Module 11 of 12

DP optimisations

Make the transition cheap: prefix sums over the previous row, a monotonic deque for a sliding-window maximum, and bitmask DP over subsets of up to twenty items.

A dynamic programme costs its number of states times the cost of one transition. The states are usually fixed by the problem; the transition is where an O(n · k) solution becomes O(n) — the difference between 10^{10} steps and 10^{5}.

The lesson shows the three speed-ups that come up most. When a state sums a contiguous range of the previous row, **prefix sums** of that row read the range in O(1). When it takes the maximum over the last k states, a **monotonic deque** keeps only the candidates that can still win, so each index enters and leaves once. And when the state is a subset of at most twenty items, a **bitmask** indexes it — the travelling-salesman DP over $2^n \cdot n$ states.

The exercises are the jumping game with a window of k (the deque) and sharing sweets among children with caps (prefix sums under a modulus). The timed checkpoint is the shortest round trip through up to twelve cities.

CONTENTS

1	DP optimisations — prefix sums, a monotonic deque and bitmasks	16 min
✓	Checkpoint — DP optimisations	60 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 16 MIN

DP optimisations — prefix sums, a monotonic deque and bitmasks

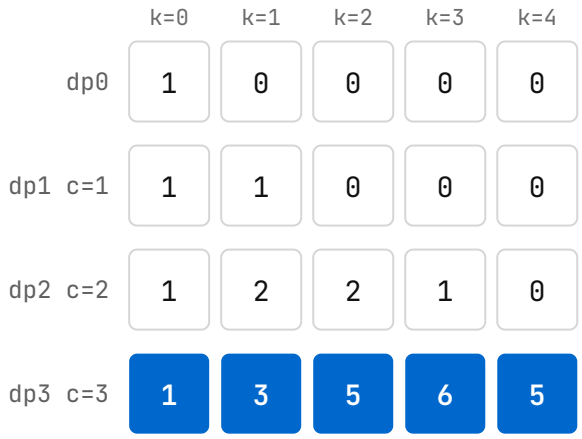
A DP is states × transition cost. The states are usually fixed by the problem; the transition is where the speed hides. If one state reads a whole window of earlier states, the loop is O(n · k) — and $10^5 \times 10^5$ does not fit.

Prefix sums: a range of the previous row in O(1)

Count the ways to share K sweets between n children when child i takes $0..c_i$:

- `dp[i][k] = dp[i-1][k-c_i] + … + dp[i-1][k]` — a contiguous range of the previous row.
- Build prefix sums `P` of row i−1 once, then `dp[i][k] = P[k+1] - P[max(0, k-c_i)]`.
- O(n · K · c) becomes O(n · K). Under a modulus, add the prime before taking `%`.

Prefix sums make each DP cell one subtraction




```
answer dp3[4] = 5
```

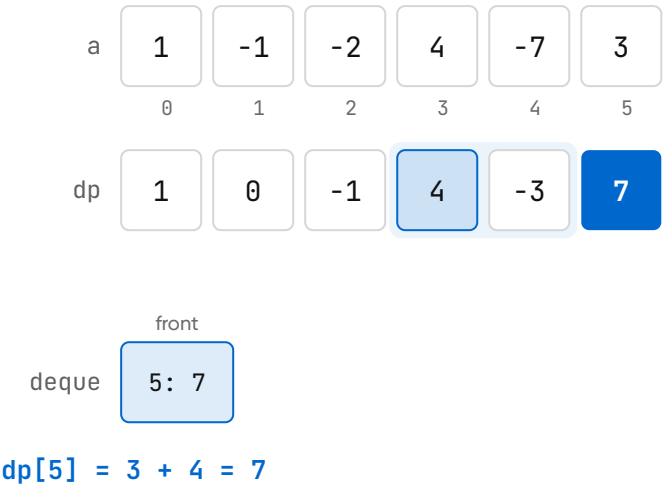
1. dp[i][k] counts the ways to give k sweets to the first i children. With no children there is one way to give 0 sweets and none to give more: row 0 is 1, 0, 0, 0, 0.
2. Child 1 takes 0..1, so dp1[k] sums dp0[k−1..k]. Build prefix sums P of row 0 once; then dp1[1] = P2 − P0 = 1 − 0 = 1, one subtraction instead of 2 additions.
3. Every cell of row 1 is one such subtraction: 1, 1, 0, 0, 0. The row costs O(K), not O(K · c).
4. Child 2 takes 0..2, so dp2[k] sums dp1[k−2..k]. Build prefix sums P of row 1 once; then dp2[3] = P4 − P1 = 2 − 1 = 1, one subtraction instead of 3 additions.
5. Every cell of row 2 is one such subtraction: 1, 2, 2, 1, 0. The row costs O(K), not O(K · c).
6. Child 3 takes 0..3, so dp3[k] sums dp2[k−3..k]. Build prefix sums P of row 2 once; then dp3[4] = P5 − P1 = 6 − 1 = 5, one subtraction instead of 4 additions.
7. Row 3 is 1, 3, 5, 6, 5: there are 5 ways to share 4 sweets with caps 1, 2, 3. The table cost O(n · K) in all.

A monotonic deque: the maximum of a sliding window

Jump through an array, at most k cells at a time, maximising the sum of the cells you land on: `dp[i] = a[i] + max(dp[i-k..i-1])`.

- Keep a deque of indices whose dp values **decrease** from front to back.
- Pop the front when it falls out of the window; read the maximum there.
- Before pushing i, pop every back index with `dp ≤ dp[i]`: it is older *and* no better, so it can never win again.
- Each index enters and leaves once: O(n) instead of O(n · k).

A monotonic deque keeps the window's maximum at its front



1. dp[i] is the best score standing on cell i, moving 1 to 2 cells at a time: dp[i] = a[i] + the maximum of the last 2 dp values. Start with dp[0] = 1 and index 0 in the deque.
2. i = 1: the front, index 0, gives the window's best 1, so dp[1] = -1 + 1 = 0. Nothing at the back is beaten, so 1 joins behind.
3. i = 2: the front, index 0, gives the window's best 1, so dp[2] = -2 + 1 = -1. Nothing at the back is beaten, so 2 joins behind.
4. i = 3: index 0 left the window and was dropped from the front; the front, index 1, gives the window's best 0, so dp[3] = 4 + 0 = 4. Indexes 2 and 1 are no better than 4 and older, so they are popped from the back before 3 is pushed.
5. i = 4: the front, index 3, gives the window's best 4, so dp[4] = -7 + 4 = -3. Nothing at the back is beaten, so 4 joins behind.
6. i = 5: the front, index 3, gives the window's best 4, so dp[5] = 3 + 4 = 7. Indexes 4 and 3 are no better than 7 and older, so they are popped from the back before 5 is pushed. The answer is dp[5] = 7.

```
deque<int> dq{0};
for (int i = 1; i < n; ++i) {
    while (dq.front() < i - k) dq.pop_front();       // left the window
    dp[i] = a[i] + dp[dq.front()];                    // the window's best
    while (!dq.empty() && dp[dq.back()] ⩽ dp[i]) dq.pop_back();
    dq.push_back(i);
}
```

Bitmasks: DP over subsets

When n ≤ 20, a subset fits in an `int`: bit j set means item j is taken.

- `dp[mask][v]` = cheapest path that has visited exactly `mask` and stands at `v`.
- Transition: for each `u` not in `mask`, relax `dp[mask | 1 << u][u]`.

- $2^n \cdot n$ states, n moves each: $O(2^n \cdot n^2)$ — 600 000 steps at n = 12, 8.5×10^7 at n = 18.

GAME · MATCH THE PAIRS

Transition shape to speed-up

Match each transition to the trick that makes it fast.

Choose a term on the left, then its partner on the right.

A	dp[i][k] sums dp[i−1][k−c..k]	·	bitmask over subsets
B	dp[i] = a[i] + max of the last k dp values	·	monotonic deque
C	the state is which of 12 cities are visited	·	a running minimum
D	dp[i] needs the minimum dp[j] for all j < i	·	prefix sums of the previous row

Check

More practice

- Sliding Window Maximum
- Jump Game VI
- Constrained Subsequence Sum
- Number of Dice Rolls With Target Sum
- Shortest Path Visiting All Nodes

EXERCISE 1

Jumps within a window

You stand on cell 1 of a row of `n` cells and must reach cell `n`. Each move goes forward by **1 to k** cells. Your score is the sum of `a[i]` over every cell you stand on, the first and the last included. Print the largest possible score.

`dp[i] = a[i] + max(dp[i-k..i-1])` is O(n · k); keep a **monotonic deque** of indices with decreasing dp values to make it O(n).

Constraints: `1 ≤ n, k ≤ 10^5`, `|a[i]| ≤ 10^4`. **Input:** `n k`, then the `n` values. **Output:** the best score.

```
6 2
1 -1 -2 4 -7 3
```

prints `7` — the cells 1, −1, 4, 3.

INPUT 1

```
6 2
1 -1 -2 4 -7 3
```

OUTPUT 1

```
7
```

INPUT 2

```
1 3
-5
```

OUTPUT 2

```
-5
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    vector<ll> a(n);
    for (auto& x : a) cin >> x;
    vector<ll> dp(n);
    dp[0] = a[0];
    deque<int> dq;
    dq.push_back(0);
    for (int i = 1; i < n; ++i) {
        while (dq.front() < i - k) dq.pop_front();
        dp[i] = a[i] + dp[dq.front()];
        while (!dq.empty() && dp[dq.back()] ≤ dp[i]) dq.pop_back();
        dq.push_back(i);
    }
    cout << dp[n - 1] << '\n';
    return 0;
}
```

EXERCISE 2

Sharing sweets with caps

Share exactly `K` identical sweets among `n` children so that child `i` gets between `0` and `a[i]` sweets. Print the number of ways **modulo 10^9 + 7**.

`dp[i][k] = dp[i-1][k-a[i]] + … + dp[i-1][k]` is a contiguous range of the previous row: keep its **prefix sums** so each state costs O(1), and the whole table O(n · K).

Constraints: `1 ≤ n ≤ 100`, `0 ≤ K ≤ 10^5`, `0 ≤ a[i] ≤ K`. **Input:** `n K`, then `a[1..n]`. **Output:** the count modulo `1 000 000 007`.

```
3 4
1 2 3
```

prints `5`: (0, 1, 3), (0, 2, 2), (1, 0, 3), (1, 1, 2) and (1, 2, 1).

INPUT 1

```
3 4
1 2 3
```

OUTPUT 1

```
5
```

INPUT 2

```
1 10
9
```

OUTPUT 2

```
0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll MOD = 1'000'000'007;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, K;
    cin >> n >> K;
    vector<ll> dp(K + 1, 0), pre(K + 2, 0);
    dp[0] = 1;
    for (int i = 0; i < n; ++i) {
        int c;
        cin >> c;
        pre[0] = 0;
        for (int k = 0; k <= K; ++k) pre[k + 1] = (pre[k] + dp[k]) % MOD;
        for (int k = 0; k <= K; ++k) dp[k] = (pre[k + 1] - pre[max(0, k - c)] + MOD)
    }
    cout << dp[K] << '\n';
    return 0;
}
```

QUIZ

1. `dp[i] = a[i] + max(dp[i-k..i-1])` with n = k = 10^5. What does the monotonic deque change?

A. O(n · k) ≈ 10^10 becomes O(n), because each index is pushed and popped at most once

B. Nothing — it is still O(n · k)

C. O(n · k) becomes O(n log n)

D. It reduces the memory, not the time

2. The deque holds indices with dp values 9, 6, 4 (front to back). The new dp[i] is 7. What happens before pushing i?

A. Pop the 4 and the 6; the deque becomes 9, 7

B. Pop the 9

C. Push without popping: 9, 6, 4, 7

D. Clear the deque

3. Under a modulus, why is `(P[k+1] - P[k-c] + MOD) % MOD` written with `+ MOD`?

A. Both prefixes are reduced mod p, so the smaller-looking one can be the larger number: the difference can be negative, and `%` keeps the sign in C++

B. To avoid overflow of `long long`

C. Because `MOD` is prime

D. It is not needed

4. How many `dp[mask][v]` states does the TSP bitmask DP have at n = 12?

A. 2^12 · 12 = 49 152

B. 12! ≈ 4.8 × 10^8

C. 12^2 = 144

D. 2^12 = 4096

ANSWER KEY · DP OPTIMISATIONS — PREFIX SUMS, A MONOTONIC DEQUE AND BITMASKS

1. A The amortised argument: n pushes and at most n pops across the whole run, plus O(1) work to read the front. A `std::multiset` of the window would give O(n log n); the deque gives O(n).

2. A Back entries with a value ≤ 7 are older and no better, so they can never be a window's maximum while i is in the window. 9 is better and stays — it may still win until it leaves the window.

3. A `-3 % 7` is `-3` in C++, not 4. Adding MOD once brings any difference of two reduced values into 0..2p−2, and `%` finishes it.

4. A A mask for every subset times an end city. Each state has up to 12 moves, about 600 000 transitions — against 12! orders for trying every tour.

CHECKPOINT · 60 MIN

Checkpoint — DP optimisations

A timed checkpoint on the module: prefix sums over the previous row, the monotonic deque for a window maximum, the cost arithmetic that says when an optimisation is needed, and DP over subsets.

How it works. Sixty minutes from the moment you start the timer: ten questions and one program. 70% on the questions and the program accepted clears the module.

Before you start, make sure you can answer these from memory:

- The prefix-sum form of `dp[i-1][k-c] + … + dp[i-1][k]`, and why the difference needs `+ MOD`.
- Which deque entries are popped from the back, and why that is safe.
- How many states and transitions the TSP bitmask DP has at n = 12, and at what n it stops fitting.
- How to test and set bit j of a mask.

The program is the shortest round trip through up to 12 cities.

EXERCISE

Shortest round trip

There are `n` cities and `d[i][j]`, the cost of travelling from city `i` to city `j` (not necessarily equal to `d[j][i]`). Starting at city 0, visit every city exactly once and return to city 0. Print the smallest total cost (`0` when `n = 1`).

Use **bitmask DP**: `dp[mask][v]` = cheapest path from 0 that has visited exactly `mask` and ends at `v`.

Constraints: `1 ≤ n ≤ 12`, `0 ≤ d[i][j] ≤ 1000`, `d[i][i] = 0`. **Input:** `n`, then `n` rows of `n` integers. **Output:** the cost.

```
4
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
```

prints `80` — 0 → 1 → 3 → 2 → 0.

INPUT 1

```
4
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
```

OUTPUT 1

```
80
```

INPUT 2

```
1
0
```

OUTPUT 2

```
0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<vector<ll>> d(n, vector<ll>(n));
    for (auto& row : d)
        for (auto& x : row) cin >> x;
    if (n == 1) {
        cout << 0 << '\n';
        return 0;
    }
    const ll INF = LLONG_MAX / 4;
    const int FULL = 1 << n;
    vector<vector<ll>> dp(FULL, vector<ll>(n, INF));
    dp[1][0] = 0;
    for (int mask = 1; mask < FULL; ++mask) {
        if (!(mask & 1)) continue;                  // every path starts at city 0
        for (int v = 0; v < n; ++v) {
            if (dp[mask][v] >= INF) continue;
            for (int u = 0; u < n; ++u) {
                if (mask >> u & 1) continue;
                int next = mask | 1 << u;
                dp[next][u] = min(dp[next][u], dp[mask][v] + d[v][u]);
            }
        }
    }
    ll best = INF;
    for (int v = 1; v < n; ++v) best = min(best, dp[FULL - 1][v] + d[v][0]);
    cout << best << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. `dp[i][k] = dp[i-1][k-c] + … + dp[i-1][k]` for n = 100 rows, K = 10^5, c up to K. Without prefix sums the table costs about…

A. 10^12 additions — too slow

B. 10^7 additions — fine

C. 10^5 additions

D. 10^9 additions — borderline

2. Which prefix-sum read gives `dp[i-1][lo] + … + dp[i-1][hi]` when `P[j] = dp[i-1][0] + … + dp[i-1][j-1]`?

A. `P[hi + 1] - P[lo]`

B. `P[hi] - P[lo]`

C. `P[hi] - P[lo - 1]`

D. `P[hi + 1] - P[lo + 1]`

3. In the sliding-window-maximum deque, when is the **front** popped?

A. When its index has left the window

B. When a larger value arrives

C. After every read

D. Never — only the back is popped

4. Why may a back entry with `dp[back] ⩽ dp[i]` be discarded for good?

A. It is older than i, so it leaves every future window first, and it is never larger than dp[i] while both are inside

B. Its value will be recomputed later

C. The deque has a fixed capacity

D. Because dp values only increase

5. What does `mask >> j & 1` test?

A. Whether bit j of mask is set — item j is in the subset

B. Whether mask is odd

C. Whether mask has j bits set

D. The j-th lowest set bit

6. How is subset `mask` without item j written?

A. `mask & ~(1 << j)`

B. `mask ^ j`

C. `mask - j`

D. `mask | ~(1 << j)`

7. TSP bitmask DP is O(2^n · n²). Roughly the largest n that fits in a second (~10^8 simple steps)?

A. About 17–18

B. About 30

C. About 12 only

D. About 25

8. In the TSP DP, why is it safe to loop masks in increasing numeric order?

A. Every transition sets a new bit, so it goes from a mask to a strictly larger one

B. Because masks are visited in order of their popcount

C. It is not safe; masks must be sorted by popcount first

D. Because city 0 is bit 0

9. Counting sweets: with K = 0 and every cap 0, how many ways are there?

A. 1 — everyone gets nothing

B. 0

C. n

D. Undefined

10. A DP needs `dp[i] = cost[i] + min(dp[j])` over **all** `j < i`. What is the cheapest speed-up?

A. Keep a running minimum as i grows: O(1) per state

B. A monotonic deque

C. Bitmask DP

D. A segment tree

ANSWER KEY · CHECKPOINT — DP OPTIMISATIONS

1. A 100 · 10^5 states, each summing up to 10^5 values: 10^12. With prefix sums every state is O(1) and the table is 10^7.

2. A With the one-shifted convention P[j] covers indexes 0..j−1, so the slice lo..hi is P[hi + 1] − P[lo]. The shift is what makes lo = 0 need no special case.

3. A The back is popped for being dominated; the front is popped for being too old. The front is always the window's maximum, so it leaves only by expiring.

4. A Any window that contains the old index also contains i, and there i is at least as good. So the old index can never be the unique maximum again.

5. A Shift bit j down to position 0 and keep only it. `>>` binds tighter than `&`, so no parentheses are needed — though many people add them for the reader.

6. A `~(1 << j)` has every bit set except j; AND clears exactly that bit. `mask ^ (1 << j)` also works, but only when the bit is known to be set.

7. A 2^17 · 17² ≈ 3.8 × 10^7 and 2^18 · 18² ≈ 8.5 × 10^7; at 20 it is 4.2 × 10^8. Memory for dp[2^n][n] of `long long` also grows — at n = 20 it is 160 MB.

8. A `mask | 1 << u` with u not in mask is mask + 2^u > mask. Numeric order therefore respects every dependency — sorting by popcount is unnecessary.

9. A The empty distribution is one way. That is exactly dp[0][0] = 1, the base case every row is built from.

10. A A window that only grows never loses an element, so one variable holding the minimum so far is enough. The deque is for windows that also shrink from the left.