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

ICPC 2026 Reference

CPCFI UNAM

>:)

RacsoFractal, zum, edwardsal17

Contents

1 Template

```cpp
// 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 *BS & friends

2.3 *STL DS Usage

2.4 *Strings methods

2.5 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`

Gray code: `G = B ⊕ (B >> 1)`

2.6 *Pragmas

2.7 Comparing Floats

```
1  long double a, b, EPS = 1e-9;
2  if( abs(a - b) < EPs ) {
3      // 'a' equals 'b'
4  }
```

Listing 2: Check if two real numbers are equal using an épsilon scope.

2.8 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 $a = 0$)

- Set bit: `n | (1 << i)`

- Clear bit: `n & ~(1 << i)`

- Toggle bit: `n ^ (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 ^ B`

```
1  int myCeil(long long a, long long b) {
2      return (a + b - 1)/b;
3  }
```

Listing 3: 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

```
1  int binary_search( vector<int>& list, int n, int
       target ) {
2      int x0 = 0, x1 = n-1, mid;
3      while( x0 <= x1 ) {
4          mid = (x0 + x1) / 2; // (x1 - x0) / 2 + x0;
5          if( list[mid] == target )
6              return mid;
7          list[mid] < target ? x0 = mid + 1 : x1 = mid
       - 1;
8      }
9      return -1;
10 }
```

Listing 4: 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.

```cpp
int binary_search( vector<int>& list, int n, int
    target ) {
  int ans = 0;
  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 5: 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
```

```cpp
      if( list[mid] < target )
        left = mid + 1;
      else
        right = mid;
  }
}
```

Listing 6: 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 7: 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

```
1  ll arns = v[0], maxSum = 0;
2  fori(i,0,n)
3  {
4    maxSum += v[i];
5    arns = max(arns, maxSum);
6    maxSum = max(0LL,maxSum);
7  }
```

Listing 8: 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

```
1  typedef long long ll;
2  typedef vector<ll> vll;
3  typedef vector<int> vi;
4  const int INF = INT_MAX;
5
```

```
6  class Segment_tree
7  {
8    public: vll t;
9    Segment_tree( int n = 1e5+10 ) {
10       t.assign(n*4,INF);
11   }
12   void update(int node, int index, int tl, int tr,
       int val) {
13       if( tr < index || tl > index ) return;
14       if( tr == tl ) t[node] = val;
15       else {
16           int mid = tl + ((tr-tl)>>1);
17           int lft = node << 1;
18           int rght = lft + 1;
19           update(lft,index,tl,mid,val);
20           update(rght,index,mid+1,tr,val);
21           t[node] = min(t[lft],t[rght]);
22       }
23   }
24   ll query(int node, int l, int r, int tl, int tr)
     {
25       if( tl > r || tr < l ) return INF;
26       if( tl >= l and tr <= r ) return t[node];
27       else {
28           int mid = tl + ((tr-tl)>>1);
29           int lft = node << 1;
30           int rght = lft + 1;
31           ll q1 = query(lft,l,r,tl,mid);
```

```
32          ll q2 = query(rght,l,r,mid+1,tr);
33          return min(q1,q2);
34      }
35   }
36   void build(vi &v, int node, int tl, int tr) {
37       if( tl == tr ) t[node] = v[tl];
38       else {
39           int mid = tl + ((tr-tl)>>1);
40           int lft = node << 1;
41           int rght = lft + 1;
42           build(v,lft,tl,mid);
43           build(v,rght,mid+1,tr);
44           t[node] = min(t[lft],t[rght]);
45       }
46   }
47 };
48 Segment_tree st(n);
49 st.build(v,1,0,n-1);
50 st.update(1,a-1,0,n-1,b);
51 st.query(1,a-1,b-1,0,n-1));
```

Listing 9: Segment Tree for Dynamic Range **Minimum** Queries. Racso's Implementation.

```
1 vector<long long> v, sex;
2 int n;
3 void build(int node, int l, int r){
4     if(l == r) sex[node] = v[l];
5     else{
6         int mid = (l+r)/2;
7         build(2*node, l, mid);
8         build(2*node + 1, mid+1, r);
9         sex[node] = sex[2*node] + sex[2*node +1];
10    }
11 }
12 void update(int node, int l, int r, int idx, int val
   ){
13    if(l == r){
14        v[idx] = val;
15        sex[node] = val;
16    }
17    else{
18        int mid = (l+r)/2;
19        if(l <= idx && idx <= mid) update(2*node, l,
   mid, idx, val);
20        else update(2*node +1, mid+1, r, idx, val);
21        sex[node] = sex[2*node] + sex[2*node + 1];
22    }
23 }
24 int query(int node, int tl, int tr, int l, int r){
25    if(r < tl || tr < l) return 0;
26    if(l <= tl && tr <= r) return sex[node];
27    int tm = (tl+tr)/2;
28    return query(2*node, tl, tm, l, r) + query(2*
   node +1, tm+1, tr, l, r);
29 }
30 v.resize(n);
```

```
31  sex.resize(4 * n);
32  build(1, 0, n - 1);
33  query(1, 0, n-1, l - 1, r - 1)
```

Listing 10: 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)

```
1  vector<vector<int>> graph;
2  vector<bool> visited;
3  graph.assign(n, vector<int>() ); // <--- main
4  visited.assign(n, false); // <--- main
5
6  void bfs( int s ) {
7    queue<int> q;
8    q.push( s );
9    visited[ s ] = true;
10   while( ! q.empty() ) {
11     int u = q.front();
```

```
12     q.pop();
13     for( auto v : graph[ u ] ) {
14       if( ! visited[ u ] ) {
15         visited[ u ] = true;
16         q.push( v );
17         // --- ToDo logic here ---
18       }
19     }
20   }
21   return;
22 }
```

Listing 11: Iterative implementation of BFS graph traversal over a graph represented as a AdjacencyList on vector of vectors. BFS runs in $O(|V| + |E|)$.

```
1  int n, m;
2  string arns = "";
3  cin >> n >> m;
4  vector<vector<bool>> visited(n,vector<bool>(m,false)
       );
5  vector<string> path(n,string(m,'0'));
6  vector<string> grid(n);
7  vii dirs = {{0,1},{0,-1},{1,0},{-1,0}};
8  string commands = "LRUD";
9  queue<ii> q;
10 ii start, end, curr;
11 function<bool(int,int)> valid = [&](int i, int j) ->
       bool {
12   return ( i >= 0 && i < n && j >= 0 && j < m &&
```

```
        grid[i][j] != '#' && ! visited[i][j] );
13  };
14  fori(i,0,n) cin >> grid[i];
15  fori(i,0,n) {
16      fori(j,0,m)
17          if( grid[i][j] == 'A' ) {
18              visited[i][j] = true;
19              q.push( {i,j} );
20          }
21  }
22  while( ! q.empty() ) {
23      curr = q.front();
24      q.pop();
25      int i = curr.fi;
26      int j = curr.se;
27      if( grid[i][j] == 'B' ) {
28          end.fi = i;
29          end.se = j;
30          break;
31      }
32      int newI, newJ;
33      fori(I,0,4) {
34          newI = i + dirs[I].fi;
35          newJ = j + dirs[I].se;
36          if( valid(newI,newJ) ) {
37              visited[newI][newJ] = true;
38              q.push( {newI,newJ} );
39              path[newI][newJ] = commands[I];
40          }
41      }
42  }
43  while( path[ end.fi ][ end.se ] != 'O' ) {
44      fori(i,0,4) {
45          if( path[ end.fi ][ end.se ] == commands[i] )
46          {
47              arns += i & 1 ? commands[i-1] : commands[i
                    +1];
48              end.fi -= dirs[i].fi;
49              end.se -= dirs[i].se;
50          }
51      }
52  }
53  reverse(all(arns));
54  if( arns == "" ) cout << "NO" << endl;
55  else cout << "YES" << endl << arns.size() << endl <<
        arns << endl;
```

Listing 12: 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)

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

```
5
6  void dfs( int s ) {
7      if( visited[s] == true ) return;
8      visited[s] = true;
9      vector<int>::iterator i;
10     for( i = graph[s].begin(); i < graph[s].end();
        ++i) {
11         if( ! visited[*i] ) {
12             // --- ToDo logic here ---
13             dfs(*i);
14         }
15     }
16 }
```

Listing 13: Recursive implementation of DFS graph traversal over a graph represented as a AdjacencyList on vector of vectors. DFS runs in $O(|V| + |E|)$.

```
1  vector<vector<int>> graph;
2  vector<bool> visited;
3  void dfs( int s ) {
4      stack<int> stk;
5      stk.push(s);
6      while (!stk.empty()) {
7          int u = stk.top();
8          stk.pop();
9          if ( visited[u] ) continue;
10         visited[u] = true;
11         // --- ToDo logic here ---
12         for(auto it = graph[u].rbegin(); it != graph
```

```
           [u].rend(); ++it)
13             if (!visited[*it])
14                 stk.push(*it);
15     }
16 }
```

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

```
1  typedef long long ll;
2  vector<vector<ll>> adj;
3  vector<bool> visited;
4  function<void(ll)> dfs = [&](ll u) -> void {
5      if( visited[u] ) return;
6      visited[u] = true;
7      for( ll v : adj[u] )
8          dfs(v);
9  };
10 dfs(n);
```

Listing 15: DFS implementation with a lambda function (adjacency list and visited don't need to be passed thorough argument). DFS runs in $O(|V| + |E|)$.

```
1  typedef long long ll;
2  typedef vector<ll> vll;
3  map<ll,vll> adj;
4  set<ll> visited;
5  function<void(ll)> dfs = [&](ll u) -> void {
6      if( visited.count(u) ) return;
7      visited.insert(u);
```

```
8        for( ll v : adj[u] )
9            dfs(v);
10 };
11 dfs(n);
```

Listing 16: 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

```
1  typedef long long ll;
2  typedef pair<ll,ll> pll;
3
4  vector<vector<ll>> graph;
5  vector<ll> visited;
6  graph.assign(n, vector<ll>() ); // <--- main
7  visited.assign(n, false); // <--- main
8
9  vector<ll> dijkstra( int n, int source, vector<
       vector<pll>> &graph ) {
10   vector<ll> dist( n, INFTY );
11   priority_queue<pll, vector<pll>, greater<pll>> pq;
12   dist[ source ] = 0;
13   pq.push( {0, source} );
14   while( ! pq.empty() ) {
15     ll d = pq.top().first;
16     ll u = pq.top().second;
```

```
17     pq.pop();
18     if( d > dist[ u ] ) continue;
19     for( auto &edge : graph[ u ] ) {
20       ll v = edge.first;
21       ll weight = edge.second;
22       if( dist[ u ] + weight < dist[ v ] ) {
23         dist[ v ] = dist[ u ] + weight;
24         pq.push( {dist[ v ], v} );
25       }
26     }
27   }
28   return dist;
29 }
```

Listing 17: 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.

```
1  vvpll graph(n+1,vpll());
2  vector<bool> visited(n+1,false);
3  function<vll(int)> dijkstra = [&](int source) -> vll
       {
4    vll dist(n+1,INF);
5    priority_queue<pll,vpll,greater<pll>> pq;
6    dist[source] = 0;
7    pq.push({0,source});
8    while( ! pq.empty() ) {
9      ll d = pq.top().fi;
```

```
10        ll u = pq.top().se;
11        pq.pop();
12        if( d > dist[u] ) continue;
13        for(pll edge : graph[u]) {
14            ll v = edge.fi;
15            ll w = edge.se;
16            if( dist[u] + w < dist[v] ) {
17                dist[v] = dist[u] + w;
18                pq.push({dist[v],v});
19            }
20        }
21    }
22    return dist;
23 };
```

Listing 18: 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

```
1 int V, E;
2 cin >> V >> E;
3 vvi edges(E,vi(3,0));
4
5 for(int i = 1; i <= E; i++)
6    cin >> edges[i][0] >> edges[i][1] >> edges[i][2];
```

```
7
8 function<vi(int)> bellman_ford = [&](int src) -> vi
9 {
10   vi dist(V,INF);
11   dist[src] = 0;
12   for(int i = 0; i < V; i++)
13   {
14     for( vi edge : edges )
15     {
16         int u = edge[0];
17         int v = edge[1];
18         int w = edge[2];
19         if( dist[u] != INF and dist[u] + w < dist[v]
     )
20         {
21             if( i == V-1 )
22                 return = {-1};
23             dist[v] = dist[u] + w;
24         }
25     }
26   }
27   return dist;
28 };
```

Listing 19: 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 20: 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 21: 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

```cpp
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 22: 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;
```

```cpp
37      return 0;
38  }
39  // Multi-Source BFS
40  while( ! q.empty() ) {
41    iii u = q.front();
42    q.pop();
43    for(ii dir : dirs) {
44      int newI = u[0] + dir.fi;
45      int newJ = u[1] + dir.se;
46      int w = u[2] + 1;
47      if( valid(newI,newJ) ) {
48        times[newI][newJ] = w;
49        q.push({newI,newJ,w});
50      }
51    }
52  }
53  // Player BFS
54  q.push({start.fi,start.se,0});
55  visited[start.fi][start.se] = true;
56  while( ! q.empty() and !escaped ) {
57    iii u = q.front();
58    q.pop();
59    for(int i = 0; i < 4; i++) {
60      int newI = u[0] + dirs[i].fi;
61      int newJ = u[1] + dirs[i].se;
62      int w = u[2] + 1;
63      if( valid_player(newI,newJ) and w < times[newI][
          newJ] ) {
64        visited[newI][newJ] = true;
65        prev[newI][newJ] = i;
66        q.push({newI,newJ,w});
67        if( is_border(newI,newJ) ) {
68          end.fi = newI;
69          end.se = newJ;
70          escaped = true;
71          break;
72        }
73      }
74    }
75  }
76  if( !escaped ) {
77    cout << "NO" << endl;
78    return 0;
79  }
80  // Path reconstruction
81  cout << "YES" << endl;
82  int i = end.fi;
83  int j = end.se;
84  while( prev[i][j] != -1 ) {
85    int oldI = i;
86    arns += path[ prev[i][j] ];
87    i -= dirs[prev[i][j]].fi;
88    j -= dirs[prev[oldI][j]].se;
89  }
90  reverse(all(arns));
91  cout << sz(arns) << endl << arns << endl;
```

Listing 23: 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

```
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.
```

```cpp
34      next.resize(n); // Prepare next-edge array for
    DFS.
35      fill(all(level),-1); // Mark all levels as
    unvisited (-1).
36      level[s] = 0; // The source has level 0.
37      q.push(s); // Start BFS from the source.
38    }
39    /**
40     * @brief Adds a directed edge and its residual
    reverse edge to the flow network. */
41    void addEdge( int u, int v, ll cap ) {
42      edges.emplace_back(u,v,cap); // Original edge: u
      -> v
43      edges.emplace_back(v,u,0); // Residual edge: v
      -> u
44      adj[u].pb(id++);
45      adj[v].pb(id++);
46    }
47    /**
48     * @brief Performs BFS to construct the level
    graph (Layered Network) from source 's' to sink '
    t'.
49     * @details Assigns levels (minimum distances
    from 's') to all nodes and checks if 't' is
    reachable.
50     *          Levels are used to guide the DFS
    phase in Dinic's algorithm.
51     * @return bool True if the sink 't' is reachable
    (i.e., there exists an augmenting path), false
    otherwise. */
52    bool bfs() {
53      while( ! q.empty() ) {
54        int curr = q.front();
55        q.pop();
56        for( auto e : adj[curr] ) {
57          if( edges[e].cap - edges[e].flow < 1 ) //
    Skip saturated edges (no residual capacity).
58            continue;
59          if( level[ edges[e].v ] != -1 ) // Skip
    already visited nodes (level assigned).
60            continue;
61          // Assign level to the neighbor node.
62          level[ edges[e].v ] = level[ edges[e].u ] +
    1; // Next level = current + 1.
63          q.push( edges[e].v ); // Add neighbor to the
      queue for further BFS.
64        }
65      }
66      return level[t] != -1; // Return whether the
    sink 't' was reached (level[t] != -1).
67    }
68    /**
69     * @brief Finds a blocking flow using DFS in the
    level graph constructed by BFS.
70     * @param u Current node being processed.
```

```
71      * @param flow Maximum flow that can be sent from
        'u' to the sink 't'.
72      * @return ll The amount of flow successfully
        sent to 't'. */
73   ll dfs( int u, ll flow ) {
74     if( flow == 0 ) // No remaining flow to send.
75        return 0;
76     if( u == t )    // Reached the sink; return
        accumulated flow.
77        return flow;
78     // Explore edges from 'u' using 'next[u]' to
        avoid revisiting processed edges.
79     for( int& cid = next[u]; cid < sz(adj[u]); cid++
        ) {
80        int e = adj[u][cid]; // Index of the edge in '
        edges'.
81        int v = edges[e].v;  // Destination node of
        the edge.
82        // Skip invalid edges:
83        // 1. Not in the level graph (level[u] + 1 !=
        level[v]). Just edges in exactly one leve ahead (
        ensures shortest paths).
84        // 2. No residual capacity (cap - flow < 1).
85        if( level[edges[e].u] + 1 != level[v] || edges
        [e].cap - edges[e].flow < 1 )
86           continue;
87        ll f = dfs( v, min(flow, edges[e].cap - edges[
        e].flow ) ); // Recursively send flow to 'v'.
88        if( f == 0 ) // No flow could be sent via this
        edge.
89           continue;
90        // Update residual capacities:
91        edges[e].flow += f;        // Increase flow in
        the original edge.
92        edges[ e ^ 1 ].flow -= f; // Decrease flow in
        the reverse edge. (All reverse edges have
        distinct parity)
93        return f;                  // Return the flow
        sent.
94     }
95     return 0; // No augmenting path found from 'u'.
96   }
97   /**
98     * @brief Computes the maximum flow from source '
        s' to sink 't' using Dinic's algorithm.
99     * @details Iterates through BFS and DFS phases
        to find the maximum flow.
100        Accumulates flow while there exists
        augmention paths in the residual graph.
101        Restart auxiliary structures for every new
        phase.
102     * @return ll The maximum flow value. */
103   ll maxFlow() {
104     ll flow = 0; // Tracks the total flow sent.
105     while( bfs() ) { // While there are augmenting
        paths:
```

```
106        fill(all(next),0); // Reset 'next' for DFS.
107        for( ll f = dfs(s,INF); f != 0ll; f = dfs(s,
      INF) ) // Send blocking flow in the level graph:
108            flow += f;
109        // Reset for next BFS phase:
110        fill(all(level),-1);
111        level[s] = 0;
112        q.push(s);
113      }
114      return flow;
115    }
116    /**
117     * @brief Finds edges belonging to the minimum
      cut after maxFlow().
118     * @details First, it marks all the reachable
      nodes from 's' with an augmention path after
      obtained the max flow
119        and all the saturated edges coming out from
      any of the nodes who belong to the min-cut.
120        For 'minCut()' to work, 'maxFlow()' must be
      first executed to get the min-cut.
121        If only is needed the value, is enough
      returning the value of 'maxFlow()'.
122     * @return vii List of edges (u, v) in the min-
      cut. Its size is the minimum number of 'roads' to
       close. */
123    vii minCut() {
124      vii ans;
125      fill(all(level),-1); // Reset levels.
126      level[s] = 0;        // Mark source as reachable
      .
127      q.push(s);
128      while( ! q.empty() ) { // BFS to mark nodes
      reachable from 's' in the residual graph.
129        int curr = q.front();
130        q.pop();
131        for( int id = 0; id < sz(adj[curr]); id++ ) {
      // For every edge going out from 'curr'.
132            int e = adj[curr][id];
133            // If 'v' is has not been visited yet, and
      the edge have residual capacity.
134            if( level[edges[e].v] == -1 && edges[e].cap
      - edges[e].flow > 0 ) {
135                q.push(edges[e].v);
136                level[edges[e].v] = level[edges[e].u] + 1;
137            }
138        }
139      }
140      for( int i = 0; i < sz(level); i++ ) {
141        if( level[i] != -1 ) {
142            for( int id = 0; id < sz(adj[i]); id++ ) {
143                int e = adj[i][id];
144                if( level[edges[e].v] == -1 && edges[e].
      cap - edges[e].flow == 0 )
145                    ans.emplace_back(edges[e].u,edges[e].v);
146            }
```

```
147        }
148      }
149      return ans;
150    }
151    /**
152     * @brief Reconstructs the maximum bipartite
       matching after running 'maxFlow()'.
153     * @details Every edge that belong to the
       original graph and have flow greater than zero,
       belongs to the matching.
154         For 'maximumMatching()' to work, 'maxFlow()
       '.
155     * @return vii List of matched pairs (boy, girl).
       */
156    vii maximumMatching() {
157      vii ans;
158      fill(all(level),-1); // Reset levels.
159      level[s] = 0;        // Mark source as reachable
       .
160      q.push(s);
161      while( ! q.empty() ) { // BFS to mark nodes
       reachable via saturated edges with flow greater
       than zero.
162        int curr = q.front();
163        q.pop();
164        for( int id = 0; id < sz(adj[curr]); id++ ) {
165          int e = adj[curr][id];
166          // If 'v' has not been visited yet, the edge
       is saturated and have flow greater than zero.
167          if( level[edges[e].v] == -1 && edges[e].cap
       - edges[e].flow == 0 && edges[e].flow != 0ll ) {
168            q.push(edges[e].v);
169            level[edges[e].v] = level[edges[e].u] + 1;
170          }
171        }
172      }
173      for( int i = 0; i < sz(level); i++ ) { //
       Collect original edges (boy -> girl) that are
       saturated and have flow > 0.
174        if( level[i] != -1 ) {
175          for( int id = 0; id < sz(adj[i]); id++ ) {
176            int e = adj[i][id];
177            if( edges[e].u != s && edges[e].v != t
       && edges[e].cap - edges[e].flow == 0 && edges[e].
       flow != 0ll )
178              ans.emplace_back(edges[e].u,edges[e].v
       );
179          }
180        }
181      }
182      return ans;
183    }
184  };
```

Listing 24: 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)

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

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

    return 0;
}
```

Listing 25: 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.

```
1  int main()
2  {
3      fastIO();
4
5      int n, m, u, v;
6      vii minCut;
7
8      cin >> n >> m;
9
10     Dinic flow(n+1,1,n); // size n+1 to fix 0-
        indexed indexes, 1 is the source (bank), 'n' is
        the sink (harbor)
11
12     fori(i,0,m)
13     {
14         cin >> u >> v;
15         flow.addEdge(u,v,1);
16         flow.addEdge(v,u,1);
17     }
18
19     flow.maxFlow();
20     minCut = flow.minCut();
21
22     cout << (sz(minCut)/2) << endl;
```

```
23     for(int i = 0; i < sz(minCut); i += 2)
24         cout << minCut[i].fi << " " << minCut[i].se
        << endl;
25
26     return 0;
27 }
```

Listing 26: 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.

```
1  int main()
2  {
3      fastIO();
```

```
4
5    int n, m, k, a, b;
6    ll maxPairs;
7    vii pairs;
8
9    cin >> n >> m >> k;
10
11   Dinic flow(n+m+2,0,n+m+1);
12
13   fori(boy,0,n+1)
14       flow.addEdge(0,boy,1);
15
16   fori(girl,n+1,n+m+1)
17       flow.addEdge(girl,n+m+1,1);
18
19   fori(i,0,k)
20   {
21       cin >> a >> b;
22       flow.addEdge(a,n+b,1);
23   }
24
25   maxPairs = flow.maxFlow();
26   pairs = flow.maximumMatching();
27
28   cout << maxPairs << endl;
29   fori(i,0,sz(pairs))
30       cout << pairs[i].fi << " " << (pairs[i].se -
     n) << endl;;
```

```
31
32   return 0;
33 }
```

Listing 27: 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

```
1  vi childrens(n+1,0);
2  vvi graph(n+1);
3  vector<bool> visited(n+1, false);
4  fori(i,2,n+1) {
5      cin >> tmp;
6      graph[tmp].pb(i);
7      graph[i].pb(tmp);
8  }
9  function<int(int)> dfs = [&](int u) -> int {
10     visited[u] = true;
11     for(int v : graph[u]) {
12         if( !visited[v] )
13             childrens[u] += dfs(v);
14     }
15     return childrens[u] + 1;
16 };
```

```
17   dfs(1);
```

Listing 28: Algorithm that counts how many childrens does every node have, from 2..n in a rooted tree (root = 1).

```
1  int gcd(int a, int b) {
2    if (a == 0) return b;
3    if (b == 0) return a;
4    if (a == b) return a;
5    if (a > b)
6      return gcd(a - b, b);
7    return gcd(a, b - a);
8  }
```

Listing 29: 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 $O(1)$.

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

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

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

Listing 30: 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

```
1  const int MOD = 1e9+7;
2  int binpow( long long a, long long b ) { // a^b
3      long long sol = 1;
4      a %= MOD;
5      while( b > 0 ) {
6          if( b & 1 )
7              sol = ( 1LL * sol * a ) % MOD;
8          a = ( 1LL * a * a ) % MOD;
9          b >>= 1;
10     }
11     return sol % MOD;
12 }
```

Listing 31: Applying binary exponentiation to a problem requiring $a^b \bmod(10^9 + 7)$ in $O(log_2(b))$.

9.1.8 Matrix Exponentiation (Linear Recurrency)

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

```
4      vector<vector<T>> c(n,vector<T>(p));
5      for(size_t i = 0; i < n; i++)
6         for(size_t j = 0; j < p; j++)
7            for(size_t k = 0; k < m; k++)
8               c[i][j] = (c[i][j] + a[i][k] * b[k][j])
       % MOD;
9      a = c;
10  }
11  template <typename T> struct Matrix {
12     vector<vector<T>> mat;
13     Matrix() {}
14     Matrix(vector<vector<T>> a) { mat = a; }
15     Matrix(int n, int m) {
16        mat.resize(n);
17        for(int i = 0; i < n; i++) {mat[i].resize(m);
       }
18     }
19     int rows() const { return mat.size(); }
20     int cols() const { return mat[0].size(); }
21     void makeIden() {
22        for(int i = 0; i < rows(); i++)
23           for(int j = 0; j < cols(); j++)
24              mat[i][j] = (i == j ? 1 : 0);
25     }
26     Matrix operator*=(const Matrix &b) {
27        matmul(mat, b.mat);
28        return *this;
29     }
30     void print() {
31        for(int i = 0; i < rows(); i++) {
32           for(int j = 0; j < cols(); j++)
33              cout << mat[i][j] << " ";
34           cout << endl;
35        }
36     }
37     Matrix operator*(const Matrix &b) { return Matrix
       (*this) *= b; }
38  };
39  int main() {
40     Matrix<ll> A( {{1,1},{1,0}} );
41     Matrix<ll> ini(2,1);
42     ini.mat[0][0] = 0;
43     ini.mat[1][0] = 1;
44     Matrix<ll> iden(2,2);
45     iden.makeIden();
46     ll n;
47     cin >> n;
48     while(n > 0) {
49        if( n & 1 ) iden *= A;
50        A *= A;
51        n >>= 1;
52     }
53     Matrix<ll> res = iden * ini;
54     cout << res.mat[0][0] << endl;
55     return 0;
56  }
```

Listing 32: 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 33: 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;
```

```cpp
        }
    }
    if( n > 1 )
        factorization.push_back(n);
}
```

Listing 34: 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 35: 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;
```

```
3     for (long long i = 2; i * i <= n; i++) {
4       if(n % i == 0) {
5           int e = 0;
6           do {
7             e++;
8             n /= i;
9           } while (n % i == 0);
10          ll s = 0, pow = 1;
11          do {
12            s += pow;
13            pow *= i;
14          } while (e-- > 0);
15          sum *= s;
16        }
17    }
18    if(n > 1)
19      sum *= (1 + n);
20    return sum;
21 }
```

Listing 36: Calculates the sum of all divisors of number n. e.g. $sum\_of\_divisors(12) = 18$. Runs in $O(\sqrt{n})$.

```
1 void sum_of_divisors_sieve( vll& sigma, int n ) {
2     sigma.assign(n+1,0);
3     for(int i = 1; i <= n; i++)
4         for(int j = i; j <= n; j+=i)
5             sigma[j] += i;
6 }
```

Listing 37: 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

```
1  const int MAXN = 1e6+1;
2  vll fact(MAXN+1), inv(MAXN+1);
3  int binpow( ll a, ll b ) { // a^b
4      ll sol = 1;
5      a %= MOD;
6      while( b > 0 ) {
7          if( b & 1 )
8              sol = ( 1LL * sol * a ) % MOD;
9          a = ( 1LL * a * a ) % MOD;
10         b >>= 1;
11     }
12     return sol % MOD;
13 }
14 void combi() {
15     fact[0] = inv[0] = 1;
16     fori(i,1,MAXN+1) {
17         fact[i] = fact[i-1] * i % MOD;
18         inv[i] = binpow( fact[i], MOD - 2 );
19     }
20 }
21 ll nCr( ll n, ll r ) {
```

```
22        return fact[n] * inv[r] % MOD * inv[n-r] % MOD;
23 }
24 combi();
25 nCr(a,b);
```

Listing 38: 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

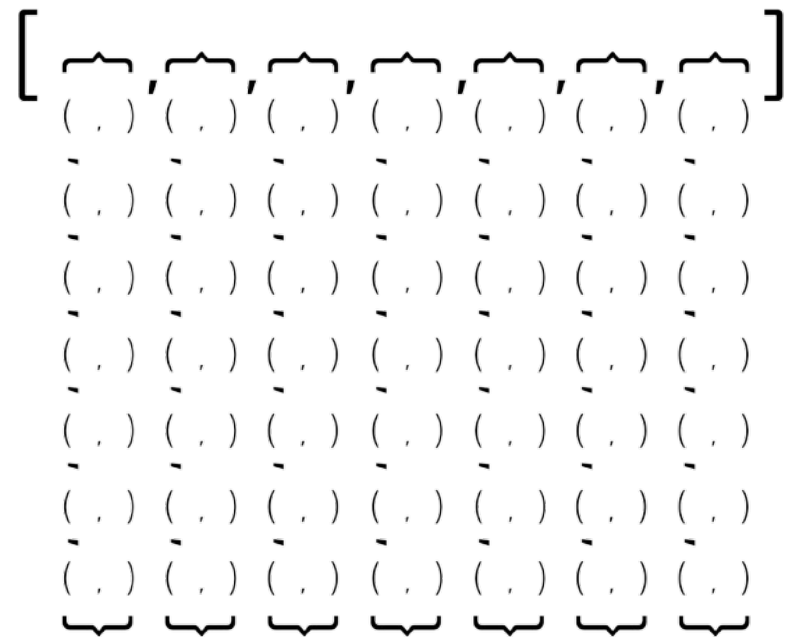## 10.5 How a `vector<vector<pair<int,int>>>` looks like

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

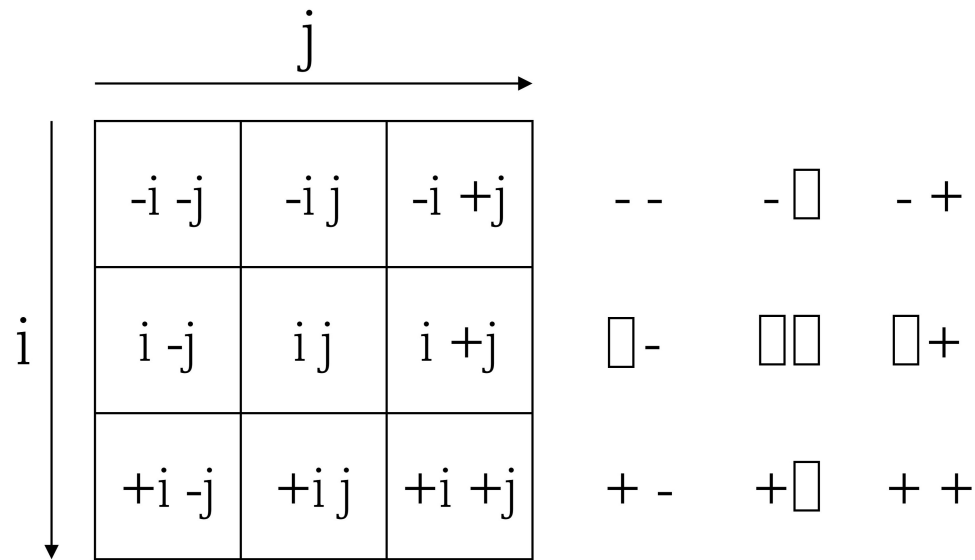## 10.6   How all neighbours of a grid looks like

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