Competitive Programming Handbook

FardinMahadi

December 9, 2025

Contents

1 Templates and Boilerplates

1.1 Main C++ Boilerplate

```
// In the name of Allah, the Most Gracious, the Most Merciful
// C: FardinMahadi

#include <bits/stdc++.h>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>

using namespace __gnu_pbds;
using namespace std;

template<typename T> using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag, tree_order_statistics_node_update>;

#define sp                  ' '
#define nl                  '\n'
#define F                   first
#define S                   second
#define ll                  long long
#define pb                  push_back
#define pf                  push_front
#define popb                pop_back
#define popf                pop_front
#define gcd(x,y)            __gcd(x,y)
#define lcm(x,y)            y*x/__gcd(x,y)
#define no                  cout << "NO" << nl
#define yes                 cout << "YES" << nl
#define all(a)              (a.begin()),(a.end())
#define SUM(a)              accumulate(all(a),0LL)
#define cinv(v)             for(auto &i : v) cin >> i
#define coutv(v)            for(auto &i : v) cout << i << sp
#define fixedpoint(x)       cout << fixed << setprecision(x)
#define UNIQUE(X)           (X).erase(unique(all(X)),(X).end())
#define print(v)            for(auto x : v) cout << x << " "; cout << nl
#define SORT_UNIQUE(c)      (sort(c.begin(),c.end()), c.resize(distance(c.
    begin(),unique(c.begin(),c.end()))))

// Enhanced Macros
#define min3(a,b,c)         min(a,min(b,c))
#define max3(a,b,c)         max(a,max(b,c))
#define min4(a,b,c,d)       min(a,min(b,min(c,d)))
#define max4(a,b,c,d)       max(a,max(b,max(c,d)))
#define sz(x)               ((int)(x).size())
#define sqr(x)              ((x)*(x))
#define ceildiv(a,b)        ((a+b-1)/b)

// Debug Macros
#ifdef LOCAL
#define debug(x)            cerr << #x << " = " << x << nl
#define debug2(x,y)         cerr << #x << " = " << x << ", " << #y << " = " <<
     y << nl
```

```cpp
#define debug3(x,y,z)        cerr << #x << " = " << x << ", " << #y << " = " << y << ", " << #z << " = " << z << nl
#define debugv(v)            cerr << #v << " = ["; for(auto x : v) cerr << x << ", "; cerr << "]" << nl
#else
#define debug(x)
#define debug2(x,y)
#define debug3(x,y,z)
#define debugv(v)
#endif

// Constants
const double PI = acos(-1);
const int INF = 1e9 + 7;
const ll LINF = 1e18 + 7;
const int MOD = 1e9 + 7;
const int N = 1e5 + 5;

int n, m;

void Solve(int tc) {
    // Your code here
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);

    int t = 1;
    cin >> t;
    for (int tc = 1; tc <= t; tc++) Solve(tc);

    return 0;
}
```

1.2 Binary Search Template

```cpp
// Binary search on answer
ll binarySearch(ll left, ll right, function<bool(ll)> check) {
    ll ans = right;
    while (left <= right) {
        ll mid = left + (right - left) / 2;
        if (check(mid)) {
            ans = mid;
            right = mid - 1;  // For minimum valid answer
            // left = mid + 1;  // For maximum valid answer
        } else {
            left = mid + 1;  // For minimum valid answer
            // right = mid - 1;  // For maximum valid answer
        }
    }
    return ans;
}
```

1.3 Binary Exponentiation

```cpp
// Binary exponentiation (a^b) mod m
ll binpow(ll a, ll b, ll m = MOD) {
    a %= m;
    ll res = 1;
    while (b > 0) {
        if (b & 1) res = (res * a) % m;
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}
```

2 Data Structures

2.1 DSU (Disjoint Set Union)

```
// DSU with path compression and union by rank
class DSU {
private:
    vector<int> parent, rank;
public:
    DSU(int n) {
        parent.resize(n+1);
        rank.resize(n+1, 0);
        for (int i = 0; i <= n; i++) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }

    void unite(int x, int y) {
        x = find(x), y = find(y);
        if (x == y) return;
        if (rank[x] < rank[y]) swap(x, y);
        parent[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }

    bool same(int x, int y) {
        return find(x) == find(y);
    }
};
```

2.2 Segment Tree

```
// Segment Tree for range queries and updates
class SegmentTree {
private:
    vector<ll> tree;
    int n;

    void build(vector<ll> &arr, int node, int start, int end) {
        if (start == end) {
            tree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            build(arr, 2*node, start, mid);
            build(arr, 2*node+1, mid+1, end);
            tree[node] = tree[2*node] + tree[2*node+1];  // Change operation as needed
        }
    }

    void update(int node, int start, int end, int idx, ll val) {
        if (start == end) {
            tree[node] = val;
        } else {
            int mid = (start + end) / 2;
            if (idx <= mid) update(2*node, start, mid, idx, val);
            else update(2*node+1, mid+1, end, idx, val);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    ll query(int node, int start, int end, int l, int r) {
        if (r < start || l > end) return 0;  // Change identity element as needed
        if (l <= start && end <= r) return tree[node];
        int mid = (start + end) / 2;
        return query(2*node, start, mid, l, r) + query(2*node+1, mid+1, end, l, r);
    }

public:
    SegmentTree(vector<ll> &arr) {
        n = arr.size();
        tree.resize(4*n);
        build(arr, 1, 0, n-1);
    }

    void update(int idx, ll val) {
        update(1, 0, n-1, idx, val);
    }

    ll query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }
};
```

2.3 Fenwick Tree (BIT)

```
// Fenwick Tree (Binary Indexed Tree)
class FenwickTree {
private:
    vector<ll> tree;
    int n;

public:
    FenwickTree(int size) {
        n = size;
        tree.resize(n+1, 0);
    }

    void update(int idx, ll delta) {
        for (; idx <= n; idx += idx & -idx) {
            tree[idx] += delta;
        }
```

```
    }

    ll query(int idx) {
        ll sum = 0;
        for (; idx > 0; idx -= idx & -idx) {
            sum += tree[idx];
        }
        return sum;
    }

    ll rangeQuery(int l, int r) {
        return query(r) - query(l-1);
    }
};
```

2.4 Trie

```
// Trie (Prefix Tree)
class Trie {
private:
    struct Node {
        vector<Node*> children;
        bool isEnd;
        Node() : children(26, nullptr), isEnd(false) {}
    };
    Node* root;

public:
    Trie() { root = new Node(); }

    void insert(string word) {
        Node* curr = root;
        for (char c : word) {
            int idx = c - 'a';
            if (!curr->children[idx]) {
                curr->children[idx] = new Node();
            }
            curr = curr->children[idx];
        }
        curr->isEnd = true;
    }

    bool search(string word) {
        Node* curr = root;
        for (char c : word) {
            int idx = c - 'a';
            if (!curr->children[idx]) return false;
            curr = curr->children[idx];
        }
        return curr->isEnd;
    }

    bool startsWith(string prefix) {
        Node* curr = root;
        for (char c : prefix) {
            int idx = c - 'a';
            if (!curr->children[idx]) return false;
            curr = curr->children[idx];
        }
        return true;
    }
};
```

2.5 Sparse Table

```
// Sparse Table for RMQ (Range Minimum Query)
class SparseTable {
private:
    vector<vector<ll>> table;
    vector<int> log;

public:
    SparseTable(vector<ll> &arr) {
        int n = arr.size();
        int maxLog = log2(n) + 1;
        table.assign(n, vector<ll>(maxLog));
        log.resize(n+1);

        for (int i = 0; i < n; i++) table[i][0] = arr[i];

        for (int j = 1; j < maxLog; j++) {
            for (int i = 0; i + (1 << j) <= n; i++) {
                table[i][j] = min(table[i][j-1], table[i + (1 << (j-1))][j-1])
    ;
            }
        }

        for (int i = 2; i <= n; i++) log[i] = log[i/2] + 1;
    }

    ll query(int l, int r) {
        int j = log[r - l + 1];
        return min(table[l][j], table[r - (1 << j) + 1][j]);
    }
};
```

2.6 Ordered Set (PBDS)

```
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

template<typename T> using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag, tree_order_statistics_node_update>;

// Usage:
ordered_set<int> os;
```

```
os.insert(5);
os.insert(2);
os.insert(7);
os.order_of_key(5);  // Returns number of elements < 5
*os.find_by_order(1);  // Returns element at index 1
```

3 Graph Algorithms

3.1 Graph Representation

```
// Graph representation - Adjacency list
vector<vector<int>> adj(N);  // Unweighted
vector<vector<pair<int, int>>> adjWeighted(N);  // Weighted: {node, weight}

// For directed graph, add edge once
// For undirected graph, add edge twice
void addEdge(int u, int v) {
    adj[u].pb(v);
    // adj[v].pb(u);  // Uncomment for undirected
}

void addWeightedEdge(int u, int v, int w) {
    adjWeighted[u].pb({v, w});
    // adjWeighted[v].pb({u, w});  // Uncomment for undirected
}
```

3.2 DFS (Depth First Search)

```
// DFS - Depth First Search
vector<bool> visited(N, false);

void dfs(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs(v);
        }
    }
}

// DFS with parent tracking
void dfs(int u, int parent) {
    for (int v : adj[u]) {
        if (v != parent) {
            dfs(v, u);
        }
    }
}
```

3.3 BFS (Breadth First Search)

```
// BFS - Breadth First Search
vector<bool> visited(N, false);
vector<int> dist(N, -1);

void bfs(int start) {
    queue<int> q;
    q.push(start);
```

```
    visited[start] = true;
    dist[start] = 0;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

3.4 Dijkstra's Algorithm

```
// Dijkstra's Algorithm - Single source shortest path
vector<ll> dijkstra(int start, int n) {
    vector<ll> dist(n+1, LINF);
    dist[start] = 0;
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair<ll, int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        ll d = pq.top().F;
        int u = pq.top().S;
        pq.pop();

        if (d > dist[u]) continue;

        for (auto [v, w] : adjWeighted[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

3.5 Floyd-Warshall

```
// Floyd-Warshall - All pairs shortest path
vector<vector<ll>> floydWarshall(int n) {
    vector<vector<ll>> dist(n+1, vector<ll>(n+1, LINF));

    // Initialize distances
    for (int i = 1; i <= n; i++) dist[i][i] = 0;
    for (int u = 1; u <= n; u++) {
        for (auto [v, w] : adjWeighted[u]) {
            dist[u][v] = min(dist[u][v], (ll)w);
        }
    }

    // Floyd-Warshall algorithm
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (dist[i][k] != LINF && dist[k][j] != LINF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
    return dist;
}
```

3.6 Topological Sort

```
// Topological Sort using DFS
vector<bool> visited(N, false);
vector<int> topoOrder;

void dfsTopo(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfsTopo(v);
        }
    }
    topoOrder.pb(u);
}

vector<int> topologicalSort(int n) {
    topoOrder.clear();
    fill(all(visited), false);
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) dfsTopo(i);
    }
    reverse(all(topoOrder));
    return topoOrder;
}
```

3.7 Cycle Detection

```
// Cycle detection in directed graph
vector<int> color(N, 0);  // 0: white, 1: gray, 2: black

bool hasCycle(int u) {
    color[u] = 1;  // Gray
    for (int v : adj[u]) {
        if (color[v] == 1) return true;  // Back edge found
```

```cpp
        if (color[v] == 0 && hasCycle(v)) return true;
    }
    color[u] = 2;  // Black
    return false;
}

// Cycle detection in undirected graph
bool hasCycleUndirected(int u, int parent) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            if (hasCycleUndirected(v, u)) return true;
        } else if (v != parent) {
            return true;
        }
    }
    return false;
}
```

3.8 Strongly Connected Components (Kosaraju)

```cpp
// Kosaraju's algorithm for Strongly Connected Components
vector<bool> visited(N, false);
vector<int> order, component;

void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) dfs1(v);
    }
    order.pb(u);
}

void dfs2(int u, vector<vector<int>> &adjRev) {
    visited[u] = true;
    component.pb(u);
    for (int v : adjRev[u]) {
        if (!visited[v]) dfs2(v, adjRev);
    }
}

vector<vector<int>> findSCC(int n) {
    // Build reverse graph
    vector<vector<int>> adjRev(n+1);
    for (int u = 1; u <= n; u++) {
        for (int v : adj[u]) adjRev[v].pb(u);
    }

    // First DFS
    fill(all(visited), false);
    order.clear();
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) dfs1(i);
    }

    // Second DFS on reverse graph
    fill(all(visited), false);
    vector<vector<int>> scc;
    reverse(all(order));
    for (int u : order) {
        if (!visited[u]) {
            component.clear();
            dfs2(u, adjRev);
            scc.pb(component);
        }
    }
    return scc;
}
```

4 String Algorithms

4.1 KMP Algorithm

```
// KMP Algorithm for pattern matching
vector<int> buildLPS(string pattern) {
    int m = pattern.length();
    vector<int> lps(m, 0);
    int len = 0, i = 1;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) len = lps[len-1];
            else { lps[i] = 0; i++; }
        }
    }
    return lps;
}

vector<int> kmpSearch(string text, string pattern) {
    vector<int> lps = buildLPS(pattern);
    vector<int> matches;
    int n = text.length(), m = pattern.length();
    int i = 0, j = 0;

    while (i < n) {
        if (text[i] == pattern[j]) { i++; j++; }
        if (j == m) {
            matches.pb(i - j);
            j = lps[j-1];
        } else if (i < n && text[i] != pattern[j]) {
            if (j != 0) j = lps[j-1];
            else i++;
        }
    }
    return matches;
}
```

4.2 Z-Algorithm

```
// Z-Algorithm for string preprocessing
vector<int> buildZ(string s) {
    int n = s.length();
    vector<int> z(n, 0);
    int l = 0, r = 0;

    for (int i = 1; i < n; i++) {
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
        if (i + z[i] - 1 > r) {
            l = i;
            r = i + z[i] - 1;
        }
    }
    return z;
}

// Find all occurrences of pattern in text
vector<int> zSearch(string text, string pattern) {
    string combined = pattern + "$" + text;
    vector<int> z = buildZ(combined);
    vector<int> matches;
    int m = pattern.length();
    for (int i = m + 1; i < combined.length(); i++) {
        if (z[i] == m) matches.pb(i - m - 1);
    }
    return matches;
}
```

4.3 String Utilities

```
// String reverse
string reverseStr(string s) {
    reverse(all(s));
    return s;
}

// Check if string is palindrome
bool isPalindrome(string s) {
    int n = s.length();
    for (int i = 0; i < n/2; i++) {
        if (s[i] != s[n-1-i]) return false;
    }
    return true;
}

// Convert string to lowercase
string toLower(string s) {
    transform(all(s), s.begin(), ::tolower);
    return s;
}

// Convert string to uppercase
string toUpper(string s) {
    transform(all(s), s.begin(), ::toupper);
    return s;
}

// Split string by delimiter
vector<string> split(const string &str, const string &delimiter) {
    vector<string> tokens;
    size_t start = 0, end, delimLength = delimiter.length();

```

```cpp
    while ((end = str.find(delimiter, start)) != string::npos) {
        tokens.push_back(str.substr(start, end - start));
        start = end + delimLength;
    }
    tokens.push_back(str.substr(start));
    return tokens;
}
```

5 Mathematical Algorithms

5.1 GCD and LCM

```cpp
// Recursive GCD
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

// Efficient and overflow-safe LCM function
ll lcm(ll a, ll b) {
    return (a / __gcd(a, b)) * b;
}
```

5.2 Extended Euclidean Algorithm

```cpp
// Extended Euclidean Algorithm - returns gcd and coefficients
ll extgcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    ll x1, y1;
    ll g = extgcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}

// Modular inverse using extended Euclidean
ll modInverse(ll a, ll m = MOD) {
    ll x, y;
    ll g = extgcd(a, m, x, y);
    if (g != 1) return -1;  // No inverse exists
    return (x % m + m) % m;
}
```

5.3 Modular Arithmetic

```cpp
// Modular arithmetic operations
ll modAdd(ll a, ll b, ll m = MOD) { return ((a % m) + (b % m)) % m; }
ll modSub(ll a, ll b, ll m = MOD) { return ((a % m) - (b % m) + m) % m; }
ll modMul(ll a, ll b, ll m = MOD) { return ((a % m) * (b % m)) % m; }
ll modDiv(ll a, ll b, ll m = MOD) { return modMul(a, modInverse(b, m), m); }
```

5.4 Sieve of Eratosthenes

```cpp
// Sieve of Eratosthenes - Generate primes up to n
vector<bool> isPrime(N, true);
vector<int> primes;

```

```cpp
void sieve(int n) {
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }
    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) primes.pb(i);
    }
}
```

5.5 Prime Factorization

```cpp
// Prime factorization
vector<pair<ll, int>> factorize(ll n) {
    vector<pair<ll, int>> factors;
    for (ll i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            int cnt = 0;
            while (n % i == 0) {
                n /= i;
                cnt++;
            }
            factors.pb({i, cnt});
        }
    }
    if (n > 1) factors.pb({n, 1});
    return factors;
}
```

5.6 Euler Totient Function

```cpp
// Euler Totient Function using sieve
vector<int> phi(N);

void eulerTotient(int n) {
    for (int i = 1; i <= n; i++) phi[i] = i;
    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {  // i is prime
            for (int j = i; j <= n; j += i) {
                phi[j] -= phi[j] / i;
            }
        }
    }
}
```

5.7 Combinatorics

```cpp
// Factorial with mod
vector<ll> fact(N);

void precomputeFactorial(int n, ll m = MOD) {
    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (fact[i-1] * i) % m;
    }
}

// nCr with mod
ll nCr(ll n, ll r, ll m = MOD) {
    if (r > n || r < 0) return 0;
    ll num = fact[n];
    ll den = (fact[r] * fact[n-r]) % m;
    return (num * modInverse(den, m)) % m;
}
```

5.8 Matrix Exponentiation

```cpp
// Matrix multiplication
vector<vector<ll>> matMul(vector<vector<ll>> &a, vector<vector<ll>> &b, ll m =
     MOD) {
    int n = a.size();
    vector<vector<ll>> res(n, vector<ll>(n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                res[i][j] = (res[i][j] + (a[i][k] * b[k][j]) % m) % m;
            }
        }
    }
    return res;
}

// Matrix exponentiation
vector<vector<ll>> matPow(vector<vector<ll>> base, ll exp, ll m = MOD) {
    int n = base.size();
    vector<vector<ll>> res(n, vector<ll>(n, 0));
    for (int i = 0; i < n; i++) res[i][i] = 1;  // Identity matrix

    while (exp > 0) {
        if (exp & 1) res = matMul(res, base, m);
        base = matMul(base, base, m);
        exp >>= 1;
    }
    return res;
}
```

6 STL Containers and Utilities

6.1 Vector

```
vector<int> v;
v.push_back(5);           // Add element
v.pop_back();             // Remove last element
v.size();                 // Get size
v.empty();                // Check if empty
v.clear();                // Clear all elements
sort(v.begin(), v.end()); // Sort
reverse(v.begin(), v.end()); // Reverse
v.resize(n, val);         // Resize with default value
v.insert(v.begin() + i, val); // Insert at position
v.erase(v.begin() + i);   // Erase at position
auto it = find(v.begin(), v.end(), val); // Find element
```

6.2 Map and Unordered Map

```
map<string, int> mp;
mp["key"] = value;        // Insert/update
mp.count("key");          // Check existence (0 or 1)
mp.find("key");           // Returns iterator
mp.erase("key");          // Erase by key
mp.size();                // Number of elements
for (auto [key, val] : mp) { } // Iterate

unordered_map<string, int> ump; // Faster, no ordering
```

6.3 Set and Unordered Set

```
set<int> s;
s.insert(5);              // Insert
s.erase(5);               // Erase
s.count(5);               // Check existence
s.find(5);                // Returns iterator
s.lower_bound(5);         // First element >= 5
s.upper_bound(5);         // First element > 5
s.size();

multiset<int> ms;         // Allows duplicates
ms.erase(ms.find(5));    // Erase one occurrence
ms.erase(5);              // Erase all occurrences

unordered_set<int> us;    // Faster, no ordering
```

6.4 Priority Queue

```
// Max heap (default)
priority_queue<int> pq;
pq.push(5);
pq.top();                 // Get max element
pq.pop();                 // Remove max element
pq.size();
pq.empty();

// Min heap
priority_queue<int, vector<int>, greater<int>> minpq;

// Custom comparator
auto cmp = [](int a, int b) { return a > b; };
priority_queue<int, vector<int>, decltype(cmp)> custompq(cmp);
```

6.5 Queue and Deque

```
queue<int> q;
q.push(5);                // Add to back
q.pop();                  // Remove from front
q.front();                // Get front element
q.back();                 // Get back element
q.size();
q.empty();

deque<int> dq;
dq.push_front(5);         // Add to front
dq.push_back(5);          // Add to back
dq.pop_front();           // Remove from front
dq.pop_back();            // Remove from back
dq.front();
dq.back();
```

6.6 Stack

```
stack<int> st;
st.push(5);               // Add to top
st.pop();                 // Remove from top
st.top();                 // Get top element
st.size();
st.empty();
```

6.7 Pair

```
pair<int, int> p = {1, 2};
p.first;                  // Access first element
p.second;                 // Access second element
p = make_pair(3, 4);

// Vector of pairs
vector<pair<int, int>> vp;
vp.push_back({1, 2});
sort(vp.begin(), vp.end()); // Sorts by first, then second
```

6.8 Common STL Algorithms

```
// Sorting
sort(v.begin(), v.end());                      // Ascending
sort(v.begin(), v.end(), greater<int>());      // Descending
sort(v.begin(), v.end(), [](int a, int b) { return a > b; }); // Custom

// Binary search (requires sorted container)
binary_search(v.begin(), v.end(), val);        // Returns bool
lower_bound(v.begin(), v.end(), val);          // First >= val
upper_bound(v.begin(), v.end(), val);          // First > val

// Other useful functions
max(a, b); min(a, b);
max_element(v.begin(), v.end());               // Returns iterator
min_element(v.begin(), v.end());
accumulate(v.begin(), v.end(), 0LL);           // Sum
count(v.begin(), v.end(), val);                // Count occurrences
find(v.begin(), v.end(), val);                 // Find element
reverse(v.begin(), v.end());                   // Reverse
unique(v.begin(), v.end());                    // Remove consecutive duplicates
next_permutation(v.begin(), v.end());          // Next permutation
prev_permutation(v.begin(), v.end());          // Previous permutation
```

7 Miscellaneous Utilities

7.1 Grid Helpers

```
// 4-direction movement
int dx4[] = {-1, 0, +1, 0};
int dy4[] = {0, -1, 0, +1};

// 8-direction movement (including diagonals)
int dx8[] = {-1, -1, -1, 0, 0, +1, +1, +1};
int dy8[] = {-1, 0, +1, -1, +1, -1, 0, +1};

// Boundary check
inline bool in(int i, int j) {
    return (0 <= i && i < n && 0 <= j && j < m);
}

// Grid traversal helper
void traverseGrid(int i, int j) {
    for (int k = 0; k < 4; k++) {  // Change to 8 for diagonal
        int ni = i + dx4[k];
        int nj = j + dy4[k];
        if (in(ni, nj)) {
            // Process cell (ni, nj)
        }
    }
}
```

7.2 Coordinate Compression

```
// Coordinate compression
vector<int> compress(vector<int> &arr) {
    vector<int> sorted = arr;
    sort(all(sorted));
    sorted.erase(unique(all(sorted)), sorted.end());

    vector<int> compressed;
    for (int x : arr) {
        int idx = lower_bound(all(sorted), x) - sorted.begin();
        compressed.pb(idx);
    }
    return compressed;
}
```

7.3 Timing Utilities

```
#include <chrono>
using namespace std::chrono;

auto start_time = high_resolution_clock::now();
// Your code here
auto end_time = high_resolution_clock::now();
```

```cpp
auto duration = duration_cast<microseconds>(end_time - start_time);
cout << "Time: " << duration.count() << " microseconds" << endl;
```

7.4 Merge Sort with Inversion Count

```cpp
ll inversionCount = 0;

void merge(vector<int>& arr, int low, int mid, int high) {
    vector<int> temp(high - low + 1);
    int i = low, j = mid + 1, k = 0;
    while (i <= mid && j <= high) {
        if (arr[i] <= arr[j]) temp[k++] = arr[i++];
        else {
            temp[k++] = arr[j++];
            inversionCount += (mid - i + 1);  // Count inversions
        }
    }
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= high) temp[k++] = arr[j++];
    for (int idx = 0; idx < temp.size(); idx++) {
        arr[low + idx] = temp[idx];
    }
}

void mergeSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}
```

7.5 Useful Constants and Macros

```cpp
const double PI = acos(-1);
const int INF = 1e9 + 7;
const ll LINF = 1e18 + 7;
const int MOD = 1e9 + 7;
const int N = 1e5 + 5;

// Common macros already in boilerplate:
// sp, nl, F, S, ll, pb, popb, gcd, lcm, all, SUM, etc.
```