Boilerplate

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

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  freopen("a.in", "r", stdin);

  // To flush for interactive problems
  cout << endl;
}
```

Run

```
#!/bin/bash
g++ $1 && ./a.out
```

Common Bugs

- Stray `freopen()` or `cout`s.
- Integer overflow.
- Out of bounds array access.
- Edge-case input like 0.

Custom Comparator

```
struct Foo {
  int a, b;
  int width;
};
bool cmp(const Foo &x, const Foo &y) {
  return x.width < y.width;
}

vector<Foo> vec;
sort(vec.begin(), vec.end(), cmp);
```

BFS

```
vector<vector<int>> adj(n);
vector<bool> visited(n);
for (int i = 0; i < n; i++) {
  if (!visited[i]) {
    queue<int> q;
    q.push(i);
    visited[i] = true;
    while (!q.empty()) {
      int current_node = q.front();
      q.pop();
      for (int neighbor : adj[current_node]) {
        if (!visited[neighbor]) {
          visited[neighbor] = true;
          q.push(neighbor);
        }
      }
    }
  }
}
```

DFS

```
vector<vector<int>> adj(n);
vector<bool> visited(n);

void dfs(int current_node) {
  if (visited[current_node]) { return; }
  visited[current_node] = true;

  for (int neighbor : adj[current_node])
{ dfs(neighbor); }
}
```

Segment Tree

```
template <class T> class MinSegmentTree {
  private:
  const T DEFAULT = std::numeric_limits<T>().max();
  int len;
  vector<T> segtree;  // index 0 is not in use
  T combine(const T &a, const T &b) { return std::min(a, b); }
  void build(const vector<T> &arr, int at, int at_left, int at_right) {
    if (at_left == at_right) {
      segtree[at] = arr[at_left];
      return;
    }
    int mid = (at_left + at_right) / 2;
    build(arr, 2 * at, at_left, mid);
    build(arr, 2 * at + 1, mid + 1, at_right);
    segtree[at] = combine(segtree[2 * at], segtree[2 * at + 1]);
  }
  void set(int ind, T val, int at, int at_left, int at_right) {
    if (at_left == at_right) {
      segtree[at] = val;
      return;
    }
    int mid = (at_left + at_right) / 2;
    if (ind <= mid) { set(ind, val, 2 * at, at_left, mid); }
    else { set(ind, val, 2 * at + 1, mid + 1, at_right); }
    segtree[at] = combine(segtree[2 * at], segtree[2 * at + 1]);
  }
  T range_min(int start, int end, int at, int at_left, int at_right) {
    if (at_right < start || end < at_left) { return DEFAULT; }
    if (start <= at_left && at_right <= end) { return segtree[at]; }
    int mid = (at_left + at_right) / 2;
    T left_res = range_min(start, end, 2 * at, at_left, mid);
    T right_res = range_min(start, end, 2 * at + 1, mid + 1, at_right);
    return combine(left_res, right_res);
  }
  public:
  MinSegmentTree(int len) : len(len) { segtree = vector<T>(len * 4, DEFAULT); };
  MinSegmentTree(const vector<T> &arr) : len(arr.size()) {
    segtree = vector<T>(len * 4, DEFAULT);
    build(arr, 1, 0, len - 1);
  }
  /** Sets the value at ind to val. */
  void set(int ind, T val) { set(ind, val, 1, 0, len - 1); }
  /** @return the minimum element in the range [start, end] */
  T range_min(int start, int end) { return range_min(start, end, 1, 0, len - 1); }
};
```

DSU

```
class DisjointSets {
  private:
  vector<int> parents;
  vector<int> sizes;

  public:
  DisjointSets(int size) : parents(size), sizes(size, 1) {
    for (int i = 0; i < size; i++) { parents[i] = i; }
  }

  /** @return the "representative" node in x's component */
  int find(int x) { return parents[x] == x ? x : (parents[x] = find(parents[x])); }

  /** @return whether the merge changed connectivity */
  bool unite(int x, int y) {
    int x_root = find(x);
    int y_root = find(y);
    if (x_root == y_root) { return false; }

    if (sizes[x_root] < sizes[y_root]) { swap(x_root, y_root); }
    sizes[x_root] += sizes[y_root];
    parents[y_root] = x_root;
    return true;
  }

  /** @return whether x and y are in the same connected component */
  bool connected(int x, int y) { return find(x) == find(y); }
};
```