

Dynamic programming II

Chapter 3.5 · Overall level: Upper intermediate

	ε	a	b	c	d	e
ε	0	0	0	0	0	0
a	0	1	1	1	1	1
c	0	1	1	2	2	2
e	0	1	1	2	2	3

LCS = 3

A backpack holds 5 kilograms. Three pieces of equipment weigh 2, 3 and 4 kilograms and are worth 4, 5 and 7 points. *Which ones should you pack?* Each piece exists once, so you cannot pack the best one twice.

Chapter 3.4 built tables indexed by one number. This chapter's states need more: an item prefix *and* a capacity, two string prefixes, or a whole set of used jobs written as the bits of an integer. The four design questions (state, transition, base, order) stay the same.

3.5.1 Items used at most once: the 0/1 knapsack

Number the items 1 to n ; item k has weight $w[k]$ and value $v[k]$. A one-number state such as “best value with capacity c ” is not enough, because it does not record which items were already used. So the state remembers how far through the list of items we have decided:

State. $dp[i][c]$ is the largest total value of a selection that uses only items 1 to i , each at most once, with total weight at most c .

Transition: cases on item i . A best selection for (i, c) either leaves item i out, and is then a selection from items 1 to $i - 1$ with the same capacity; or it packs item i once, and the rest is a selection from items 1 to $i - 1$ that fits into $c - w[i]$:

$$dp[i][c] = \max(dp[i - 1][c], \quad dp[i - 1][c - w[i]] + v[i])$$

(second case only if $w[i] \leq c$)

Base. With no items, every capacity gives value 0: $dp[0][c] = 0$. **Order.** Row i reads only row $i - 1$, so fill the rows for $i = 1, 2, \dots, n$. The answer is $dp[n][W]$ for capacity W .

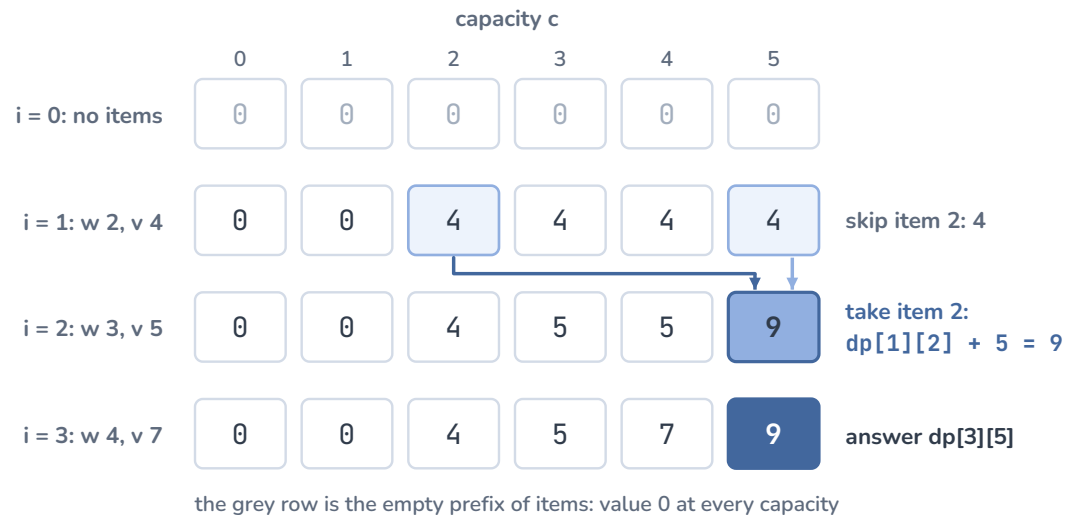


Figure 3.5.1 — The full table for the backpack. Each entry looks one row up: straight up to skip item i , or up and $w[i]$ columns left to take it. The best pack is items 1 and 2, value 9.

Why it is correct. The two cases cover every selection, because any selection either contains item i or does not. In the first case its value is at most $dp[i - 1][c]$ by the meaning of that entry. In the second case, removing item i leaves a selection of earlier items with weight at most $c - w[i]$, worth at most $dp[i - 1][c - w[i]]$. Both bounds can actually be reached, so their maximum is the optimum. Notice how the state's phrase “only items 1 to i ” is exactly what stops item i from being counted twice: the second case looks up a row in which item i is not available.

3.5.2 One array, capacities downward

The table has $(n + 1)(W + 1)$ entries; with 100 items and capacity 100,000 that is about ten million numbers. Since each row reads only the row above it, one array `best[0..W]` can hold “the current row” and be overwritten item by item. The question is the order in which to overwrite it.

Updating `best[c]` needs the *old* value at $c - w$, the one from before this item. The figure shows what happens with a single item of weight 2 and value 4 and capacity 4.

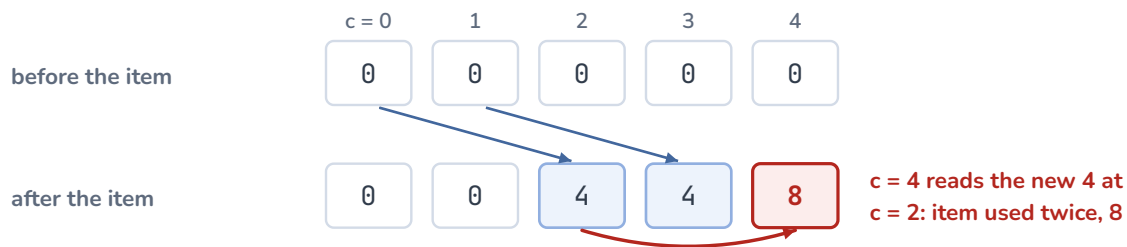
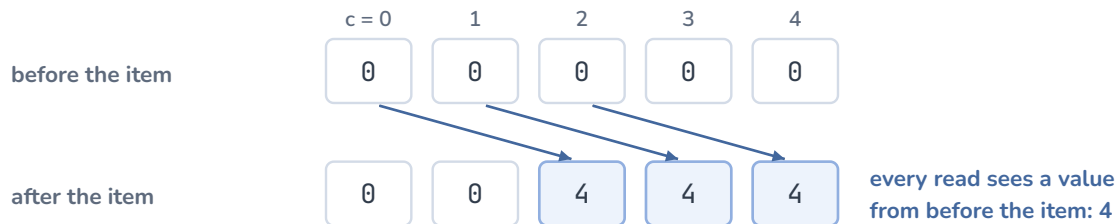
Upward: $c = 2, 3, 4$ (wrong)**Downward: $c = 4, 3, 2$ (right)**

Figure 3.5.2 — Descending capacities read states from before this item, enforcing at most one use. Arrows show which entry each update reads; the red arrow reads an entry that already contains the item.

Why downward is right: an invariant. During the pass for item i , going from $c = W$ down to $w[i]$, every entry below the current c has not been touched yet in this pass, so it still holds row $i - 1$. In particular $\text{best}[c - w[i]]$ (a smaller index) is the old value, exactly as the recurrence demands. Upward order breaks this: the smaller index has already been updated, may already contain item i , and the item gets packed again.

Upward order is not useless. It is exactly the recurrence for **unlimited** copies, like the coins of Chapter 3.4, where using an item again is allowed. The direction of one loop is the whole difference between the two problems, so choose it on purpose.

Pseudocode

0/1 knapsack: $O(nW)$ time, $O(W)$ memory

```

read n, W
best[0..W] ← 0
for each item (w, v):
    for c from W down to w:
        best[c] ← max(best[c], best[c - w] + v)
output best[W]
```

The capacity is “at most c ”, not “exactly c ”, so every entry starts at 0 (choosing nothing always fits) and the answer is simply $\text{best}[W]$. No sentinel is needed.

3.5.3 Code in four languages

Each program solves the first practice problem, **One-use equipment**. The first line holds n and W , then n lines hold a weight and a value. The output is the largest total value within weight W .

Sample input	Sample output
3 5 2 4 3 5 4 7	9

C++, Java and JavaScript follow the pseudocode. Python, where loops are slow, first copies the old values `best[0..W - w]` and builds all new entries from that copy at once; since the copy predates the item, every read is an old value, just as with the downward loop.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n, cap = int(data[0]), int(data[1])
    best = [0] * (cap + 1)          # best[c]: max value with weight <= c
    for k in range(n):
        w, v = int(data[2 + 2 * k]), int(data[3 + 2 * k])
        # old[c - w] for c = w..cap, copied before anything changes
        old = best[:cap + 1 - w]
        best[w:] = [max(keep, prev + v) for keep, prev in zip(best[w:], old)]
    print(best[cap])

main()
```

C++17

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, cap;
    cin >> n >> cap;
    vector<long long> best(cap + 1, 0);    // best[c]: max value with weight <= c
    for (int k = 0; k < n; k++) {
        int w;
        long long v;
        cin >> w >> v;
        for (int c = cap; c >= w; c--)    // downward: each item used at most once
            best[c] = max(best[c], best[c - w] + v);
    }
    cout << best[cap] << '\n';
    return 0;
}
```

Java 17

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken();
        int n = (int) in.nval;
        in.nextToken();
        int cap = (int) in.nval;
        long[] best = new long[cap + 1]; // best[c]: max value with weight <= c
        for (int k = 0; k < n; k++) {
            in.nextToken(); int w = (int) in.nval;
            in.nextToken(); long v = (long) in.nval;
            for (int c = cap; c >= w; c--) // downward: each item used at most once
                best[c] = Math.max(best[c], best[c - w] + v);
        }
        System.out.println(best[cap]);
    }
}
```

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8")
    .trim().split(/\s+/).map(Number);
let pos = 0;
const n = data[pos++], cap = data[pos++];
const best = new Float64Array(cap + 1); // best[c]: max value with weight <= c
for (let k = 0; k < n; k++) {
    const w = data[pos++], v = data[pos++];
    for (let c = cap; c >= w; c--) { // downward: each item used at most once
        if (best[c - w] + v > best[c]) best[c] = best[c - w] + v;
    }
}
console.log(best[cap]);
```

Cost. Each item updates at most $W + 1$ entries, so the time is $O(nW)$, here at most $100 \times 100,001 \approx 10^7$ updates, and the memory is one array of $W + 1$ numbers. The largest possible answer is $100 \times 10^6 = 10^8$, which fits in 32 bits; the programs use 64-bit values anyway so that larger value limits need no change. In JavaScript every value is an exact Number far below 2^{53} .

$O(nW)$ looks polynomial, but it grows with the *numeric value* W , not with the length of the input. A capacity of 10^9 would need a billion entries. Such “pseudo-polynomial” tables work only when the capacity itself is modest.

3.5.4 Two strings: subsequences, not substrings

A **substring** is a consecutive block of characters: “bcd” is a substring of “abcde”. A **subsequence** keeps the order but may skip characters: “ace” is a subsequence of “abcde” (take a, skip b, take c, skip d, take e), but not a substring. The longest common subsequence (LCS) of two strings is the longest string that is a subsequence of both. For “abcde” and “ace” it is “ace”, length 3.

Here the state must describe a pair of prefixes, one from each string:

State. $dp[i][j]$ is the length of a longest common subsequence of the first i characters of s and the first j characters of t .

Transition: cases on the last characters $s[i]$ and $t[j]$.

```
if  $s[i] = t[j]$ :     $dp[i][j] = dp[i - 1][j - 1] + 1$ 
otherwise:         $dp[i][j] = \max(dp[i - 1][j], dp[i][j - 1])$ 
```

Base. If either prefix is empty, the only common subsequence is empty: $dp[0][j] = dp[i][0] = 0$. These are the extra row and column labelled ϵ (the empty string) in the figure. **Order.** Row by row, left to right: up, left and up-left are then all known.

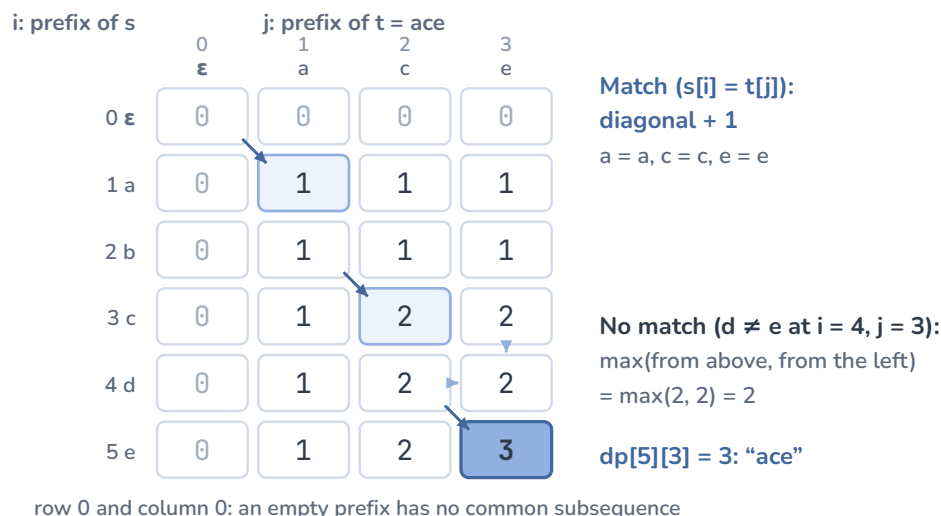


Figure 3.5.3 — The LCS table for abcde and ace. Index i counts characters of s (rows) and j counts characters of t (columns); row 0 and column 0 are the empty prefixes. Diagonal arrows mark matches; the other entries copy the larger of the entry above and the entry to the left.

Why it is correct. Take any common subsequence of the two prefixes. Its last character sits at some position of s at most i and some position of t at most j ; everything before it is a common subsequence of $s[1..i - 1]$ and $t[1..j - 1]$. So $dp[i][j] \leq dp[i - 1][j - 1] + 1$ always, and when $s[i] = t[j]$ this is reached by appending the matching letter. When the letters differ, no common subsequence can use both $s[i]$ and $t[j]$ as its final pair, so at least one of them is unused, and the subsequence lies in $(i - 1, j)$ or in $(i, j - 1)$. The maximum of those two entries is therefore exact.

The same table structure with a different rule computes the longest common *substring*: there a mismatch resets the entry to 0, because a block must be consecutive, and the answer is the largest entry anywhere. Mixing the two rules is a classic mistake.

3.5.5 Edit distance: the cheapest last edit

The edit distance from s to t is the fewest single-character insertions, deletions and substitutions that turn s into t . For example, “kitten” becomes “sitting” in three: substitute $k \rightarrow s$, substitute $e \rightarrow i$ and insert g .

State. $dp[i][j]$ is the fewest operations that turn the first i characters of s into the first j characters of t .

Transition: what happens to the last characters. In a cheapest edit, either $s[i]$ is deleted (then the first $i - 1$ characters must become $t[1..j]$), or $t[j]$ is inserted at the end (then $s[1..i]$ must become $t[1..j - 1]$), or $s[i]$ ends up as $t[j]$, for free if they are equal and by one substitution if not:

$$dp[i][j] = \min \left(\begin{array}{ll} dp[i - 1][j] + 1, & \text{(delete } s[i]) \\ dp[i][j - 1] + 1, & \text{(insert } t[j]) \\ dp[i - 1][j - 1] + \text{diff} \end{array} \right) \quad \text{(keep or substitute)}$$

where $\text{diff} = 0$ if $s[i] = t[j]$, and 1 otherwise

Base. Turning i characters into the empty string takes i deletions, and building j characters from nothing takes j insertions: $dp[i][0] = i$ and $dp[0][j] = j$. Unlike LCS, the boundary row and column are not zeros.

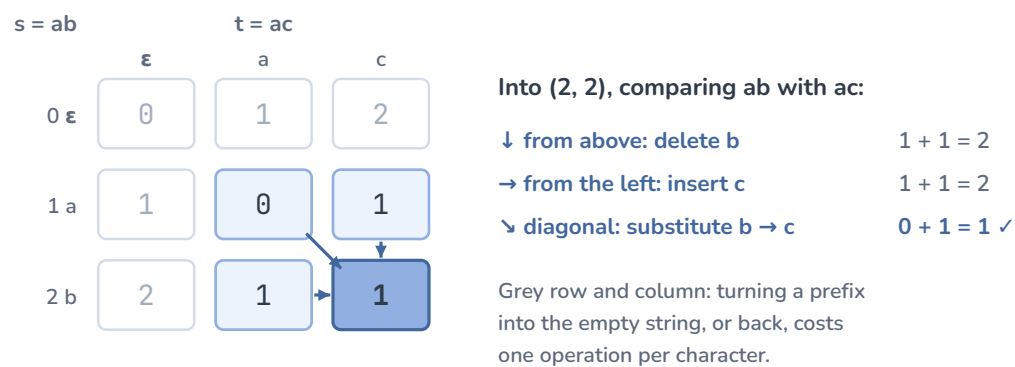


Figure 3.5.4 — Each transition describes the final edit applied to two prefixes. For ab and ac, keeping a and substituting the final $b \rightarrow c$ costs 1, cheaper than deleting b or inserting c.

Why it is correct. Write any edit script as an alignment: each character of s is deleted or paired with a character of t , and unpaired characters of t are inserted, with pairs in order. Look at the last column of the alignment: it is “ $s[i]$ deleted”, “ $t[j]$ inserted” or “ $s[i]$ paired with $t[j]$ ”. Removing that column leaves an alignment of the matching smaller prefixes, which costs at least the corresponding table entry. So every script costs at least one of the three candidates, and each candidate is achievable: the minimum is exact.

Both string tables take $O(|s| \cdot |t|)$ time, 4×10^6 entries for two strings of length 2000. Each row needs only the row above, so two rows of length $|t| + 1$ suffice. With a single row, remember to save the old up-left value before overwriting it; reading a freshly written entry by accident is the same bug as the upward knapsack.

3.5.6 A set as the bits of an integer

The last kind of state is a **set**. If a problem has at most about 20 objects, a set of them can be stored as one integer, called a **bitmask**: bit j (counting from 0 at the right) is 1 exactly when object $j + 1$ is in the set. With jobs 1, 2, 3, the mask written in binary as 011 (decimal 3) has bits 0 and 1 set, so it means {job 1, job 2}. Each of the 2^n subsets is one integer from 0 to $2^n - 1$.

Task	Expression	Example with $n = 3$, mask = 011 (3)
Is job $j + 1$ in the set?	<code>(mask >> j) & 1</code>	$j = 1$: $(3 >> 1) \& 1 = 1$, so job 2 is in
Add job $j + 1$	<code>mask (1 << j)</code>	$j = 2$: $011 100 = 111$ (7)
Number of jobs in the set	<code>popcount(mask)</code>	<code>popcount(011) = 2</code>
The set of all n jobs	<code>(1 << n) - 1</code>	$(1 << 3) - 1 = 7 = 111$
Visit every subset	for mask from 0 to <code>(1 << n) - 1</code>	0, 1, ..., 7: eight subsets

Popcount (the number of 1 bits) is built in almost everywhere: `mask.bit_count()` in Python 3.10+, `__builtin_popcount(mask)` in C++, `Integer.bitCount(mask)` in Java. It can also be tabulated in one pass, `cnt[mask] = cnt[mask >> 1] + (mask & 1)`, because dropping the last bit gives a smaller mask that is already done. Keep masks small: in JavaScript the bitwise operators work on 32-bit integers, which is plenty for $n \leq 30$.

3.5.7 Assignment DP over used jobs

Now n workers must each get exactly one of n jobs, all different, and worker i doing job j costs `cost[i][j]`. Trying all $n!$ assignments is hopeless for $n = 18$ ($18! \approx 6.4 \times 10^{15}$). The trick is to give out jobs to the workers *in order*: worker 1 first, then worker 2, and so on. Then a partial assignment is described completely by the set of jobs already taken, because the number of taken jobs says how many workers are done.

State. $dp[mask]$ is the smallest total cost of giving the jobs in $mask$ to workers 1, 2, ..., k , where $k = \text{popcount}(mask)$, each of those workers getting one of these jobs.

Transition. From $mask$, the next worker is number $k + 1$. Give that worker any job j not in the $mask$: $dp[mask \mid \text{bit } j] \leftarrow \min(\dots, dp[mask] + \text{cost}[k + 1][j])$. **Base.** $dp[0] = 0$ (nobody assigned yet); every other entry starts at ∞ . **Order.** Increasing $mask$: adding a bit always makes the integer larger, so every state is final before it is used. The answer is $dp[(1 \ll n) - 1]$.

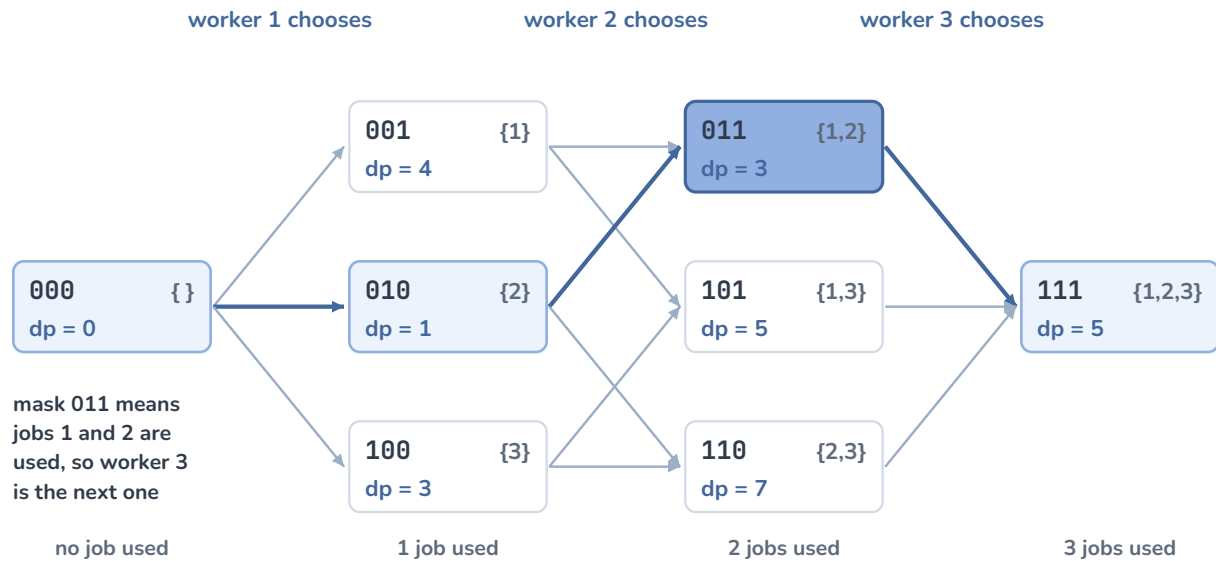


Figure 3.5.5 — The number of set bits determines which worker is assigned next. Each arrow adds one unused job; the highlighted path is an optimal assignment for the costs $[[4, 1, 3], [2, 5, 6], [7, 8, 2]]$.

With the cost matrix $[[4, 1, 3], [2, 5, 6], [7, 8, 2]]$ (row = worker, column = job), the table fills as follows. Mask 011, for example, is reached from 001 (worker 1 took job 1 for 4, then worker 2 takes job 2 for 5: total 9) or from 010 (worker 1 took job 2 for 1, then worker 2 takes job 1 for 2: total 3), so $dp[011] = 3$.

mask	000	001	010	100	011	101	110	111
jobs used	{ }	{1}	{2}	{3}	{1, 2}	{1, 3}	{2, 3}	{1, 2, 3}
next worker	1	2	2	2	3	3	3	—
$dp[mask]$	0	4	1	3	3	5	7	5

The minimum total is 5: job 2, job 1, job 3 for workers 1, 2, 3 ($1 + 2 + 2$). Here each worker happens to take their cheapest free job, but that greedy rule fails in general: for costs $[[1, 2], [1, 9]]$ worker 1 grabs job 1 for 1, leaving job 2 to worker 2 for 9 (total 10), while the opposite choice costs $2 + 1 = 3$.

Why it is correct: cases on the last assignment. In the best way to give the jobs of `mask` to workers 1 to k , worker k receives some job j in the mask. The other workers then hold exactly the jobs of the mask without j , and they must do so as cheaply as possible, at cost `dp[mask without j]`, or the whole could be improved. So `dp[mask]` is the minimum over j of `dp[mask without j] + cost[k][j]`, which is exactly what the forward updates compute. Every complete assignment corresponds to one path from 000 to 111 in the figure.

Counting time and memory first

There are 2^n masks and each tries at most n jobs, so the time is $O(n \cdot 2^n)$. For $n = 18$ that is 262,144 states and about 4.7 million transitions; with 8-byte entries the table takes 2 MB. Each extra object doubles both: $n = 25$ needs 33.5 million entries (268 MB at 8 bytes), and $n = 30$ needs over a billion. Before writing a bitmask DP, multiply out $2^n \times (\text{bytes per entry})$ and compare it with the memory you have.

3.5.8 Common mistakes

- **Comparing substring and subsequence recurrences.** A subsequence may skip characters, so a mismatch takes the larger neighbour; a substring must be consecutive, so a mismatch resets to 0. Check your rule against “abcde” and “ace”: the LCS is 3, the longest common substring only 1.
- **In-place updates reading the new row.** Overwriting a one-dimensional table in the wrong direction (upward in 0/1 knapsack) or losing the old up-left value in a one-row string table silently reads values from the current step. Trace one item or one row by hand.
- **Choosing a bitmask when n is too large.** 2^n grows by a factor of 1000 every 10 objects. Compute the memory before coding; around $n = 20$ to 23 is the usual limit.
- **Boundary rows that do not match the state.** LCS starts from zeros, edit distance from `dp[i][0] = i` and `dp[0][j] = j`. Read the state sentence for an empty prefix to get them right.
- **Wrong worker for a mask.** The worker to place next is `popcount(mask) + 1` in 1-based numbering, or `popcount(mask)` as a 0-based row index. Mixing the two assigns every job to the wrong row.

3.5.9 Four states side by side, and where they stop

Problem	State	Cases of the transition	Boundary	Time	Memory
0/1 knapsack	item prefix i , capacity c	skip or take item i	row $0 = 0$	$O(nW)$	$O(W)$
LCS	prefixes i of s , j of t	match, or drop $s[i]$ or $t[j]$	row, column $0 = 0$	$O(s t)$	$O(t)$
Edit distance	prefixes i of s , j of t	delete, insert, keep/substitute	$dp[i][0] = i$, $dp[0][j] = j$	$O(s t)$	$O(t)$
Assignment	set of used jobs	next worker takes job j	$dp[0] = 0$, others ∞	$O(n \cdot 2^n)$	$O(2^n)$

All three techniques depend on the number of states. Knapsack needs $n \times W$ work, so a capacity of 10^9 calls for a different idea, such as a table indexed by value when values are small. Two-string tables need $|s| \cdot |t|$ entries, fine for 2000×2000 but not for $10^5 \times 10^5$. Bitmask DP stops in the low twenties; larger assignment problems are solved by specialised algorithms beyond this library.

When the order of the items or workers is not fixed and the dependencies form cycles, the states no longer form a table that can be filled in order; that is a shortest-path problem (Chapter 3.3). And when values change between queries, a stored table must be maintained instead of rebuilt; Chapter 4.2 shows how for sums and minima.

Exercises

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

- Why must the capacity loop of the one-array 0/1 knapsack run downward?
- What does $dp[\text{mask}]$ mean in the assignment DP?
- Items have weights 1, 3, 4 and values 15, 20, 30; capacity 4. Show $\text{best}[0..4]$ after each item when updating downward. What is the answer?
- Fill the LCS table for $s = \text{abcb}$ and $t = \text{bcab}$. What is the LCS length, and which common subsequence achieves it?
- The edit distance from “stone” to “tones” is 2. Describe two edits that achieve it, and say which boundary entries $dp[5][0]$ and $dp[0][5]$ the table starts with.
- With $n = 4$ jobs, mask 1011 in binary. Which jobs are used, which worker is next, and which mask can follow it?

Answers and explanations

Read these after attempting the exercises.

1. Updating capacity c reads $\text{best}[c - w]$, which must still describe the items before the current one. Going downward, smaller capacities have not been updated yet in this pass; going upward, $\text{best}[c - w]$ may already contain the current item, which would then be used twice.
2. The minimum total cost of giving exactly the jobs in mask to the first $\text{popcount}(\text{mask})$ workers, one job each.
3. After item (1, 15): [0, 15, 15, 15, 15]. After (3, 20): [0, 15, 15, 20, 35]. After (4, 30): $\text{best}[4] = \max(35, 0 + 30) = 35$, so the array stays [0, 15, 15, 20, 35]. The answer is **35** (items 1 and 2).
4. Rows $i = 0..4$ (ϵ , a, b, c, b) over columns ϵ , b, c, a, b: 0 0 0 0 0 / 0 0 0 1 1 / 0 1 1 1 2 / 0 1 2 2 2 / 0 1 2 2 3. The length is **3**, for example "bcb" (b, c and the last b of each string).
5. Delete the leading s ("tone"), then insert s at the end ("tones"). The boundaries are $\text{dp}[5][0] = 5$ (five deletions) and $\text{dp}[0][5] = 5$ (five insertions).
6. Bits 0, 1 and 3 are set, so jobs **1, 2 and 4** are used. Three workers are done, so worker **4** is next; the only unused job is 3 (bit 2), so the next mask is 1111 (15).

Practice this topic

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

One-use equipment

[Apply](#)

There are n items with weight w_i and value v_i . Choose each item at most once with total weight at most W . Print the maximum total value. Choosing nothing is allowed.

Shared subsequence

[Connect](#)

Find the length of a longest sequence of characters appearing in both strings in the same order, not necessarily consecutively.

Edit the identifier

[Reason](#)

Find the minimum operations to turn s into t . Each operation inserts one character, deletes one character or substitutes one character, at cost 1.

Assign the jobs

[Stretch](#)

Assign each of n workers exactly one distinct job among n jobs. Worker i doing job j costs $c[i][j]$. Print the minimum total cost.