National Autonomous University of México
School of Engineering
Division of Electrical Engineering

ICPC 2026 Reference

CPCFI UNAM

>:)

RacsoFractal, zum, edwardsal17

Contents

1 Template

```
// Racso programmed here
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long double ld;
typedef __int128 sll;
typedef pair<int,int> ii;
typedef pair<ll,ll> pll;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef vector<ii> vii;
typedef vector<vi> vvi;
typedef vector<vii> vvii;
typedef vector<pll> vpll;
typedef unsigned int uint;
typedef unsigned long long ull;
#define fi first
#define se second
#define pb push_back
#define all(v) v.begin(),v.end()
#define rall(v) v.rbegin(),v.rend()
#define sz(a) (int)(a.size())
#define fori(i,a,n) for(int i = a; i < n; i++)
#define endl '\n'
const int MOD = 1e9+7;
const int INFTY = INT_MAX;
const long long LLINF = LLONG_MAX;
const double EPS = DBL_EPSILON;
void printVector( auto& v ){ fori(i,0,sz(v)) cout << v[i] << " "; cout << endl; }
void fastIO() { ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0); }
```

Listing 1: Racso's template.

2 C++ Sintaxis

2.1 Compilation sentences

To compile and execute in C++:

```
g++-13 -Wall -o solucion.exe solucion.cpp
g++ -std=c++20 -Wall -o main a.cpp ./solucion.exe < input.txt > output.txt
```

To compile and execute in Java:

```
javac -Xlint Solucion.java
java Solucion < input.txt > output.txt
```

To execute in Python (two options):

```
python3 solucion.py < input.txt > output.txt
pypy3 solucion.py < input.txt > output.txt
```

2.2 Custom comparators

```
// Using function
bool cmpFunction(const pair<int,int> &a, const pair<int,int> &b) {
  return a.second < b.second; // ascending by second
```

```cpp
}
sort(all(v), cmpFunction);
// Using functor
struct CmpFunctor {
  bool operator()(const pair<int,int> &a, const pair
   <int,int> &b) const {
    return a.second < b.second; // ascending by
   second
  }
};
sort(all(v), CmpFunctor());
// Using lambda function
sort(all(v), [](const pair<int,int> &a, const pair<
   int,int> &b) {
  return a.second < b.second; // ascending by second
});
```

Listing 2: Template for custom sorting, using as example ordering a vii depending on second element.

2.3 *BS & friends

2.4 *STL DS Usage

2.5 *Strings methods

2.6 *Pragmas

2.7 Bit Manipulation Cheat Sheet

Bitwise operators:

- `&` (AND): Sets each bit to 1 if both bits are 1.
- `|` (OR): Sets each bit to 1 if al teast

one bit is 1.

- ^ (XOR): Sets each bit to 1 if bits are different.
- ∼ (NOT): Inverts all bits.
- `<<` (Left shift): Shifts bits left, fills with 0.
- `>>` (Right shift): Shifts bits right.

Basic Bit Tasks:

- Get bit: `(n & (1 << i)) != 0`
- Set bit: `n | (1 << i)`
- Clear bit: `n & ∼(1 << i)`
- Toggle bit: `n ˆ(1 << i)`
- Clear LSB: `n & (n - 1)`
- Get LSB: `n & -n`

Set Operations:

- Subset check: `(A & B) == B`
- Set union: `A | B`
- Set intersection: `A & B`
- Set difference: `A & ∼B`
- Toggle subset: `A ˆB`

Gray code: `G = B` ⊕ `(B >> 1)`

2.8 Comparing Floats

```
long double a, b, EPS = 1e-9;
if( abs(a - b) < EPs ) {
    // 'a' equals 'b'
}
```

Listing 3: Check if two real numbers are equal using an épsilon scope.

2.9 Ceil

$$\left\lceil \frac{a}{b} \right\rceil = \frac{a+b-1}{b}$$

(Originally I though $\lceil \frac{a}{b} \rceil = \frac{a-1}{b} + 1$, but this calculates the wrong way in the cases where $a = 0$)

```
int myCeil(long long a, long long b) {
    return (a + b - 1)/b;
}
```

Listing 4: A way to do ceil operation between integers without making explicit conversions. `a` and `b` are `int`, but the operation `a+b-1` can cause an overflow, so they must be casted into `long long` to avoid this. The result must be `int` anyway.

3 Miscellaneous

3.1 Binary Search

```
int binary_search( vector<int>& list, int n, int
    target ) {
    int x0 = 0, x1 = n-1, mid;
    while( x0 <= x1 ) {
        mid = (x0 + x1) / 2; // (x1 - x0) / 2 + x0;
        if( list[mid] == target )
            return mid;
        list[mid] < target ? x0 = mid + 1 : x1 = mid
    - 1;
    }
    return -1;
}
```

Listing 5: Classic (Vanilla) implementation of Binary Search. Returns the index where `target` was found. Binary Search works on $O(log_2(n))$, let n be the size of the container.

```
int binary_search( vector<int>& list, int n, int
    target ) {
    int ans = 0;
```

```cpp
    function<bool(int)> check = [&](int idx)->bool {
        return idx < n && list[idx] <= target;
    };
    for( int i = 31; i >= 0; i-- ) {
        if( check( ans + (1 << i) ) )
            ans += 1 << i;
    }
    return list[ans] == target ? ans : -1;
}
```

Listing 6: Logarithmic jumps implementation of Binary Search. Returns the index where `target` was found. Binary Search works on $O(log_2(n))$, let n be the size of the container.

```cpp
int binary_search( vector<int>& list, int n, int target ) {
  int  left = 1, right = n + 1, mid;
  while(right - left >= 1) {
    mid = left + (right - left) / 2;
    if( list[mid] >= target && target > list[mid - 1]) {
      //--------- TODO logic here ---------//
      break;
    }
    else
      if( list[mid] < target )
        left = mid + 1;
      else
        right = mid;
    }
}
```

Listing 7: Binary Search implementation for searching an element in the interval $(numbers_{i-1}, numbers_i]$. Originally used on Codeforces problem 474 - B (Worms). Returns the index where `target` was found. Binary Search works on $O(\log_2(n))$, let n be the size of the container.

```cpp
function<bool(ll)> check = [&](ll t) -> bool {
  ll products = 0;
  for(ll machine : v) {
    products += t / machine;
    if( products >= target )
      return true;
  }
  return products >= target;
};
for(int i = 0; i < 70; i++) {
  mid = (x0 + x1) / 2;
  check(mid) ? x1 = mid : x0 = mid + 1;
}
```

Listing 8: Implementation of Binary Search in the Answer. Originally used on CSES problem *Factory Machines*. Binary Search works on $O(\log_2(n))$, let n be the size of the container.

3.1.1 *Parallel Binary Search

3.1.2 *Ternary Search

3.2 Kadane's Algorithm

```cpp
ll arns = v[0], maxSum = 0;
fori(i,0,n)
{
  maxSum += v[i];
  arns = max(arns, maxSum);
  maxSum = max(0LL,maxSum);
}
```

Listing 9: Uses Kadane's Algorithm to find maximum subarray sum in $O(n)$.

4 Queries

4.1 *Prefix Sum 2D

4.2 *Sparse Table

4.3 *Sqrt Decomposition

4.4 *Fenwick Tree

4.5 *Fenwick Tree 2D

4.6 Segment Tree

```cpp
typedef long long ll;
typedef vector<ll> vll;
typedef vector<int> vi;
const int INF = INT_MAX;

class Segment_tree
{
   public: vll t;
   Segment_tree( int n = 1e5+10 ) {
      t.assign(n*4,INF);
   }
   void update(int node, int index, int tl, int tr,
   int val) {
      if( tr < index || tl > index ) return;
      if( tr == tl ) t[node] = val;
      else {
         int mid = tl + ((tr-tl)>>1);
         int lft = node << 1;
         int rght = lft + 1;
         update(lft,index,tl,mid,val);
         update(rght,index,mid+1,tr,val);
         t[node] = min(t[lft],t[rght]);
      }
   }
   ll query(int node, int l, int r, int tl, int tr)
   {
      if( tl > r || tr < l ) return INF;
      if( tl >= l and tr <= r ) return t[node];
      else {
         int mid = tl + ((tr-tl)>>1);
         int lft = node << 1;
         int rght = lft + 1;
         ll q1 = query(lft,l,r,tl,mid);
         ll q2 = query(rght,l,r,mid+1,tr);
         return min(q1,q2);
      }
   }
```

```
    void build(vi &v, int node, int tl, int tr) {
        if( tl == tr ) t[node] = v[tl];
        else {
            int mid = tl + ((tr-tl)>>1);
            int lft = node << 1;
            int rght = lft + 1;
            build(v,lft,tl,mid);
            build(v,rght,mid+1,tr);
            t[node] = min(t[lft],t[rght]);
        }
    }
};
Segment_tree st(n);
st.build(v,1,0,n-1);
st.update(1,a-1,0,n-1,b);
st.query(1,a-1,b-1,0,n-1));
```

Listing 10: Segment Tree for Dynamic Range **Minimum** Queries. Racso's Implementation.

```
vector<long long> v, sex;
int n;
void build(int node, int l, int r){
    if(l == r) sex[node] = v[l];
    else{
        int mid = (l+r)/2;
        build(2*node, l, mid);
        build(2*node + 1, mid+1, r);
        sex[node] = sex[2*node] + sex[2*node +1];
    }
}
void update(int node, int l, int r, int idx, int val
  ){
    if(l == r){
        v[idx] = val;
        sex[node] = val;
    }
    else{
        int mid = (l+r)/2;
        if(l <= idx && idx <= mid) update(2*node, l,
    mid, idx, val);
        else update(2*node +1, mid+1, r, idx, val);
        sex[node] = sex[2*node] + sex[2*node + 1];
    }
}
int query(int node, int tl, int tr, int l, int r){
    if(r < tl || tr < l) return 0;
    if(l <= tl && tr <= r) return sex[node];
    int tm = (tl+tr)/2;
    return query(2*node, tl, tm, l, r) + query(2*
  node +1, tm+1, tr, l, r);
}
v.resize(n);
sex.resize(4 * n);
build(1, 0, n - 1);
query(1, 0, n-1, l - 1, r - 1)
```

Listing 11: Segment Tree for Dynamic Range **Sum** Queries. Zum's Implementation.

4.6.1 *2D Segment Tree

4.6.2 *Persistent Segment Tree

4.6.3 *Lazy Propagation

4.7 *Ordered Set

4.7.1 Multi-Ordered Set

4.8 *Treap

4.9 *Trie

5 Graph Theory

5.1 Breadth-First Search (BFS)

```
vector<vector<int>> graph;
vector<bool> visited;
graph.assign(n, vector<int>() ); // <--- main
visited.assign(n, false); // <--- main

void bfs( int s ) {
  queue<int> q;
  q.push( s );
  visited[ s ] = true;
  while( ! q.empty() ) {
    int u = q.front();
    q.pop();
    for( auto v : graph[ u ] ) {
      if( ! visited[ u ] ) {
        visited[ u ] = true;
        q.push( v );
        // --- ToDo logic here ---
      }
    }
  }
  return;
}
```

Listing 12: Iterative implementation of BFS graph traversal over a graph represented as a AdjacencyList on vector of vectors. BFS runs in $O(|V| + |E|)$.

```
int n, m;
string arns = "";
cin >> n >> m;
vector<vector<bool>> visited(n,vector<bool>(m,false)
    );
vector<string> path(n,string(m,'0'));
vector<string> grid(n);
vii dirs = {{0,1},{0,-1},{1,0},{-1,0}};
string commands = "LRUD";
queue<ii> q;
ii start, end, curr;
function<bool(int,int)> valid = [&](int i, int j) ->
     bool {
   return ( i >= 0 && i < n && j >= 0 && j < m &&
   grid[i][j] != '#' && ! visited[i][j] );
};
fori(i,0,n) cin >> grid[i];
```

```cpp
fori(i,0,n) {
    fori(j,0,m)
        if( grid[i][j] == 'A' ) {
            visited[i][j] = true;
            q.push( {i,j} );
        }
}
while( ! q.empty() ) {
    curr = q.front();
    q.pop();
    int i = curr.fi;
    int j = curr.se;
    if( grid[i][j] == 'B' ) {
        end.fi = i;
        end.se = j;
        break;
    }
    int newI, newJ;
    fori(I,0,4) {
        newI = i + dirs[I].fi;
        newJ = j + dirs[I].se;
        if( valid(newI,newJ) ) {
            visited[newI][newJ] = true;
            q.push( {newI,newJ} );
            path[newI][newJ] = commands[I];
        }
    }
}
while( path[ end.fi ][ end.se ] != '0' ) {
    fori(i,0,4) {
        if( path[ end.fi ][ end.se ] == commands[i] )
    {
            arns += i & 1 ? commands[i-1] : commands[i
    +1];
            end.fi -= dirs[i].fi;
            end.se -= dirs[i].se;
        }
    }
}
reverse(all(arns));
if( arns == "" ) cout << "NO" << endl;
else cout << "YES" << endl << arns.size() << endl <<
     arns << endl;
```

Listing 13: BFS on Grid to find shortest path from an starting point A to an end B. Once the path is found, it reconstruct it with movements $LRUD$. Works in $O(n \cdot m)$. Originally used on problem *Labyrinth* from CSES.

5.2 Deep-First Search (DFS)

```cpp
vector<vector<int>> graph;
vector<bool> visited;
graph.assign(n, vector<int>() ); // <--- main
visited.assign(n, false); // <--- main

void dfs( int s ) {
    if( visited[s] == true ) return;
    visited[s] = true;
    vector<int>::iterator i;
    for( i = graph[s].begin(); i < graph[s].end(); ++i) {
        if( ! visited[*i] ) {
            // --- ToDo logic here ---
            dfs(*i);
        }
    }
}
```

Listing 14: Recursive implementation of DFS graph traversal over a graph represented as a AdjacencyList on vector of vectors. DFS runs in $O(|V| + |E|)$.

```cpp
vector<vector<int>> graph;
vector<bool> visited;
void dfs( int s ) {
    stack<int> stk;
    stk.push(s);
    while (!stk.empty()) {
        int u = stk.top();
        stk.pop();
        if ( visited[u] ) continue;
        visited[u] = true;
        // --- ToDo logic here ---
        for(auto it = graph[u].rbegin(); it != graph[u].rend(); ++it)
            if (!visited[*it])
                stk.push(*it);
        }
}
```

Listing 15: Iterative implementation of DFS graph traversal over a graph represented as a AdjacencyList on vector of vectors. DFS runs in $O(|V| + |E|)$.

```cpp
typedef long long ll;
vector<vector<ll>> adj;
vector<bool> visited;
function<void(ll)> dfs = [&](ll u) -> void {
    if( visited[u] ) return;
    visited[u] = true;
    for( ll v : adj[u] )
        dfs(v);
};
dfs(n);
```

Listing 16: DFS implementation with a lambda function (adjacency list and visited don't need to be passed thorough argument). DFS runs in $O(|V| + |E|)$.

```cpp
typedef long long ll;
typedef vector<ll> vll;
map<ll,vll> adj;
set<ll> visited;
function<void(ll)> dfs = [&](ll u) -> void {
    if( visited.count(u) ) return;
    visited.insert(u);
    for( ll v : adj[u] )
        dfs(v);
};
```

```cpp
dfs(n);
```

Listing 17: DFS implementation with a lambda function implemented with a map instead of vector of vectors, and a set to track visited nodes. DFS runs in $O(|V| + |E|)$.

5.3 Shortest Path

5.3.1 Dijkstra's Algorithm

```cpp
typedef long long ll;
typedef pair<ll,ll> pll;

vector<vector<ll>> graph;
vector<ll> visited;
graph.assign(n, vector<ll>() ); // <--- main
visited.assign(n, false); // <--- main

vector<ll> dijkstra( int n, int source, vector<
    vector<pll>> &graph ) {
  vector<ll> dist( n, INFTY );
  priority_queue<pll, vector<pll>, greater<pll>> pq;
  dist[ source ] = 0;
  pq.push( {0, source} );
  while( ! pq.empty() ) {
    ll d = pq.top().first;
    ll u = pq.top().second;
    pq.pop();
    if( d > dist[ u ] ) continue;
    for( auto &edge : graph[ u ] ) {
      ll v = edge.first;
      ll weight = edge.second;
      if( dist[ u ] + weight < dist[ v ] ) {
        dist[ v ] = dist[ u ] + weight;
        pq.push( {dist[ v ], v} );
      }
    }
  }
  return dist;
}
```

Listing 18: Iterative implementation of Dijkstra's Algorithm for shortest path over a graph represented as a AdjacencyList on vector of vectors. Returns a vector with the shortest path to every other vertex in the graph. $O(|E| \times log_2(|v|))$. doesn't work with negative weights.

```cpp
vvpll graph(n+1,vpll());
vector<bool> visited(n+1,false);
function<vll(int)> dijkstra = [&](int source) -> vll
    {
  vll dist(n+1,INF);
  priority_queue<pll,vpll,greater<pll>> pq;
  dist[source] = 0;
  pq.push({0,source});
  while( ! pq.empty() ) {
    ll d = pq.top().fi;
    ll u = pq.top().se;
    pq.pop();
    if( d > dist[u] ) continue;
```

```
        for(pll edge : graph[u]) {
            ll v = edge.fi;
            ll w = edge.se;
            if( dist[u] + w < dist[v] ) {
                dist[v] = dist[u] + w;
                pq.push({dist[v],v});
            }
        }
    }
    return dist;
};
```

Listing 19: Iterative implementation of Dijkstra's Algorithm as a Lambda Function for shortest path over a graph represented as a AdjacencyList on vector of vectors. Returns a vector with the shortest path to every other vertex in the graph. $O(|E| \times log_2(|V|))$. Doesn't work with negative weights.

5.3.2 *Floyd-Warshall's Algorithm

5.3.3 Bellman–Ford Algorithm

```
int V, E;
cin >> V >> E;
vvi edges(E,vi(3,0));

for(int i = 1; i <= E; i++)
  cin >> edges[i][0] >> edges[i][1] >> edges[i][2];

function<vi(int)> bellman_ford = [&](int src) -> vi
{
  vi dist(V,INF);
  dist[src] = 0;
  for(int i = 0; i < V; i++)
  {
    for( vi edge : edges )
    {
      int u = edge[0];
      int v = edge[1];
      int w = edge[2];
      if( dist[u] != INF and dist[u] + w < dist[v]
    )
      {
        if( i == V-1 )
          return = {-1};
        dist[v] = dist[u] + w;
      }
    }
  }
  return dist;
};
```

Listing 20: Finds the shortest route from a source vertex, to every other one in the graph. Works over a list of edges. Runs in $O(|V| \times |E|)$. Can be used to find negative cycles.

5.4 Minimum Spanning Tree (MST)

5.4.1 *Prim's Algorithm

5.4.2 *Kruskal's Algorithm

5.5 *Bipartite Checking

5.6 *Negative Cycles

5.7 Topological Sort

```cpp
vi topo;
vvi graph(V+1,vi());
vector<bool> visited(V+1,false);
function<void(int)> dfs = [&](int u) -> void {
    visited[u] = true;
    for(int v : graph[u])
        if( ! visited[v] )
            dfs(v);
    topo.pb(u);
};
function<void()> topological_sort = [&]() -> void {
    for(int i = 1; i <= V; i++)
        if( ! visited[i] )
            dfs(i);
    reverse(all(topo));
};
topological_sort();
for(int i = 0; i < V; i++) cout << topo[i] << " ";
```

Listing 21: Recursive toposort implementation for unweighted DAG through vvi with DFS with inverted postorder. Runs in $O(|V| \times |E|)$.

```cpp
vi indegree(V+1,0);
vvi graph(V+1,vi());
vector<bool> visited(V+1,false);
fori(i,0,E) {
    graph[u].pb(v);
    indegree[v]++;
}
function<vi()> topological_sort = [&]() -> vi {
  vi order, deg = indegree; // copy
  queue<int> q;
  for(int i = 1; i <= V; i++)
    if( deg[i] == 0 )
      q.push(i);
  while( ! q.empty() ) {
    int u = q.front(); q.pop();
    order.pb(u);
    for(int v : graph[u]) {
      deg[v]--;
      if(deg[v] == 0)
        q.push(v);
    }
  }
  return order;
};
vi topo = topological_sort();
if( (int)(topo.size()) != V ) cout << "IMPOSSIBLE"
    << endl;
```

Listing 22: Kahn's Algorithm for Topological Sorting using BFS and indegree vertex analysis (nodes in a cycle will never have indegree zero). Works over unweighted directed graphs containing cycles through vvi. Runs in $O(|V| \times |E|)$.

5.8 *Disjoint Set Union (DSU)

5.9 *Condensation Graph

5.10 *Strongly Connected Components (SCC)

5.11 *2-SAT

5.12 *Bridges and point articulation

5.13 Flood Fill

```
vector<string> grid(n);
vii dirs = {{0,1},{0,-1},{1,0},{-1,0}};
ii start;
int arns = 0;
function<void(int,int)> traverse = [&](int i, int j)
     -> void {
  if( grid[i][j] == '#' ) return;
  int newI, newJ;
  if( grid[i][j] != '.' ) arns += grid[i][j] - '0';
  grid[i][j] = '#';
  for( ii move : dirs ) {
    newI = i + move.fi; newJ = j + move.se;
    if( newI >= 0 && newI < n && newJ >= 0 && newJ <
    m && grid[newI][newJ] == 'T' )
      return;
  }
  for( ii move : dirs ) {
    newI = i + move.fi; newJ = j + move.se;
    if( newI >= 0 && newI < n && newJ >= 0 && newJ <
    m )
      traverse(newI, newJ);
  }
};
fori(i,0,n)
  cin >> grid[i];
fori(i,0,n) {
  fori(j,0,m) {
    if( grid[i][j] == 'S' ) {
      grid[i][j] = '.';
      start.fi = i;
      start.se = j;
    }
  }
}
traverse(start.fi, start.se);
  cout << arns << endl;
```

Listing 23: Traverse a matrix of 'n' x 'm' on grid representation. The matrix is composed of '.' for a valid space (empty), '#' for a wall, 'T' for a trap, and a number for a treasure. This implementation takes the sum of every treasure in the maze. The condition for moving to the next location is that there are no Traps nearby (up, down, left, right), so the player will never be killed while traversing. It also implements a way to read numerous test cases, but without knowing beforehand how many there are. Runs in $O(n \cdot m)$. Originally used on the problem *Treasures* from *2024-2025 ICPC Bolivia Pre-National Contest.*

5.14 Lava Flow (Multi-source BFS)

```cpp
typedef array<int,3> iii;

vii dirs = {{1,0},{0,1},{-1,0},{0,-1}};
map<int,string> path = {{0,"D"},{1,"R"},{2,"U"},{3,"
    L"}};
int n, m;
string arns = "";
bool escaped = false;
cin >> n >> m;
vector<string> grid(n);
vvi times(n,vi(m,INF)), prev(n,vi(m,-1));
vector<vector<bool>> visited(n,vector<bool>(m,false)
    );
queue<iii> q;
ii start, end;
for(int i = 0; i < n; i++) {
  cin >> grid[i];
  for(int j = 0; j < m; j++) {
    if( grid[i][j] == 'M' ) {
      q.push({i,j,0});
      times[i][j] = 0;
    }
    else if( grid[i][j] == 'A' )
      start = {i,j};
  }
}
function<bool(int,int)> valid = [&](int I, int J) ->
    bool {
  return (I >= 0) and (I < n) and (J >= 0 ) and (J <
    m) and (grid[I][J] != '#') and (times[I][J] ==
   INF);
};
function<bool(int,int)> valid_player = [&](int I,
   int J) -> bool {
  return (I >= 0) and (I < n) and (J >= 0) and (J <
   m) and (!visited[I][J]) and (grid[I][J] != '#');
};
function<bool(int,int)> is_border = [&](int I, int J
   ) -> bool {
  return I == 0 || I == n-1 || J == 0 || J == m-1;
};
// Corner cases
if( is_border(start.fi,start.se) ) {
  cout << "YES" << endl << "0" << endl;
  return 0;
}
// Multi-Source BFS
while( ! q.empty() ) {
  iii u = q.front();
  q.pop();
  for(ii dir : dirs) {
    int newI = u[0] + dir.fi;
    int newJ = u[1] + dir.se;
    int w = u[2] + 1;
    if( valid(newI,newJ) ) {
      times[newI][newJ] = w;
      q.push({newI,newJ,w});
    }
  }
}
// Player BFS
q.push({start.fi,start.se,0});
visited[start.fi][start.se] = true;
while( ! q.empty() and !escaped ) {
  iii u = q.front();
  q.pop();
  for(int i = 0; i < 4; i++) {
    int newI = u[0] + dirs[i].fi;
    int newJ = u[1] + dirs[i].se;
    int w = u[2] + 1;
    if( valid_player(newI,newJ) and w < times[newI][
    newJ] ) {
      visited[newI][newJ] = true;
      prev[newI][newJ] = i;
      q.push({newI,newJ,w});
      if( is_border(newI,newJ) ) {
        end.fi = newI;
        end.se = newJ;
        escaped = true;
        break;
      }
    }
  }
}
if( !escaped ) {
  cout << "NO" << endl;
  return 0;
}
// Path reconstruction
cout << "YES" << endl;
int i = end.fi;
int j = end.se;
while( prev[i][j] != -1 ) {
  int oldI = i;
  arns += path[ prev[i][j] ];
  i -= dirs[prev[i][j]].fi;
  j -= dirs[prev[oldI][j]].se;
}
reverse(all(arns));
cout << sz(arns) << endl << arns << endl;
```

Listing 24: Classic Lava Flow problem implementation, where the timer from the starting point A needs to be less than every other in the MS-BFS starting in M places. Once one edge is reached, the path is reconstructed from the output. Runs in BFS complexity $O(|V| + |E|)$. Originally used in the CSES problem *Monsters*.

5.15 MaxFlow

5.15.1 Dinic's Algorihtm

```cpp
const ll INF = 1e17;
/**
 * @brief Represents a directed edge in a flow
   network.
 * @details Stores the edge's source, destination,
   capacity, and current flow.
 *          Used in max-flow algorithms like Dinic
   or Ford-Fulkerson. */
struct flowEdge {
  int u; // Source node
  int v; // Destination node
  ll cap; // Maximum flow capacity of the edge
  ll flow = 0; // Current flow through the edge (
   initially 0)
  flowEdge( int u, int v, ll cap ) : u(u), v(v), cap
   (cap) {};
};
/**
 * @brief Implementation of Dinic's max-flow
   algorithm.
 * @details Manages a flow network with BFS (Level
   Graph) and DFS (Blocking Flow) optimizations. */
struct Dinic {
  vector<flowEdge> edges; // All edges in the flow
   network (including reverse edges)
  vector<vi> adj;
  int n; // Total number of nodes in the graph
  int s; // Source node
  int t; // Sink node (destination of flow)
  int id = 0; // Counter for edge indexing
  vi level; // Stores the level (distance from 's')
   of each node during BFS
  vi next; // Optimization for DFS: tracks the next
   edge to explore for each node
  queue<int> q; // Queue for BFS traversal
  /**
   * @brief Constructs a Dinic solver for a flow
   network.
   * @param n Number of nodes.
   * @param s Source node.
   * @param t Sink node. */
  Dinic( int n, int s, int t ) : n(n), s(s), t(t) {
    adj.resize(n); // Initialize adjacency list for
   `n` nodes.
    level.resize(n); // Prepare level array for BFS.
  next.resize(n); // Prepare next-edge array for DFS.
  fill(all(level),-1); // Mark all levels as unvisited (-1).
  level[s] = 0; // The source has level 0.
  q.push(s); // Start BFS from the source.
}
/**
 * @brief Adds a directed edge and its residual reverse edge to the flow network. */
void addEdge( int u, int v, ll cap ) {
  edges.emplace_back(u,v,cap); // Original edge: u -> v
  edges.emplace_back(v,u,0); // Residual edge: v -> u
  adj[u].pb(id++);
  adj[v].pb(id++);
}
/**
  * @brief Performs BFS to construct the level graph (Layered Network) from source 's' to sink 't'.
  * @details Assigns levels (minimum distances from 's') to all nodes and checks if 't' is reachable.
  *          Levels are used to guide the DFS phase in Dinic's algorithm.
  * @return bool True if the sink 't' is reachable (i.e., there exists an augmenting path), false otherwise. */
bool bfs() {
  while( ! q.empty() ) {
    int curr = q.front();
    q.pop();
    for( auto e : adj[curr] ) {
      if( edges[e].cap - edges[e].flow < 1 ) // Skip saturated edges (no residual capacity).
        continue;
      if( level[ edges[e].v ] != -1 ) // Skip already visited nodes (level assigned).
        continue;
      // Assign level to the neighbor node.
      level[ edges[e].v ] = level[ edges[e].u ] + 1; // Next level = current + 1.
      q.push( edges[e].v ); // Add neighbor to the queue for further BFS.
    }
  }
  return level[t] != -1; // Return whether the sink 't' was reached (level[t] != -1).
}
/**
  * @brief Finds a blocking flow using DFS in the level graph constructed by BFS.
  * @param u Current node being processed.
```

```
   * @param flow Maximum flow that can be sent from
    'u' to the sink 't'.
   * @return ll The amount of flow successfully
  sent to 't'. */
 ll dfs( int u, ll flow ) {
   if( flow == 0 ) // No remaining flow to send.
     return 0;
   if( u == t )    // Reached the sink; return
  accumulated flow.
     return flow;
   // Explore edges from 'u' using 'next[u]' to
  avoid revisiting processed edges.
   for( int& cid = next[u]; cid < sz(adj[u]); cid++
    ) {
     int e = adj[u][cid]; // Index of the edge in '
  edges'.
     int v = edges[e].v;  // Destination node of
  the edge.
     // Skip invalid edges:
     // 1. Not in the level graph (level[u] + 1 !=
  level[v]). Just edges in exactly one leve ahead (
  ensures shortest paths).
     // 2. No residual capacity (cap - flow < 1).
     if( level[edges[e].u] + 1 != level[v] || edges
  [e].cap - edges[e].flow < 1 )
       continue;
     ll f = dfs( v, min(flow, edges[e].cap - edges[
  e].flow ) ); // Recursively send flow to 'v'.
     if( f == 0 ) // No flow could be sent via this
   edge.
       continue;
     // Update residual capacities:
     edges[e].flow += f;        // Increase flow in
  the original edge.
     edges[ e ^ 1 ].flow -= f; // Decrease flow in
  the reverse edge. (All reverse edges have
  distinct parity)
     return f;                  // Return the flow
  sent.
   }
   return 0; // No augmenting path found from 'u'.
 }
 /**
   * @brief Computes the maximum flow from source '
  s' to sink 't' using Dinic's algorithm.
   * @details Iterates through BFS and DFS phases
  to find the maximum flow.
       Accumulates flow while there exists
  augmention paths in the residual graph.
       Restart auxiliary structures for every new
  phase.
   * @return ll The maximum flow value. */
 ll maxFlow() {
   ll flow = 0; // Tracks the total flow sent.
   while( bfs() ) { // While there are augmenting
  paths:
      fill(all(next),0); // Reset 'next' for DFS.
      for( ll f = dfs(s,INF); f != 0ll; f = dfs(s,INF) ) // Send blocking flow in the level graph:
        flow += f;
      // Reset for next BFS phase:
      fill(all(level),-1);
      level[s] = 0;
      q.push(s);
    }
    return flow;
  }
  /**
   * @brief Finds edges belonging to the minimum cut after maxFlow().
   * @details First, it marks all the reachable nodes from 's' with an augmention path after obtained the max flow
       and all the saturated edges coming out from any of the nodes who belong to the min-cut.
       For 'minCut()' to work, 'maxFlow()' must be first executed to get the min-cut.
       If only is needed the value, is enough returning the value of 'maxFlow()'.
   * @return vii List of edges (u, v) in the min-cut. Its size is the minimum number of 'roads' to close. */
  vii minCut() {
    vii ans;
    fill(all(level),-1); // Reset levels.
    level[s] = 0;         // Mark source as reachable.
    q.push(s);
    while( ! q.empty() ) { // BFS to mark nodes reachable from 's' in the residual graph.
      int curr = q.front();
      q.pop();
      for( int id = 0; id < sz(adj[curr]); id++ ) { // For every edge going out from 'curr'.
        int e = adj[curr][id];
        // If 'v' is has not been visited yet, and the edge have residual capacity.
        if( level[edges[e].v] == -1 && edges[e].cap - edges[e].flow > 0 ) {
          q.push(edges[e].v);
          level[edges[e].v] = level[edges[e].u] + 1;
        }
      }
    }
    for( int i = 0; i < sz(level); i++ ) {
      if( level[i] != -1 ) {
        for( int id = 0; id < sz(adj[i]); id++ ) {
          int e = adj[i][id];
          if( level[edges[e].v] == -1 && edges[e].cap - edges[e].flow == 0 )
            ans.emplace_back(edges[e].u,edges[e].v);
        }
```

```cpp
      }
    }
    return ans;
  }
  /**
    * @brief Reconstructs the maximum bipartite matching after running 'maxFlow()'.
    * @details Every edge that belong to the original graph and have flow greater than zero, belongs to the matching.
        For 'maximumMatching()' to work, 'maxFlow()'.
    * @return vii List of matched pairs (boy, girl).
    */
  vii maximumMatching() {
    vii ans;
    fill(all(level),-1); // Reset levels.
    level[s] = 0;        // Mark source as reachable.
    q.push(s);
    while( ! q.empty() ) { // BFS to mark nodes reachable via saturated edges with flow greater than zero.
      int curr = q.front();
      q.pop();
      for( int id = 0; id < sz(adj[curr]); id++ ) {
        int e = adj[curr][id];
        // If 'v' has not been visited yet, the edge is saturated and have flow greater than zero.
        if( level[edges[e].v] == -1 && edges[e].cap - edges[e].flow == 0 && edges[e].flow != 0ll ) {
          q.push(edges[e].v);
          level[edges[e].v] = level[edges[e].u] + 1;
        }
      }
    }
    for( int i = 0; i < sz(level); i++ ) { // Collect original edges (boy -> girl) that are saturated and have flow > 0.
      if( level[i] != -1 ) {
        for( int id = 0; id < sz(adj[i]); id++ ) {
          int e = adj[i][id];
            if( edges[e].u != s && edges[e].v != t && edges[e].cap - edges[e].flow == 0 && edges[e].flow != 0ll )
              ans.emplace_back(edges[e].u,edges[e].v);
        }
      }
    }
    return ans;
  }
};
```

Listing 25: Commented template for solving MaxFlow problems with Dinic's algorithm. Works in complexity $O(|V|^2 \times |E|)$. In bipartite graphs and graphs with unitary max capacity the complexity turns $O(|E| \times \sqrt{|V|})$.

Download Speed How does a particular flow problem looks like? Following is the statement for the problem CSES 1694 (Download Speed):

Consider a network consisting of n computers and m connections. Each connection specifies how fast a computer can send data to another computer.

Kotivalo wants to download some data from a server. What is the maximum speed he can do this, using the connections in the network?

Input

The first input line has two integers n and m: the number of computers and connections. The computers are numbered $1, 2, \ldots, n$. Computer 1 is the server and computer n is Kotivalo's computer.

After this, there are m lines describing the connections. Each line has three integers a, b, and c: computer a can send data to computer b at speed c.

Output

Print one integer: the maximum speed Kotivalo can download data.

```cpp
int main()
{
    fastIO();

    int n, m, u, v, w;

    cin >> n >> m;

    Dinic flow(n+1,1,n); // size n+1 to fix 0-
        indexed indexes, 1 is the source (server), 'n' is
        the sink (Kotivalo)

    fori(i,0,m)
    {
        cin >> u >> v >> w;
        flow.addEdge(u,v,w);
    }

    cout << flow.maxFlow() << endl;

    return 0;
}
```

Listing 26: Main method for solving CSES 1697 Download Speed using MaxFlow template.

Police Chase **Max Flow-Min Cut Theorem:** MaxFlow = MinCut.
Following is the statement for the problem CSES 1695 (Police Chase):

Kaaleppi has just robbed a bank and is now heading to the harbor. However, the police wants to stop him by closing some streets of the city.

What is the minimum number of streets that should be closed so that there is no route between the bank and the harbor?

Input

The first input line has two integers n and m: the number of crossings and streets. The crossings are numbered $1, 2, \ldots, n$. The bank is located at crossing 1, and the harbor is located at crossing n.

After this, there are m lines that describing the streets. Each line has two integers a and b: there is a street between crossings a and b. All streets are

two-way streets, and there is at most one street between two crossings.
Output
First print an integer k: the minimum number of streets that should be closed. After this, print k lines describing the streets. You can print any valid solution.

```cpp
int main()
{
    fastIO();

    int n, m, u, v;
    vii minCut;

    cin >> n >> m;

    Dinic flow(n+1,1,n); // size n+1 to fix 0-indexed indexes, 1 is the source (bank), 'n' is the sink (harbor)

    fori(i,0,m)
    {
        cin >> u >> v;
        flow.addEdge(u,v,1);
        flow.addEdge(v,u,1);
    }

    flow.maxFlow();
    minCut = flow.minCut();

    cout << (sz(minCut)/2) << endl;
    for(int i = 0; i < sz(minCut); i += 2)
        cout << minCut[i].fi << " " << minCut[i].se << endl;

    return 0;
}
```

Listing 27: Main method for solving CSES 1695 Police Chase using MaxFlow template.

School Dance MaxFlow = MinCut = MaxMatching.
Following is the statement for the problem CSES 1696 (School Dance):

There are n boys and m girls in a school. Next week a school dance will be organized. A dance pair consists of a boy and a girl, and there are k potential pairs.

Your task is to find out the maximum number of dance pairs and show how this number can be achieved.
Input
The first input line has three integers n, m and k: the number of boys, girls, and potential pairs. The boys are numbered $1, 2, \ldots, n$, and the girls are numbered $1, 2, \ldots, m$.

After this, there are k lines describing the potential pairs. Each line has two integers a and b: boy a and girl b are willing to dance together.
Output
First print one integer r: the maximum number of dance pairs. After this, print r lines describing the pairs. You can print any valid solution.

```cpp
int main()
{
    fastIO();

    int n, m, k, a, b;
    ll maxPairs;
    vii pairs;

    cin >> n >> m >> k;

    Dinic flow(n+m+2,0,n+m+1);

    fori(boy,0,n+1)
        flow.addEdge(0,boy,1);

    fori(girl,n+1,n+m+1)
        flow.addEdge(girl,n+m+1,1);

    fori(i,0,k)
    {
        cin >> a >> b;
        flow.addEdge(a,n+b,1);
    }

    maxPairs = flow.maxFlow();
    pairs = flow.maximumMatching();

    cout << maxPairs << endl;
    fori(i,0,sz(pairs))
        cout << pairs[i].fi << " " << (pairs[i].se -
n) << endl;;

    return 0;
}
```

Listing 28: Main method for solving CSES 1696 School Dancing using MaxFlow template.

5.15.2 *Ford-Fulkerson Algorithm

5.15.3 *Goldber-Tarjan Algorithm

6 Trees

6.1 Counting Childrens

```
vi childrens(n+1,0);
vvi graph(n+1);
vector<bool> visited(n+1, false);
fori(i,2,n+1) {
   cin >> tmp;
   graph[tmp].pb(i);
   graph[i].pb(tmp);
}
function<int(int)> dfs = [&](int u) -> int {
   visited[u] = true;
   for(int v : graph[u]) {
      if( !visited[v] )
         childrens[u] += dfs(v);
   }
   return childrens[u] + 1;
};
```

```
dfs(1);
```

Listing 29: Algorithm that counts how many childrens does every node have, from 2..n in a rooted tree (root = 1).

6.2 *Tree Diameter

6.3 *Centroid Decomposition

6.4 *Euler Tour

6.5 *Lowest Common Ancestor (LCA)

6.6 *Heavy-Light Decomposition (HLD)

7 Strings

7.1 *Knuth-Morris-Pratt Algorithm

7.2 *Suffix Array

7.3 *Rolling Hashing

7.4 *Z Function

7.5 *Aho-Corasick Algorithm

8 Dynamic Programming

8.1 *Coins

8.2 *Longest Increasing Subsequence (LIS)

8.3 *Edit Distance

8.4 *Knapsack

8.5 *SOS DP

8.6 *Digit DP

8.7 *Bitmask DP

9 Mathematics

9.1 Number Theory

9.1.1 Greatest Common Divisor (GCD)

```cpp
int gcd(int a, int b) {
  if (a == 0) return b;
  if (b == 0) return a;
  if (a == b) return a;
  if (a > b)
    return gcd(a - b, b);
  return gcd(a, b - a);
}
```

Listing 30: Implementation of handmade GCD, because using `gcd()` runs slow with long long, also `__gcd()`.

9.1.2 Gauss Sum

The sum of the first n natural numbers in $O(1)$.

$$S = \frac{n\,(n+1)}{2} \tag{1}$$

$$n = \sqrt{2S + \frac{1}{4}} - \frac{1}{2} \tag{2}$$

```cpp
int S = (1LL * n * (1LL * n + 1LL))/2;
int n = (int)( sqrt( 2 * S + 0.25 ) - 0.5 )
```

Listing 31: Implementation of the Gauss Sum.

9.1.3 *Modular Theory

9.1.4 *Modulo Inverse

9.1.5 *Fermat's Little Theorem

9.1.6 *Chinese Remainder Theorem

9.1.7 Binpow

```cpp
const int MOD = 1e9+7;
int binpow( long long a, long long b ) { // a^b
    long long sol = 1;
    a %= MOD;
    while( b > 0 ) {
        if( b & 1 )
            sol = ( 1LL * sol * a ) % MOD;
        a = ( 1LL * a * a ) % MOD;
        b >>= 1;
    }
    return sol % MOD;
}
```

Listing 32: Applying binary exponentiation to a problem requiring $a^b\, mod(10^9+7)$ in $O(log_2(b))$.

9.1.8 Matrix Exponentiation (Linear Recurrency)

```cpp
template <typename T> void matmul(vector<vector<T>>
    &a, const vector<vector<T>>& b) {
    size_t n = a.size(), m = a[0].size(), p = b[0].
    size();
    assert(m == b.size());
```

```cpp
   vector<vector<T>> c(n,vector<T>(p));
   for(size_t i = 0; i < n; i++)
      for(size_t j = 0; j < p; j++)
         for(size_t k = 0; k < m; k++)
            c[i][j] = (c[i][j] + a[i][k] * b[k][j]) % MOD;
   a = c;
}
template <typename T> struct Matrix {
   vector<vector<T>> mat;
   Matrix() {}
   Matrix(vector<vector<T>> a) { mat = a; }
   Matrix(int n, int m) {
      mat.resize(n);
      for(int i = 0; i < n; i++) {mat[i].resize(m); }
   }
   int rows() const { return mat.size(); }
   int cols() const { return mat[0].size(); }
   void makeIden() {
      for(int i = 0; i < rows(); i++)
         for(int j = 0; j < cols(); j++)
            mat[i][j] = (i == j ? 1 : 0);
   }
   Matrix operator*=(const Matrix &b) {
      matmul(mat, b.mat);
      return *this;
   }
   void print() {
      for(int i = 0; i < rows(); i++) {
         for(int j = 0; j < cols(); j++)
            cout << mat[i][j] << " ";
         cout << endl;
      }
   }
   Matrix operator*(const Matrix &b) { return Matrix (*this) *= b; }
};
int main() {
   Matrix<ll> A( {{1,1},{1,0}} );
   Matrix<ll> ini(2,1);
   ini.mat[0][0] = 0;
   ini.mat[1][0] = 1;
   Matrix<ll> iden(2,2);
   iden.makeIden();
   ll n;
   cin >> n;
   while(n > 0) {
      if( n & 1 ) iden *= A;
      A *= A;
      n >>= 1;
   }
   Matrix<ll> res = iden * ini;
   cout << res.mat[0][0] << endl;
   return 0;
}
```

Listing 33: Template to pow a matrix of size n to a certain exponent with logarithmic time (using binpow), and multiply it to another matrix, with modulo operation, as well as how to use it. Full implementation for calculating `n-th` Fibonacci term with linear recurrency.

9.1.9 Prime checking

```cpp
bool prime( int n ){
    if( n == 2 )
        return true;
    if( n % 2 == 0 || n <= 1 )
        return false;
    for( int i = 3; i * i <= n; i += 2 )
            if( ( n % i ) == 0 )
                return false;
    return true;
}
```

Listing 34: Returns if n is a prime number in $O(\sqrt{n})$. Avoids overflow $\forall\ n \leq 10^6$ ($\approx INT\_MAX$).

9.1.10 Prime factorization

```cpp
void prime_factorization(vll& factorization, ll n) {
    for(long long d = 2; d*d <= n; d++) {
        while(n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if( n > 1 )
        factorization.push_back(n);
}
```

Listing 35: Returns prime factorization of the number n using *trial division*, simplest way. Runs in $O(\sqrt{n})$. e.g. for 12 the result is 2x2x3.

9.1.11 Sieve of Eratosthenes

```cpp
void sieve_of_eratosthenes(vector<bool>& is_prime,
   int n) {
    is_prime.assign(n+1,true);
    is_prime[0] = is_prime[1] = false;
    for(int i = 2; i <= n; i++) {
        if( is_prime[i] && (long long)i * i <= n ) {
            for(int j = i*i; j <= n; j += i)
            is_prime[j] = false;
        }
    }
}
```

Listing 36: Calculates every prime number up to n with sieve of eratosthenes in a boolean 1-indexed vector. Runs in $O(n\ log\ log\ n)$.

9.1.12 Sum of Divisors

```cpp
ll sum_of_divisors(ll n) {
  ll sum = 1;
  for (long long i = 2; i * i <= n; i++) {
    if(n % i == 0) {
      int e = 0;
      do {
        e++;
        n /= i;
      } while (n % i == 0);
      ll s = 0, pow = 1;
      do {
        s += pow;
        pow *= i;
      } while (e-- > 0);
      sum *= s;
    }
  }
  if(n > 1)
    sum *= (1 + n);
  return sum;
}
```

Listing 37: Calculates the sum of all divisors of number n. e.g. $sum\_of\_divisors(12) = 18$. Runs in $O(\sqrt{n})$.

```cpp
void sum_of_divisors_sieve( vll& sigma, int n ) {
    sigma.assign(n+1,0);
    for(int i = 1; i <= n; i++)
        for(int j = i; j <= n; j+=i)
            sigma[j] += i;
}
```

Listing 38: Calculates the sum of all divisors of all numbers from 1 to n. Runs in $O(n\,log\,(n))$.

9.2 Combinatorics

9.2.1 Binomial Coefficients

```cpp
const int MAXN = 1e6+1;
vll fact(MAXN+1), inv(MAXN+1);
int binpow( ll a, ll b ) { // a^b
    ll sol = 1;
    a %= MOD;
    while( b > 0 ) {
        if( b & 1 )
            sol = ( 1LL * sol * a ) % MOD;
        a = ( 1LL * a * a ) % MOD;
        b >>= 1;
    }
    return sol % MOD;
}
void combi() {
    fact[0] = inv[0] = 1;
    fori(i,1,MAXN+1) {
        fact[i] = fact[i-1] * i % MOD;
        inv[i] = binpow( fact[i], MOD - 2 );
    }
}
ll nCr( ll n, ll r ) {
    return fact[n] * inv[r] % MOD * inv[n-r] % MOD;
}
combi();
nCr(a,b);
```

Listing 39: Template for calculating binomial coefficients $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$. Precalculate *fact* and *inv* runs in $O(MAXN \cdot log_2(MOD))$ ($log_2(MOD) \approx 30$). So, in general case when $NMAX = 10^6$ and $MOD = 10^9 + 7$ can be generalizad to $O(n \cdot log(n))$, $n \leq 10^6$.

9.2.2 Common combinatorics formulas

$$\binom{n}{2} = \frac{n\,(n-1)}{2} \tag{3}$$

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \tag{4}$$

$$\sum_{k=0}^{n} \binom{n}{k}\binom{n}{n-k} = \binom{2n}{n} \tag{5}$$

$$\sum_{k=0}^{n} k\binom{n}{k} = n2^{n-1} \tag{6}$$

$$\sum_{k=0}^{\infty} \binom{2k}{k}\binom{2n-2k}{n-k} = 4^n \tag{7}$$

$$\tag{8}$$

9.3 *Stars and Bars

9.4 Probability

9.5 Computational Geometry

9.5.1 *Cross Product

9.5.2 *Convex Hull

9.6 *Fast Fourier Transform (FFT)

10 Appendix

10.1 What to do against WA?

1. Have you done the correct complexity analysis?
2. Have you understood well the statement?
3. Have you corroborated yet the trivial test cases?
4. Have you checked all the corner cases?
5. Have you proposed a lot of non-trivial test cases?
6. Isn't there any posisbilitie of overflow? (Multiplying two `int` needs to be fitted into a `long long`)
7. Have you done a desktop test?
8. Have you red all the variables? (`tc` variable on `main`)
9. Every part of your code works as it's meant to?

10.2 Primitive sizes

Data type	[B]	Minimun value it takes	Maximum value it takes
bool	1	0	1
signed char	1	0	255
unsigned char	1	-128	127
signed int	4	-2,147,483,648 $\approx -2 \times 10^9$	2,147,483,647 $\approx 2 \times 10^9$
unsigned int	4	0	4,294,967,295 $\approx 4 \times 10^9$
signed short	2	-32,768	32,767
unsigned short	2	0	65,535
signed long long int	8	-9,223,372,036,854,775,808 $\approx -9 \times 10^{18}$	9,223,372,036,854,775,807 $\approx 9 \times 10^{18}$
unsigned long long int	8	0	18,446,744,073,709,551,615 $\approx 18 \times 10^{18}$
float	4	1.1×10^{-38}	3.4×10^{38}
double	8	2.2×10^{-308}	1.7×10^{308}
long double	12	3.3×10^{-4932}	1.1×10^{4932}

Table 1: Capacity of primitive data types in C++.

10.3 *ASCII table

10.4 *Numbers bit representation

10.5 How a `vector<vector<pair<int,int>>>` looks like

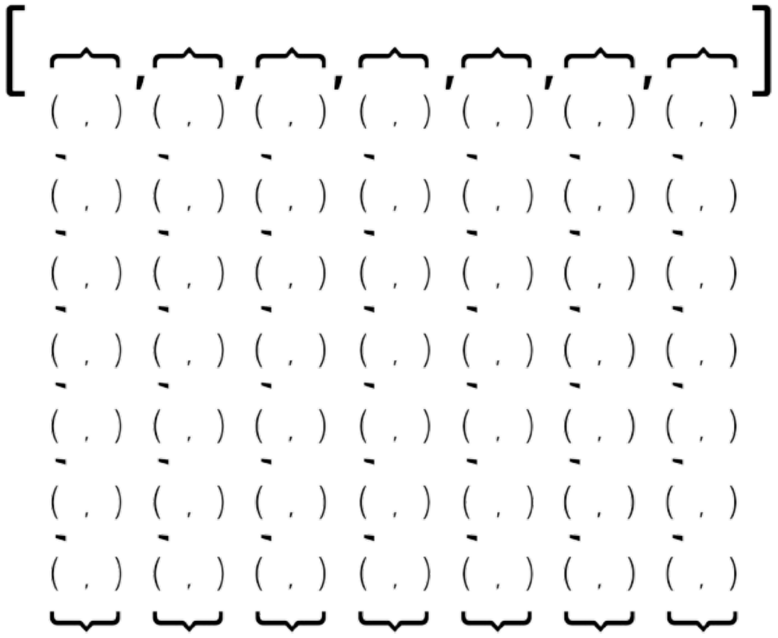

Figure 1: Visual representation of a vector of vector of pairs.

10.6 How all neighbours of a grid looks like

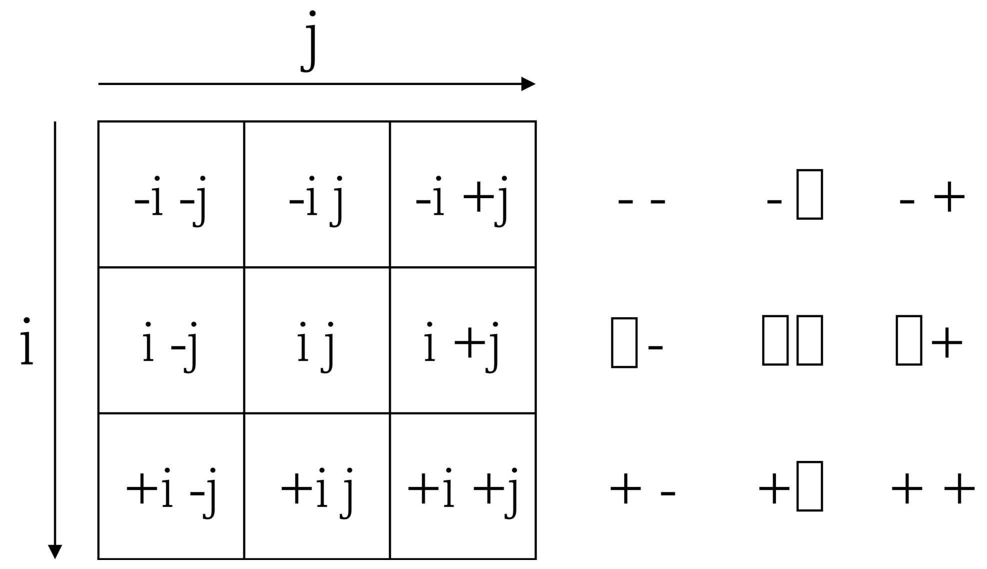

Figure 2: Visual representation of how all adjacent cells in a grid looks like.