` `a = 2` `b = ?`

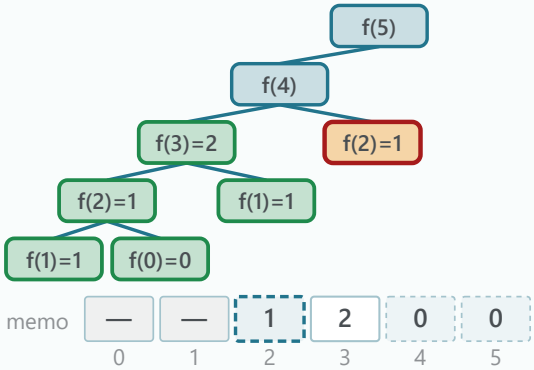
CALL STACK

`fib(4)`

`fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;            // base
   case
▶    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(2)` : `memo[2] = 1` is **already** in the table; no recomputation.

Step 13 / 17

VARIABLES

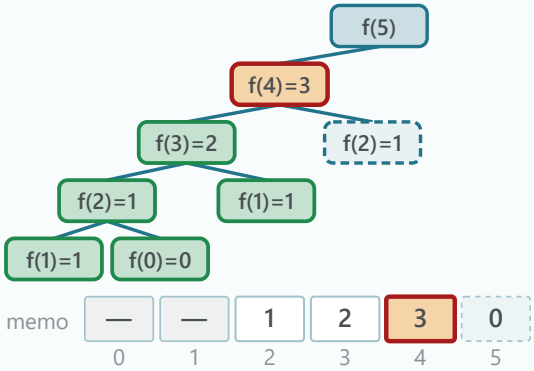
`n = 2` `a = –` `b = –`

CALL STACK

- `fib(2)`
- `fib(4)`
- `fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
▶    memo[n] = a + b;
▶    return memo[n];
10 }
```



`memo[4] = 2 + 1 = 3` is stored in the table and returned.

Step 14 / 17

VARIABLES

`n = 4` `a = 2` `b = 1`

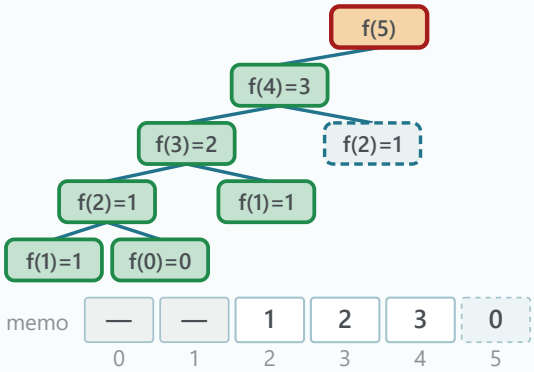
CALL STACK

`fib(4)`

`fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
▶    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(4)` returned (a = 3); now call `fib(3)`.

Step 15 / 17

VARIABLES

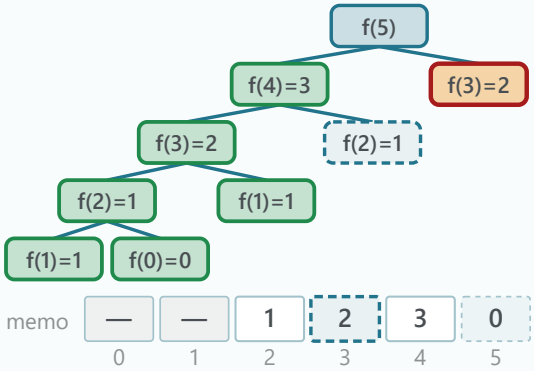
`n = 5` `a = 3` `b = ?`

CALL STACK

`fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;            // base
   case
     if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
8    memo[n] = a + b;
9    return memo[n];
10 }
```



`fib(3)` : `memo[3] = 2` is **already** in the table; no recomputation.

Step 16 / 17

VARIABLES

`n = 3` `a = –` `b = –`

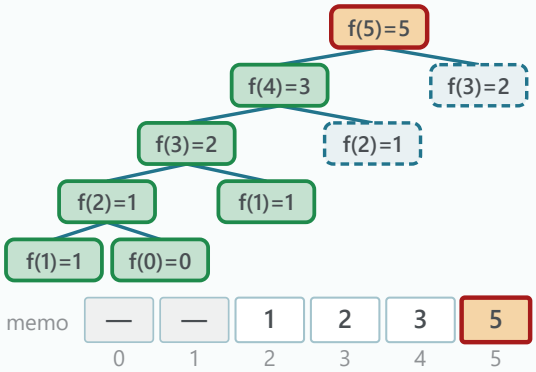
CALL STACK

`fib(3)`

`fib(5)`

Memoization: fib(5) Step by Step

```
1  static long[] memo = new long[50];
2
3  static long fib(int n) {
4    if (n <= 1) return n;           // base
   case
5    if (memo[n] != 0) return memo[n];
6    long a = fib(n - 1);
7    long b = fib(n - 2);
▶    memo[n] = a + b;
▶    return memo[n];
10 }
```



`memo[5] = 3 + 2 = 5` is stored in the table and returned.

Step 17 / 17

VARIABLES

`n = 5` `a = 3` `b = 2`

CALL STACK

`fib(5)`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
▶   if (n <= 1) return n;   // n = 0 and n = 1
▶   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
6   for (int i = 2; i <= n; i++)
7     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

The `fib` array is created with n + 1 = 10 elements (all 0 initially in Java).

Step 1 / 11

VARIABLES

`n = 9` `i = –`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n = 1
3   int[] fib = new int[n + 1];
▶   fib[0] = 0;   // base cases
▶   fib[1] = 1;
6   for (int i = 2; i <= n; i++)
7     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



The base cases are written: `fib[0] = 0`, `fib[1] = 1`. Empty cells are not computed yet.

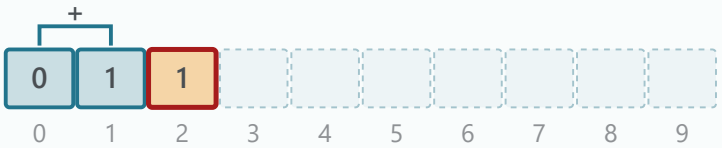
Step 2 / 11

VARIABLES

`n = 9` `i = –`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n
  = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 2: `fib[2] = fib[1] + fib[0] = 1 + 0 = 1`

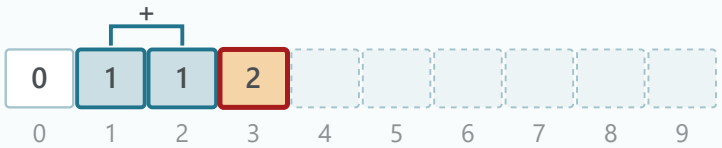
Step 3 / 11

VARIABLES

`n = 9` `i = 2` `fib[i] = 1`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n
  = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 3: `fib[3] = fib[2] + fib[1] = 1 + 1 = 2`

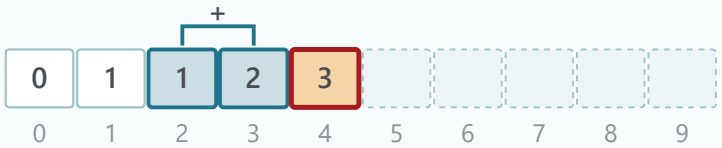
Step 4 / 11

VARIABLES

`n = 9` `i = 3` `fib[i] = 2`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n
  = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 4: `fib[4] = fib[3] + fib[2] = 2 + 1 = 3`

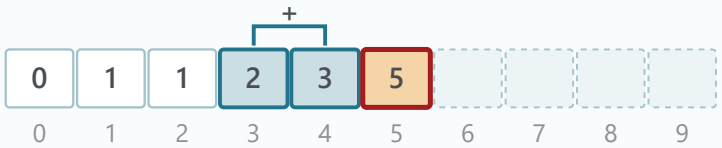
Step 5 / 11

VARIABLES

`n = 9` `i = 4` `fib[i] = 3`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n
  = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 5: `fib[5] = fib[4] + fib[3] = 3 + 2 = 5`

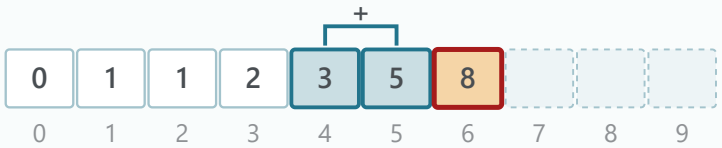
Step 6 / 11

VARIABLES

`n = 9` `i = 5` `fib[i] = 5`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n
  = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 6: `fib[6] = fib[5] + fib[4] = 5 + 3 = 8`

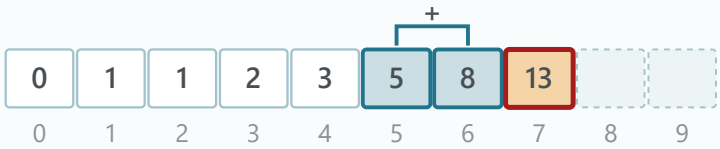
Step 7 / 11

VARIABLES

`n = 9` `i = 6` `fib[i] = 8`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 7: `fib[7] = fib[6] + fib[5] = 8 + 5 = 13`

Step 8 / 11

VARIABLES

`n = 9` `i = 7` `fib[i] = 13`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 8: `fib[8] = fib[7] + fib[6] = 13 + 8 = 21`

Step 9 / 11

VARIABLES

`n = 9` `i = 8` `fib[i] = 21`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n
  = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
▶   for (int i = 2; i <= n; i++)
▶     fib[i] = fib[i - 1] + fib[i - 2];
8   return fib[n];
9 }
```



i = 9: `fib[9] = fib[8] + fib[7] = 21 + 13 = 34`

Step 10 / 11

VARIABLES

`n = 9` `i = 9` `fib[i] = 34`

Tabulation: fibonacci(9) Step by Step

```
1 static int fibonacci(int n) {
2   if (n <= 1) return n;   // n = 0 and n = 1
3   int[] fib = new int[n + 1];
4   fib[0] = 0;   // base cases
5   fib[1] = 1;
6   for (int i = 2; i <= n; i++)
7     fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n];
9 }
```

0	1	1	2	3	5	8	13	21	34
0	1	2	3	4	5	6	7	8	9

The loop ends; `fib[9] = 34` is returned. Each cell was computed **once**: O(n) time, O(n) memory.

Step 11 / 11

VARIABLES

`n = 9` `i = 10` `fib[i] = –`

Fibonacci: O(1) Memory and Comparison

```java
static long fib(int n) {
  if (n <= 1) return n;
  long prev = 0, curr = 1;
  for (int i = 2; i <= n; i++) {
    long next = prev + curr;
    prev = curr;
    curr = next;
  }
  return curr;
}
```

Method	Time	Extra memory
Plain recursion	$\Theta(\varphi^n)$	O(n) stack
Memoization	O(n)	O(n) table + stack
Tabulation	O(n)	O(n) table
Two variables	O(n)	**O(1)**

Saving memory: fib[i] depends only on the last **two** values; there is no need to store the whole array. The same idea will return in the knapsack as "only the previous row".

The 0/1 Knapsack Problem

- n items: each item has a **value** v_i and a **weight** (size) w_i.
- The knapsack has **capacity** W.
- Each item is either **taken whole (1)** or **not taken (0)**; it cannot be split.
- Goal: **maximize the total value** subject to total weight ≤ W.



Question: Which items should go into the knapsack in the picture?

▶ **Answer**

Why Not Brute Force or Greedy?

Brute force

2 options per item → 2^n subsets. For n = 50, more than 10^{15} subsets: impossible in practice.

Greedy (value/weight ratio)

W = 50; items (60 ₺, 10 kg), (100 ₺, 20 kg), (120 ₺, 30 kg). Ratios 6, 5, 4 → greedy takes the first two: **160 ₺**. Optimal: second + third = **220 ₺**.



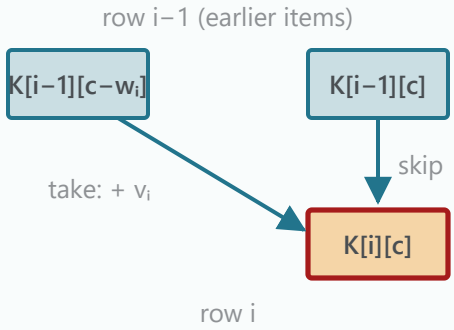
The greedy choice is optimal only when items **can be split** (fractional knapsack). In the 0/1 case, taking an item changes the later options: both choices must be tried → **dynamic programming**.

0/1 Knapsack: State and Recurrence

State: K[i][c] = the largest value achievable with the first **i** items and capacity **c**.

$K[0][c] = 0,\ K[i][0] = 0$
if $w_i > c$: $K[i][c] = K[i-1][c]$
otherwise: $K[i][c] = \max(K[i-1][c],\ v_i + K[i-1][c - w_i])$

- **Skip it:** the capacity stays the same; the best with the earlier items.
- **Take it:** gain v_i and fill the remaining capacity $c - w_i$ as well as possible with the earlier items.
- Answer: **K[n][W]**. The table is $(n+1) \times (W+1)$, filled row by row.



0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
10         K[i][c] = Math.max(K[i - 1][c],
11                            vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0						
2	4	3	0						
3	3	2	0						
4	2	1	0						

The table is (n+1) × (W+1) = 5 × 7. Row 0 (no items) and column 0 (no capacity) are **0**.

Step 1 / 26

VARIABLES

n = 4 | W = 6 | i = – | c = –

0/1 Knapsack: Filling the Table (W = 6)

```java
int knapsack(int[] v, int[] w, int W) {
  int n = v.length;
  int[][] K = new int[n + 1][W + 1];
  for (int i = 1; i <= n; i++)
    for (int c = 1; c <= W; c++) {
      int vi = v[i - 1], wi = w[i - 1];
      if (wi > c)              // does not fit
        K[i][c] = K[i - 1][c];
      else                     // skip / take
        K[i][c] = Math.max(K[i - 1][c],
                           vi + K[i - 1][c - wi]);
    }
  return K[n][W];
}
```

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0					
2	4	3	0						
3	3	2	0						
4	2	1	0						

i=1, c=1: w=4 > 1, the item **does not fit** → copy the value above: K[1][1] = **0**

Step 2 / 26

VARIABLES

`i = 1` `c = 1` `vi = 5` `wi = 4` `skip = 0` `take = –`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
▶        if (wi > c)              // does not fit
▶          K[i][c] = K[i - 1][c];
9        else                     // skip / take
10         K[i][c] = Math.max(K[i - 1][c],
11                            vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0				
2	4	3	0						
3	3	2	0						
4	2	1	0						

i=1, c=2: w=4 > 2, the item **does not fit** → copy the value above: K[1][2] = **0**

Step 3 / 26

VARIABLES

`i = 1` `c = 2` `vi = 5` `wi = 4` `skip = 0` `take = –`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
▶        if (wi > c)              // does not fit
▶          K[i][c] = K[i - 1][c];
9        else                     // skip / take
10         K[i][c] = Math.max(K[i - 1][c],
11                            vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0			
2	4	3	0						
3	3	2	0						
4	2	1	0						

i=1, c=3: w=4 > 3, the item **does not fit** → copy the value above: K[1][3] = **0**

Step 4 / 26

VARIABLES

`i = 1` `c = 3` `vi = 5` `wi = 4` `skip = 0` `take = –`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5		
2	4	3	0						
3	3	2	0						
4	2	1	0						

i=1, c=4: max(skip = 0, take = 5 + K[0][0] = 5) = **5** (take)

Step 5 / 26

VARIABLES

`i = 1` `c = 4` `vi = 5` `wi = 4` `skip = 0` `take = 5`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	
2	4	3	0						
3	3	2	0						
4	2	1	0						

i=1, c=5: max(skip = 0, take = 5 + K[0][1] = 5) = **5** (take)

Step 6 / 26

VARIABLES

`i = 1` `c = 5` `vi = 5` `wi = 4` `skip = 0` `take = 5`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0						
3	3	2	0						
4	2	1	0						

i=1, c=6: max(skip = 0, take = 5 + K[0][2] = 5) = **5** (take)

Step 7 / 26

VARIABLES

`i = 1` `c = 6` `vi = 5` `wi = 4` `skip = 0` `take = 5`

0/1 Knapsack: Filling the Table (W = 6)

```java
int knapsack(int[] v, int[] w, int W) {
  int n = v.length;
  int[][] K = new int[n + 1][W + 1];
  for (int i = 1; i <= n; i++)
    for (int c = 1; c <= W; c++) {
      int vi = v[i - 1], wi = w[i - 1];
      if (wi > c)                 // does not fit
        K[i][c] = K[i - 1][c];
      else                        // skip / take
        K[i][c] = Math.max(K[i - 1][c],
                           vi + K[i - 1][c - wi]);
    }
  return K[n][W];
}
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0					
3	3	2	0						
4	2	1	0						

i=2, c=1: w=3 > 1, the item **does not fit** → copy the value above: K[2][1] = **0**

Step 8 / 26

VARIABLES

`i = 2` `c = 1` `vi = 4` `wi = 3` `skip = 0` `take = –`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
▶        if (wi > c)              // does not fit
▶          K[i][c] = K[i - 1][c];
9        else                     // skip / take
10         K[i][c] = Math.max(K[i - 1][c],
11                            vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0				
3	3	2	0						
4	2	1	0						

i=2, c=2: w=3 > 2, the item **does not fit** → copy the value above: K[2][2] = **0**

Step 9 / 26

VARIABLES

`i = 2` `c = 2` `vi = 4` `wi = 3` `skip = 0` `take = –`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4			
3	3	2	0						
4	2	1	0						

i=2, c=3: max(skip = 0, take = 4 + K[1][0] = 4) = **4** (take)

Step 10 / 26

VARIABLES

`i = 2` `c = 3` `vi = 4` `wi = 3` `skip = 0` `take = 4`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5		
3	3	2	0						
4	2	1	0						

i=2, c=4: max(skip = 5, take = 4 + K[1][1] = 4) = **5** (skip)

Step 11 / 26

VARIABLES

`i = 2` `c = 4` `vi = 4` `wi = 3` `skip = 5` `take = 4`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	
3	3	2	0						
4	2	1	0						

i=2, c=5: max(skip = 5, take = 4 + K[1][2] = 4) = **5** (skip)

Step 12 / 26

VARIABLES

`i = 2` `c = 5` `vi = 4` `wi = 3` `skip = 5` `take = 4`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0						
4	2	1	0						

i=2, c=6: max(skip = 5, take = 4 + K[1][3] = 4) = **5** (skip)

Step 13 / 26

VARIABLES

`i = 2` `c = 6` `vi = 4` `wi = 3` `skip = 5` `take = 4`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
▶        if (wi > c)              // does not fit
▶          K[i][c] = K[i - 1][c];
9        else                     // skip / take
10         K[i][c] = Math.max(K[i - 1][c],
11                            vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0					
4	2	1	0						

i=3, c=1: w=2 > 1, the item **does not fit** → copy the value above: K[3][1] = **0**

Step 14 / 26

VARIABLES

`i = 3` `c = 1` `vi = 3` `wi = 2` `skip = 0` `take = –`

0/1 Knapsack: Filling the Table (W = 6)

```java
int knapsack(int[] v, int[] w, int W) {
  int n = v.length;
  int[][] K = new int[n + 1][W + 1];
  for (int i = 1; i <= n; i++)
    for (int c = 1; c <= W; c++) {
      int vi = v[i - 1], wi = w[i - 1];
      if (wi > c)              // does not fit
        K[i][c] = K[i - 1][c];
      else                     // skip / take
        K[i][c] = Math.max(K[i - 1][c],
                           vi + K[i - 1][c - wi]);
    }
  return K[n][W];
}
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3				
4	2	1	0						

i=3, c=2: max(skip = 0, take = 3 + K[2][0] = 3) = **3** (take)

Step 15 / 26

VARIABLES

`i = 3` `c = 2` `vi = 3` `wi = 2` `skip = 0` `take = 3`

0/1 Knapsack: Filling the Table (W = 6)

```java
int knapsack(int[] v, int[] w, int W) {
  int n = v.length;
  int[][] K = new int[n + 1][W + 1];
  for (int i = 1; i <= n; i++)
    for (int c = 1; c <= W; c++) {
      int vi = v[i - 1], wi = w[i - 1];
      if (wi > c)              // does not fit
        K[i][c] = K[i - 1][c];
      else                     // skip / take
        K[i][c] = Math.max(K[i - 1][c],
                           vi + K[i - 1][c - wi]);
    }
  return K[n][W];
}
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4			
4	2	1	0						

i=3, c=3: max(skip = 4, take = 3 + K[2][1] = 3) = **4** (skip)

Step 16 / 26

VARIABLES

`i = 3` `c = 3` `vi = 3` `wi = 2` `skip = 4` `take = 3`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5		
4	2	1	0						

i=3, c=4: max(skip = 5, take = 3 + K[2][2] = 3) = **5** (skip)

Step 17 / 26

VARIABLES

`i = 3` `c = 4` `vi = 3` `wi = 2` `skip = 5` `take = 3`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	
4	2	1	0						

i=3, c=5: max(skip = 5, take = 3 + K[2][3] = 7) = **7** (take)

Step 18 / 26

VARIABLES

`i = 3` `c = 5` `vi = 3` `wi = 2` `skip = 5` `take = 7`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0						

i=3, c=6: max(skip = 5, take = 3 + K[2][4] = 8) = **8** (take)

Step 19 / 26

VARIABLES

`i = 3` `c = 6` `vi = 3` `wi = 2` `skip = 5` `take = 8`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2					

i=4, c=1: max(skip = 0, take = 2 + K[3][0] = 2) = **2** (take)

Step 20 / 26

VARIABLES

`i = 4` `c = 1` `vi = 2` `wi = 1` `skip = 0` `take = 2`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3				

i=4, c=2: max(skip = 3, take = 2 + K[3][1] = 2) = **3** (skip)

Step 21 / 26

VARIABLES

`i = 4` `c = 2` `vi = 2` `wi = 1` `skip = 3` `take = 2`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5			

i=4, c=3: max(skip = 4, take = 2 + K[3][2] = 5) = **5** (take)

Step 22 / 26

VARIABLES

`i = 4` `c = 3` `vi = 2` `wi = 1` `skip = 4` `take = 5`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6		

i=4, c=4: max(skip = 5, take = 2 + K[3][3] = 6) = **6** (take)

Step 23 / 26

VARIABLES

`i = 4` `c = 4` `vi = 2` `wi = 1` `skip = 5` `take = 6`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	

i=4, c=5: max(skip = 7, take = 2 + K[3][4] = 7) = **7** (skip)

Step 24 / 26

VARIABLES

`i = 4` `c = 5` `vi = 2` `wi = 1` `skip = 7` `take = 7`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
▶          K[i][c] = Math.max(K[i - 1][c],
▶                             vi + K[i - 1][c - wi]);
12     }
13   return K[n][W];
14 }
```

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	9

i=4, c=6: max(skip = 8, take = 2 + K[3][5] = 9) = **9** (take)

Step 25 / 26

VARIABLES

`i = 4` `c = 6` `vi = 2` `wi = 1` `skip = 8` `take = 9`

0/1 Knapsack: Filling the Table (W = 6)

```
1  int knapsack(int[] v, int[] w, int W) {
2    int n = v.length;
3    int[][] K = new int[n + 1][W + 1];
4    for (int i = 1; i <= n; i++)
5      for (int c = 1; c <= W; c++) {
6        int vi = v[i - 1], wi = w[i - 1];
7        if (wi > c)              // does not fit
8          K[i][c] = K[i - 1][c];
9        else                     // skip / take
10         K[i][c] = Math.max(K[i - 1][c],
11                            vi + K[i - 1][c - wi]);
12     }
▶   return K[n][W];
14 }
```

c (capacity) →

i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	**9**

The table is full. The answer is `K[4][6]` = **9**. 4 × 6 = 24 cells in total, each O(1).

Step 26 / 26

VARIABLES

`i = 5` `c = –` `result = 9`

Reconstruction: Which Items Were Taken?

Rule: if K[i][c] ≠ K[i−1][c], item i was taken (the value rose only by taking it) → c ← c − w_i; if equal, it was not taken.

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	9

Reconstruction starts at cell `K[4][6] = 9`. Which items were taken?

Reconstruction: Which Items Were Taken?

Rule: if K[i][c] ≠ K[i−1][c], item i was taken (the value rose only by taking it) → c ← c − w_i; if equal, it was not taken.

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	9

K[4][6] = 9 ≠ K[3][6] = 8 → item 4 **taken** (v=2, w=1); c = 6 − 1 = 5

Reconstruction: Which Items Were Taken?

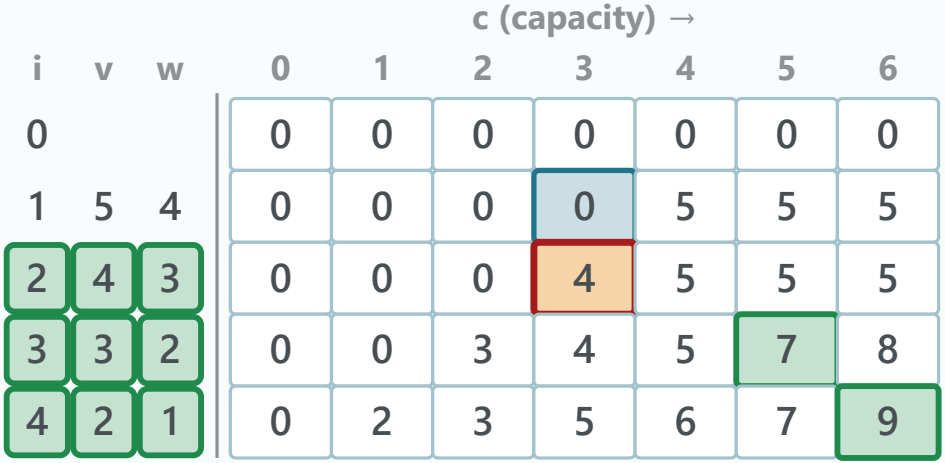
Rule: if K[i][c] ≠ K[i−1][c], item i was taken (the value rose only by taking it) → c ← c − w_i; if equal, it was not taken.

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	9

K[3][5] = 7 ≠ K[2][5] = 5 → item 3 **taken** (v=3, w=2); c = 5 − 2 = 3

Reconstruction: Which Items Were Taken?

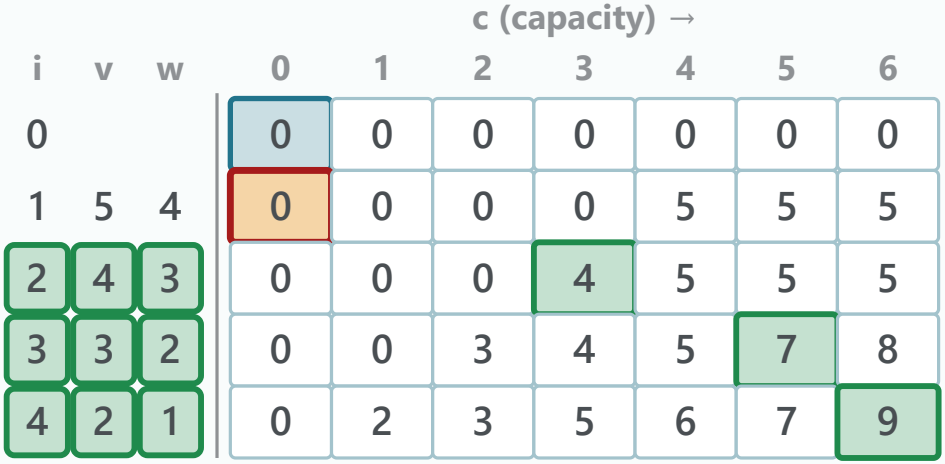
Rule: if K[i][c] ≠ K[i−1][c], item i was taken (the value rose only by taking it) → c ← c − w_i; if equal, it was not taken.



K[2][3] = 4 ≠ K[1][3] = 0 → item 2 **taken** (v=4, w=3); c = 3 − 3 = 0

Reconstruction: Which Items Were Taken?

Rule: if K[i][c] ≠ K[i−1][c], item i was taken (the value rose only by taking it) → c ← c − w_i; if equal, it was not taken.



K[1][0] = K[0][0] = 0 → item 1 **not taken**; c is unchanged.

Reconstruction: Which Items Were Taken?

Rule: if K[i][c] ≠ K[i−1][c], item i was taken (the value rose only by taking it) → c ← c − w_i; if equal, it was not taken.

			c (capacity) →						
i	v	w	0	1	2	3	4	5	6
0			0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	9

Items chosen: 2, 3, 4 → value 4 + 3 + 2 = **9**, weight 3 + 2 + 1 = **6** = W.

0/1 Knapsack: Complexity and Memory

- **Time: O(n · W)**, **memory: O(n · W)** — (n+1)(W+1) cells, each O(1).
- This time is **pseudo-polynomial**: it depends on the *value* of W, not on its bit length. With W = 10^9, the table cannot be built.
- 0/1 knapsack is **NP-hard**; DP is practical only when W is small.
- **Saving memory** (code on the right): row i uses only row i−1 → a one-dimensional array, O(W). The capacity must be scanned **from high to low**; otherwise the same item is taken twice.
- For reconstruction, the one-dimensional version also keeps a `boolean[][] taken` table.

```
int[] dp = new int[W + 1];
for (int i = 0; i < n; i++)
  for (int c = W; c >= w[i]; c--)
    dp[c] = Math.max(dp[c],
                     v[i] + dp[c - w[i]]);
return dp[W];
```

0/1 Knapsack: With a Single Row (W = 6)

```
int[] dp = new int[W + 1];
for (int i = 0; i < n; i++)
  for (int c = W; c >= w[i]; c--)
    dp[c] = Math.max(dp[c],
          v[i] + dp[c - w[i]]);
return dp[W];
```

dp	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

item: `v=5 w=4` `v=4 w=3` `v=3 w=2` `v=2 w=1`

Items (v, w): (5, 4), (4, 3), (3, 2), (2, 1); W = 6. One row instead of the 2D table: dp[c] = the best value with capacity c using the items so far.

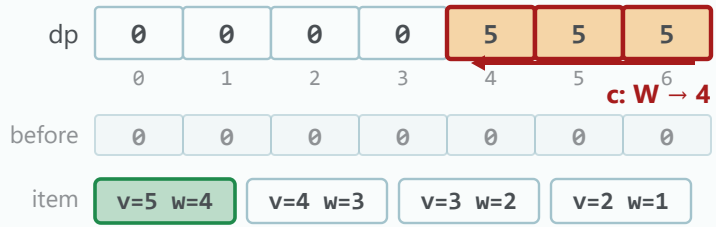
Step 1 / 6

VARIABLES

`i = –` `c = –` `dp[W] = 0`

0/1 Knapsack: With a Single Row (W = 6)

```
1 int[] dp = new int[W + 1];
▶ for (int i = 0; i < n; i++)
▶   for (int c = W; c >= w[i]; c--)
▶     dp[c] = Math.max(dp[c],
▶             v[i] + dp[c - w[i]]);
6 return dp[W];
```



Item 0 (v = 5, w = 4), c from 6 **down** to 4: dp[6]: 0 → 5, dp[5]: 0 → 5, dp[4]: 0 → 5. Thanks to the decreasing order, dp[c − 4] is always the old value (0): the item cannot be taken twice.

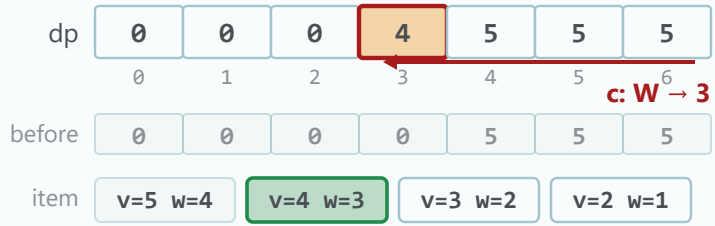
Step 2 / 6

VARIABLES

`i = 0` `c = 4` `dp[W] = 5`

0/1 Knapsack: With a Single Row (W = 6)

```
1 int[] dp = new int[W + 1];
▶ for (int i = 0; i < n; i++)
▶   for (int c = W; c >= w[i]; c--)
▶     dp[c] = Math.max(dp[c],
▶             v[i] + dp[c - w[i]]);
6 return dp[W];
```



Item 1 (v = 4, w = 3), c from 6 **down** to 3: dp[3]: 0 → 4.

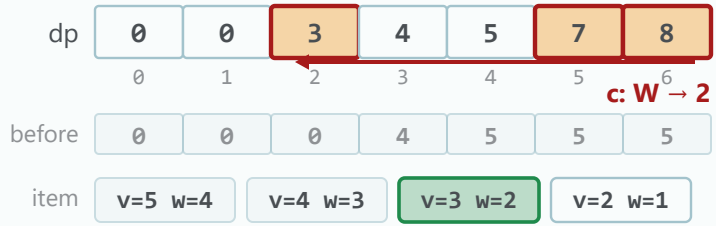
Step 3 / 6

VARIABLES

`i = 1` `c = 3` `dp[W] = 5`

0/1 Knapsack: With a Single Row (W = 6)

```
1 int[] dp = new int[W + 1];
▶ for (int i = 0; i < n; i++)
▶   for (int c = W; c >= w[i]; c--)
▶     dp[c] = Math.max(dp[c],
▶             v[i] + dp[c - w[i]]);
6 return dp[W];
```



Item 2 (v = 3, w = 2), c from 6 **down** to 2: dp[6]: 5 → 8, dp[5]: 5 → 7, dp[2]: 0 → 3.

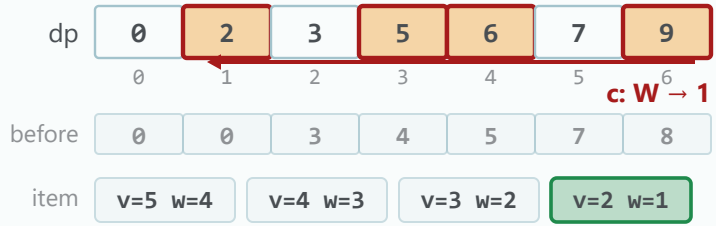
Step 4 / 6

VARIABLES

`i = 2` `c = 2` `dp[W] = 8`

0/1 Knapsack: With a Single Row (W = 6)

```
1 int[] dp = new int[W + 1];
▶ for (int i = 0; i < n; i++)
▶   for (int c = W; c >= w[i]; c--)
▶     dp[c] = Math.max(dp[c],
▶             v[i] + dp[c - w[i]]);
6 return dp[W];
```



Item 3 (v = 2, w = 1), c from 6 **down** to 1: dp[6]: 8 → 9, dp[4]: 5 → 6, dp[3]: 4 → 5, dp[1]: 0 → 2.

Step 5 / 6

VARIABLES

`i = 3` `c = 1` `dp[W] = 9`

0/1 Knapsack: With a Single Row (W = 6)

```
1 int[] dp = new int[W + 1];
2 for (int i = 0; i < n; i++)
3   for (int c = W; c >= w[i]; c--)
4     dp[c] = Math.max(dp[c],
5           v[i] + dp[c - w[i]]);
▶ return dp[W];
```

dp	0	2	3	5	6	7	9
	0	1	2	3	4	5	6

item	v=5 w=4	v=4 w=3	v=3 w=2	v=2 w=1

dp[6] = **9** (items 1, 2, 3: 4 + 3 + 2), the same as the bottom-right cell of the 2D table; memory O(W) instead of O(n · W).

Step 6 / 6

VARIABLES

`i = 4` `c = –` `dp[W] = 9`

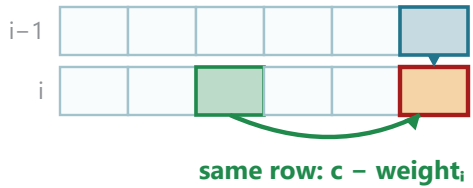
The Unbounded Knapsack

- Each item can be taken **as many times as you like** (unlimited copies); otherwise as in 0/1.
- **State:** dp[i][c] = the largest profit with item types 0..i and capacity c.

$$dp[i][c] = \max(dp[i-1][c] , profit_i + dp[\mathbf{i}][c - weight_i]) \text{ (if } weight_i \le c)$$

The only difference from 0/1

"Take it" looks at **the same row** (i), not the row above (i−1): a taken item can be taken again.



Example

Profit = [15, 50, 60, 90], weight = [1, 3, 4, 5], capacity = 8. The table is 4 × 9; the first row can use only item 0 (profit 15, 1 kg).

Row indices start at 0 (take dp[−1][c] = 0).

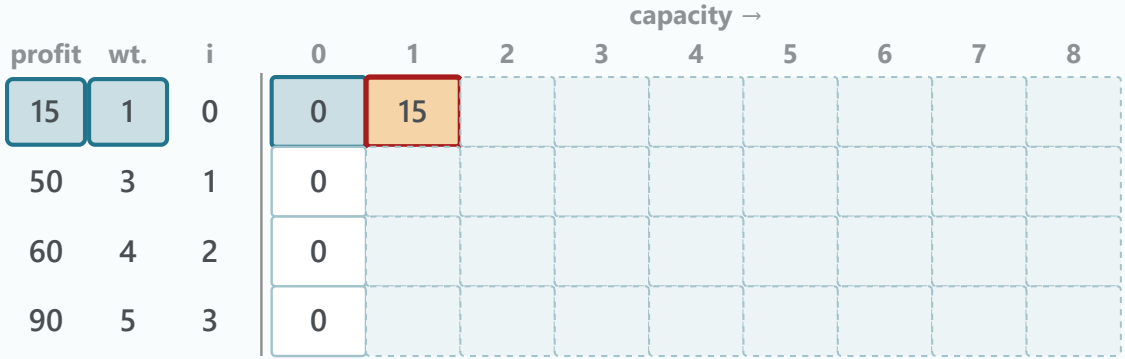
Unbounded Knapsack: Filling the Table

capacity →

profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0								
50	3	1	0								
60	4	2	0								
90	5	3	0								

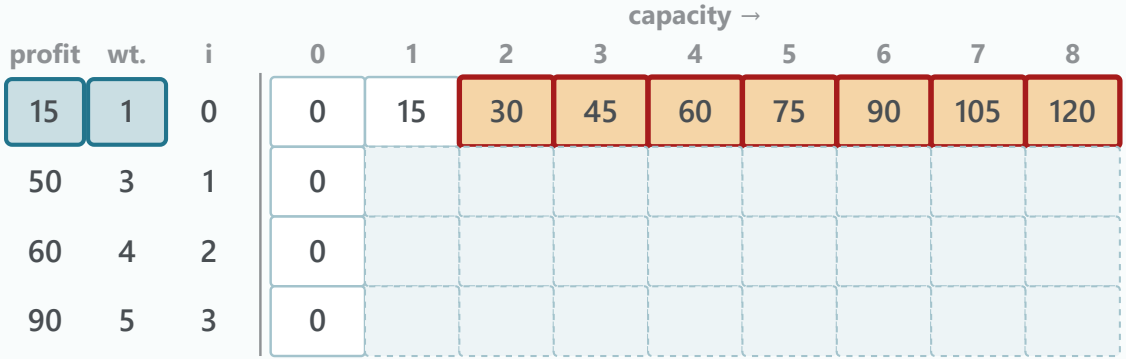
The capacity-0 column is 0: no item fits.

Unbounded Knapsack: Filling the Table



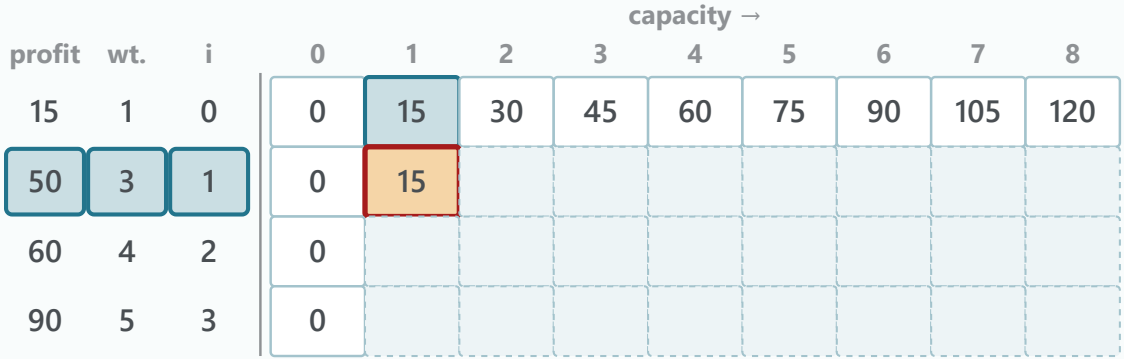
i=0, c=1: only item 0 is available; its weight 1 fits → 15 + dp[0][0] = **15**.

Unbounded Knapsack: Filling the Table



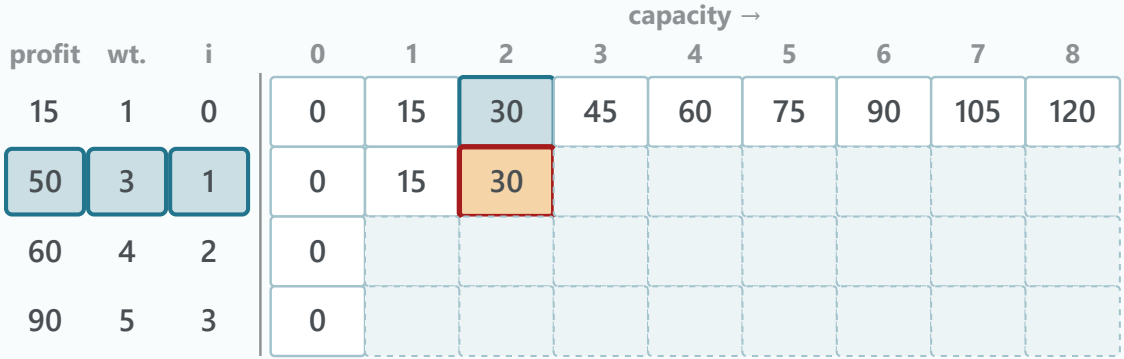
i=0, c=2..8: dp[0][c] = 15 + dp[0][c−1] → c copies of item 0: 30, 45, …, **120**. The same row is reused!

Unbounded Knapsack: Filling the Table



i=1, c=1: weight 3 > 1, does not fit → dp[1][1] = dp[0][1] = **15**

Unbounded Knapsack: Filling the Table



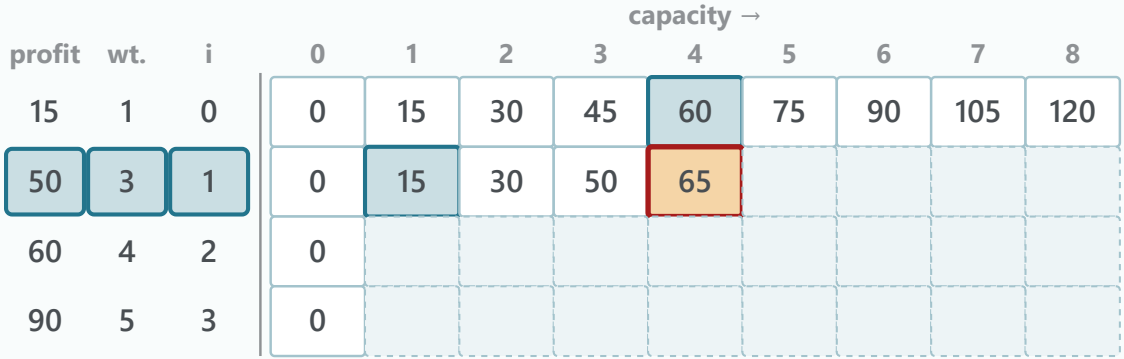
i=1, c=2: weight 3 > 2, does not fit → dp[1][2] = dp[0][2] = **30**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50					
60	4	2	0								
90	5	3	0								

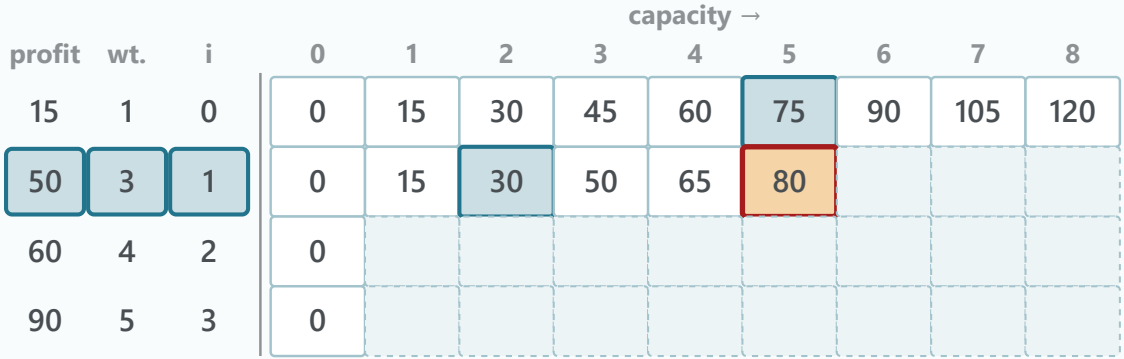
i=1, c=3: max(dp[0][3] = 45, 50 + dp[1][0] = 50) = **50**

Unbounded Knapsack: Filling the Table



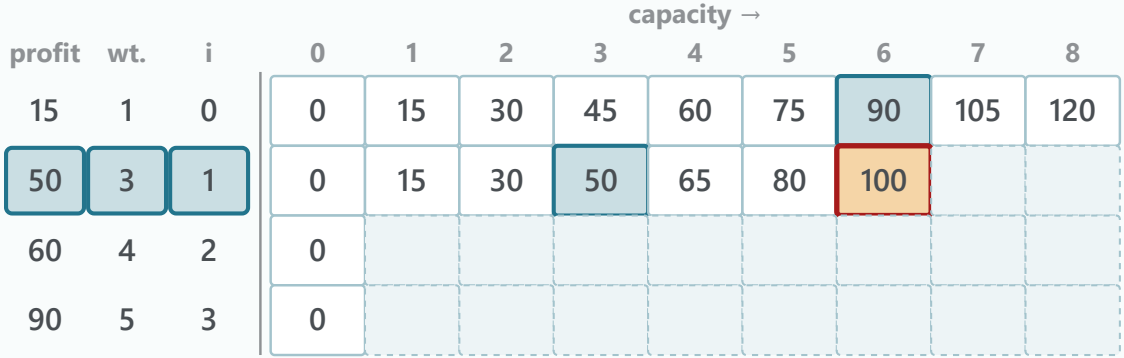
i=1, c=4: max(dp[0][4] = 60, 50 + dp[1][1] = 65) = **65**

Unbounded Knapsack: Filling the Table



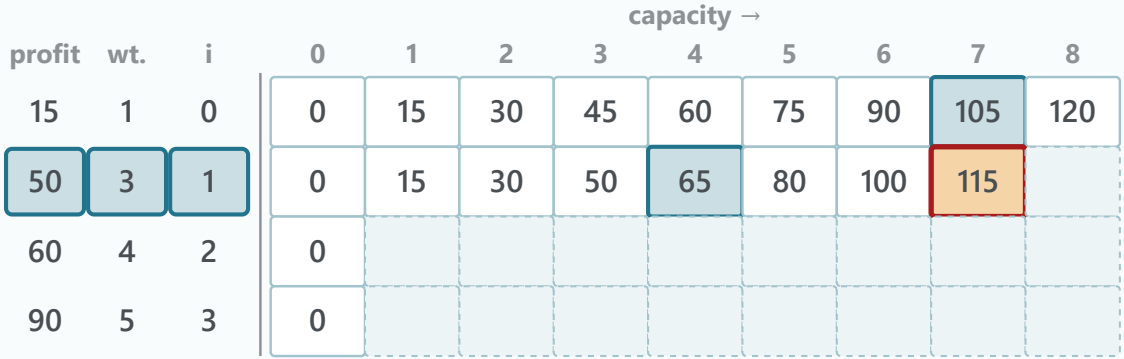
i=1, c=5: max(dp[0][5] = 75, 50 + dp[1][2] = 80) = **80**

Unbounded Knapsack: Filling the Table



i=1, c=6: max(dp[0][6] = 90, 50 + dp[1][3] = 100) = **100**

Unbounded Knapsack: Filling the Table



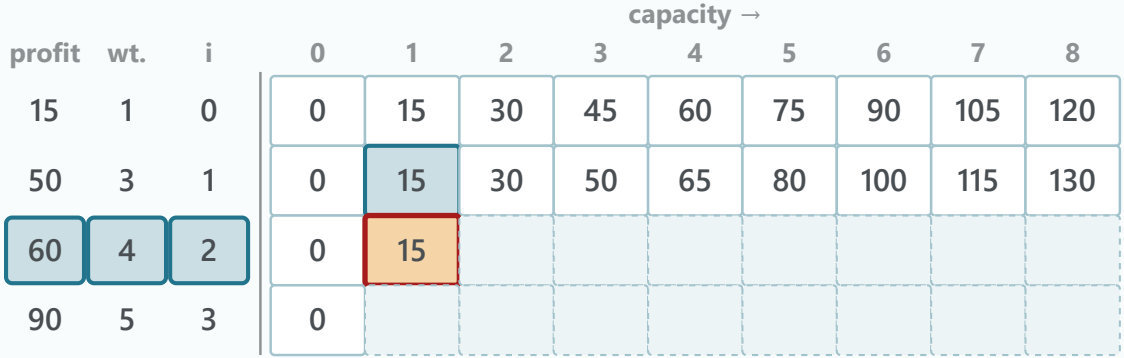
i=1, c=7: max(dp[0][7] = 105, 50 + dp[1][4] = 115) = **115**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0								
90	5	3	0								

i=1, c=8: max(dp[0][8] = 120, 50 + dp[1][5] = 130) = **130**

Unbounded Knapsack: Filling the Table



i=2, c=1: weight 4 > 1, does not fit → dp[2][1] = dp[1][1] = **15**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30						
90	5	3	0								

i=2, c=2: weight 4 > 2, does not fit → dp[2][2] = dp[1][2] = **30**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50					
90	5	3	0								

i=2, c=3: weight 4 > 3, does not fit → dp[2][3] = dp[1][3] = **50**

Unbounded Knapsack: Filling the Table

profit	wt.	i	capacity → 0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65				
90	5	3	0								

i=2, c=4: max(dp[1][4] = 65, 60 + dp[2][0] = 60) = **65**

Unbounded Knapsack: Filling the Table

profit	wt.	i	capacity → 0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80			
90	5	3	0								

i=2, c=5: max(dp[1][5] = 80, 60 + dp[2][1] = 75) = **80**

Unbounded Knapsack: Filling the Table

profit	wt.	i	capacity → 0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100		
90	5	3	0								

i=2, c=6: max(dp[1][6] = 100, 60 + dp[2][2] = 90) = **100**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	
90	5	3	0								

i=2, c=7: max(dp[1][7] = 115, 60 + dp[2][3] = 110) = **115**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0								

i=2, c=8: max(dp[1][8] = 130, 60 + dp[2][4] = 125) = **130**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15							

i=3, c=1: weight 5 > 1, does not fit → dp[3][1] = dp[2][1] = **15**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30						

i=3, c=2: weight 5 > 2, does not fit → dp[3][2] = dp[2][2] = **30**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50					

i=3, c=3: weight 5 > 3, does not fit → dp[3][3] = dp[2][3] = **50**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65				

i=3, c=4: weight 5 > 4, does not fit → dp[3][4] = dp[2][4] = **65**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90			

i=3, c=5: max(dp[2][5] = 80, 90 + dp[3][0] = 90) = **90**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90	105		

i=3, c=6: max(dp[2][6] = 100, 90 + dp[3][1] = 105) = **105**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90	105	120	

i=3, c=7: max(dp[2][7] = 115, 90 + dp[3][2] = 120) = **120**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90	105	120	140

i=3, c=8: max(dp[2][8] = 130, 90 + dp[3][3] = 140) = **140**

Unbounded Knapsack: Filling the Table

			capacity →								
profit	wt.	i	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90	105	120	140

Answer dp[3][8] = **140**: 90 + dp[3][3] = 90 + 50 → one item 3 (90, 5 kg) + one item 1 (50, 3 kg).

Unbounded Knapsack: A One-Dimensional Solution

```java
static int unboundedKnapsack(int[] w, int[] v, int W) {
  int[] dp = new int[W + 1];        // dp[c]: best with capacity c
  for (int c = 1; c <= W; c++)
    for (int j = 0; j < w.length; j++)
      if (w[j] <= c)
        dp[c] = Math.max(dp[c], dp[c - w[j]] + v[j]);
  return dp[W];
}
```

- The rows can be merged: dp[c] = max(dp[c − w_j] + v_j) over **all items**.
- For the example, dp = [0, 15, 30, 50, 65, 90, 105, 120, **140**] — the last row of the table.
- Time **O(n · W)**, memory **O(W)**.

Loop direction decides everything

In a one-dimensional array, scanning capacity **from low to high** means dp[c − w] has already been updated in this round → the item can be taken again (**unbounded**). Scanning **from high to low** reads the old value → **0/1**.

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
▶     int[] dp = new int[W + 1];  // dp[c]
4    for (int c = 1; c <= W; c++)
5      for (int j = 0; j < w.length; j++)
6        if (w[j] <= c)
7          dp[c] = Math.max(dp[c],
8                    dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



Profit [15, 50, 60, 90], weight [1, 3, 4, 5], W = 8. dp[c] = the largest profit with capacity c; dp[0] = 0 and the others are not computed yet.

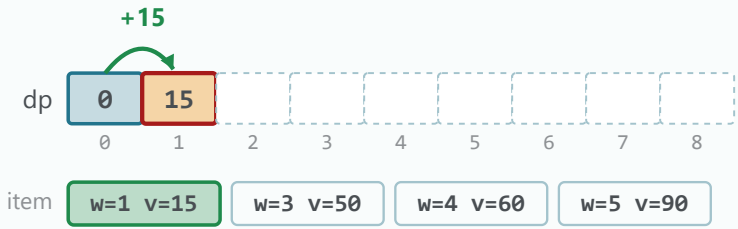
Step 1 / 10

VARIABLES

W = 8 | c = – | dp[c] = –

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 1: dp[0] + 15 = 15 → dp[1] = **15**.

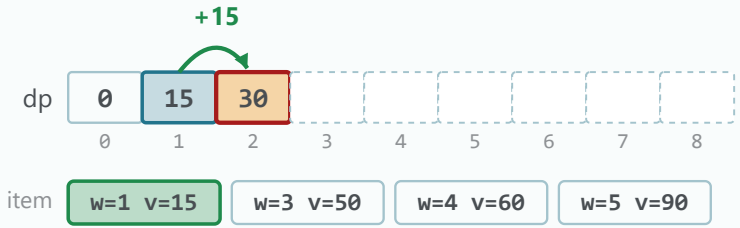
Step 2 / 10

VARIABLES

W = **8** c = **1** dp[c] = **15**

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 2: dp[1] + 15 = 30 → dp[2] = **30**. Item 0 is taken a second time: dp[1] was already updated in this round (unbounded).

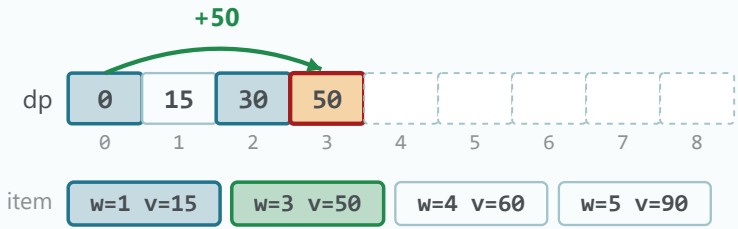
Step 3 / 10

VARIABLES

W = 8 | c = 2 | dp[c] = 30

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 3: dp[2] + 15 = 45, dp[0] + 50 = 50 → dp[3] = **50**.

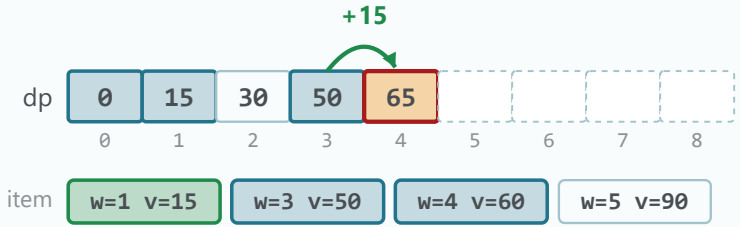
Step 4 / 10

VARIABLES

W = 8 | c = 3 | dp[c] = 50

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 4: dp[3] + 15 = 65, dp[1] + 50 = 65, dp[0] + 60 = 60 → dp[4] = **65**.

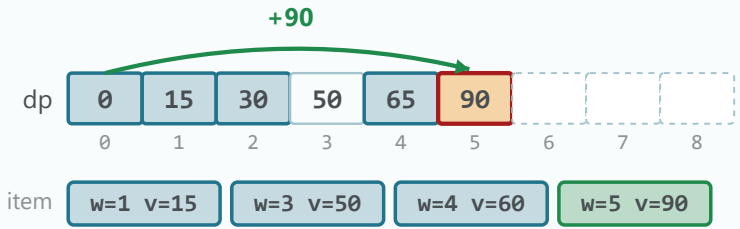
Step 5 / 10

VARIABLES

`W = 8` `c = 4` `dp[c] = 65`

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 5: dp[4] + 15 = 80, dp[2] + 50 = 80, dp[1] + 60 = 75, dp[0] + 90 = 90 → dp[5] = **90**.

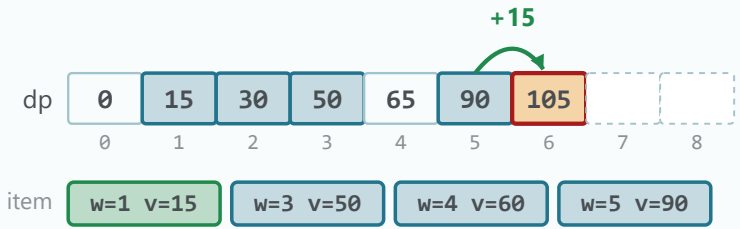
Step 6 / 10

VARIABLES

W = 8 | c = 5 | dp[c] = 90

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 6: dp[5] + 15 = 105, dp[3] + 50 = 100, dp[2] + 60 = 90, dp[1] + 90 = 105 → dp[6] = **105**.

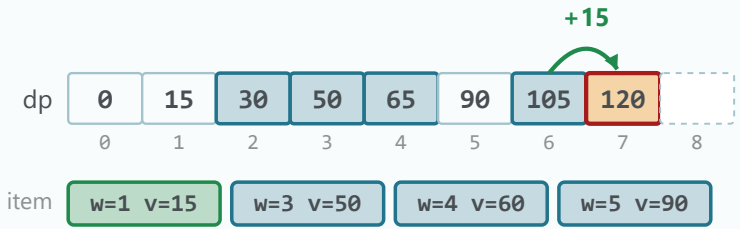
Step 7 / 10

VARIABLES

`W = 8` `c = 6` `dp[c] = 105`

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 7: dp[6] + 15 = 120, dp[4] + 50 = 115, dp[3] + 60 = 110, dp[2] + 90 = 120 → dp[7] = **120**.

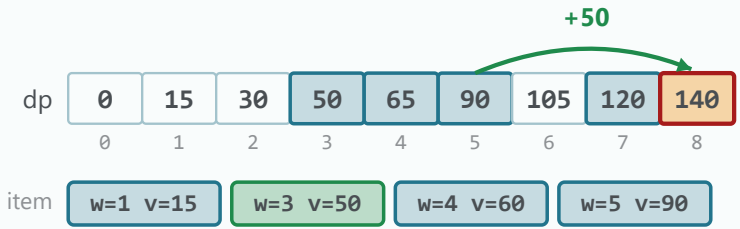
Step 8 / 10

VARIABLES

`W = 8` `c = 7` `dp[c] = 120`

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
▶    for (int c = 1; c <= W; c++)
▶      for (int j = 0; j < w.length; j++)
▶        if (w[j] <= c)
▶          dp[c] = Math.max(dp[c],
▶                           dp[c - w[j]] + v[j]);
9    return dp[W];
10 }
```



c = 8: dp[7] + 15 = 135, dp[5] + 50 = 140, dp[4] + 60 = 125, dp[3] + 90 = 140 → dp[8] = **140**.

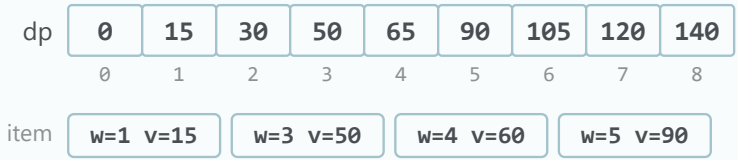
Step 9 / 10

VARIABLES

`W = 8` `c = 8` `dp[c] = 140`

unboundedKnapsack(w, v, 8)

```
1  static int unboundedKnapsack(
2      int[] w, int[] v, int W) {
3    int[] dp = new int[W + 1];  // dp[c]
4    for (int c = 1; c <= W; c++)
5      for (int j = 0; j < w.length; j++)
6        if (w[j] <= c)
7          dp[c] = Math.max(dp[c],
8                    dp[c - w[j]] + v[j]);
▶    return dp[W];
10 }
```



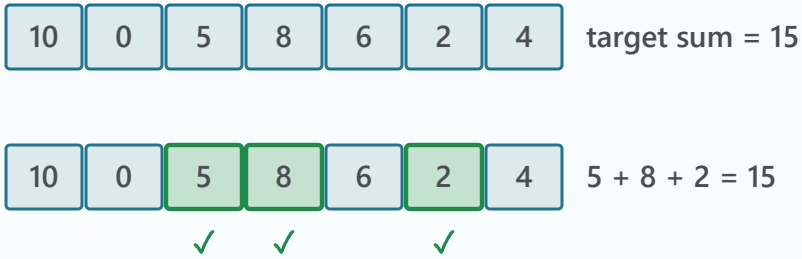
dp[8] = **140**: 50 + 90 (3 kg + 5 kg). The equally profitable 15 + 15 + 50 + 60 weighs 9 kg and does not fit; the table tried all combinations in O(n · W).

Step 10 / 10

VARIABLES

`W = 8` `c = 9` `dp[c] = –`

Subset Sum

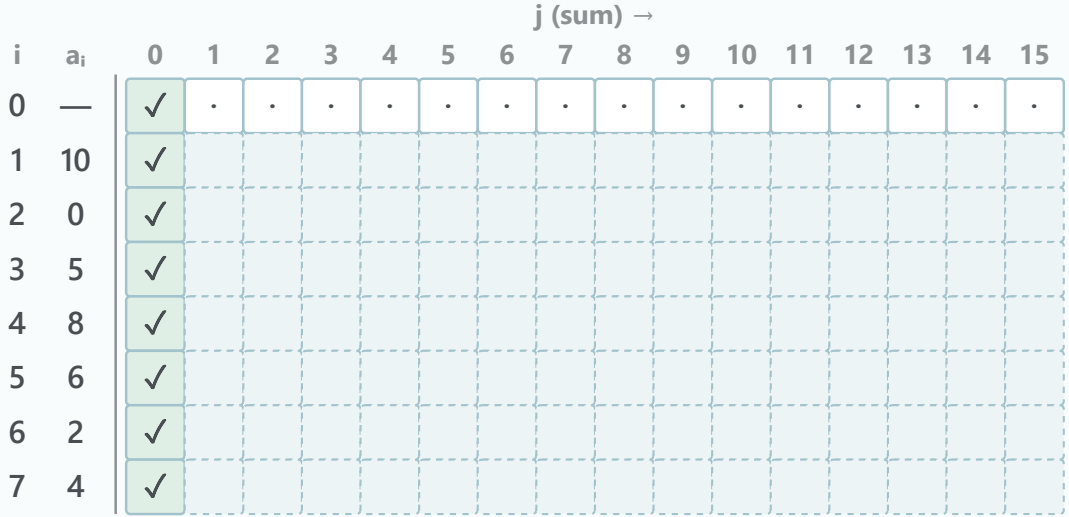


- Given nonnegative integers (target ≥ 0), is there a subset whose sum equals the **target** exactly? (Decision problem: yes / no.)
- **State:** dp[i][j] = can the first i elements make the sum j? (boolean)

dp[i][0] = true · dp[0][j] = false (j > 0)
dp[i][j] = dp[i−1][j] **or** ($a_i \le j$ **and** dp[i−1][j − a_i])

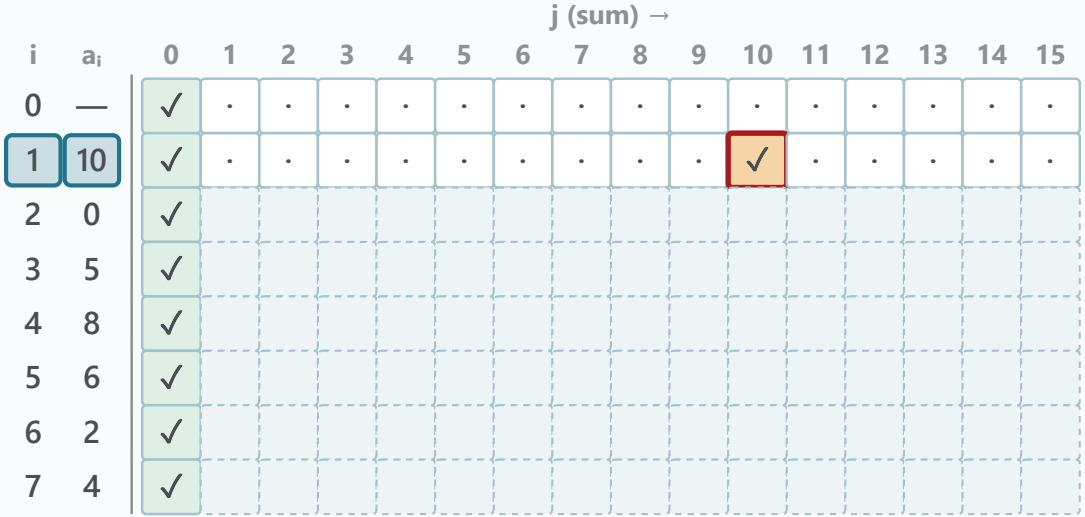
It is the special case of knapsack with "value = weight": logical **or** replaces max.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



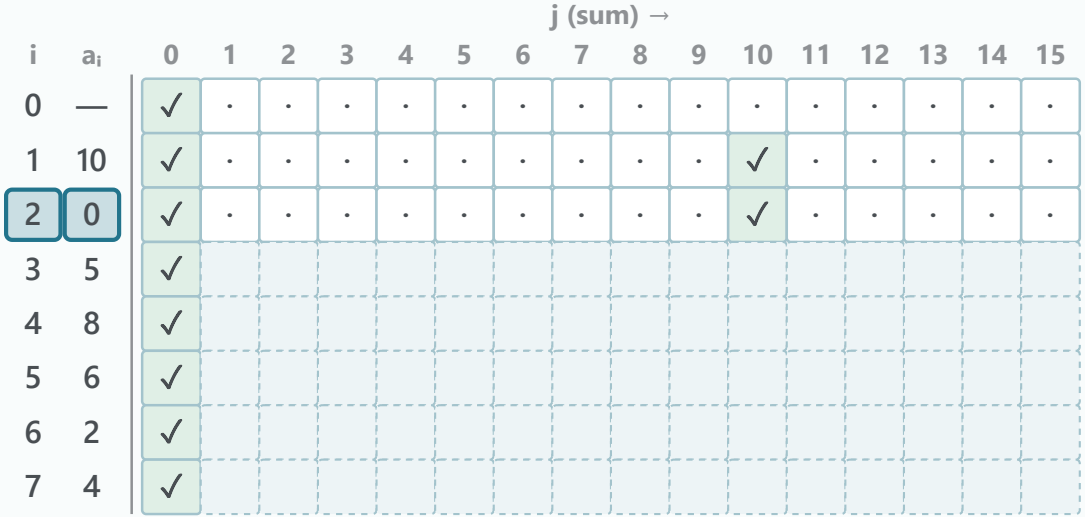
Column 0 is always ✓ (the empty set sums to 0). In row 0 (no elements), no other sum is reachable (·).

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



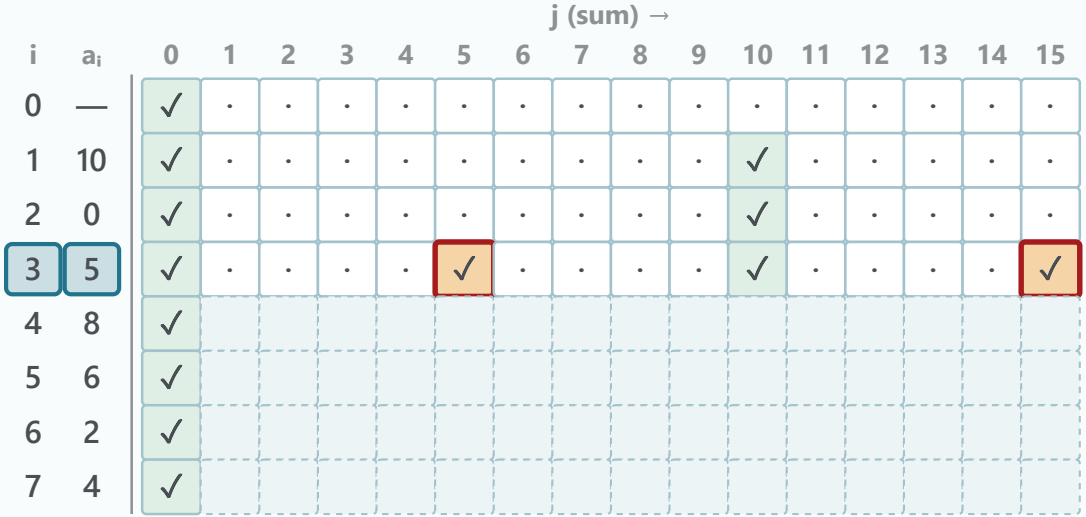
i=1, a=10: copy row i−1, and mark ✓ every j with dp[i−1][j−10] ✓ → new sums reached: 10.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



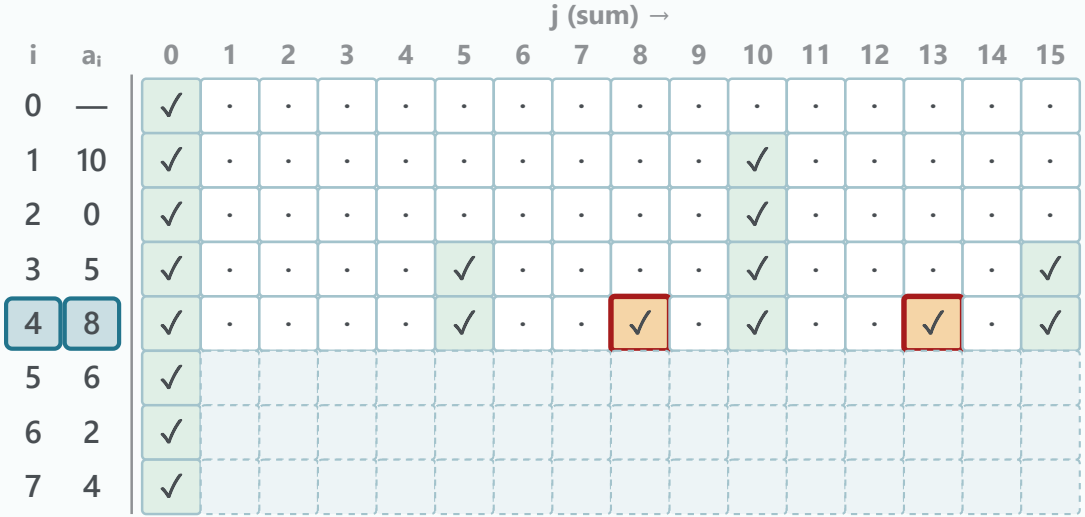
i=2, a=0: copy row i−1, and mark ✓ every j with dp[i−1][j−0] ✓ → no new sums (adding 0 changes nothing).

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



i=3, a=5: copy row i−1, and mark ✓ every j with dp[i−1][j−5] ✓ → new sums reached: 5, 15.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



i=4, a=8: copy row i−1, and mark ✓ every j with dp[i−1][j−8] ✓ → new sums reached: 8, 13.

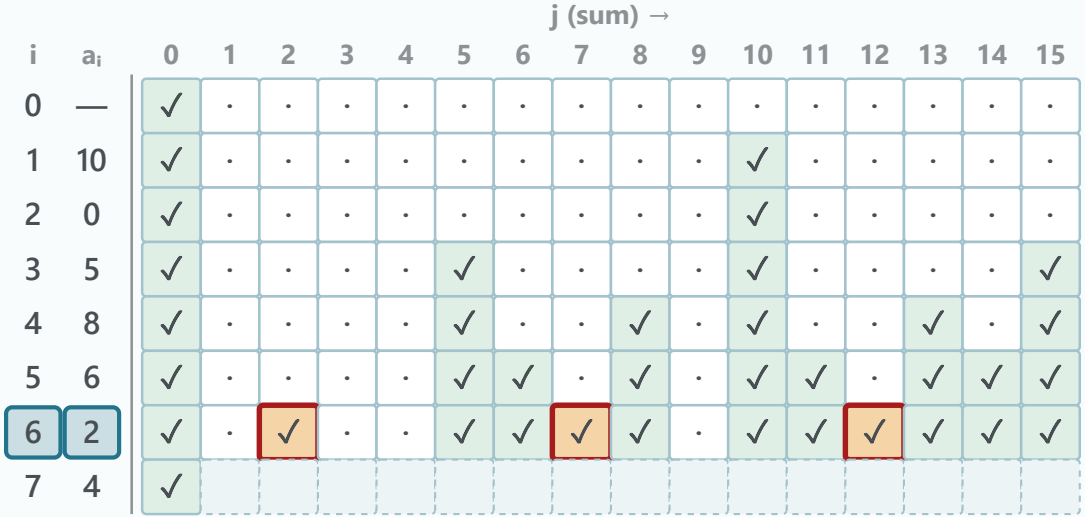
Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

j (sum) →

i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓															
7	4	✓															

i=5, a=6: copy row i−1, and mark ✓ every j with dp[i−1][j−6] ✓ → new sums reached: 6, 11, 14.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



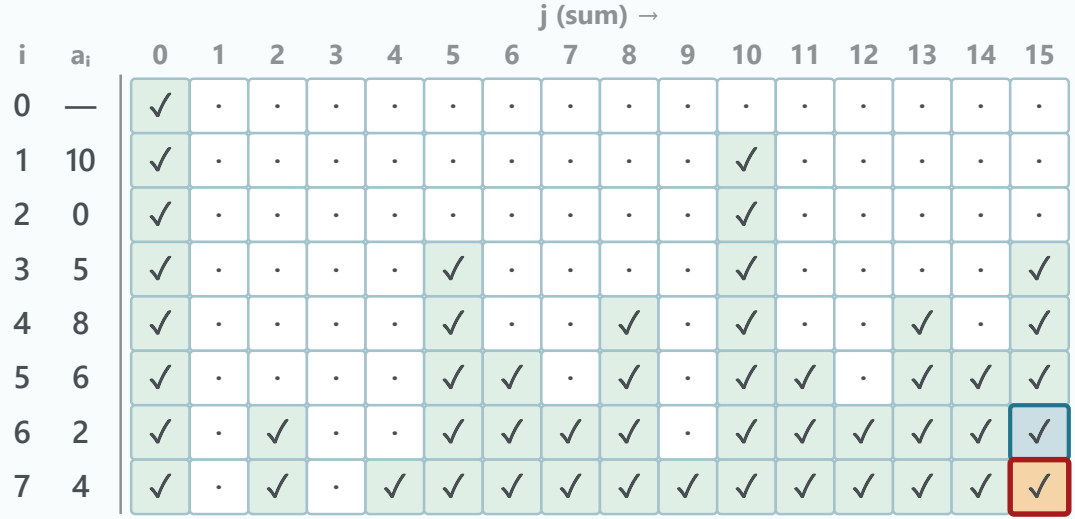
i=6, a=2: copy row i−1, and mark ✓ every j with dp[i−1][j−2] ✓ → new sums reached: 2, 7, 12.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

i=7, a=4: copy row i−1, and mark ✓ every j with dp[i−1][j−4] ✓ → new sums reached: 4, 9.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)



dp[7][15] = dp[6][15] = ✓ → sum 15 is possible without 4; not chosen, move up.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

dp[6][15] = dp[5][15] = ✓ → sum 15 is possible without 2; not chosen, move up.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

dp[5][15] = dp[4][15] = ✓ → sum 15 is possible without 6; not chosen, move up.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

dp[4][15] = dp[3][15] = ✓ → sum 15 is possible without 8; not chosen, move up.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

j (sum) →

i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

dp[3][15] ✓ but dp[2][15] · → 5 **chosen**; j = 15 − 5 = 10

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

dp[2][10] = dp[1][10] = ✓ → sum 10 is possible without 0; not chosen, move up.

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

dp[1][10] ✓ but dp[0][10] · → 10 **chosen**; j = 10 − 10 = 0

Subset Sum: Table (a = 10, 0, 5, 8, 6, 2, 4; target 15)

		j (sum) →															
i	a_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	—	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	10	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
2	0	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
3	5	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓
4	8	✓	·	·	·	·	✓	·	·	✓	·	✓	·	·	✓	·	✓
5	6	✓	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	✓	✓
6	2	✓	·	✓	·	·	✓	✓	✓	✓	·	✓	✓	✓	✓	✓	✓
7	4	✓	·	✓	·	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Reconstruction finds the subset {5, 10}: 10 + 5 = 15. {5, 8, 2} and {5, 6, 4} also work; the table yields only **one**.

Subset Sum: Code and Analysis

```
boolean subsetSum(int[] a, int target) {
  int n = a.length; // a[i] ≥ 0, target ≥ 0
  boolean[][] dp = new boolean[n + 1][target + 1];
  for (int i = 0; i <= n; i++)
    dp[i][0] = true;           // empty set
  for (int i = 1; i <= n; i++) {
    int x = a[i - 1];
    for (int j = 1; j <= target; j++)
      dp[i][j] = dp[i - 1][j]  // x not taken
        || (x <= j && dp[i - 1][j - x]);
  }
  return dp[n][target];
}
```

- Time and memory: **O(n · target)** — again pseudo-polynomial.
- The problem is **NP-complete**; if the target is large, the table cannot be built.
- This example has 6 different solutions: {10, 5}, {5, 8, 2}, {5, 6, 4}, and each of them with 0 added.
- One-dimensional version: scan j **from high to low** (as in 0/1).
- Application: the **partition** problem — splitting an array with sum S into two equal halves = subset sum with target S/2.

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
▶    boolean[][] dp =
▶        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;        // empty set
7    for (int i = 1; i <= n; i++) {
8      int x = a[i - 1];
9      for (int j = 1; j <= target; j++)
10       dp[i][j] = dp[i - 1][j] // x not taken
11           || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	·	·	·	·	·	·	·	·	·	·
3	1	·	·	·	·	·	·	·	·	·	·
34	2	·	·	·	·	·	·	·	·	·	·
4	3	·	·	·	·	·	·	·	·	·	·
12	4	·	·	·	·	·	·	·	·	·	·
5	5	·	·	·	·	·	·	·	·	·	·
2	6	·	·	·	·	·	·	·	·	·	·

a = [3, 34, 4, 12, 5, 2], target = 9. dp[i][j]: can the first i elements make the sum j? In Java a boolean array starts out false (·).

Step 1 / 9

VARIABLES

n = 6 | i = – | x = –

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
▶    for (int i = 0; i <= n; i++)
▶      dp[i][0] = true;        // empty set
7    for (int i = 1; i <= n; i++) {
8      int x = a[i - 1];
9      for (int j = 1; j <= target; j++)
10       dp[i][j] = dp[i - 1][j] // x not taken
11           || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	·	·	·	·	·	·	·
34	2	T	·	·	·	·	·	·	·	·	·
4	3	T	·	·	·	·	·	·	·	·	·
12	4	T	·	·	·	·	·	·	·	·	·
5	5	T	·	·	·	·	·	·	·	·	·
2	6	T	·	·	·	·	·	·	·	·	·

dp[i][0] = true in every row: the empty set gives sum 0. The rest of row 0 is false: with no elements there is no positive sum.

Step 2 / 9

VARIABLES

n = 6 | i = – | x = –

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
▶      int x = a[i - 1];
▶      for (int j = 1; j <= target; j++)
▶        dp[i][j] = dp[i - 1][j] // x not taken
▶            || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	·	·	·	·	·	·	·
4	3	T	·	·	·	·	·	·	·	·	·
12	4	T	·	·	·	·	·	·	·	·	·
5	5	T	·	·	·	·	·	·	·	·	·
2	6	T	·	·	·	·	·	·	·	·	·

i = 1, x = 3: the row above carries down; in addition, every j with dp[0][j − 3] true gives a new sum: **3**.

Step 3 / 9

VARIABLES

n = **6** i = **1** x = **3**

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
▶      int x = a[i - 1];
▶      for (int j = 1; j <= target; j++)
▶        dp[i][j] = dp[i - 1][j] // x not taken
▶            || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	T	·	·	·	·	·	·
4	3	T	·	·	·	·	·	·	·	·	·
12	4	T	·	·	·	·	·	·	·	·	·
5	5	T	·	·	·	·	·	·	·	·	·
2	6	T	·	·	·	·	·	·	·	·	·

i = 2, x = 34 > 9: x fits no j; the row is a copy of the one above.

Step 4 / 9

VARIABLES

n = 6 · i = 2 · x = 34

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
▶      int x = a[i - 1];
▶      for (int j = 1; j <= target; j++)
▶        dp[i][j] = dp[i - 1][j] // x not taken
▶            || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	T	·	·	·	·	·	·
4	3	T	·	·	T	T	·	·	T	·	·
12	4	T	·	·	·	·	·	·	·	·	·
5	5	T	·	·	·	·	·	·	·	·	·
2	6	T	·	·	·	·	·	·	·	·	·

i = 3, x = 4: the row above carries down; in addition, every j with dp[2][j − 4] true gives new sums: **4**, **7**.

Step 5 / 9

VARIABLES

n = 6 | i = 3 | x = 4

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
▶      int x = a[i - 1];
▶      for (int j = 1; j <= target; j++)
▶        dp[i][j] = dp[i - 1][j] // x not taken
▶            || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	T	·	·	·	·	·	·
4	3	T	·	·	T	T	·	·	T	·	·
12	4	T	·	·	T	T	·	·	T	·	·
5	5	T	·	·	·	·	·	·	·	·	·
2	6	T	·	·	·	·	·	·	·	·	·

i = 4, x = 12 > 9: x fits no j; the row is a copy of the one above.

Step 6 / 9

VARIABLES

n = 6 i = 4 x = 12

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
▶      int x = a[i - 1];
▶      for (int j = 1; j <= target; j++)
▶        dp[i][j] = dp[i - 1][j] // x not taken
▶            || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	T	·	·	·	·	·	·
4	3	T	·	·	T	T	·	·	T	·	·
12	4	T	·	·	T	T	·	·	T	·	·
5	5	T	·	·	T	T	T	·	T	T	T
2	6	T	·	·	·	·	·	·	·	·	·

i = 5, x = 5: the row above carries down; in addition, every j with dp[4][j − 5] true gives new sums: **5**, **8**, **9**. dp[5][9] = dp[4][4] → 4 + 5 = 9 found.

Step 7 / 9

VARIABLES

n = **6** i = **5** x = **5**

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
▶      int x = a[i - 1];
▶      for (int j = 1; j <= target; j++)
▶        dp[i][j] = dp[i - 1][j] // x not taken
▶            || (x <= j && dp[i - 1][j - x]);
12   }
13   return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	T	·	·	·	·	·	·
4	3	T	·	·	T	T	·	·	T	·	·
12	4	T	·	·	T	T	·	·	T	·	·
5	5	T	·	·	T	T	T	·	T	T	T
2	6	T	·	T	T	T	T	T	T	T	T

i = 6, x = 2: the row above carries down; in addition, every j with dp[5][j − 2] true gives new sums: **2**, **6**.

Step 8 / 9

VARIABLES

n = 6 i = 6 x = 2

subsetSum({3, 34, 4, 12, 5, 2}, 9)

```
1  boolean subsetSum(int[] a, int target) {
2    int n = a.length; // a[i] ≥ 0, target ≥ 0
3    boolean[][] dp =
4        new boolean[n + 1][target + 1];
5    for (int i = 0; i <= n; i++)
6      dp[i][0] = true;          // empty set
7    for (int i = 1; i <= n; i++) {
8      int x = a[i - 1];
9      for (int j = 1; j <= target; j++)
10       dp[i][j] = dp[i - 1][j] // x not taken
11           || (x <= j && dp[i - 1][j - x]);
12   }
▶    return dp[n][target];
14 }
```

x		0	1	2	3	4	5	6	7	8	9
–	0	T	·	·	·	·	·	·	·	·	·
3	1	T	·	·	T	·	·	·	·	·	·
34	2	T	·	·	T	·	·	·	·	·	·
4	3	T	·	·	T	T	·	·	T	·	·
12	4	T	·	·	T	T	·	·	T	·	·
5	5	T	·	·	T	T	T	·	T	T	T
2	6	T	·	T	T	T	T	T	T	T	T

dp[6][9] = **true**: some subset sums to 9 (e.g., {4, 5} or {3, 4, 2}). The table has (n + 1) × (target + 1) = 70 cells.

Step 9 / 9

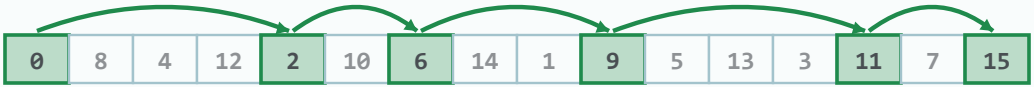
VARIABLES

n = 6 | i = 7 | x = 2

Longest Increasing Subsequence

- A **subsequence** keeps the order of the elements but may **skip** some. A **substring / subarray** must be **contiguous**.
- LIS: the longest subsequence whose elements are **strictly increasing**.

0, 8, 4, 12, **2**, 10, **6**, 14, 1, **9**, 5, 13, 3, **11**, 7, **15** → LIS = **0, 2, 6, 9, 11, 15** (length 6)



$O(n^2)$ DP

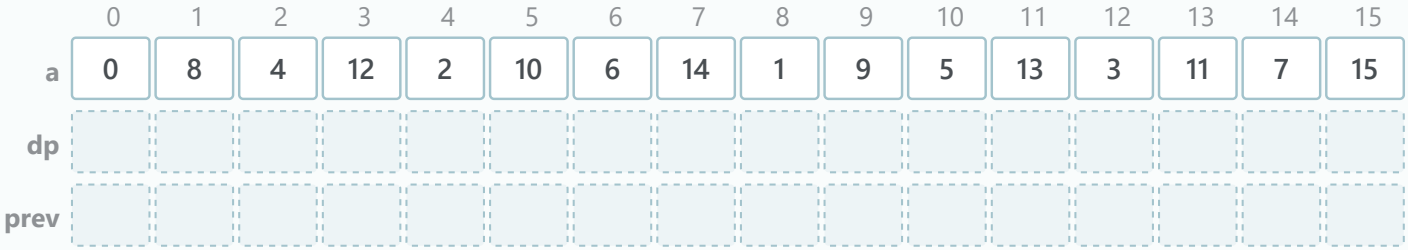
dp[i] = length of the longest increasing subsequence **ending** at a[i]. dp[i] = 1 + max{ dp[j] : j < i, a[j] < a[i] }.

O(n log n): active lists

Keep, for each length, the list with the smallest last element; place each new element by binary search.

This sequence has more than one LIS (e.g., 0, 4, 6, 9, 13, 15); the length is always unique.

LIS: $O(n^2)$ Dynamic Programming

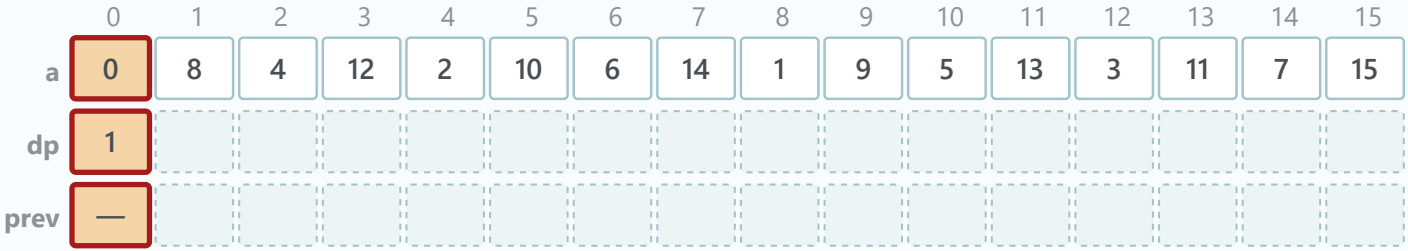


Start: every element alone is an increasing subsequence of length 1.

Step 1 / 18

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

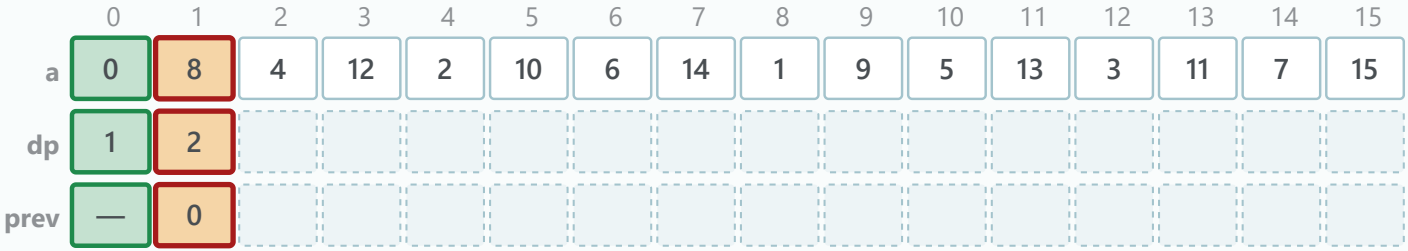
LIS: O(n^2) Dynamic Programming



i=0: a[0]=0, no earlier elements → dp[0] = **1**.

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

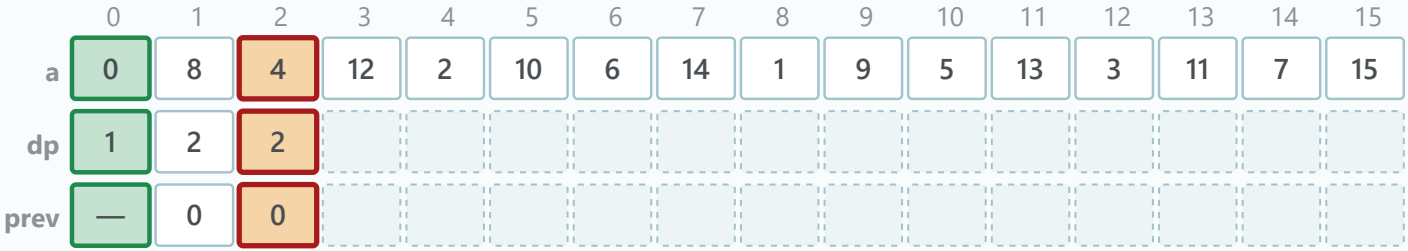
LIS: $O(n^2)$ Dynamic Programming



i=1: a[1]=8; largest dp among smaller earlier elements (blue): j=0 (dp=1) → dp[1] = **2**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

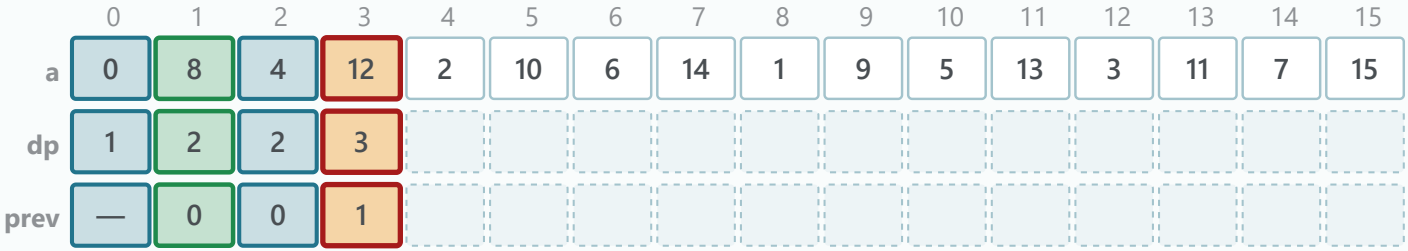
LIS: $O(n^2)$ Dynamic Programming



i=2: a[2]=4; largest dp among smaller earlier elements (blue): j=0 (dp=1) → dp[2] = **2**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

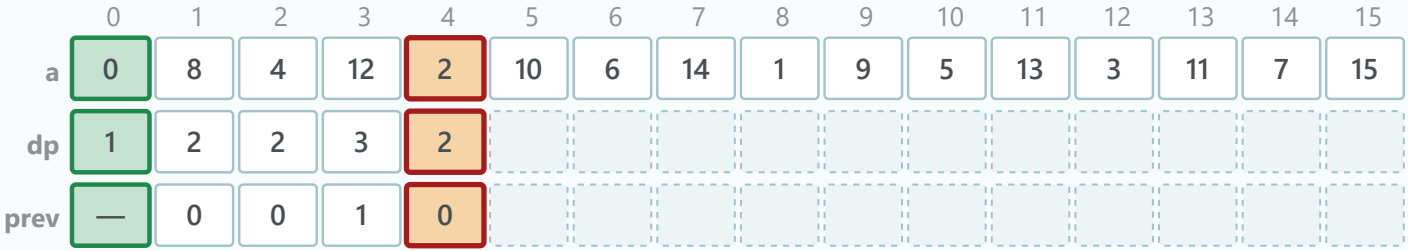
LIS: $O(n^2)$ Dynamic Programming



i=3: a[3]=12; largest dp among smaller earlier elements (blue): j=1 (dp=2) → dp[3] = **3**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

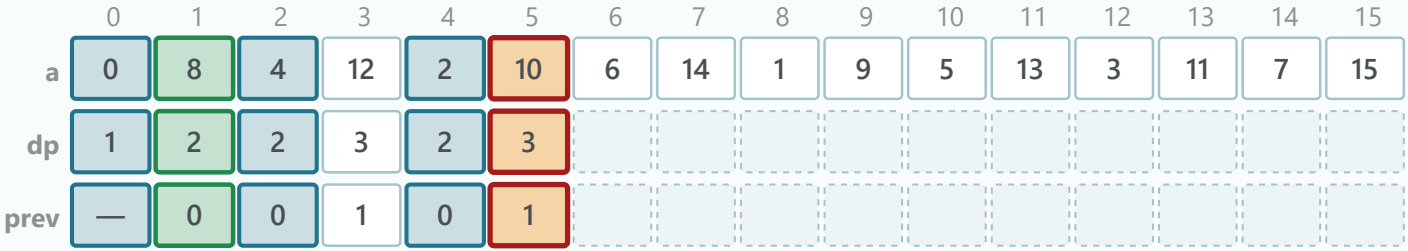
LIS: $O(n^2)$ Dynamic Programming



i=4: a[4]=2; largest dp among smaller earlier elements (blue): j=0 (dp=1) → dp[4] = **2**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

LIS: $O(n^2)$ Dynamic Programming



i=5: a[5]=10; largest dp among smaller earlier elements (blue): j=1 (dp=2) → dp[5] = **3**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

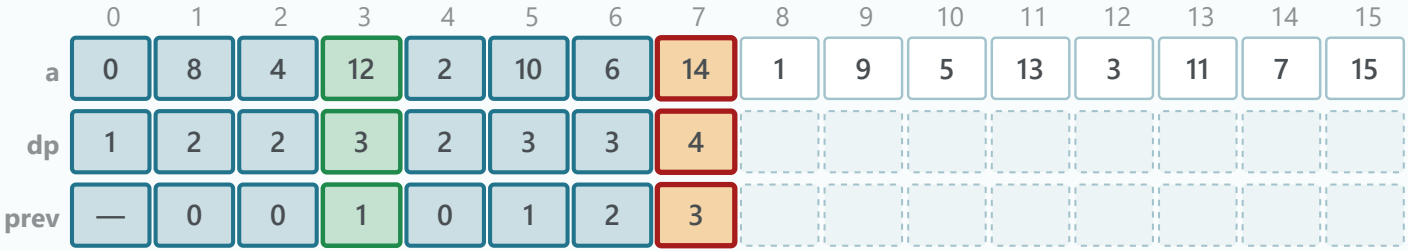
LIS: $O(n^2)$ Dynamic Programming

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp	1	2	2	3	2	3	3									
prev	—	0	0	1	0	1	2									

i=6: a[6]=6; largest dp among smaller earlier elements (blue): j=2 (dp=2) → dp[6] = **3**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

LIS: O(n^2) Dynamic Programming



i=7: a[7]=14; largest dp among smaller earlier elements (blue): j=3 (dp=3) → dp[7] = **4**

Step 9 / 18

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

LIS: O(n^2) Dynamic Programming

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp	1	2	2	3	2	3	3	4	2							
prev	—	0	0	1	0	1	2	3	0							

i=8: a[8]=1; largest dp among smaller earlier elements (blue): j=0 (dp=1) → dp[8] = **2**

Step 10 / 18

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

LIS: $O(n^2)$ Dynamic Programming



i=9: a[9]=9; largest dp among smaller earlier elements (blue): j=6 (dp=3) → dp[9] = **4**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

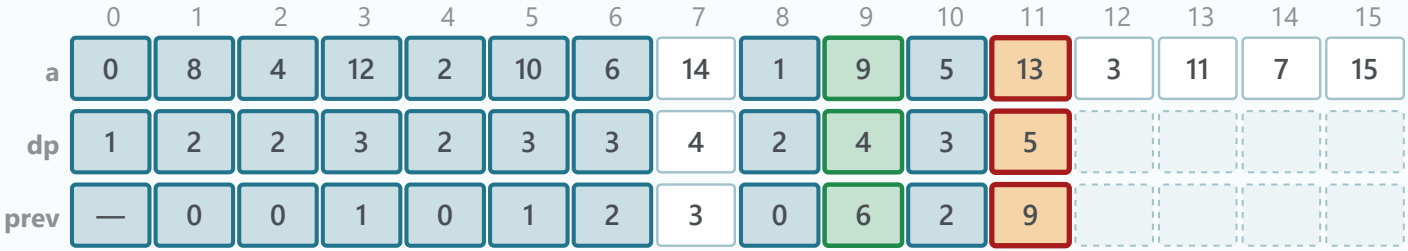
LIS: $O(n^2)$ Dynamic Programming

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp	1	2	2	3	2	3	3	4	2	4	3					
prev	—	0	0	1	0	1	2	3	0	6	2					

i=10: a[10]=5; largest dp among smaller earlier elements (blue): j=2 (dp=2) → dp[10] = **3**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

LIS: O(n^2) Dynamic Programming



i=11: a[11]=13; largest dp among smaller earlier elements (blue): j=9 (dp=4) → dp[11] = **5**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

LIS: O(n^2) Dynamic Programming

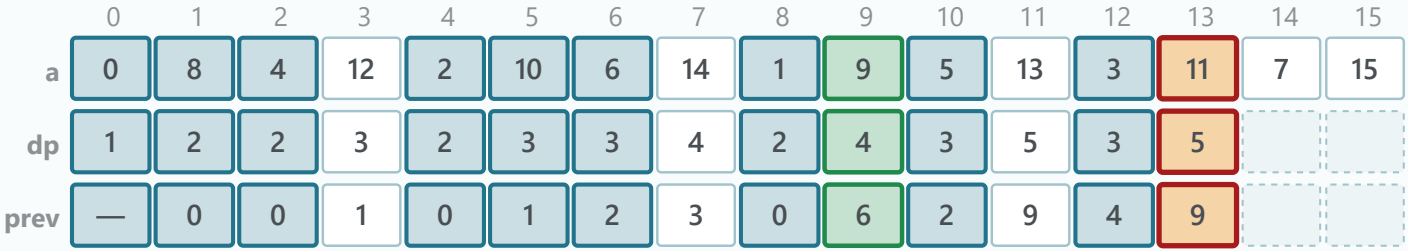
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp	1	2	2	3	2	3	3	4	2	4	3	5	3			
prev	—	0	0	1	0	1	2	3	0	6	2	9	4			

i=12: a[12]=3; largest dp among smaller earlier elements (blue): j=4 (dp=2) → dp[12] = **3**

Step 14 / 18

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

LIS: $O(n^2)$ Dynamic Programming



i=13: a[13]=11; largest dp among smaller earlier elements (blue): j=9 (dp=4) → dp[13] = **5**

Step 15 / 18

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

LIS: O(n^2) Dynamic Programming



i=14: a[14]=7; largest dp among smaller earlier elements (blue): j=6 (dp=3) → dp[14] = **4**

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

LIS: $O(n^2)$ Dynamic Programming

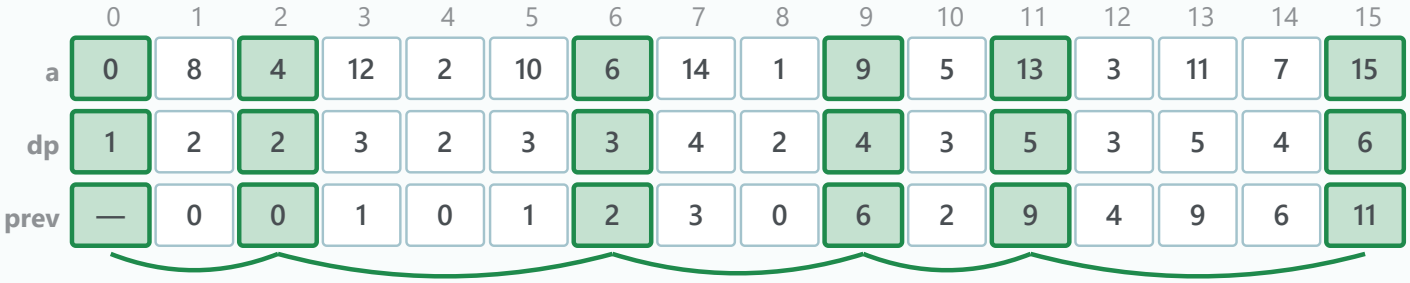
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp	1	2	2	3	2	3	3	4	2	4	3	5	3	5	4	6
prev	—	0	0	1	0	1	2	3	0	6	2	9	4	9	6	11

i=15: a[15]=15; largest dp among smaller earlier elements (blue): j=11 (dp=5) → dp[15] = **6**

Step 17 / 18

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. $O(n^2)$ comparisons in total.

LIS: O(n^2) Dynamic Programming



Largest dp = 6 (i=15). Follow the `prev` indices: 15 → 11 → 9 → 6 → 2 → 0 ⇒ **0, 4, 6, 9, 13, 15** (ties pick the first j; 0, 2, 6, 9, 11, 15 is also an LIS of length 6).

Orange: element being processed · blue: candidates with a[j] < a[i] · green: best previous element. O(n^2) comparisons in total.

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

No lists yet: start the first active list with [0].

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, **8,** 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 8

8 is larger than every end: copy the longest list and append **8** (length 2).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 4

~~0, 8~~ dropped

4 extends the list ending in 0 → [0, 4]; [0, 8] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, **12,** 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

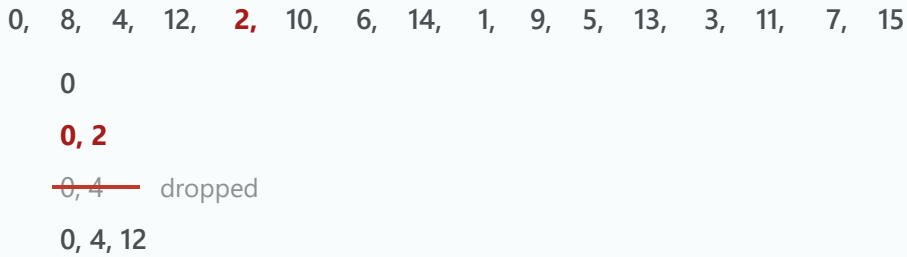
0, 4

0, 4, 12

12 is larger than every end: copy the longest list and append **12** (length 3).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.



2 extends the list ending in 0 → [0, 2]; [0, 4] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, **10,** 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 2

0, 2, 10

~~0, 4, 12~~ dropped

10 extends the list ending in 2 → [0, 2, 10]; [0, 4, 12] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 2

0, 2, 6

~~0, 2, 10~~ dropped

6 extends the list ending in 2 → [0, 2, 6]; [0, 2, 10] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, **14,** 1, 9, 5, 13, 3, 11, 7, 15

0

0, 2

0, 2, 6

0, 2, 6, 14

14 is larger than every end: copy the longest list and append **14** (length 4).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 1

~~0, 2~~ dropped

0, 2, 6

0, 2, 6, 14

1 extends the list ending in 0 → [0, 1]; [0, 2] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, **9,** 5, 13, 3, 11, 7, 15

0

0, 1

0, 2, 6

0, 2, 6, 9

~~0, 2, 6, 14~~ dropped

9 extends the list ending in 6 → [0, 2, 6, 9]; [0, 2, 6, 14] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, **5,** 13, 3, 11, 7, 15

0

0, 1

0, 1, 5

~~0, 2, 6~~ dropped

0, 2, 6, 9

5 extends the list ending in 1 → [0, 1, 5]; [0, 2, 6] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 1

0, 1, 5

0, 2, 6, 9

0, 2, 6, 9, 13

13 is larger than every end: copy the longest list and append **13** (length 5).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 1

0, 1, 3

~~0, 1, 5~~ dropped

0, 2, 6, 9

0, 2, 6, 9, 13

3 extends the list ending in 1 → [0, 1, 3]; [0, 1, 5] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 1

0, 1, 3

0, 2, 6, 9

0, 2, 6, 9, 11

~~0, 2, 6, 9, 13~~ dropped

11 extends the list ending in 9 → [0, 2, 6, 9, 11]; [0, 2, 6, 9, 13] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 1

0, 1, 3

0, 1, 3, 7

~~0, 2, 6, 9~~ dropped

0, 2, 6, 9, 11

7 extends the list ending in 3 → [0, 1, 3, 7]; [0, 2, 6, 9] of the same length is now redundant (larger end).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, **15**

0

0, 1

0, 1, 3

0, 1, 3, 7

0, 2, 6, 9, 11

0, 2, 6, 9, 11, 15

15 is larger than every end: copy the longest list and append **15** (length 6).

LIS: The Active-Lists Method

For each length, keep the increasing list with the **smallest last element**; a smaller end is more open to future growth.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

0

0, 1

0, 1, 3

0, 1, 3, 7

0, 2, 6, 9, 11

0, 2, 6, 9, 11, 15 ← **an LIS (length 6)**

The sequence is done: the longest active list is **0, 2, 6, 9, 11, 15** → LIS length 6.

LIS: $O(n^2)$ Code

```java
static int lis(int[] a) {
  int n = a.length, best = 0;
  int[] dp = new int[n];
  for (int i = 0; i < n; i++) {
    dp[i] = 1;                // on its own
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;    // extend
    best = Math.max(best, dp[i]);
  }
  return best;
}
```

- The inner loop tries to extend the sequence of every earlier element smaller than a[i].
- Time **$O(n^2)$**, memory **$O(n)$**.
- To build the sequence, store `prev[i] = j`, or scan the dp values downward from the end.
- The answer is **max dp[i]**, not dp[n−1]: an LIS can end anywhere.

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
▶    int n = a.length, best = 0;
▶    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
5      dp[i] = 1;          // on its own
6      for (int j = 0; j < i; j++)
7        if (a[j] < a[i] &&
8            dp[j] + 1 > dp[i])
9          dp[i] = dp[j] + 1;  // extend
10     best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



a = [5, 2, 8, 6, 3, 6, 9, 7]. dp[i] = length of the longest increasing subsequence **ending** at a[i]; not computed yet.

Step 1 / 10

VARIABLES

`i = –` `dp[i] = –` `best = 0`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
       dp[i] = 1;          // on its own
       for (int j = 0; j < i; j++)
         if (a[j] < a[i] &&
             dp[j] + 1 > dp[i])
           dp[i] = dp[j] + 1;  // extend
       best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 0, a[i] = 5: no smaller element on its left → dp[0] = 1 (on its own).

Step 2 / 10

VARIABLES

i = 0 | dp[i] = 1 | best = 1

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
▶      dp[i] = 1;          // on its own
▶      for (int j = 0; j < i; j++)
▶        if (a[j] < a[i] &&
▶            dp[j] + 1 > dp[i])
▶          dp[i] = dp[j] + 1;  // extend
▶      best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 1, a[i] = 2: no smaller element on its left → dp[1] = 1 (on its own).

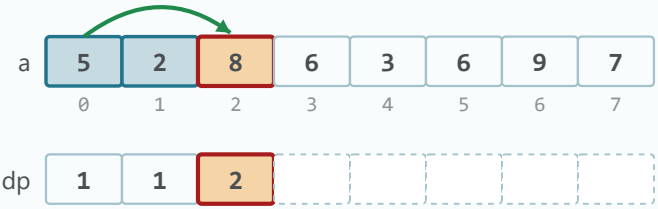
Step 3 / 10

VARIABLES

`i = 1` `dp[i] = 1` `best = 1`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
▶      dp[i] = 1;          // on its own
▶      for (int j = 0; j < i; j++)
▶        if (a[j] < a[i] &&
▶            dp[j] + 1 > dp[i])
▶          dp[i] = dp[j] + 1;  // extend
▶      best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 2, a[i] = 8: smaller elements on the left: 5, 2. Extending the longest, j = 0 (dp = 1) → dp[2] = **2**.

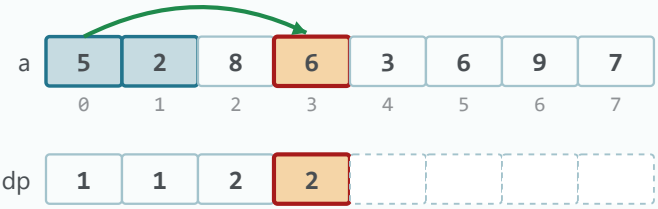
Step 4 / 10

VARIABLES

`i = 2` `dp[i] = 2` `best = 2`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
▶      dp[i] = 1;          // on its own
▶      for (int j = 0; j < i; j++)
▶        if (a[j] < a[i] &&
▶            dp[j] + 1 > dp[i])
▶          dp[i] = dp[j] + 1;  // extend
▶      best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 3, a[i] = 6: smaller elements on the left: 5, 2. Extending the longest, j = 0 (dp = 1) → dp[3] = **2**.

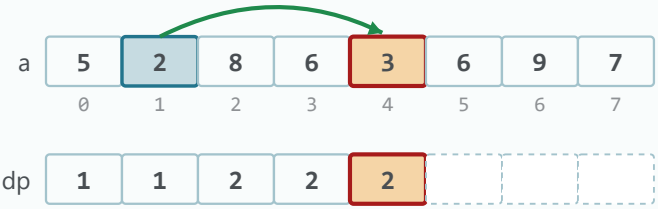
Step 5 / 10

VARIABLES

`i = 3` `dp[i] = 2` `best = 2`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
▶      dp[i] = 1;          // on its own
▶      for (int j = 0; j < i; j++)
▶        if (a[j] < a[i] &&
▶            dp[j] + 1 > dp[i])
▶          dp[i] = dp[j] + 1;  // extend
▶      best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 4, a[i] = 3: smaller elements on the left: 2.
Extending the longest, j = 1 (dp = 1) → dp[4] = **2**.

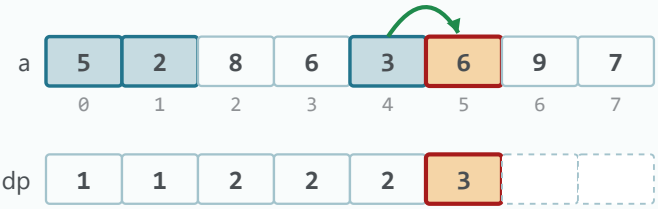
Step 6 / 10

VARIABLES

`i = 4` `dp[i] = 2` `best = 2`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
       dp[i] = 1;          // on its own
       for (int j = 0; j < i; j++)
         if (a[j] < a[i] &&
             dp[j] + 1 > dp[i])
           dp[i] = dp[j] + 1;  // extend
       best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 5, a[i] = 6: smaller elements on the left: 5, 2, 3. Extending the longest, j = 4 (dp = 2) → dp[5] = **3**.

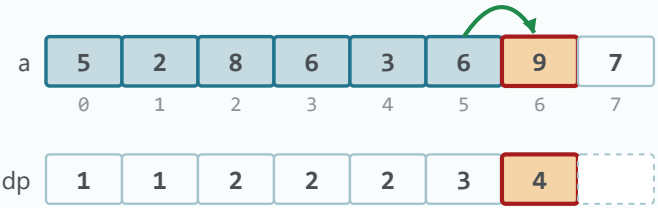
Step 7 / 10

VARIABLES

`i = 5` `dp[i] = 3` `best = 3`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
       dp[i] = 1;          // on its own
       for (int j = 0; j < i; j++)
         if (a[j] < a[i] &&
             dp[j] + 1 > dp[i])
           dp[i] = dp[j] + 1;  // extend
       best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 6, a[i] = 9: smaller elements on the left: 5, 2, 8, 6, 3, 6. Extending the longest, j = 5 (dp = 3) → dp[6] = **4**.

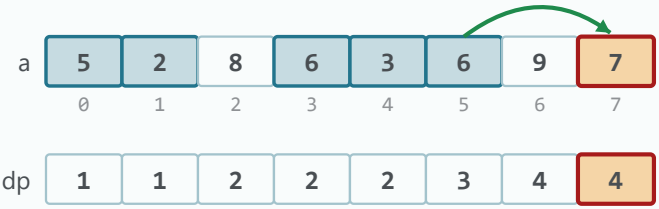
Step 8 / 10

VARIABLES

`i = 6` `dp[i] = 4` `best = 4`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
▶      dp[i] = 1;          // on its own
▶      for (int j = 0; j < i; j++)
▶        if (a[j] < a[i] &&
▶            dp[j] + 1 > dp[i])
▶          dp[i] = dp[j] + 1;  // extend
▶      best = Math.max(best, dp[i]);
11   }
12   return best;
13 }
```



i = 7, a[i] = 7: smaller elements on the left: 5, 2, 6, 3, 6. Extending the longest, j = 5 (dp = 3) → dp[7] = **4**.

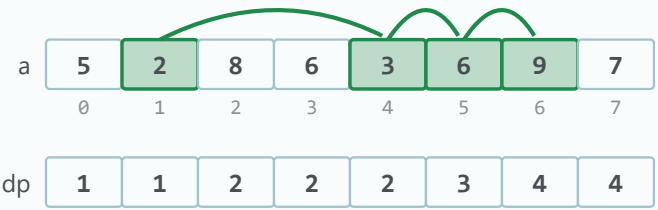
Step 9 / 10

VARIABLES

`i = 7` `dp[i] = 4` `best = 4`

lis({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lis(int[] a) {
2    int n = a.length, best = 0;
3    int[] dp = new int[n];
4    for (int i = 0; i < n; i++) {
5      dp[i] = 1;          // on its own
6      for (int j = 0; j < i; j++)
7        if (a[j] < a[i] &&
8            dp[j] + 1 > dp[i])
9          dp[i] = dp[j] + 1;  // extend
10     best = Math.max(best, dp[i]);
11   }
▶    return best;
13 }
```



best = **4** = max dp[i] (not dp[n − 1]). A sample LIS: 2, 3, 6, 9 (the chain with dp values 1, 2, 3, 4).

Step 10 / 10

VARIABLES

`i = 8` `dp[i] = –` `best = 4`

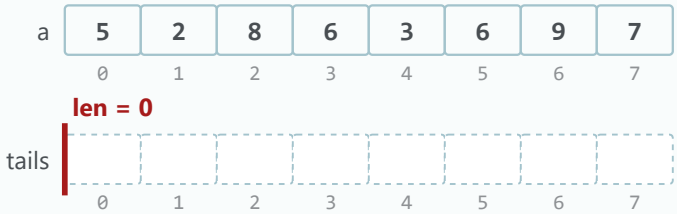
LIS: O(n log n) — The Tails Array and Binary Search

```java
static int lisFast(int[] a) {
  int[] tails = new int[a.length];
  int len = 0;            // list count
  for (int x : a) {
    int lo = 0, hi = len; // tails[lo] >= x
    while (lo < hi) {
      int mid = (lo + hi) / 2;
      if (tails[mid] < x) lo = mid + 1;
      else hi = mid;
    }
    tails[lo] = x;        // replace / extend
    if (lo == len) len++;
  }
  return len;
}
```

- Only the **last elements** of the active lists are stored; tails[] is always increasing, so the position is found by **binary search** → O(n log n).
- For the sample sequence, tails = [0, 1, 3, 7, 11, 15]: the length is right (6), but this is **not** an LIS (7 comes after 11).
- To build the sequence, also store each element's predecessor index.

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
▶    int[] tails = new int[a.length];
▶    int len = 0;       // number of lists
4    for (int x : a) {
5      int lo = 0, hi = len;
6      while (lo < hi) {
7        int mid = (lo + hi) / 2;
8        if (tails[mid] < x) lo = mid + 1;
9        else hi = mid;
10     }
11     tails[lo] = x;     // place / extend
12     if (lo == len) len++;
13   }
14   return len;
15 }
```



tails[k] = the **smallest** last element of an increasing subsequence of length k + 1. tails[0..len−1] is always increasing; initially len = 0.

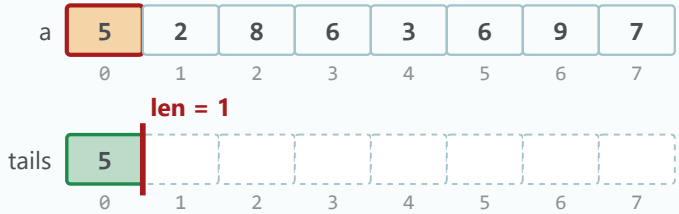
Step 1 / 9

VARIABLES

`x = –` `lo = –` `len = 0`

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
▶      int lo = 0, hi = len;
▶      while (lo < hi) {
▶        int mid = (lo + hi) / 2;
▶        if (tails[mid] < x) lo = mid + 1;
▶        else hi = mid;
10     }
▶      tails[lo] = x;    // place / extend
▶      if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 5 is larger than every end: binary search gives lo = len = 0 → tails[0] = 5, the list grows (len = 1).

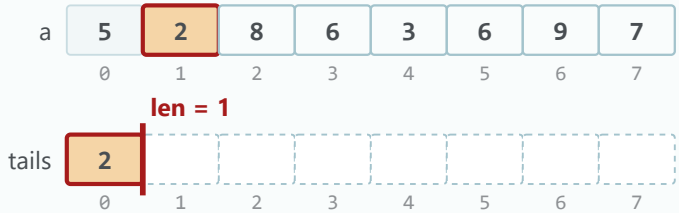
Step 2 / 9

VARIABLES

`x = 5` `lo = 0` `len = 1`

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
       int lo = 0, hi = len;
       while (lo < hi) {
         int mid = (lo + hi) / 2;
         if (tails[mid] < x) lo = mid + 1;
         else hi = mid;
10     }
       tails[lo] = x;     // place / extend
       if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 2: the first position with tails[lo] ≥ x is lo = 0 (1 comparison) → 2 replaces 5; len is unchanged, but the length-1 sequence now ends with a smaller number.

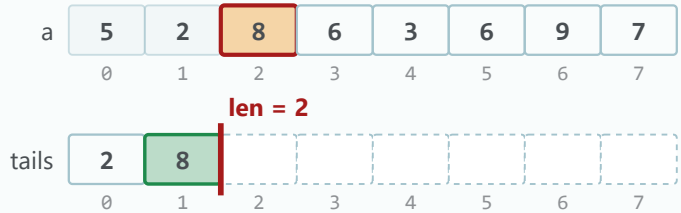
Step 3 / 9

VARIABLES

x = 2 | lo = 0 | len = 1

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
▶      int lo = 0, hi = len;
▶      while (lo < hi) {
▶        int mid = (lo + hi) / 2;
▶        if (tails[mid] < x) lo = mid + 1;
▶        else hi = mid;
10     }
▶      tails[lo] = x;    // place / extend
▶      if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 8 is larger than every end: binary search gives lo = len = 1 → tails[1] = 8, the list grows (len = 2).

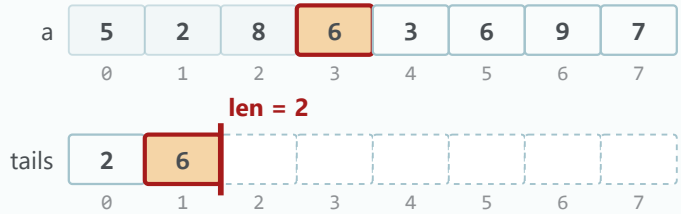
Step 4 / 9

VARIABLES

`x = 8` `lo = 1` `len = 2`

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
       int lo = 0, hi = len;
       while (lo < hi) {
         int mid = (lo + hi) / 2;
         if (tails[mid] < x) lo = mid + 1;
         else hi = mid;
10     }
       tails[lo] = x;     // place / extend
       if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 6: the first position with tails[lo] ≥ x is lo = 1 (2 comparisons) → 6 replaces 8; len is unchanged, but the length-2 sequence now ends with a smaller number.

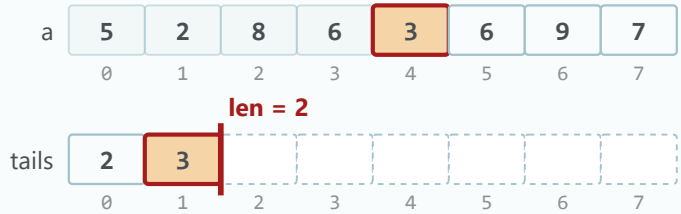
Step 5 / 9

VARIABLES

x = 6 | lo = 1 | len = 2

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
       int lo = 0, hi = len;
       while (lo < hi) {
         int mid = (lo + hi) / 2;
         if (tails[mid] < x) lo = mid + 1;
         else hi = mid;
10     }
       tails[lo] = x;     // place / extend
       if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 3: the first position with tails[lo] ≥ x is lo = 1 (2 comparisons) → 3 replaces 6; len is unchanged, but the length-2 sequence now ends with a smaller number.

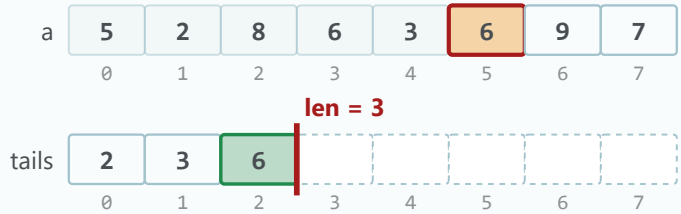
Step 6 / 9

VARIABLES

x = 3 | lo = 1 | len = 2

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
▶      int lo = 0, hi = len;
▶      while (lo < hi) {
▶        int mid = (lo + hi) / 2;
▶        if (tails[mid] < x) lo = mid + 1;
▶        else hi = mid;
10     }
▶      tails[lo] = x;     // place / extend
▶      if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 6 is larger than every end: binary search gives lo = len = 2 → tails[2] = 6, the list grows (len = 3).

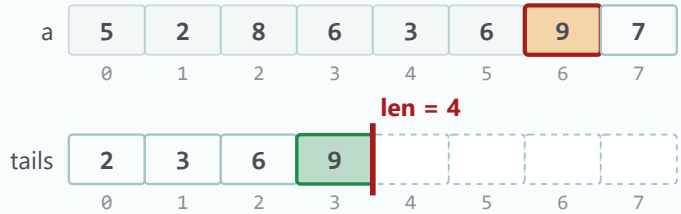
Step 7 / 9

VARIABLES

`x = 6` `lo = 2` `len = 3`

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
▶      int lo = 0, hi = len;
▶      while (lo < hi) {
▶        int mid = (lo + hi) / 2;
▶        if (tails[mid] < x) lo = mid + 1;
▶        else hi = mid;
10     }
▶      tails[lo] = x;    // place / extend
▶      if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 9 is larger than every end: binary search gives lo = len = 3 → tails[3] = 9, the list grows (len = 4).

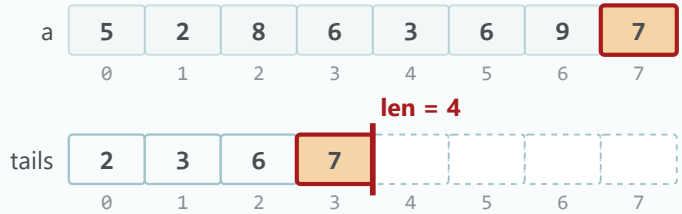
Step 8 / 9

VARIABLES

`x = 9` `lo = 3` `len = 4`

lisFast({5, 2, 8, 6, 3, 6, 9, 7})

```
1  static int lisFast(int[] a) {
2    int[] tails = new int[a.length];
3    int len = 0;       // number of lists
4    for (int x : a) {
       int lo = 0, hi = len;
       while (lo < hi) {
         int mid = (lo + hi) / 2;
         if (tails[mid] < x) lo = mid + 1;
         else hi = mid;
10     }
       tails[lo] = x;     // place / extend
       if (lo == len) len++;
13   }
14   return len;
15 }
```



x = 7: the first position with tails[lo] ≥ x is lo = 3 (2 comparisons) → 7 replaces 9; len is unchanged, but the length-4 sequence now ends with a smaller number. Result **4**; tails = [2, 3, 6, 7] need not be an LIS.

Step 9 / 9

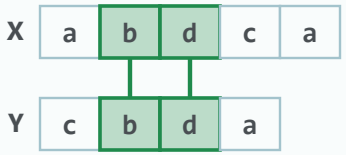
VARIABLES

`x = 7` `lo = 3` `len = 4`

Longest Common Substring

- The longest run of **contiguous** characters shared by two strings.
- Example: `"abdca"` and `"cbda"` → common substrings "a", "b", "d", "c", **"bd"** → answer **"bd"**, length 2.
- **State:** L[i][j] = length of the longest common piece **ending exactly at** position i of Y and position j of X.

if Y[i−1] = X[j−1]: L[i][j] = L[i−1][j−1] + 1
otherwise: L[i][j] = **0** (contiguity breaks, the counter resets)
Answer = the **largest** value in the table (not the last cell!)



"bd": contiguous length 2

Brute force: search for each of the O(m^2) substrings of X in Y → O($m^2 \cdot n$). DP: O($m \cdot n$) time.

Common Substring: Filling the Table

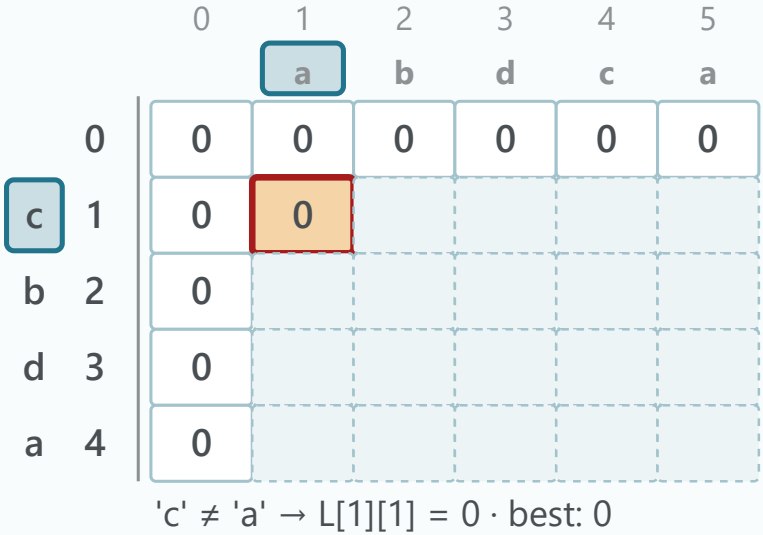
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0					
b	2	0					
d	3	0					
a	4	0					

Row 0 and column 0 are zero: nothing is shared with the empty string.

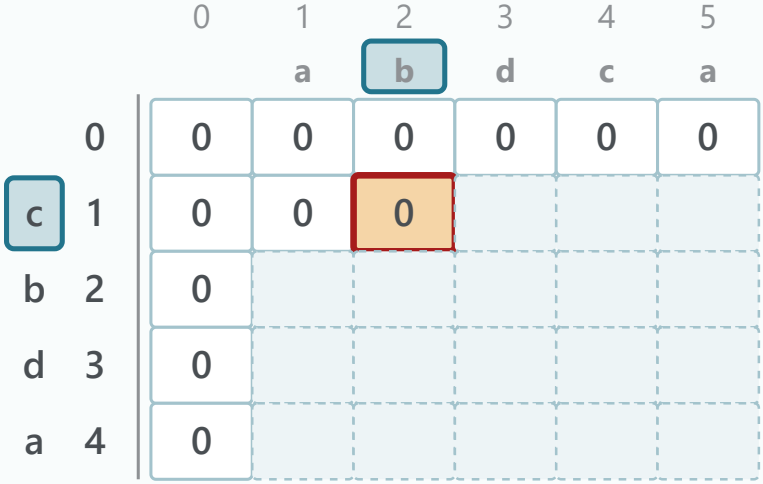
Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



Common Substring: Filling the Table

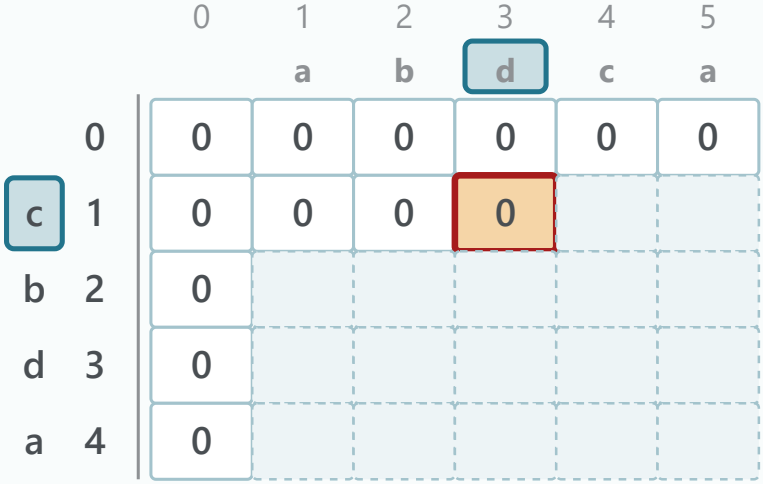
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'c' ≠ 'b' → L[1][2] = 0 · best: 0

Common Substring: Filling the Table

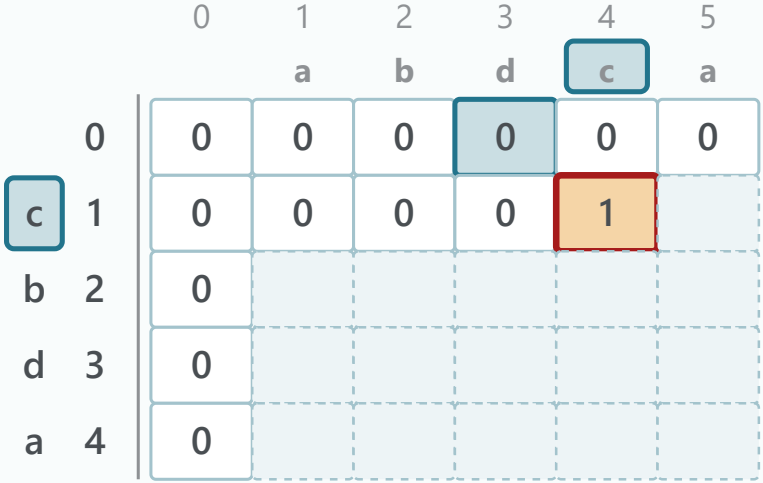
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'c' ≠ 'd' → L[1][3] = 0 · best: 0

Common Substring: Filling the Table

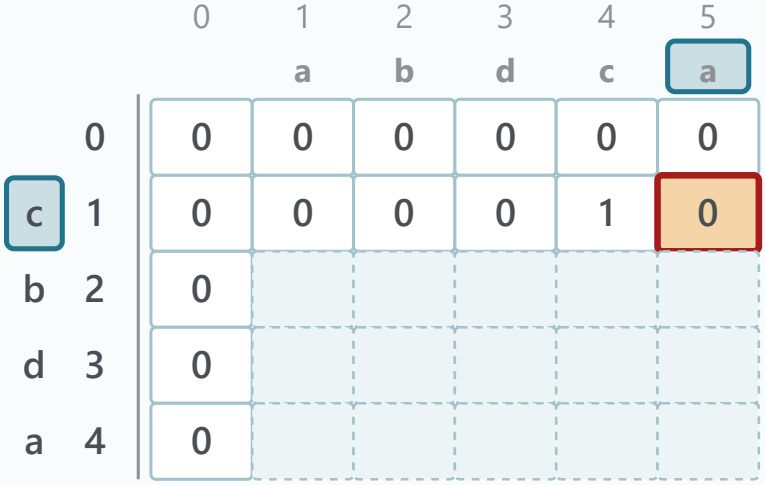
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



Y[0]='c' = X[3]='c' → L[1][4] = L[0][3] + 1 = **1** · best: 1

Common Substring: Filling the Table

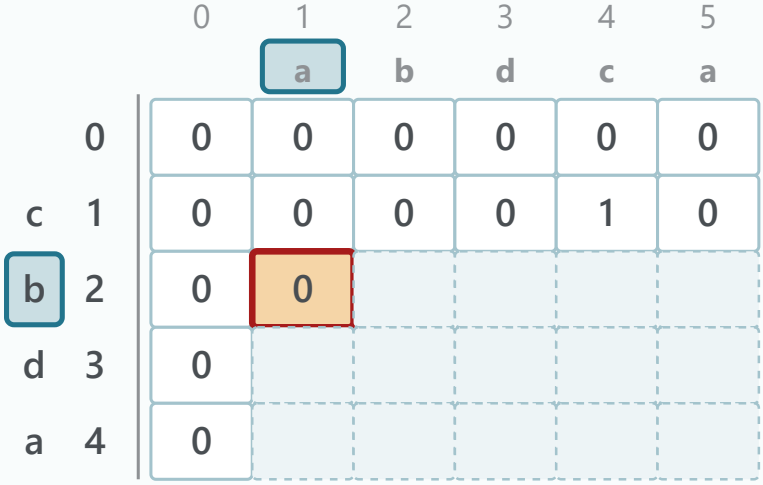
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'c' ≠ 'a' → L[1][5] = 0 · best: 1

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'b' ≠ 'a' → L[2][1] = 0 · best: 1

Common Substring: Filling the Table

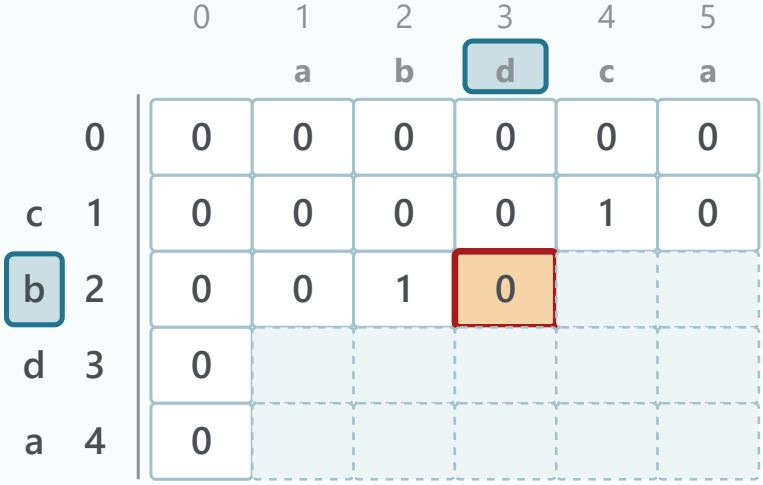
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1			
d	3	0					
a	4	0					

Y[1]='b' = X[1]='b' → L[2][2] = L[1][1] + 1 = **1** · best: 1

Common Substring: Filling the Table

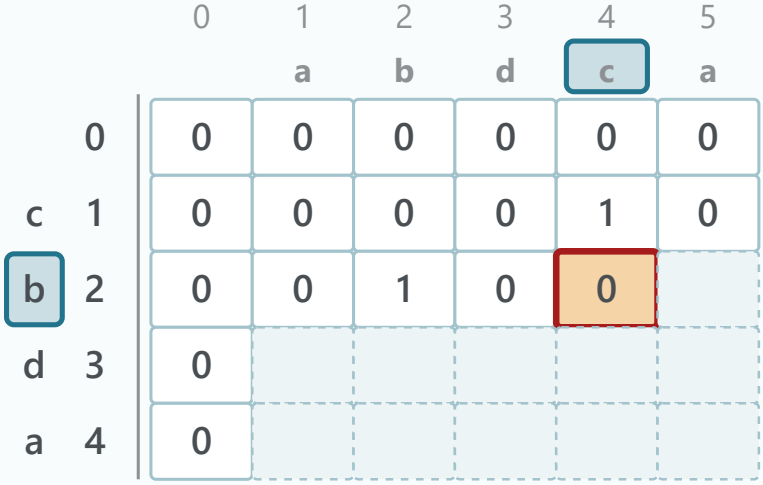
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'b' ≠ 'd' → L[2][3] = 0 · best: 1

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'b' ≠ 'c' → L[2][4] = 0 · best: 1

Common Substring: Filling the Table

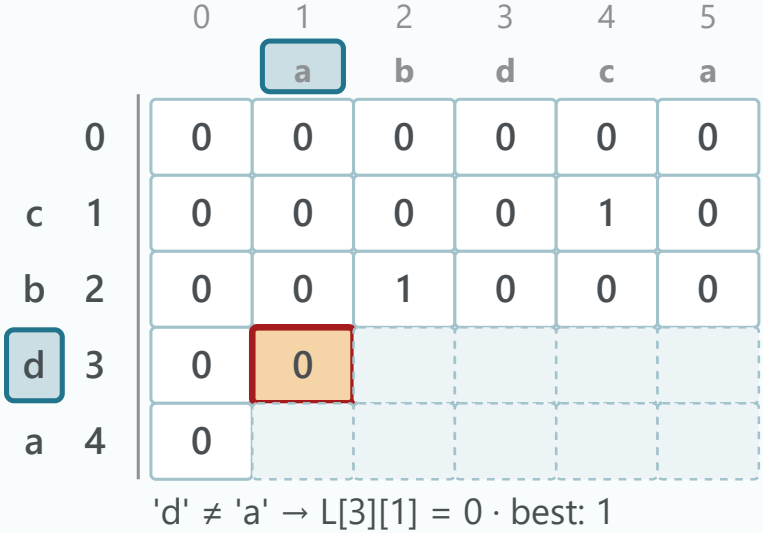
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0					
a	4	0					

'b' ≠ 'a' → L[2][5] = 0 · best: 1

Common Substring: Filling the Table

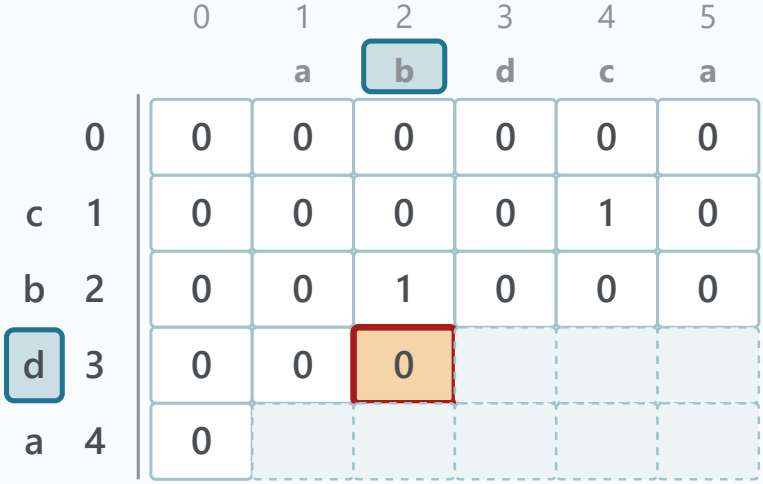
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'd' ≠ 'a' → L[3][1] = 0 · best: 1

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'd' ≠ 'b' → L[3][2] = 0 · best: 1

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2		
a	4	0					

Y[2]='d' = X[2]='d' → L[3][3] = L[2][2] + 1 = **2** · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	
a	4	0					

'd' ≠ 'c' → L[3][4] = 0 · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0					

'd' ≠ 'a' → L[3][5] = 0 · best: 2

Common Substring: Filling the Table

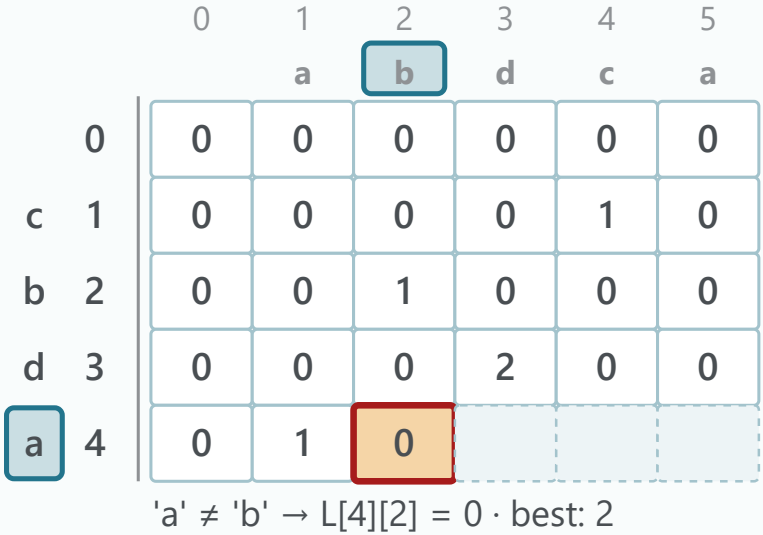
Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1				

Y[3]='a' = X[0]='a' → L[4][1] = L[3][0] + 1 = **1** · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.



'a' ≠ 'b' → L[4][2] = 0 · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0		

'a' ≠ 'd' → L[4][3] = 0 · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0	0	

'a' ≠ 'c' → L[4][4] = 0 · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0	0	1

Y[3]='a' = X[4]='a' → L[4][5] = L[3][4] + 1 = **1** · best: 2

Common Substring: Filling the Table

Columns X = "abdca", rows Y = "cbda". Orange: cell being computed · blue: the diagonal cell used.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0	0	1

The largest value is L[3][3] = **2**; 2 steps back along the diagonal: X[1..2] = **"bd"**. The last cell, L[4][5] = 1, is not the answer.

Longest Common Subsequence

- The longest subsequence common to two sequences, chosen in order (with skips allowed).
- Example: `"abdca"` and `"cbda"` → **"bda"**, length 3 (the common substring was only 2). The LCS need not be unique, but its **length** is: AGGTAB · GXTXAYB → GTAB (4); GTAB is a subsequence but not a substring.
- Uses: `diff` / git comparison, similarity of DNA sequences, plagiarism detection.

if Y[i−1] = X[j−1]: L[i][j] = L[i−1][j−1] + 1 (take the common character)
otherwise: L[i][j] = max(L[i−1][j] , L[i][j−1]) (skip one)
Answer = **L[n][m]** (the last cell)

	Match	Mismatch	Answer
Common substring	diagonal + 1	**0**	maximum of the table
Common subsequence (LCS)	diagonal + 1	**max(up, left)**	last cell

LCS: Code Trace

```
1  static int lcs(String x, String y) {
▶    int m = x.length(), n = y.length();
▶    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0					
b	2	0					
d	3	0					
a	4	0					

Array L is (n+1) × (m+1) = 5 × 6; in Java every cell starts at 0.

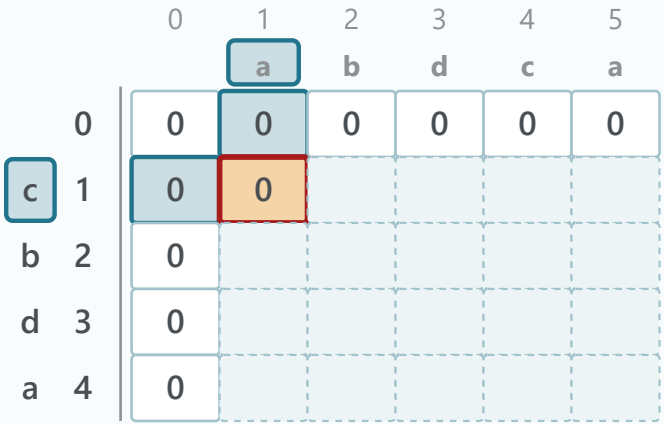
Step 1 / 22

VARIABLES

`x = "abdca"` `y = "cbda"` `i = –` `j = –`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```



'c' ≠ 'a' → max(up 0, left 0) = **0**

Step 2 / 22

VARIABLES

`i = 1` `j = 1` `y[i-1] = 'c'` `x[j-1] = 'a'` `L[i][j] = 0`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0			
b	2	0					
d	3	0					
a	4	0					

'c' ≠ 'b' → max(up 0, left 0) = **0**

Step 3 / 22

VARIABLES

`i = 1` `j = 2` `y[i-1] = 'c'` `x[j-1] = 'b'` `L[i][j] = 0`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0		
b	2	0					
d	3	0					
a	4	0					

'c' ≠ 'd' → max(up 0, left 0) = **0**

Step 4 / 22

VARIABLES

`i = 1` `j = 3` `y[i-1] = 'c'` `x[j-1] = 'd'` `L[i][j] = 0`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
▶        if (y.charAt(i - 1) == x.charAt(j - 1))
▶          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	
b	2	0					
d	3	0					
a	4	0					

'c' = 'c' → diagonal + 1: L[1][4] = 0 + 1 = **1**

Step 5 / 22

VARIABLES

`i = 1` `j = 4` `y[i-1] = 'c'` `x[j-1] = 'c'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0					
d	3	0					
a	4	0					

'c' ≠ 'a' → max(up 0, left 1) = **1**

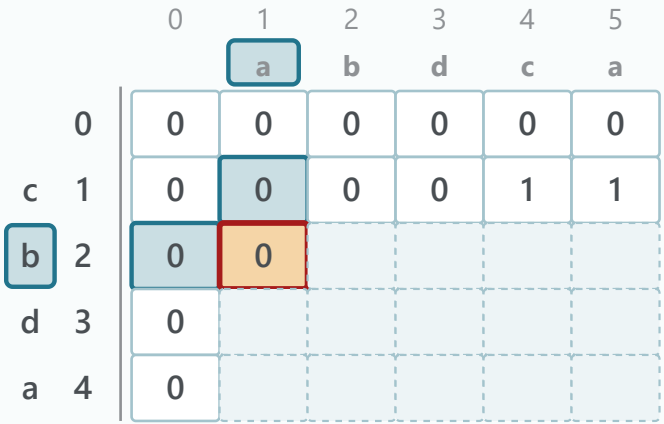
Step 6 / 22

VARIABLES

`i = 1` `j = 5` `y[i-1] = 'c'` `x[j-1] = 'a'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```



'b' ≠ 'a' → max(up 0, left 0) = **0**

Step 7 / 22

VARIABLES

`i = 2` `j = 1` `y[i-1] = 'b'` `x[j-1] = 'a'` `L[i][j] = 0`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
▶        if (y.charAt(i - 1) == x.charAt(j - 1))
▶          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1			
d	3	0					
a	4	0					

'b' = 'b' → diagonal + 1: L[2][2] = 0 + 1 = **1**

Step 8 / 22

VARIABLES

`i = 2` `j = 2` `y[i-1] = 'b'` `x[j-1] = 'b'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1		
d	3	0					
a	4	0					

'b' ≠ 'd' → max(up 0, left 1) = **1**

Step 9 / 22

VARIABLES

`i = 2` `j = 3` `y[i-1] = 'b'` `x[j-1] = 'd'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	
d	3	0					
a	4	0					

'b' ≠ 'c' → max(up 1, left 1) = **1**

Step 10 / 22

VARIABLES

`i = 2` `j = 4` `y[i-1] = 'b'` `x[j-1] = 'c'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0					
a	4	0					

'b' ≠ 'a' → max(up 1, left 1) = **1**

Step 11 / 22

VARIABLES

`i = 2` `j = 5` `y[i-1] = 'b'` `x[j-1] = 'a'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0				
a	4	0					

'd' ≠ 'a' → max(up 0, left 0) = **0**

Step 12 / 22

VARIABLES

`i = 3` `j = 1` `y[i-1] = 'd'` `x[j-1] = 'a'` `L[i][j] = 0`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1			
a	4	0					

'd' ≠ 'b' → max(up 1, left 0) = **1**

Step 13 / 22

VARIABLES

`i = 3` `j = 2` `y[i-1] = 'd'` `x[j-1] = 'b'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
▶        if (y.charAt(i - 1) == x.charAt(j - 1))
▶          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2		
a	4	0					

'd' = 'd' → diagonal + 1: L[3][3] = 1 + 1 = **2**

Step 14 / 22

VARIABLES

`i = 3` `j = 3` `y[i-1] = 'd'` `x[j-1] = 'd'` `L[i][j] = 2`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	
a	4	0					

'd' ≠ 'c' → max(up 1, left 2) = **2**

Step 15 / 22

VARIABLES

`i = 3` `j = 4` `y[i-1] = 'd'` `x[j-1] = 'c'` `L[i][j] = 2`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0					

'd' ≠ 'a' → max(up 1, left 2) = **2**

Step 16 / 22

VARIABLES

`i = 3` `j = 5` `y[i-1] = 'd'` `x[j-1] = 'a'` `L[i][j] = 2`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
▶        if (y.charAt(i - 1) == x.charAt(j - 1))
▶          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1				

'a' = 'a' → diagonal + 1: L[4][1] = 0 + 1 = **1**

Step 17 / 22

VARIABLES

`i = 4` `j = 1` `y[i-1] = 'a'` `x[j-1] = 'a'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1			

'a' ≠ 'b' → max(up 1, left 1) = **1**

Step 18 / 22

VARIABLES

`i = 4` `j = 2` `y[i-1] = 'a'` `x[j-1] = 'b'` `L[i][j] = 1`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2		

'a' ≠ 'd' → max(up 2, left 1) = **2**

Step 19 / 22

VARIABLES

`i = 4` `j = 3` `y[i-1] = 'a'` `x[j-1] = 'd'` `L[i][j] = 2`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
▶          L[i][j] = Math.max(L[i - 1][j],
▶                             L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	

'a' ≠ 'c' → max(up 2, left 2) = **2**

Step 20 / 22

VARIABLES

`i = 4` `j = 4` `y[i-1] = 'a'` `x[j-1] = 'c'` `L[i][j] = 2`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
▶        if (y.charAt(i - 1) == x.charAt(j - 1))
▶          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
11   return L[n][m];
12 }
```

		0	1 a	2 b	3 d	4 c	5 a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

'a' = 'a' → diagonal + 1: L[4][5] = 2 + 1 = **3**

Step 21 / 22

VARIABLES

`i = 4` `j = 5` `y[i-1] = 'a'` `x[j-1] = 'a'` `L[i][j] = 3`

LCS: Code Trace

```
1  static int lcs(String x, String y) {
2    int m = x.length(), n = y.length();
3    int[][] L = new int[n + 1][m + 1];
4    for (int i = 1; i <= n; i++)
5      for (int j = 1; j <= m; j++)
6        if (y.charAt(i - 1) == x.charAt(j - 1))
7          L[i][j] = L[i - 1][j - 1] + 1;
8        else
9          L[i][j] = Math.max(L[i - 1][j],
10                            L[i][j - 1]);
▶    return L[n][m];
12 }
```

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	**3**

The table is full: `L[4][5]` = **3**. 20 cells, each O(1) → O(m · n).

Step 22 / 22

VARIABLES

`i = 5` `j = –` `result = 3`

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

Reconstruction starts at the last cell, L[4][5] = 3; the characters found are collected from back to front.

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

'a' = 'a' → the character joins the LCS ("a"); move diagonally.

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

'd' ≠ 'c'; left (2) > up (1) → move left.

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

'd' = 'd' → the character joins the LCS ("da"); move diagonally.

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

'b' = 'b' → the character joins the LCS ("bda"); move diagonally.

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

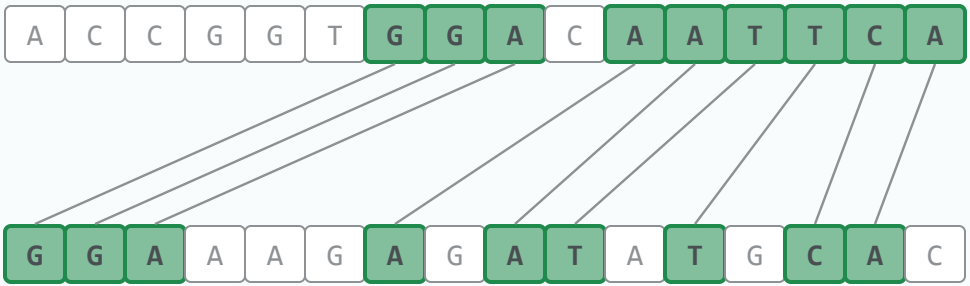
'c' ≠ 'a'; up (0) ≥ left (0) → move up.

LCS: Reconstructing the Solution

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

The border is reached. Green cells are matches: LCS = **"bda"**, length 3. Reconstruction takes O(m + n).

Example: LCS of DNA Sequences



- The lines joining matched characters **do not cross**: the relative order is preserved. Every matching that obeys this rule is a common subsequence.
- DP result: LCS = **GGAAATTCA**, length **9**.

Matching by eye is misleading. For instance, AAAATTCA (8 characters) is also a valid common subsequence, but not the **longest**; the DP table gives the right answer.

Try It: LCS and Common Substring Tables

X abdca Y cbda LCS common substring

		a	b	d	c	a
	0	0	0	0	0	0
c	0	0	0	0	1	1
b	0	0	1	1	1	1
d	0	0	1	2	2	2
a	0	1	1	2	2	3

LCS length **3**: "bda" (last cell).

X has at most 9 characters, Y at most 7. Green cells show the reconstruction path.

Other Classic DP Problems

Problem	State and recurrence	Time
Edit distance	$D[i][j] = \min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + [a_i \neq b_j])$; $D[i][0] = i$, $D[0][j] = j$	$O(m \cdot n)$
Coin change	fewest coins: $P[x] = 1 + \min P[x - c]$ $(c \leq x)$; $P[0] = 0$	$O(k \cdot X)$
Maximum subarray	$V(i) = \max(a[i], V(i-1) + a[i])$ — Kadane; answer $\max V(i)$	$O(n)$
Matrix-chain multiplication	$M[i][j] = \min \{ M[i][k] + M[k+1][j] + p_{i-1} \cdot p_k \cdot p_j : i \leq k < j \}$	$O(n^3)$
Traveling salesman (Held-Karp)	$dp[S][i]$ = shortest path visiting set S and ending at i; $dp[S][i] = \min dp[S - i][j] + d(j, i)$	$O(n^2 \cdot 2^n)$
Bellman-Ford	$d_k(v)$ = shortest path with at most k edges; $d_k(v) = \min(d_{k-1}(v), d_{k-1}(u) + w(u, v))$	$O(V \cdot E)$
Optimal binary search tree	$e[i][j] = \min \{ e[i][r-1] + e[r+1][j] : i \leq r \leq j \} + w(i, j)$	$O(n^3)$

Summary

Problem	State / recurrence	Time · memory
Fibonacci	$F(n) = F(n-1) + F(n-2)$; memoization or a table	$O(n) \cdot O(1)$
0/1 knapsack	$K[i][c] = \max(K[i-1][c], v_i + K[i-1][c-w_i])$	$O(nW) \cdot O(W)$
Unbounded knapsack	$dp[i][c] = \max(dp[i-1][c], p_i + dp[\mathbf{i}][c-w_i])$	$O(nW) \cdot O(W)$
Subset sum	$dp[i][j] = dp[i-1][j] \vee dp[i-1][j-a_i]$	$O(n \cdot \text{target})$
LIS	$dp[i] = 1 + \max dp[j]$ ($j < i$, $a[j] < a[i]$); or active lists	$O(n^2)$ / $O(n \log n)$
Common substring	if equal, diagonal + 1, else 0; answer is the table maximum	$O(mn)$
LCS	if equal, diagonal + 1, else max(up, left); answer is the last cell	$O(mn)$

DP = **overlapping subproblems** + **optimal substructure**. Time ≈ number of states × work per state. The solution itself is found by **reading the table backward**.

SUMMARY

Common Mistakes

Forgetting the base cases

If K[0][c], dp[i][0], or fib(0)/fib(1) is initialized wrongly, the whole table shifts.

Off-by-one indices

The table has size n+1 but the array has n: row i uses `v[i - 1]`. Writing `v[i]` overflows on the last item.

Wrong loop direction

In the one-dimensional 0/1 knapsack, scanning capacity from low to high takes an item more than once (it becomes unbounded).

Substring ≠ subsequence

For the common substring, a mismatch writes 0 and the answer is the table maximum; for LCS, write max(up, left) and the answer is the last cell.

Assuming greedy works

Greedy is not optimal for the 0/1 knapsack or for making change (6 with {1, 3, 4}).

Mistaking O(nW) for polynomial

W is a numeric value; the input size is log W bits. For large W, the table cannot be built.

SUMMARY

Sample Problems

1. 0/1 knapsack: W = 5; items (v, w) = (3, 2), (4, 3), (5, 4), (6, 5). What is the largest value?
2. What is the length of LCS("ABCBDAB", "BDCABA")? Write one LCS.
3. Coins {1, 3, 4}, amount 6: how few coins suffice? How many does greedy use?
4. What are the LIS length and one LIS of [10, 22, 9, 33, 21, 50, 41, 60]?
5. What are the longest common substring (contiguous) and LCS lengths for "ABAB" and "BABA"?
6. How many calls does plain recursive fib(6) make? How many with memoization?

▶ **Show answers**

SUMMARY

Exercises

1. Write the 0/1 knapsack with **memoization** (top-down).
2. In **how many ways** can the amount 5 be paid with coins {1, 2, 5}?
3. Find the longest common substring of "ababc" and "babca".
4. Does {3, 34, 4, 12, 5, 2} have a subset summing to 9? To 30?
5. What is the maximum subarray of [−2, 1, −3, 4, −1, 2, 1, −5, 4] (Kadane)?
6. Weights {2, 3, …, 8}, values {3, 5, 8, 6, 10, 11, 15}, W = 10: what are the 0/1 and unbounded results?

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 14: Dynamic Programming).
- Kleinberg and Tardos, *Algorithm Design*, Pearson, 2005 (Chapter 6).
- Dasgupta, Papadimitriou, and Vazirani, *Algorithms*, McGraw-Hill, 2006 (Chapter 6).
- R. Bellman, *Dynamic Programming*, Princeton University Press, 1957.

The End

Chapter 3: Dynamic Programming

Previous chapter: Chapter 2: Greedy Algorithms · Next chapter: Chapter 4: Backtracking