ICPC Division 2
Team Reference Document

Contents

Topic	Page
Choose an approach from the constraints	2
C++17 competition starter	3
Build, run, and compare samples	4
Bash shortcuts and short filenames	5
C++ headers and compiler compatibility	6
Trace a small failing case	7
Arrays and scans	8
Strings and parsing	9
Grid scans and simulation	10
Frequency arrays, maps, and sets	11
Sorting and grouping	12
Brute force and small search spaces	13
Prefix sums	14
Two pointers and fixed windows	15
Binary search	16
Stacks and queues	17
Integer arithmetic and GCD	18
Graph representation and BFS	19
Iterative DFS and components	20
Grid flood fill and shortest paths	21
Greedy interval scheduling	22
Heaps and priority queues	23
Basic dynamic programming	24
Submission checklist and sources	25

Before printing Set `\TeamName` to the registered team name. This edition uses US Letter and includes this index in the 25 pages. Check the site instructions for the allowed materials, copy count, and compiler. Print single-sided if the site uses the NAC format. Code fragments use the headers and aliases on page 3. Indices are zero-based and ranges are half-open unless stated otherwise.

Choose an approach from the constraints

Start with the input limits. Multiply across test cases if the statement does not give a bound on the total. Estimate memory as well as runtime.

Scale	Plausible starting points
$n \leq 8$	Enumerate permutations if each check is cheap.
$n \leq 20$	Consider all subsets if the work per subset is small.
$n \approx 100$	A cubic loop makes about one million iterations.
$n \approx 2{,}000$	All pairs make about two million unordered pairs.
$n \approx 200{,}000$	Aim for a scan or $O(n \log n)$ sorting.
$R \times C$ grid	Count cells first. A four-neighbor traversal is $O(RC)$.

These estimates depend on the time limit, work per iteration, and input. A full $200{,}000 \times 200{,}000$ matrix of four-byte integers would need about 160 GB.

Match the wording to a pattern

- Repeated values or membership queries → frequency array, map, or set.
- Repeated sums on unchanged data → prefix sums.
- Adjacent ranks or groups of equal keys → sort and scan.
- First position meeting a threshold in sorted data → binary search.
- Few moves in a maze → BFS with a queue.
- Reachable or in the same group → BFS or DFS.
- Best answer built from earlier smaller states → try a small DP table.

Before typing State what each loop maintains. Work a tiny case by hand and check whether you need an original index, a tie-breaker, or a 64-bit total.

C++17 competition starter

```cpp
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <functional>
#include <iomanip>
#include <iostream>
#include <limits>
#include <map>
#include <numeric>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <string>
#include <tuple>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

using namespace std;
using ll = long long;

void solve() {
    int n;
    cin >> n;                    // Example: one n, then n integers
    vector<ll> a(n);
    for (ll& x : a) cin >> x;
    cout << accumulate(a.begin(), a.end(), 0LL) << '\n';
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    solve();
}
```

Multiple cases Read `t`, then call `solve()` in a `while (t-)` loop. For input until EOF, use the first read as the loop condition. Keep per-case state local.

I/O Use `cin`/`cout` consistently with fast I/O. Interactive problems require explicit flushing. Use `fixed` and `setprecision(k)` for k decimal places.

Build, run, and compare samples

Use when You have a source file, a sample input, and the expected answer.

```
g++ --version
g++ -std=c++17 -O2 -Wall -Wextra a.cpp -o a
./a < sample.in
./a < sample.in > actual.out
diff -u sample.ans actual.out
```

`diff` is silent when the files match. Lines starting with `-` come from the expected file. Lines starting with `+` come from your output. Check the problem's output rules when several answers are valid.

Create a sample without an editor

```
cat > sample.in <<'EOF_SAMPLE'
4
2 5 -1 7
EOF_SAMPLE
printf '%s\n' 13 > sample.ans

g++ -std=c++17 -O2 -Wall -Wextra a.cpp -o a &&
    ./a < sample.in > actual.out &&
    diff -u sample.ans actual.out
```

The quoted heredoc delimiter keeps the pasted text literal. Put the closing delimiter at the start of its line. The `&&` chain stops if compilation or execution fails, which avoids testing a stale binary.

Recommended build flags

- `-std=c++17` selects the language standard used in this reference. Match the judge's C++ option.
- `-O2` enables optimization for a contest-style build.
- `-Wall -Wextra` enables useful compiler warnings. Start with the first error or warning that points into your code.
- Add `-DLOCAL` when you want to enable your own trace. See page 7.

Bash shortcuts and short filenames

Use when You are compiling and running the same problem repeatedly.

Start each problem from one boilerplate file

Save the page 3 starter once as `cpb.cpp`. Copy it to the problem's source file and edit the copy.

```
cp -i cpb.cpp a.cpp
cp -i cpb.cpp b.cpp
```

Run the command for the problem you are starting. The `-i` flag asks before replacing an existing file. Keep `cpb.cpp` ready for the next problem.

Compile and compare with aliases

```
# Enter these once in an interactive Bash session.
alias cc17='g++ -std=c++17 -O2 -Wall -Wextra'
alias df='diff -u'

cc17 a.cpp -o a && ./a < a.in > a.out && df a.ans a.out
cc17 b.cpp -o b && ./b < b.in > b.out && df b.ans b.out
```

Use the problem letter as the file stem. For A, keep the source `a.cpp`, executable `a`, input `a.in`, expected answer `a.ans`, and actual output `a.out` together. A separate answer file keeps the expected result safe when you rerun the program.

Aliases defined at the prompt last for that shell session. On your own machine, put the definitions in `~/.bashrc` if you want Bash to load them in new interactive shells that read that file. At the contest, set them up in the practice session if the site allows it.

Keys worth remembering

Key	Action in Bash
Tab	Complete a filename or command.
Up	Recall the previous command and edit it.
Ctrl-R	Search history for a previous build or test command.
Ctrl-A / Ctrl-E	Move to the start or end of the current command.
Ctrl-C	Interrupt a program that is stuck or waiting for input.

Check Keep `&&` between compile, run, and compare so a failed compile cannot trigger a run of an old executable. Submit the source `a.cpp`, and select the matching language.

C++ headers and compiler compatibility

The starter on page 3 uses standard headers. Include the header that declares the tool you use. Code that compiles on your laptop can fail on the judge if it happened to get a declaration indirectly from another header.

Header	Common uses
`<iostream>`	`cin`, `cout`, `cerr`
`<vector>`, `<string>`	`vector`, `string`, `getline` with a string
`<algorithm>`	`sort`, `reverse`, `lower_bound`, `max`
`<numeric>`	`accumulate`, `gcd` (C++17)
`<queue>`, `<stack>`	`queue`, `priority_queue`, `stack`
`<functional>`	`greater` for a min-heap
`<map>`, `<unordered_map>`	Ordered and hash maps
`<set>`, `<unordered_set>`	Ordered and hash sets
`<utility>`, `<tuple>`	`pair`, `tie`
`<iomanip>`, `<limits>`	`setprecision`, `numeric_limits`

Optional catch-all header

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

This is a nonstandard header supplied by GNU libstdc++. It can save typing on a contest machine that provides it. Try compiling and submitting a small practice program with the selected judge language before relying on it. Use the standard headers in the starter when the catch-all is unavailable, such as with a typical Apple Clang and libc++ setup.

Check If a name is undeclared, check the header, spelling, and `std::` qualification. If a feature like `gcd` is still missing, check the language standard and compiler.

Trace a small failing case

Use when The sample or a small custom test differs from the answer you computed.

Write down the expected output first. Run the case with `diff -u` (page 4). Trace the loop by hand and look for the first point where the program's state differs from your expected state. Check indices, ties, and the initial value of any counter or total.

Print one useful line

Inside the loop, print the variables you want to compare with your trace. Here `i` is the current index and `sum` is the running total.

```
#ifdef LOCAL
cerr << "i=" << i << " sum=" << sum << '\n';
#endif
```

```
g++ -std=c++17 -O2 -Wall -Wextra -DLOCAL a.cpp -o a &&
    ./a < a.in > a.out && diff -u a.ans a.out
```

With the alias on page 5, the build is `cc17 -DLOCAL a.cpp -o a`. The flag defines `LOCAL` so the trace runs. `cerr` stays visible in the terminal while `cout` goes to `a.out`. Rebuild without `-DLOCAL` for the final local check.

Example to trace on paper

If the task is to sum $[2, 5, -1, 7]$, the running totals after each item are $2, 7, 6, 13$. If the trace gives $5, 4, 11$, check whether the loop started at index 1 and skipped the first element. Once this case passes, try a one-element input and an all-negative input if both are legal.

Arrays and scans

Use when Each item can update a small amount of state.

```
// First index of the largest element, or -1 if empty.
int argmax(const vector<ll>& a) {
    int best = -1;
    for (int i = 0; i < static_cast<int>(a.size()); ++i)
        if (best == -1 || a[i] > a[best]) best = i;
    return best;
}
```

Cost $O(n)$ time and $O(1)$ extra space.

Check A strict comparison keeps the first tie. Initialize from data or use a sentinel index so all-negative input works.

Useful vector operations

```
vector<int> a(n, 0);              // n elements, initialized to zero
vector<int> b;
b.reserve(n);                     // capacity only; size still zero
for (int i = 0; i < n; ++i) b.push_back(i);
reverse(b.begin(), b.end());
ll total = accumulate(b.begin(), b.end(), 0LL);
if (!b.empty()) cout << b.front() << ' ' << b.back() << '\n';
```

Appending is amortized $O(1)$. A reallocation can invalidate references, pointers, and iterators. Inserting in the middle shifts elements.

One-pass state example

```
// Longest run of equal neighbors; empty input returns zero.
int longest_run(const vector<int>& a) {
    int best = 0, run = 0;
    for (int i = 0; i < static_cast<int>(a.size()); ++i) {
        if (i == 0 || a[i] != a[i - 1]) run = 1;
        else ++run;
        best = max(best, run);
    }
    return best;
}
```

Strings and parsing

```
string word;
cin >> word;                        // one whitespace-delimited token

// If switching to full lines after a token on its own line:
cin.ignore(numeric_limits<streamsize>::max(), '\n');
string line;
getline(cin, line);                 // keeps spaces; may be empty
```

`getline(cin » ws, line)` skips all leading whitespace, including blank lines. Use it only when that matches the input format. For input consisting entirely of lines, call `getline` from the start.

Split a line and find a substring

```
istringstream fields(line);
string token;
while (fields >> token) {
    // Process one whitespace-delimited token.
}

auto pos = line.find("key=");
if (pos != string::npos) {
    string tail = line.substr(pos + 4);  // suffix after "key="
}
```

Check `substr(start, length)` takes a length. Guard against `string::npos` before adding to a search result.

Palindrome by comparing mirrored characters

```
bool palindrome(const string& s) {
    for (size_t i = 0; i < s.size() / 2; ++i)
        if (s[i] != s[s.size() - 1 - i]) return false;
    return true;
}
```

Cost $O(n)$ time and $O(1)$ extra space. The empty string passes.

Grid scans and simulation

Use when The statement gives a board or a sequence of explicit actions.

```
int rows, cols;
cin >> rows >> cols;
vector<string> grid(rows);
for (string& row : grid) cin >> row;  // No spaces within a row.

// One non-dot character per column, as in Snakey String.
string answer;
for (int c = 0; c < cols; ++c)
    for (int r = 0; r < rows; ++r)
        if (grid[r][c] != '.') answer += grid[r][c];
cout << answer << '\n';
```

Cost $O(RC)$ time. Store the grid in $O(RC)$ space.

Simulate movement with walls

```
// Directions 0,1,2,3 mean up,right,down,left.
// r,c begin on an open cell; moves contains legal directions.
const int dr[] = {-1, 0, 1, 0};
const int dc[] = {0, 1, 0, -1};
for (int d : moves) {
    int nr = r + dr[d], nc = c + dc[d];
    if (nr < 0 || nr >= rows || nc < 0 || nc >= cols)
        continue;
    if (grid[nr][nc] == '#') continue;
    r = nr;
    c = nc;
}
```

Check Test the bounds before indexing. Trace row and column separately. For simultaneous updates, read the old board and write the next board. For sequential updates, apply each move immediately.

Use a hand-traced case with a wall and a boundary step. For connected areas or shortest routes, use page 21.

Frequency arrays, maps, and sets

```
// Dense keys in [0, k).
vector<int> freq(k, 0);
for (int x : a) ++freq[x];

// Arbitrary keys.
unordered_map<int, int> count;
for (int x : a) ++count[x];

auto it = count.find(target);
int times = (it == count.end() ? 0 : it->second);
```

Cost Dense counts take $O(n + k)$ time and $O(k)$ space. Hash lookup and insertion are average $O(1)$. Ordered `map` and `set` operations take $O(\log n)$.

Check `count[target]` inserts a zero if the key is missing. Use `find` for a read that leaves the container unchanged. Use an ordered container or sort the keys if output order matters.

Detect a duplicate and find a successor

```
unordered_set<int> seen;
bool duplicate = false;
for (int x : a)
    if (!seen.insert(x).second) duplicate = true;

set<int> keys(a.begin(), a.end());
auto next_key = keys.upper_bound(target);  // first key > target
if (next_key != keys.end()) cout << *next_key << '\n';
```

A set stores one copy of a key. Use a count map when multiplicity matters. For ordered containers, use the member `lower_bound` or `upper_bound` to get logarithmic traversal.

Sorting and grouping

```
sort(a.begin(), a.end());              // ascending
sort(a.rbegin(), a.rend());            // descending

// Sort records by descending score, then ascending input ID.
struct Record { int score, id; };
sort(records.begin(), records.end(),
     [](const Record& a, const Record& b) {
         if (a.score != b.score) return a.score > b.score;
         return a.id < b.id;
     });
```

Cost Sorting uses $O(n \log n)$ comparisons. Scanning afterward is $O(n)$.

Check A comparator must return false when comparing equal keys in both directions. Use strict comparisons. Preserve original indices when the answer requires them. Use `stable_sort` when equal keys should stay in input order.

Count each group after ascending sort

```
sort(a.begin(), a.end());
int n = static_cast<int>(a.size());
for (int i = 0; i < n; ) {
    int j = i + 1;
    while (j < n && a[j] == a[i]) ++j;
    cout << a[i] << ' ' << (j - i) << '\n';
    i = j;
}
```

Keep distinct values in sorted order

```
sort(a.begin(), a.end());
a.erase(unique(a.begin(), a.end()), a.end());
```

`unique` moves distinct adjacent groups to the front and returns the new logical end. Use `erase` to shrink the vector.

Brute force and small search spaces

```
// Count unordered pairs with sum target; a contains int values.
ll pairs = 0;
for (int i = 0; i < n; ++i)
    for (int j = i + 1; j < n; ++j)
        if (1LL * a[i] + a[j] == target) ++pairs;
```

Cost $O(n^2)$ for pairs. Restrict to $i < j$ to visit each once.

Enumerate subsets for small n

```
// Require 0 <= n <= 20 here; a contains int values.
for (uint64_t mask = 0; mask < (uint64_t{1} << n); ++mask) {
    ll sum = 0;
    for (int i = 0; i < n; ++i)
        if ((mask >> i) & 1U) sum += a[i];
    // Check this subset.
}
```

Cost $O(n2^n)$. Mask zero represents the empty subset.

Enumerate distinct permutations

```
sort(a.begin(), a.end());
do {
    // Check this ordering.
} while (next_permutation(a.begin(), a.end()));
```

Cost At most $n!$ orderings, with $O(n)$ worst-case work to advance. Budget the checking cost too.

Check Start sorted to cover the full set. Repeated values give fewer distinct orderings. For triples use $i < j < k$ when order is irrelevant.

Prefix sums

Use when Many queries ask for sums on an array that stays fixed.

```
int n = static_cast<int>(a.size());
vector<ll> pref(n + 1, 0);
for (int i = 0; i < n; ++i) pref[i + 1] = pref[i] + a[i];

auto range_sum = [&](int left, int right) {
    // Sum on [left, right), with 0 <= left <= right <= n.
    return pref[right] - pref[left];
};
```

Cost $O(n)$ preparation and space. $O(1)$ per query.

`pref[i]` is the sum of the first i values. Subtract the prefix before the requested range. For input endpoints L, R that are one-based and inclusive, call `range_sum(L - 1, R)`. **Check** For $a = [4, -2, 7]$, the prefix array is $[0, 4, 2, 9]$. The range $[1, 3)$ sums to $9 - 4 = 5$. The whole array is $[0, n)$. The empty range sums to zero.

Count events within a range

Build the same prefix array with `pref[i + 1] = pref[i] + (a[i] > 0);` to count positive elements. Substitute any predicate that depends only on the item at i.

Optional grid version

Let $P[r][c]$ sum the first r rows and first c columns. Initialize the top row and left column of P to zero. Fill in row order:

$$P[r+1][c+1] = g[r][c] + P[r][c+1] + P[r+1][c] - P[r][c].$$

The rectangle $[r_1, r_2) \times [c_1, c_2)$ sums to $P[r_2][c_2] - P[r_1][c_2] - P[r_2][c_1] + P[r_1][c_1]$.

Two pointers and fixed windows

Find a pair with a target sum

```
// a is sorted ascending and contains int values.
bool has_pair(const vector<int>& a, ll target) {
    int left = 0, right = static_cast<int>(a.size()) - 1;
    while (left < right) {
        ll sum = 1LL * a[left] + a[right];
        if (sum == target) return true;
        if (sum < target) ++left;
        else --right;
    }
    return false;
}
```

If the sum is too small, even the largest remaining partner is too small for the left value. If the sum is too large, even the smallest partner is too large for the right value. **Cost** $O(n)$ after sorting. Save value-index pairs if original positions matter.

Maximum sum of exactly k consecutive items

```
ll max_window(const vector<int>& a, int k) {
    int n = static_cast<int>(a.size());
    assert(1 <= k && k <= n);
    ll sum = accumulate(a.begin(), a.begin() + k, 0LL);
    ll best = sum;
    for (int r = k; r < n; ++r) {
        sum += 1LL * a[r] - a[r - k];
        best = max(best, sum);
    }
    return best;
}
```

Cost $O(n)$ time and $O(1)$ extra space.

Check Initialize the maximum from the first real window so all-negative input works. Fixed-size windows work with negative values. Variable-size sum windows need a separate monotonicity argument.

Binary search

Use when The data are sorted, or a yes/no condition changes from false to true.

```
sort(a.begin(), a.end());
auto first = lower_bound(a.begin(), a.end(), x);  // >= x
auto after = upper_bound(a.begin(), a.end(), x);  // > x
auto count = after - first;
auto pos = first - a.begin();      // n if no value >= x
bool found = first != a.end() && *first == x;
```

Cost $O(\log n)$ per query on a sorted vector.

Check Test before dereferencing `end()`. On $[2, 4, 4, 9]$, the lower and upper positions for 4 are 1 and 3.

First true in an index range

```
// ok(i) is monotone on [0, n); returns n if all false.
template<class Predicate>
int first_true(int n, Predicate ok) {
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (ok(mid)) hi = mid;
        else lo = mid + 1;
    }
    return lo;
}
```

Each step keeps the boundary in $[lo, hi]$. The predicate is never called at n. For feasibility search, prove that every candidate after a feasible candidate is also feasible. The total cost is the predicate cost times $O(\log(n + 1))$. **Check** Try all false, all true, one element, and an empty domain.

Stacks and queues

Stack for nested delimiters

```
// Input contains only parentheses and square brackets.
bool balanced(const string& s) {
    stack<char> open;
    for (char c : s) {
        if (c == '(' || c == '[') {
            open.push(c);
        } else {
            if (open.empty()) return false;
            if (c == ')' && open.top() != '(') return false;
            if (c == ']' && open.top() != '[') return false;
            open.pop();
        }
    }
    return open.empty();
}
```

Cost $O(n)$ time and $O(n)$ space in the worst case.

The top is the most recent unmatched opening delimiter. Try an early close, an unmatched open, and crossed nesting such as `([)]`.

Queue for first-in, first-out processing

```
queue<int> q;
for (int id : arrivals) q.push(id);
while (!q.empty()) {
    int id = q.front();  // Copy before popping.
    q.pop();
    // Process id.
}
```

`front()` gives the oldest item in a queue. A stack uses `top()` for the newest item. With the default underlying containers, push, pop, and access are $O(1)$. Check for emptiness before reading or popping.

Integer arithmetic and GCD

Division, remainders, and rounding

```
// Require a >= 0 and b > 0.
ll ceil_div(ll a, ll b) {
    return a / b + (a % b != 0);
}

// Require mod > 0; result is in [0, mod).
ll nonnegative_mod(ll x, ll mod) {
    ll r = x % mod;
    if (r < 0) r += mod;
    return r;
}
```

C++ integer division truncates toward zero, so $-7/3 = -2$ and $-7 \bmod 3$ in C++ gives -1. For divisibility by a nonzero d, check `x % d == 0`. For parity, check `x % 2 != 0` for odd values, including negative ones.

GCD and LCM

```
// a,b are nonnegative ll values.
ll g = gcd(a, b);  // <numeric>, C++17; gcd(0,0) == 0
ll l = 0;
if (g != 0) {
    ll reduced = a / g;
    if (b != 0 && reduced > numeric_limits<ll>::max() / b) {
        // LCM does not fit in ll. Handle per the problem.
    } else {
        l = reduced * b;
    }
}
```

Euclid repeatedly replaces (a, b) with $(b, a \bmod b)$ until $b = 0$. Division before multiplication reduces the intermediate size.

Check Promote before doing arithmetic, as in `1LL * width * height` for two `int` values. Verify even the widened result fits. Compute `accumulate` totals with a `0LL` initial value. Avoid division or remainder by zero.

Graph representation and BFS

Read an unweighted graph

```
int n, m;
cin >> n >> m;
vector<vector<int>> adj(n);
for (int i = 0; i < m; ++i) {
    int u, v;
    cin >> u >> v;
    --u; --v;                  // If input labels run from 1 to n.
    adj[u].push_back(v);
    adj[v].push_back(u);       // Omit for directed edges.
}
```

Distances from one source

```
vector<int> bfs(const vector<vector<int>>& adj, int source) {
    vector<int> dist(adj.size(), -1);
    queue<int> q;
    dist[source] = 0;
    q.push(source);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : adj[u]) {
            if (dist[v] != -1) continue;
            dist[v] = dist[u] + 1;
            q.push(v);
        }
    }
    return dist;
}
```

Cost $O(V + E)$ time and $O(V)$ extra space beyond adjacency lists.

Check Mark a vertex when enqueuing so it enters the queue once. Unreachable vertices keep distance -1. If you need a shortest path, also assign `parent[v] = u` on discovery and trace parents back from a reachable target. BFS minimizes the number of edges.

Iterative DFS and components

Use when You need to count connected groups or mark reachable vertices.

```
// For component counting, adj represents an undirected graph.
int components(const vector<vector<int>>& adj) {
    int n = static_cast<int>(adj.size());
    vector<char> seen(n, false);
    stack<int> pending;
    int groups = 0;

    for (int start = 0; start < n; ++start) {
        if (seen[start]) continue;
        ++groups;
        seen[start] = true;
        pending.push(start);

        while (!pending.empty()) {
            int u = pending.top();
            pending.pop();
            for (int v : adj[u]) {
                if (seen[v]) continue;
                seen[v] = true;
                pending.push(v);
            }
        }
    }
    return groups;
}
```

Cost $O(V + E)$ time and $O(V)$ extra space.

One traversal marks every vertex in the current group. The outer loop finds the next group. An isolated vertex is a group of size one. For group sizes, increment a local counter each time a vertex is popped.

Check For reachability from one vertex, start one traversal. The order can differ from recursive DFS. Iteration avoids exhausting the call stack on a long chain. This component count assumes undirected edges.

Grid flood fill and shortest paths

Use when Open cells connect by up, down, left, and right moves.

```
// g is rectangular and nonempty; sr,sc is an open cell.
vector<vector<int>> grid_bfs(const vector<string>& g,
                             int sr, int sc) {
    int rows = static_cast<int>(g.size());
    int cols = static_cast<int>(g[0].size());
    vector<vector<int>> dist(rows, vector<int>(cols, -1));
    const int dr[] = {-1, 0, 1, 0};
    const int dc[] = {0, 1, 0, -1};
    queue<pair<int, int>> q;
    q.emplace(sr, sc);
    dist[sr][sc] = 0;

    while (!q.empty()) {
        auto [r, c] = q.front();
        q.pop();
        for (int d = 0; d < 4; ++d) {
            int nr = r + dr[d], nc = c + dc[d];
            if (nr < 0 || nr >= rows || nc < 0 || nc >= cols)
                continue;
            if (g[nr][nc] == '#' || dist[nr][nc] != -1)
                continue;
            dist[nr][nc] = dist[r][c] + 1;
            q.emplace(nr, nc);
        }
    }
    return dist;
}
```

Cost $O(RC)$ time and space.

For counting regions, use one shared visited grid. Scan all cells and start a traversal each time you find an unvisited open cell. Mark cells when you enqueue them and keep the marks between traversals.

Check A one-cell grid gives distance zero to its start. A walled-off destination keeps -1. Add diagonal directions only if the statement allows them.

Greedy interval scheduling

Use when Choose the largest number of nonoverlapping intervals, each with equal value.

```cpp
// Each job is {start, finish} with start < finish.
// A job may start when the previous one finishes.
int max_jobs(vector<pair<int, int>> jobs) {
    sort(jobs.begin(), jobs.end(), [](const auto& a, const auto& b) {
        return tie(a.second, a.first) < tie(b.second, b.first);
    });

    int picked = 0;
    int end = numeric_limits<int>::min();
    for (auto [start, finish] : jobs) {
        if (start < end) continue;
        ++picked;
        end = finish;
    }
    return picked;
}
```

Cost $O(n \log n)$ time. The by-value parameter copies the input.

Why earliest finish works

Take an optimal schedule. Replace its first job with the earliest-finishing job. All later jobs still fit because the replacement finishes no later. Repeat on the jobs that can follow it.

Check For $[0, 10), [1, 3), [3, 5)$, earliest start takes one job and earliest finish takes two. For closed intervals where touching counts as overlap, require a strictly later start after the first job. The proof assumes equal value per job.

For other greedy problems, try to exchange your choice into an optimal solution, or test your rule against brute force on small instances.

Heaps and priority queues

Use when You repeatedly need the smallest or largest current value while adding items.

```
priority_queue<int> max_heap;
priority_queue<int, vector<int>, greater<int>> min_heap;

min_heap.push(8);
min_heap.push(3);
if (!min_heap.empty()) {
    int smallest = min_heap.top();  // 3
    min_heap.pop();                 // pop returns void
}
```

The starter includes `<functional>` for `greater`. **Cost** $O(1)$ for top and $O(\log n)$ for push or pop.

Keep the k largest values seen so far

```
// Require 1 <= k <= a.size().
priority_queue<int, vector<int>, greater<int>> keep;
for (int x : a) {
    keep.push(x);
    if (static_cast<int>(keep.size()) > k) keep.pop();
}
int kth_largest = keep.top();
```

The smallest value among the retained k is the kth largest overall. This takes $O(n \log(k+1))$ time and $O(k)$ extra space. Duplicates count as separate values.

Priorities with tie-breakers

```
using Job = pair<int, int>;  // {priority, id}
priority_queue<Job, vector<Job>, greater<Job>> pending;
```

The smallest priority comes first, then the smallest ID on a tie. When the entire order is needed and all items are already available, sorting a vector is usually simpler.

Basic dynamic programming

Write down the state, base cases, transition, iteration order, and location of the answer.

Count ways to climb n steps using moves of 1 or 2

```
// n >= 0; require the count to fit in ll.
vector<ll> ways(n + 1, 0);
ways[0] = 1;
for (int i = 1; i <= n; ++i) {
    ways[i] = ways[i - 1];
    if (i >= 2) ways[i] += ways[i - 2];
}
cout << ways[n] << '\n';
```

`ways[i]` counts sequences ending at step i. The final move came from $i-1$ or $i-2$. At step zero there is one empty sequence. Time and space are $O(n)$.

Count right/down paths through a grid with walls

```
// rows,cols > 0; every count fits in ll.
vector<vector<ll>> ways(rows, vector<ll>(cols, 0));
for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
        if (g[r][c] == '#') continue;
        if (r == 0 && c == 0) ways[r][c] = 1;
        if (r > 0) ways[r][c] += ways[r - 1][c];
        if (c > 0) ways[r][c] += ways[r][c - 1];
    }
}
cout << ways[rows - 1][cols - 1] << '\n';
```

Cost $O(RC)$ time and space. The top and left states are ready in row order.

Check Test a blocked start, blocked finish, single row, and a 2×2 open board (two paths). If a modulus is specified, reduce during the transition and keep intermediate additions representable.

Submission checklist and sources

Before sending

1. Check the input contract. Is there one case, a leading count, or EOF?
2. Check index bases, endpoint conventions, ties, and original positions.
3. Test the sample, then try a smallest legal input and a boundary case.
4. Verify signed arithmetic, integer widths, and total work across cases.
5. Build the final source, compare output, and select the right problem and language.

If the judge rejects it

Verdict	First things to check
Wrong answer	Input shape, ties, boundaries, overflow, and format.
Time limit	Complexity at maximum input, repeated work, infinite loops.
Runtime error	Bounds, empty containers, division by zero, memory use.
Compile error	Language setting, headers, compiler diagnostics.

Format and reference sources

- ICPC North America Championship, 2024 TRD instructions. Format model for 25 US Letter pages with team and university in the upper left and page numbers in the upper right.
- ICPC Southeast USA Regional. Check current local instructions for what your team may bring and use.
- KTH KACTL and Stanford ACM-ICPC team notebook. Public examples of indexed code references. The listings here are written for this Division 2 notebook.
- FIU Southeast USA regional archive. Source for Division 2 practice sets, including the 2025 set containing *Snakey String*.

Rebuild Run `latexmk -pdf icpc-d2-trd.tex` from the source directory. After edits, verify the index and page count before printing.