

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



ICPC 2026 Reference

CPCFI UNAM

>:)

RacsoFractal, zum, edwardsal17

Contents

1 Template	3	5.3.3 Bellman–Ford Algorithm	16
2 C++ Syntax	3	5.4 Minimum Spanning Tree (MST)	17
2.1 Compilation sentences	3	5.4.1 *Prim's Algorithm	17
2.2 Custom comparators	3	5.4.2 *Kruskal's Algorithm	17
2.3 *STL DS Usage	4	5.5 *Bipartite Matching	17
2.4 *Strings methods	4	5.6 *Negative Cycles	17
2.5 Pragmas	4	5.7 Topological Sort	17
2.6 Bit Manipulation Cheat Sheet	4	5.8 Disjoint Set Union (DSU)	18
2.7 Comparing Floats	5	5.9 *Condensation Graph	19
2.8 Ceil	5	5.10 *Strongly Connected Components (SCC)	19
3 Miscellaneous	5	5.11 2-SAT	19
3.1 Binary Search	5	5.12 *Bridges and point articulation	22
3.1.1 *Parallel Binary Search	7	5.13 Flood Fill	22
3.1.2 *Ternary Search	7	5.14 Lava Flow (Multi-source BFS)	23
3.2 Kadane's Algorithm	7	5.15 MaxFlow	25
4 Queries	7	5.15.1 Dinic's Algorithm	25
4.1 Prefix Sum 2D	7	5.15.2 *Ford-Fulkerson Algorithm	32
4.2 *Sparse Table	7	5.15.3 *Goldberg-Tarjan Algorithm	32
4.3 *Sqrt Decomposition	7	6 Trees	32
4.4 *Fenwick Tree	7	6.1 Counting Childrens	32
4.5 *Fenwick Tree 2D	7	6.2 *Tree Diameter	33
4.6 Segment Tree	7	6.3 *Centroid Decomposition	33
4.6.1 *2D Segment Tree	9	6.4 *Euler Tour	33
4.6.2 *Persistent Segment Tree	9	6.5 *Lowest Common Ancestor (LCA)	33
4.6.3 Lazy Propagation	9	6.6 *Heavy-Light Decomposition (HLD)	33
4.7 Ordered Set	11	7 Strings	33
4.7.1 Multi-Ordered Set	12	7.1 Knuth-Morris-Pratt Algorithm (KMP)	33
4.8 *Treap	12	7.2 *Suffix Array	34
4.9 *Trie	12	7.3 *Rolling Hashing	34
5 Graph Theory	12	7.4 *Z Function	34
5.1 Breadth-First Search (BFS)	12	7.5 *Aho-Corasick Algorithm	34
5.2 Deep-First Search (DFS)	14	8 Dynamic Programming	34
5.3 Shortest Path	15	8.1 *Coins	34
5.3.1 Dijkstra's Algorithm	15	8.2 *Longest Increasing Subsequence (LIS)	34
5.3.2 *Floyd-Warshall's Algorithm	16	8.3 *Edit Distance	34
		8.4 *Knapsack	34
		8.5 *SOS DP	34

8.6 *Digit DP	34
8.7 *Bitmask DP	34
9 Mathematics	34
9.1 Number Theory	34
9.1.1 Greatest Common Divisor (GCD)	34
9.1.2 Gauss Sum	34
9.1.3 *Modular Theory	34
9.1.4 *Modulo Inverse	34
9.1.5 *Fermat's Little Theorem	34
9.1.6 *Chinese Remainder Theorem	34
9.1.7 Binpow	34
9.1.8 Matrix Exponentiation (Linear Recurrency)	35
9.1.9 Prime checking	36
9.1.10 Prime factorization	36
9.1.11 Sieve of Eratosthenes	36
9.1.12 Sum of Divisors	37
9.2 Combinatorics	37
9.2.1 Binomial Coefficients	37
9.2.2 Common combinatorics formulas	38
9.2.3 Stars and Bars	38
9.3 Probability	38
9.4 Computational Geometry	38
9.4.1 *Cross Product	38
9.4.2 *Convex Hull	38
9.5 *Fast Fourier Transform (FFT)	38
10 Appendix	38
10.1 What to do against WA?	38
10.2 Primitive sizes	39
10.3 Printable ASCII characters	39
10.4 Numbers bit representation	40
10.5 How a <code>vector<vector<pair<int,int>>></code> looks like	40
10.6 How all neighbours of a grid looks like	40

1 Template

```

1 // Racso programmed here
2 #include <bits/stdc++.h>
3 using namespace std;
4 typedef long long ll;
5 typedef long double ld;
6 typedef __int128 sll;
7 typedef pair<int,int> ii;
8 typedef pair<ll,ll> pll;
9 typedef vector<int> vi;
10 typedef vector<ll> vll;
11 typedef vector<ii> vii;
12 typedef vector<vi> vvi;
13 typedef vector<vii> vvii;
14 typedef vector<pll> vpll;
15 typedef unsigned int uint;
16 typedef unsigned long long ull;
17 #define fi first
18 #define se second
19 #define pb push_back
20 #define all(v) v.begin(),v.end()
21 #define rall(v) v.rbegin(),v.rend()
22 #define sz(a) (int)(a.size())
23 #define fori(i,a,n) for(int i = a; i < n; i++)
24 #define endl '\n'
25 const int MOD = 1e9+7;
26 const int INFTY = INT_MAX;

```

```

27 const long long LLINF = LLONG_MAX;
28 const double EPS = DBL_EPSILON;
29 void printVector( auto& v ){ fori(i,0,sz(v)) cout << v[i] << " "; cout << endl; }
30 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++ -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

```

1 // Using function
2 bool cmpFunction(const pair<int,int> &a, const pair<
3   int,int> &b) {
4   return a.second < b.second;
}

```

```

4 }
5 sort(all(v), cmpFunction);
6 // Using functor
7 struct CmpFunctor {
8     bool operator()(const pair<int,int> &a, const pair
9         <int,int> &b) const {
10        return a.second < b.second;
11    }
12 };
13 sort(all(v), CmpFunctor());
14 // Using lambda function
15 sort(all(v), [](const pair<int,int> &a, const pair<
16        int,int> &b) {
17        return a.second < b.second;
18 });

```

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

2.3 *STL DS Usage

2.4 *Strings methods

2.5 Pragmas

```

1 #pragma GCC optimize("O3")
2 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
3 #pragma GCC optimize("unroll-loops")

```

Listing 3: Common pragmas.

2.6 Bit Manipulation Cheat Sheet

Bitwise operators:

- **&** (AND): Sets each bit to 1 if both bits are 1.
- **|** (OR): Sets each bit to 1 if at least one bit is 1.
- **\wedge** (XOR): Sets each bit to 1 if bits are different.
- **\sim** (NOT): Inverts all bits.
- **\ll** (Left shift): Shifts bits left, fills with 0.
- **\gg** (Right shift): Shifts bits right.

Basic Bit Tasks:

- Get bit: $(n \& (1 \ll i)) != 0$
- Set bit: $n | (1 \ll i)$
- Clear bit: $n \& \sim(1 \ll i)$
- Toggle bit: $n \wedge (1 \ll i)$
- 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: $A \& \sim B$
- Toggle subset: $A \hat{B}$

Equations:

Properties of bitwise:

- $a | b = a \oplus b + a \& b$

- $a \oplus (a \& b) = (a \mid b) \oplus b$
- $b \oplus (a \& b) = (a \mid b) \oplus a$
- $(a \& b) \oplus (a \mid b) = a \oplus b$

In addition and subtraction:

- $a + b = (a \mid b) + (a \& b)$
- $a + b = a \oplus b + 2(a \& b)$
- $a - b = (a \oplus (a \& b)) - ((a \mid b) \oplus a)$
- $a - b = ((a \mid b) \oplus b) - ((a \mid b) \oplus a)$
- $a - b = (a \oplus (a \& b)) - (b \oplus (a \& b))$
- $a - b = ((a \mid b) \oplus b) - (b \oplus (a \& b))$

Gray code: $G = B \oplus (B \gg 1)$

C++ built in functions:

- `__builtin_popcount(x)` - Count the number of set bits in x.
- `__builtin_clz(x)` - Count the number of leading zeros in x.
- `__builtin_ctz(x)` - Count the number of trailing zeros in x.

2.7 Comparing Floats

```

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

Listing 4: Check if two real numbers are equal using an epsilon scope.

2.8 Ceil

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

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

```

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

Listing 5: 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
2                     target ) {
3
4     int x0 = 0, x1 = n-1, mid;
5
6     while( x0 <= x1 ) {
7
8         mid = (x0 + x1) / 2; // (x1 - x0) / 2 + x0;
9
10        if( list[mid] == target )
11            return mid;
12
13        list[mid] < target ? x0 = mid + 1 : x1 = mid
14            - 1;
15    }
16
17    return -1;
18 }
```

Listing 6: 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.

```

1 int binary_search( vector<int>& list, int n, int
2     target ) {
3
4     int ans = 0;
5
6     function<bool(int)> check = [&](int idx)->bool {
7
8         return idx < n && list[idx] <= target;
9     };
10
11    for( int i = 31; i >= 0; i-- ) {
12        if( check( ans + (1 << i) ) )
13            ans += 1 << i;
14    }
15
16    return list[ans] == target ? ans : -1;
17 }
```

Listing 7: 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.

```

1 int binary_search( vector<int>& list, int n, int
2     target ) {
3
4     int left = 1, right = n + 1, mid;
5
6     while(right - left >= 1) {
7
8         mid = left + (right - left) / 2;
9
10        if( list[mid] >= target && target > list[mid -
11]) {
```

```

6         //----- TODO logic here -----//
7         break;
8     }
9     else
10        if( list[mid] < target )
11            left = mid + 1;
12        else
13            right = mid;
14    }
15 }
```

Listing 8: 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.

```

1 function<bool(ll)> check = [&](ll t) -> bool {
2
3     ll products = 0;
4
5     for(ll machine : v) {
6
7         products += t / machine;
8
9         if( products >= target )
10            return true;
11    }
12
13    return products >= target;
14 }
15
16 for(int i = 0; i < 70; i++) {
17
18     mid = (x0 + x1) / 2;
19
20     check(mid) ? x1 = mid : x0 = mid + 1;
21 }
```

Listing 9: 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 arns = v[0], maxSum = 0;
2 for(i,0,n)
{
3     maxSum += v[i];
4     arns = max(arns, maxSum);
5     maxSum = max(0LL, maxSum);
6 }

```

Listing 10: Uses Kadane's Algorithm to find maximum subarray sum in $O(n)$.

4 Queries

4.1 Prefix Sum 2D

```

1 for(int i = 1; i <= n; i++)
2     for(int j = 1; j <= n; j++)
3     {
4         prefix[i][j] = prefix[i][j-1] + prefix[i-1][j] -
5             prefix[i-1][j-1];
6         prefix[i][j] += forest[i-1][j-1] == '*' ? 1 : 0;

```

```

6     }
7     for(int i = 0; i < q; i++)
8     {
9         pair<int,int> p1, p2;
10        cin >> p1.fi >> p1.se >> p2.fi >> p2.se;
11        int arns = prefix[p2.fi][p2.se];
12        arns -= prefix[p2.fi][p1.se-1] + prefix[p1.fi-1][
13            p2.se];
14        arns += prefix[p1.fi-1][p1.se-1];
15        cout << arns << endl;

```

Listing 11: Construction and querie of how many 1's are there in a matrix. Originally used on *Forest Queries* from CSES.

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 class Segment_tree {
6     public: vll t;
7     Segment_tree( int n = 1e5+10 ) {

```

```

8     t.assign(n*4,INF);
9 }
10 void update(int node, int index, int tl, int tr,
11   int val) {
12     if( tr < index || tl > index ) return;
13     if( tr == tl ) t[node] = val;
14     else {
15       int mid = tl + ((tr-tl)>>1);
16       int lft = node << 1;
17       int rght = lft + 1;
18       update(lft,index,tl,mid,val);
19       update(rght,index,mid+1,tr,val);
20       t[node] = min(t[lft],t[rght]);
21     }
22 }
23 ll query(int node, int l, int r, int tl, int tr)
24 {
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 Segment_tree st(n);
48 st.build(v,1,0,n-1);
49 st.update(1,a-1,0,n-1,b);
50 st.query(1,a-1,b-1,0,n-1));

```

Listing 12: 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 ) {
14     if(l == r) {
15         v[idx] = val;
16         sex[node] = val;
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 13: 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**

```

1 vector<long long> v;
2 vector<long long> sex;
3 vector<long long> lazy;
4 long long n;

5

6 void push(int node, int tl, int tr){
7     if(lazy[node] != 0){
8         sex[node] += lazy[node] * (tr - tl + 1);
9
10        if(tl != tr){
11            lazy[2*node] += lazy[node];
12            lazy[2*node + 1] += lazy[node];
13        }
14
15        lazy[node] = 0;
16    }
17 }

18

19 void build(int node, int tl, int tr){
20     if(tl == tr){
21         sex[node] = v[tl];

```

```
22     }
23 
24     else{
25         int tm = (tl + tr)/2;
26         build(2*node, tl, tm);
27         build(2*node + 1, tm+1, tr);
28         sex[node] = sex[2*node] + sex[2*node + 1];
29     }
30 }
31 void update(int node, int tl, int tr, int l, int r,
32             int val){
33     push(node, tl, tr);
34     //Si el rango del nodo actual esta fuera de
35     //rango totalmente
36     if(l > tr || r < tl){
37         return;
38     }
39     //Si el rango del nodo esta completamente dentro
40     //del rango
41     if(l <= tl && r >= tr){
42         lazy[node] += val;
43         push(node, tl, tr);
44         return;
45     }
46     int tm = (tl + tr)/2;
47     update(2*node, tl, tm, l, r, val);
48     update(2*node + 1, tm + 1, tr, l, r, val);
49 }
50
51 long long query(int node, int tl, int tr, int l, int
52                  r){
53     //Antes de cualquier consulta, se pushean las
54     //actualizaciones pendientes
55     push(node, tl, tr);
56
57     //Si el rango del nodo actual esta fuera de
58     //rango
59     if(l > tr || r < tl){
60         return 0; //Regresa el elemento neutro para
61         //la suma
62     }
63
64     //Si el rango del nodo actual esta completamente
65     //dentro del rango
66     if(l <= tl && r >= tr){
67         return sex[node];
68     }
69
70     //Si el rango del nodo esta parcialmente en el
71     //rango, desciende y combina los resultados
72     int tm = (tl + tr)/2;
73     long long lzum = query(2*node, tl, tm, l, r);
74     long long rzum = query(2*node + 1, tm + 1, tr, l, r);
75     return lzum + rzum;
76 }
```

```

68     long long rzum = query(2*node + 1, tm + 1, tr, l
69         , r);
70
71     return lzum + rzum; //Devuelve la operacion
72     aplicada a ambas partes (suma)
73 }
74
75 void solve(){
76     long long q; cin >> n >> q;
77
78     v.resize(n);
79     sex.assign(4 * n, 0); //Se inicializan en el
80     elemento neutro
81     lazy.assign(4 * n, 0);
82
83     //Lectura del arreglo inicial
84     for(int i = 0; i < n; i++){
85         cin >> v[i];
86     }
87
88     //Construye el sextree
89     build(1, 0, n - 1);
90
91     //type == 1 es actualización
92     //type == 2 es query
93     for(long long i = 0; i < q; i++){
94         int type; cin >> type;
95         if(type == 1){
96             long long l, r, val;
97             cin >> l >> r >> val;
98             update(1, 0, n - 1, l - 1, r - 1, val);
99             //Esta 1-indexed
100        }
101    }
102}

```

Listing 14: Lazy Propagation Segment Tree for Range Updates Zum Queries.

4.7 Ordered Set

```

1 //<-- Header.
2 #include <bits/stdc++.h>
3 #include <ext/pb_ds/assoc_container.hpp>
4 #include <ext/pb_ds/tree_policy.hpp>
5 using namespace std;
6 using namespace __gnu_pbds;
7 template<typename T, typename Cmp = less<T>>
8 using ordered_set = tree<T, null_type, Cmp, rb_tree_tag
9   , tree_order_statistics_node_update>;
10 //<-- Declaration.
11 ordered_set<int> oset;
12 //<-- Methods usage.

```

```

12 // K-th element in a set (counting from zero).
13 ordered_set<int>::iterator it = oset.find_by_order(k
14 );
14 // Number of items strictly smaller than k.
15 ordered_set<int>::iterator it = oset.order_of_key(k
16 ;
16 // Every other std::set method.

```

Listing 15: Ordered set necessary includes in header, declaration of the object, and usage of its new methods.

4.7.1 Multi-Ordered Set

```

1 //<-- Header.
2 #include <bits/stdc++.h>
3 #include <ext/pb_ds/assoc_container.hpp>
4 #include <ext/pb_ds/tree_policy.hpp>
5 using namespace std;
6 using namespace __gnu_pbds;
7 template<typename T, typename Cmp = less<T>>
8 using ordered_set = tree<T,null_type,Cmp,rb_tree_tag
     ,tree_order_statistics_node_update>;
9 //<-- Use in main.
10 ordered_set<pair<int,int>> multi_oiset;
11 map<int,int> cuenta;
12 function<void(int)> insertar = [&](int val) -> void
    {
13     multi_oiset.insert({val,++cuenta[val]});
14 };

```

```

15 function<void(int)> eliminar = [&](int val) -> void
    {
16     multi_oiset.erase({val,cuenta[val]--});
17 };

```

Listing 16: Declaration of multi-oiset structure.

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 17: 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) ->
12     bool {
13     return ( i >= 0 && i < n && j >= 0 && j < m &&
14         grid[i][j] != '#' && ! visited[i][j] );
15 };
16 for(i,0,n) cin >> grid[i];
17 for(i,0,n) {
18     for(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     for(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 ] != '0' ) {
44     for(i,0,4) {
```

```

45     if( path[ end.fi ][ end.se ] == commands[i] )
46     {
47         arns += i & 1 ? commands[i-1] : commands[i
48 +1];
49         end.fi -= dirs[i].fi;
50         end.se -= dirs[i].se;
51     }
52 }
53 reverse(all(arns));
54 if( arns == "") cout << "NO" << endl;
55 else cout << "YES" << endl << arns.size() << endl <<
arns << endl;

```

Listing 18: 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();
11 ++i ) {
12        if( ! visited[*i] ) {
13            // --- ToDo logic here ---
14            dfs(*i);
15        }
16    }

```

Listing 19: 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
13 [u].rend(); ++it)
14            if ( !visited[*it])
15                stk.push(*it);
16    }

```

Listing 20: 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 21: 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 22: 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 23: 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.

```

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 24: 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.

```

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

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            {
22                if( i == V-1 )
23                    return = {-1};
24                dist[v] = dist[u] + w;
25            }
26        }
27    return dist;
28 }

```

Listing 25: 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

```

1 vi topo;
2 vvi graph(V+1,vi());
3 vector<bool> visited(V+1,false);
4 function<void(int)> dfs = [&](int u) -> void {
5     visited[u] = true;
6     for(int v : graph[u])
7         if( ! visited[v] )
8             dfs(v);
9     topo.pb(u);
10 };
11 function<void()> topological_sort = [&]() -> void {
12     for(int i = 1; i <= V; i++)
13         if( ! visited[i] )
14             dfs(i);
15     reverse(all(topo));
16 };
17 topological_sort();
18 for(int i = 0; i < V; i++) cout << topo[i] << " ";

```

Listing 26: Recursive toposort implementation for unweighted DAG through vvi with DFS with inverted postorder. Runs in $O(|V| \times |E|)$.

d

```

1 vi indegree(V+1,0);
2 vvi graph(V+1,vi());
3 vector<bool> visited(V+1,false);
4 for(i,0,E) {
5     graph[u].pb(v);
6     indegree[v]++;
7 }
8 function<vi()> topological_sort = [&]() -> vi {
9     vi order, deg = indegree; // copy
10    queue<int> q;
11    for(int i = 1; i <= V; i++)
12        if( deg[i] == 0 )
13            q.push(i);
14    while( ! q.empty() ) {
15        int u = q.front(); q.pop();
16        order.pb(u);
17        for(int v : graph[u]) {
18            deg[v]--;
19            if(deg[v] == 0)
20                q.push(v);
21        }
22    }
23    return order;
24 };
25 vi topo = topological_sort();
26 if( (int)(topo.size()) != V ) cout << "IMPOSSIBLE"
    << endl;

```

Listing 27: 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)

```

1 class DisjointSets {
2 private:
3     vector<int> parents;
4 public:
5     vector<int> sizes;
6     DisjointSets(int size) : parents(size), sizes(size
7         , 1) {
8         for (int i = 0; i < size; i++) { parents[i] = i;
9         }
10    }
11    /** @return the "representative" node in x's
12     * component */
13    int find( int x ) {
14        return parents[x] == x ? x : ( parents[x] =
15            find( parents[x] ) );
16    }
17    /** @return whether the merge changed connectivity
18     */
19    bool unite( int x, int y ) {
20        int x_root = find(x);
21        int y_root = find(y);
22    }

```

```

17     if (x_root == y_root) return false;
18
19     if ( sizes[x_root] < sizes[y_root] ) swap(x_root,
20         ,y_root);
21
22     sizes[x_root] += sizes[y_root];
23     parents[y_root] = x_root;
24
25     return true;
26 }
27
28 /** @return whether x and y are in the same
29  * connected component */
30
31 bool connected( int x, int y ){
32
33     return find(x) == find(y);
34 }
35
36 void printLists() {
37
38     cout << "Printing parents..." << endl;
39
40     for(auto i : parents)
41
42         cout << i << " ";
43
44     cout << endl << "Printing sizes..." << endl;
45
46     for(auto i : sizes )
47
48         cout << i << " ";
49
50 }
51
52 };
53
54 DisjointSets dsu( V );
55
56 int number_of_components = V, largest_component = 1;
57
58 if( dsu.unite(x, y) ) {
59
60     largest_component = max( largest_component, dsu.
61         sizes[ dsu.find( x ) ] );
62
63     number_of_components--;
64 }

```

Listing 28: Template and usage of DSU with path compression. Complexity of $O(m \alpha(n))$ for a sequence of m operations over n elements. Where α denotes Ackerman function, where $\alpha(n) \leq 4, \forall n \leq 10^{18}$. Practically $O(1)$.

5.9 *Condensation Graph

5.10 *Strongly Connected Components (SCC)

5.11 2-SAT

```
1 struct TwoSatSolver {
2
3     int n_vars;
4
5     int n_vertices;
6
7     vector<vector<int>> adj, adj_t;
8
9     vector<bool> used;
10
11    vector<int> order, comp;
12
13    vector<bool> assignment;
14
15
16
17    TwoSatSolver(int _n_vars) : n_vars(_n_vars),
18    n_vertices(2 * n_vars), adj(n_vertices), adj_t(
19        n_vertices), used(n_vertices), order(), comp(
20        n_vertices, -1), assignment(n_vars) {
21
22        order.reserve(n_vertices);
23
24    }
25
26    void dfs1(int v) {
27
28        used[v] = true;
29
30        for (int u : adj[v]) {
31
32            if (!used[u])
33
34                dfs1(u);
35
36        }
37
38    }
39
40    void solve() {
41
42        for (int i = 0; i < n_vertices; ++i)
43
44            if (!order[i])
45
46                dfs1(i);
47
48        for (int i = 0; i < n_vertices; ++i)
49
50            if (order[i] < 0)
51
52                order[i] = -order[i];
53
54
55        for (int i = 0; i < n_vertices; ++i)
56
57            assignment[i] = order[i] < 0;
58
59
60        for (int i = 0; i < n_vertices; ++i)
61
62            for (int j : adj[i])
63
64                if (order[i] == order[j])
65
66                    adj_t[i].push_back(j);
67
68
69        for (int i = 0; i < n_vertices; ++i)
70
71            if (order[i] < 0)
72
73                dfs1(i);
74
75
76        for (int i = 0; i < n_vertices; ++i)
77
78            if (order[i] < 0)
79
80                assignment[i] = !assignment[i];
81
82
83        for (int i = 0; i < n_vertices; ++i)
84
85            if (order[i] < 0)
86
87                assignment[i] = !assignment[i];
88
89
90        for (int i = 0; i < n_vertices; ++i)
91
92            if (order[i] < 0)
93
94                dfs1(i);
95
96
97        for (int i = 0; i < n_vertices; ++i)
98
99            if (order[i] < 0)
100
101                assignment[i] = !assignment[i];
102
103
104        for (int i = 0; i < n_vertices; ++i)
105
106            if (order[i] < 0)
107
108                dfs1(i);
109
110
111        for (int i = 0; i < n_vertices; ++i)
112
113            if (order[i] < 0)
114
115                assignment[i] = !assignment[i];
116
117
118        for (int i = 0; i < n_vertices; ++i)
119
120            if (order[i] < 0)
121
122                dfs1(i);
123
124
125        for (int i = 0; i < n_vertices; ++i)
126
127            if (order[i] < 0)
128
129                assignment[i] = !assignment[i];
130
131
132        for (int i = 0; i < n_vertices; ++i)
133
134            if (order[i] < 0)
135
136                dfs1(i);
137
138
139        for (int i = 0; i < n_vertices; ++i)
140
141            if (order[i] < 0)
142
143                assignment[i] = !assignment[i];
144
145
146        for (int i = 0; i < n_vertices; ++i)
147
148            if (order[i] < 0)
149
150                dfs1(i);
151
152
153        for (int i = 0; i < n_vertices; ++i)
154
155            if (order[i] < 0)
156
157                assignment[i] = !assignment[i];
158
159
160        for (int i = 0; i < n_vertices; ++i)
161
162            if (order[i] < 0)
163
164                dfs1(i);
165
166
167        for (int i = 0; i < n_vertices; ++i)
168
169            if (order[i] < 0)
170
171                assignment[i] = !assignment[i];
172
173
174        for (int i = 0; i < n_vertices; ++i)
175
176            if (order[i] < 0)
177
178                dfs1(i);
179
180
181        for (int i = 0; i < n_vertices; ++i)
182
183            if (order[i] < 0)
184
185                assignment[i] = !assignment[i];
186
187
188        for (int i = 0; i < n_vertices; ++i)
189
190            if (order[i] < 0)
191
192                dfs1(i);
193
194
195        for (int i = 0; i < n_vertices; ++i)
196
197            if (order[i] < 0)
198
199                assignment[i] = !assignment[i];
200
201
202        for (int i = 0; i < n_vertices; ++i)
203
204            if (order[i] < 0)
205
206                dfs1(i);
207
208
209        for (int i = 0; i < n_vertices; ++i)
210
211            if (order[i] < 0)
212
213                assignment[i] = !assignment[i];
214
215
216        for (int i = 0; i < n_vertices; ++i)
217
218            if (order[i] < 0)
219
220                dfs1(i);
221
222
223        for (int i = 0; i < n_vertices; ++i)
224
225            if (order[i] < 0)
226
227                assignment[i] = !assignment[i];
228
229
230        for (int i = 0; i < n_vertices; ++i)
231
232            if (order[i] < 0)
233
234                dfs1(i);
235
236
237        for (int i = 0; i < n_vertices; ++i)
238
239            if (order[i] < 0)
240
241                assignment[i] = !assignment[i];
242
243
244        for (int i = 0; i < n_vertices; ++i)
245
246            if (order[i] < 0)
247
248                dfs1(i);
249
250
251        for (int i = 0; i < n_vertices; ++i)
252
253            if (order[i] < 0)
254
255                assignment[i] = !assignment[i];
256
257
258        for (int i = 0; i < n_vertices; ++i)
259
260            if (order[i] < 0)
261
262                dfs1(i);
263
264
265        for (int i = 0; i < n_vertices; ++i)
266
267            if (order[i] < 0)
268
269                assignment[i] = !assignment[i];
270
271
272        for (int i = 0; i < n_vertices; ++i)
273
274            if (order[i] < 0)
275
276                dfs1(i);
277
278
279        for (int i = 0; i < n_vertices; ++i)
280
281            if (order[i] < 0)
282
283                assignment[i] = !assignment[i];
284
285
286        for (int i = 0; i < n_vertices; ++i)
287
288            if (order[i] < 0)
289
290                dfs1(i);
291
292
293        for (int i = 0; i < n_vertices; ++i)
294
295            if (order[i] < 0)
296
297                assignment[i] = !assignment[i];
298
299
300        for (int i = 0; i < n_vertices; ++i)
301
302            if (order[i] < 0)
303
304                dfs1(i);
305
306
307        for (int i = 0; i < n_vertices; ++i)
308
309            if (order[i] < 0)
310
311                assignment[i] = !assignment[i];
312
313
314        for (int i = 0; i < n_vertices; ++i)
315
316            if (order[i] < 0)
317
318                dfs1(i);
319
320
321        for (int i = 0; i < n_vertices; ++i)
322
323            if (order[i] < 0)
324
325                assignment[i] = !assignment[i];
326
327
328        for (int i = 0; i < n_vertices; ++i)
329
330            if (order[i] < 0)
331
332                dfs1(i);
333
334
335        for (int i = 0; i < n_vertices; ++i)
336
337            if (order[i] < 0)
338
339                assignment[i] = !assignment[i];
340
341
342        for (int i = 0; i < n_vertices; ++i)
343
344            if (order[i] < 0)
345
346                dfs1(i);
347
348
349        for (int i = 0; i < n_vertices; ++i)
350
351            if (order[i] < 0)
352
353                assignment[i] = !assignment[i];
354
355
356        for (int i = 0; i < n_vertices; ++i)
357
358            if (order[i] < 0)
359
360                dfs1(i);
361
362
363        for (int i = 0; i < n_vertices; ++i)
364
365            if (order[i] < 0)
366
367                assignment[i] = !assignment[i];
368
369
370        for (int i = 0; i < n_vertices; ++i)
371
372            if (order[i] < 0)
373
374                dfs1(i);
375
376
377        for (int i = 0; i < n_vertices; ++i)
378
379            if (order[i] < 0)
380
381                assignment[i] = !assignment[i];
382
383
384        for (int i = 0; i < n_vertices; ++i)
385
386            if (order[i] < 0)
387
388                dfs1(i);
389
390
391        for (int i = 0; i < n_vertices; ++i)
392
393            if (order[i] < 0)
394
395                assignment[i] = !assignment[i];
396
397
398        for (int i = 0; i < n_vertices; ++i)
399
400            if (order[i] < 0)
401
402                dfs1(i);
403
404
405        for (int i = 0; i < n_vertices; ++i)
406
407            if (order[i] < 0)
408
409                assignment[i] = !assignment[i];
410
411
412        for (int i = 0; i < n_vertices; ++i)
413
414            if (order[i] < 0)
415
416                dfs1(i);
417
418
419        for (int i = 0; i < n_vertices; ++i)
420
421            if (order[i] < 0)
422
423                assignment[i] = !assignment[i];
424
425
426        for (int i = 0; i < n_vertices; ++i)
427
428            if (order[i] < 0)
429
430                dfs1(i);
431
432
433        for (int i = 0; i < n_vertices; ++i)
434
435            if (order[i] < 0)
436
437                assignment[i] = !assignment[i];
438
439
440        for (int i = 0; i < n_vertices; ++i)
441
442            if (order[i] < 0)
443
444                dfs1(i);
445
446
447        for (int i = 0; i < n_vertices; ++i)
448
449            if (order[i] < 0)
450
451                assignment[i] = !assignment[i];
452
453
454        for (int i = 0; i < n_vertices; ++i)
455
456            if (order[i] < 0)
457
458                dfs1(i);
459
460
461        for (int i = 0; i < n_vertices; ++i)
462
463            if (order[i] < 0)
464
465                assignment[i] = !assignment[i];
466
467
468        for (int i = 0; i < n_vertices; ++i)
469
470            if (order[i] < 0)
471
472                dfs1(i);
473
474
475        for (int i = 0; i < n_vertices; ++i)
476
477            if (order[i] < 0)
478
479                assignment[i] = !assignment[i];
480
481
482        for (int i = 0; i < n_vertices; ++i)
483
484            if (order[i] < 0)
485
486                dfs1(i);
487
488
489        for (int i = 0; i < n_vertices; ++i)
490
491            if (order[i] < 0)
492
493                assignment[i] = !assignment[i];
494
495
496        for (int i = 0; i < n_vertices; ++i)
497
498            if (order[i] < 0)
499
500                dfs1(i);
501
502
503        for (int i = 0; i < n_vertices; ++i)
504
505            if (order[i] < 0)
506
507                assignment[i] = !assignment[i];
508
509
510        for (int i = 0; i < n_vertices; ++i)
511
512            if (order[i] < 0)
513
514                dfs1(i);
515
516
517        for (int i = 0; i < n_vertices; ++i)
518
519            if (order[i] < 0)
520
521                assignment[i] = !assignment[i];
522
523
524        for (int i = 0; i < n_vertices; ++i)
525
526            if (order[i] < 0)
527
528                dfs1(i);
529
530
531        for (int i = 0; i < n_vertices; ++i)
532
533            if (order[i] < 0)
534
535                assignment[i] = !assignment[i];
536
537
538        for (int i = 0; i < n_vertices; ++i)
539
540            if (order[i] < 0)
541
542                dfs1(i);
543
544
545        for (int i = 0; i < n_vertices; ++i)
546
547            if (order[i] < 0)
548
549                assignment[i] = !assignment[i];
550
551
552        for (int i = 0; i < n_vertices; ++i)
553
554            if (order[i] < 0)
555
556                dfs1(i);
557
558
559        for (int i = 0; i < n_vertices; ++i)
560
561            if (order[i] < 0)
562
563                assignment[i] = !assignment[i];
564
565
566        for (int i = 0; i < n_vertices; ++i)
567
568            if (order[i] < 0)
569
570                dfs1(i);
571
572
573        for (int i = 0; i < n_vertices; ++i)
574
575            if (order[i] < 0)
576
577                assignment[i] = !assignment[i];
578
579
580        for (int i = 0; i < n_vertices; ++i)
581
582            if (order[i] < 0)
583
584                dfs1(i);
585
586
587        for (int i = 0; i < n_vertices; ++i)
588
589            if (order[i] < 0)
590
591                assignment[i] = !assignment[i];
592
593
594        for (int i = 0; i < n_vertices; ++i)
595
596            if (order[i] < 0)
597
598                dfs1(i);
599
600
601        for (int i = 0; i < n_vertices; ++i)
602
603            if (order[i] < 0)
604
605                assignment[i] = !assignment[i];
606
607
608        for (int i = 0; i < n_vertices; ++i)
609
610            if (order[i] < 0)
611
612                dfs1(i);
613
614
615        for (int i = 0; i < n_vertices; ++i)
616
617            if (order[i] < 0)
618
619                assignment[i] = !assignment[i];
620
621
622        for (int i = 0; i < n_vertices; ++i)
623
624            if (order[i] < 0)
625
626                dfs1(i);
627
628
629        for (int i = 0; i < n_vertices; ++i)
630
631            if (order[i] < 0)
632
633                assignment[i] = !assignment[i];
634
635
636        for (int i = 0; i < n_vertices; ++i)
637
638            if (order[i] < 0)
639
640                dfs1(i);
641
642
643        for (int i = 0; i < n_vertices; ++i)
644
645            if (order[i] < 0)
646
647                assignment[i] = !assignment[i];
648
649
650        for (int i = 0; i < n_vertices; ++i)
651
652            if (order[i] < 0)
653
654                dfs1(i);
655
656
657        for (int i = 0; i < n_vertices; ++i)
658
659            if (order[i] < 0)
660
661                assignment[i] = !assignment[i];
662
663
664        for (int i = 0; i < n_vertices; ++i)
665
666            if (order[i] < 0)
667
668                dfs1(i);
669
670
671        for (int i = 0; i < n_vertices; ++i)
672
673            if (order[i] < 0)
674
675                assignment[i] = !assignment[i];
676
677
678        for (int i = 0; i < n_vertices; ++i)
679
680            if (order[i] < 0)
681
682                dfs1(i);
683
684
685        for (int i = 0; i < n_vertices; ++i)
686
687            if (order[i] < 0)
688
689                assignment[i] = !assignment[i];
690
691
692        for (int i = 0; i < n_vertices; ++i)
693
694            if (order[i] < 0)
695
696                dfs1(i);
697
698
699        for (int i = 0; i < n_vertices; ++i)
700
701            if (order[i] < 0)
702
703                assignment[i] = !assignment[i];
704
705
706        for (int i = 0; i < n_vertices; ++i)
707
708            if (order[i] < 0)
709
710                dfs1(i);
711
712
713        for (int i = 0; i < n_vertices; ++i)
714
715            if (order[i] < 0)
716
717                assignment[i] = !assignment[i];
718
719
720        for (int i = 0; i < n_vertices; ++i)
721
722            if (order[i] < 0)
723
724                dfs1(i);
725
726
727        for (int i = 0; i < n_vertices; ++i)
728
729            if (order[i] < 0)
730
731                assignment[i] = !assignment[i];
732
733
734        for (int i = 0; i < n_vertices; ++i)
735
736            if (order[i] < 0)
737
738                dfs1(i);
739
740
741        for (int i = 0; i < n_vertices; ++i)
742
743            if (order[i] < 0)
744
745                assignment[i] = !assignment[i];
746
747
748        for (int i = 0; i < n_vertices; ++i)
749
750            if (order[i] < 0)
751
752                dfs1(i);
753
754
755        for (int i = 0; i < n_vertices; ++i)
756
757            if (order[i] < 0)
758
759                assignment[i] = !assignment[i];
760
761
762        for (int i = 0; i < n_vertices; ++i)
763
764            if (order[i] < 0)
765
766                dfs1(i);
767
768
769        for (int i = 0; i < n_vertices; ++i)
770
771            if (order[i] < 0)
772
773                assignment[i] = !assignment[i];
774
775
776        for (int i = 0; i < n_vertices; ++i)
777
778            if (order[i] < 0)
779
780                dfs1(i);
781
782
783        for (int i = 0; i < n_vertices; ++i)
784
785            if (order[i] < 0)
786
787                assignment[i] = !assignment[i];
788
789
790        for (int i = 0; i < n_vertices; ++i)
791
792            if (order[i] < 0)
793
794                dfs1(i);
795
796
797        for (int i = 0; i < n_vertices; ++i)
798
799            if (order[i] < 0)
800
801                assignment[i] = !assignment[i];
802
803
804        for (int i = 0; i < n_vertices; ++i)
805
806            if (order[i] < 0)
807
808                dfs1(i);
809
810
811        for (int i = 0; i < n_vertices; ++i)
812
813            if (order[i] < 0)
814
815                assignment[i] = !assignment[i];
816
817
818        for (int i = 0; i < n_vertices; ++i)
819
820            if (order[i] < 0)
821
822                dfs1(i);
823
824
825        for (int i = 0; i < n_vertices; ++i)
826
827            if (order[i] < 0)
828
829                assignment[i] = !assignment[i];
830
831
832        for (int i = 0; i < n_vertices; ++i)
833
834            if (order[i] < 0)
835
836                dfs1(i);
837
838
839        for (int i = 0; i < n_vertices; ++i)
840
841            if (order[i] < 0)
842
843                assignment[i] = !assignment[i];
844
845
846        for (int i = 0; i < n_vertices; ++i)
847
848            if (order[i] < 0)
849
850                dfs1(i);
851
852
853        for (int i = 0; i < n_vertices; ++i)
854
855            if (order[i] < 0)
856
857                assignment[i] = !assignment[i];
858
859
860        for (int i = 0; i < n_vertices; ++i)
861
862            if (order[i] < 0)
863
864                dfs1(i);
865
866
867        for (int i = 0; i < n_vertices; ++i)
868
869            if (order[i] < 0)
870
871                assignment[i] = !assignment[i];
872
873
874        for (int i = 0; i < n_vertices; ++i)
875
876            if (order[i] < 0)
877
878                dfs1(i);
879
880
881        for (int i = 0; i < n_vertices; ++i)
882
883            if (order[i] < 0)
884
885                assignment[i] = !assignment[i];
886
887
888        for (int i = 0; i < n_vertices; ++i)
889
890            if (order[i] < 0)
891
892                dfs1(i);
893
894
895        for (int i = 0; i < n_vertices; ++i)
896
897            if (order[i] < 0)
898
899                assignment[i] = !assignment[i];
900
901
902        for (int i = 0; i < n_vertices; ++i)
903
904            if (order[i] < 0)
905
906                dfs1(i);
907
908
909        for (int i = 0; i < n_vertices; ++i)
910
911            if (order[i] < 0)
912
913                assignment[i] = !assignment[i];
914
915
916        for (int i = 0; i < n_vertices; ++i)
917
918            if (order[i] < 0)
919
920                dfs1(i);
921
922
923        for (int i = 0; i < n_vertices; ++i)
924
925            if (order[i] < 0)
926
927                assignment[i] = !assignment[i];
928
929
930        for (int i = 0; i < n_vertices; ++i)
931
932            if (order[i] < 0)
933
934                dfs1(i);
935
936
937        for (int i = 0; i < n_vertices; ++i)
938
939            if (order[i] < 0)
940
941                assignment[i] = !assignment[i];
942
943
944        for (int i = 0; i < n_vertices; ++i)
945
946            if (order[i] < 0)
947
948                dfs1(i);
949
950
951        for (int i = 0; i < n_vertices; ++i)
952
953            if (order[i] < 0)
954
955                assignment[i] = !assignment[i];
956
957
958        for (int i = 0; i < n_vertices; ++i)
959
960            if (order[i] < 0)
961
962                dfs1(i);
963
964
965        for (int i = 0; i < n_vertices; ++i)
966
967            if (order[i] < 0)
968
969                assignment[i] = !assignment[i];
970
971
972        for (int i = 0; i < n_vertices; ++i)
973
974            if (order[i] < 0)
975
976                dfs1(i);
977
978
979        for (int i = 0; i < n_vertices; ++i)
980
981            if (order[i] < 0)
982
983                assignment[i] = !assignment[i];
984
985
986        for (int i = 0; i < n_vertices; ++i)
987
988            if (order[i] < 0)
989
990                dfs1(i);
991
992
993        for (int i = 0; i < n_vertices; ++i)
994
995            if (order[i] < 0)
996
997                assignment[i] = !assignment[i];
998
999
1000       for (int i = 0; i < n_vertices; ++i)
1001
1002           if (order[i] < 0)
1003
1004               dfs1(i);
1005
1006
1007       for (int i = 0; i < n_vertices; ++i)
1008
1009           if (order[i] < 0)
1010
1011               assignment[i] = !assignment[i];
1012
1013
1014       for (int i = 0; i < n_vertices; ++i)
1015
1016           if (order[i] < 0)
1017
1018               dfs1(i);
1019
1020
1021       for (int i = 0; i < n_vertices; ++i)
1022
1023           if (order[i] < 0)
1024
1025               assignment[i] = !assignment[i];
1026
1027
1028       for (int i = 0; i < n_vertices; ++i)
1029
1030           if (order[i] < 0)
1031
1032               dfs1(i);
1033
1034
1035       for (int i = 0; i < n_vertices; ++i)
1036
1037           if (order[i] < 0)
1038
1039               assignment[i] = !assignment[i];
1040
1041
1042       for (int i = 0; i < n_vertices; ++i)
1043
1044           if (order[i] < 0)
1045
1046               dfs1(i);
1047
1048
1049       for (int i = 0; i < n_vertices; ++i)
1050
1051           if (order[i] < 0)
1052
1053               assignment[i] = !assignment[i];
1054
1055
1056       for (int i = 0; i < n_vertices; ++i)
1057
1058           if (order[i] < 0)
1059
1060               dfs1(i);
1061
1062
1063       for (int i = 0; i < n_vertices; ++i)
1064
1065           if (order[i] < 0)
1066
1067               assignment[i] = !assignment[i];
1068
1069
1070       for (int i = 0; i < n_vertices; ++i)
1071
1072           if (order[i] < 0)
1073
1074               dfs1(i);
1075
1076
1077       for (int i = 0; i < n_vertices; ++i)
1078
1079           if (order[i] < 0)
1080
1081               assignment[i] = !assignment[i];
1082
1083
1084       for (int i = 0; i < n_vertices; ++i)
1085
1086           if (order[i] < 0)
1087
1088               dfs1(i);
1089
1090
1091       for (int i = 0; i < n_vertices; ++i)
1092
1093           if (order[i] < 0)
1094
1095               assignment[i] = !assignment[i];
1096
1097
1098       for (int i = 0; i < n_vertices; ++i)
1099
1100           if (order[i] < 0)
1101
1102               dfs1(i);
1103
1104
1105       for (int i = 0; i < n_vertices; ++i)
1106
1107           if (order[i] < 0)
1108
1109               assignment[i] = !assignment[i];
1110
1111
1112       for (int i = 0; i < n_vertices; ++i)
1113
1114           if (order[i] < 0)
1115
1116               dfs1(i);
1117
1118
1119       for (int i = 0; i < n_vertices; ++i)
1120
1121           if (order[i] < 0)
1122
1123               assignment[i] = !assignment[i];
1124
1125
1126       for (int i = 0; i < n_vertices; ++i)
1127
1128           if (order[i] < 0)
1129
1130               dfs1(i);
1131
1132
1133       for (int i = 0; i < n_vertices; ++i)
1134
1135           if (order[i] < 0)
1136
1137               assignment[i] = !assignment[i];
1138
1139
1140       for (int i = 0; i < n_vertices; ++i)
1141
1142           if (order[i] < 0)
1143
1144               dfs1(i);
1145
1146
1147       for (int i = 0; i < n_vertices; ++i)
1148
1149           if (order[i] < 0)
1150
1151               assignment[i] = !assignment[i];
1152
1153
1154       for (int i = 0; i < n_vertices; ++i)
1155
1156           if (order[i] < 0)
1157
1158               dfs1(i);
1159
1160
1161       for (int i = 0; i < n_vertices; ++i)
1162
1163           if (order[i] < 0)
1164
1165               assignment[i] = !assignment[i];
1166
1167
1168       for (int i = 0; i < n_vertices; ++i)
1169
1170           if (order[i] < 0)
1171
1172               dfs1(i);
1173
1174
1175       for (int i = 0; i < n_vertices; ++i)
1176
1177           if (order[i] < 0)
1178
1179               assignment[i] = !assignment[i];
1180
1181
1182       for (int i = 0; i < n_vertices; ++i)
1183
1184           if (order[i] < 0)
1185
1186               dfs1(i);
1187
1188
1189       for (int i = 0; i < n_vertices; ++i)
1190
1191           if (order[i] < 0)
1192
1193               assignment[i] = !assignment[i];
1194
1195
1196       for (int i = 0; i < n_vertices; ++i)
1197
1198           if (order[i] < 0)
1199
1200               dfs1(i);
1201
1202
1203       for (int i = 0; i < n_vertices; ++i)
1204
1205           if (order[i] < 0)
1206
1207               assignment[i] = !assignment[i];
1208
1209
1210       for (int i = 0; i < n_vertices; ++i)
1211
1212           if (order[i] < 0)
1213
1214               dfs1(i);
1215
1216
1217       for (int i = 0; i < n_vertices; ++i)
1218
1219           if (order[i] < 0)
1220
1221               assignment[i] = !assignment[i];
1222
1223
1224       for (int i = 0; i < n_vertices; ++i)
1225
1226           if (order[i] < 0)
1227
1228               dfs1(i);
1229
1230
1231       for (int i = 0; i < n_vertices; ++i)
1232
1233           if (order[i] < 0)
1234
1235               assignment[i] = !assignment[i];
1236
1237
1238       for (int i = 0; i < n_vertices; ++i)
1239
1240           if (order[i] < 0)
1241
1242               dfs1(i);
1243
1244
1245       for (int i = 0; i < n_vertices; ++i)
1246
1247           if (order[i] < 0)
1248
1249               assignment[i] = !assignment[i];
1250
1251
1252       for (int i = 0; i < n_vertices; ++i)
1253
1254           if (order[i] < 0)
1255
1256               dfs1(i);
1257
1258
1259       for (int i = 0; i < n_vertices; ++i)
1260
1261           if (order[i] < 0)
1262
1263               assignment[i] = !assignment[i];
1264
1265
1266       for (int i = 0; i < n_vertices; ++i)
1267
1268           if (order[i] < 0)
1269
1270               dfs1(i);
1271
1272
1273       for (int i = 0; i < n_vertices; ++i)
1274
1275           if (order[i] < 0)
1276
1277               assignment[i] = !assignment[i];
1278
1279
1280       for (int i = 0; i < n_vertices; ++i)
1281
1282           if (order[i] < 0)
1283
1284               dfs1(i);
1285
1286
1287       for (int i = 0; i < n_vertices; ++i)
1288
1289           if (order[i] < 0)
1290
1291               assignment[i] = !assignment[i];
1292
1293
1294       for (int i = 0; i < n_vertices; ++i)
1295
1296           if (order[i] < 0)
1297
1298               dfs1(i);
1299
1300
1301       for (int i = 0; i < n_vertices; ++i)
1302
1303           if (order[i] < 0)
1304
1305               assignment[i] = !assignment[i];
1306
1307
1308       for (int i = 0; i < n_vertices; ++i)
1309
1310           if (order[i] < 0)
1311
1312               dfs1(i);
1313
1314
1315       for (int i = 0; i < n_vertices; ++i)
1316
1317           if (order[i] < 0)
1318
1319               assignment[i] = !assignment[i];
1320
1321
1322       for (int i = 0; i < n_vertices; ++i)
1323
1324           if (order[i] < 0)
1325
1326               dfs1(i);
1327
1328
1329       for (int i = 0; i < n_vertices; ++i)
1330
1331           if (order[i] < 0)
1332
1333               assignment[i] = !assignment[i];
1334
1335
1336       for (int i = 0; i < n_vertices; ++i)
1337
1338           if (order[i] < 0)
1339
1340               dfs1(i);
1341
1342
1343       for (int i = 0; i < n_vertices; ++i)
1344
1345           if (order[i] < 0)
1346
1347               assignment[i] = !assignment[i];
1348
1349
1350       for (int i = 0; i < n_vertices; ++i)
1
```

```
17     }
18     order.push_back(v);
19 }
20
21 void dfs2(int v, int cl) {
22     comp[v] = cl;
23     for (int u : adj_t[v]) {
24         if (comp[u] == -1)
25             dfs2(u, cl);
26     }
27 }
28
29 bool solve_2SAT() {
30     order.clear();
31     used.assign(n_vertices, false);
32     for (int i = 0; i < n_vertices; ++i) {
33         if (!used[i])
34             dfs1(i);
35     }
36
37     comp.assign(n_vertices, -1);
38     for (int i = 0, j = 0; i < n_vertices; ++i)
39     {
40         int v = order[n_vertices - i - 1];
41         if (comp[v] == -1)
42             dfs2(v, j++);
43     }
44     assignment.assign(n_vars, false);
45     for (int i = 0; i < n_vertices; i += 2) {
46         if (comp[i] == comp[i + 1])
47             return false;
48         assignment[i / 2] = comp[i] > comp[i + 1];
49     }
50     return true;
51 }
52
53 void add_disjunction(int a, bool na, int b, bool nb) {
54     // na and nb signify whether a and b are to
55     // be negated
56     a = 2 * a ^ na;
57     b = 2 * b ^ nb;
58     int neg_a = a ^ 1;
59     int neg_b = b ^ 1;
60     adj[neg_a].push_back(b);
61     adj[neg_b].push_back(a);
62     adj_t[b].push_back(neg_a);
63     adj_t[a].push_back(neg_b);
64 }
65 static void example_usage() {
66     TwoSatSolver solver(3); // a, b, c
67     solver.add_disjunction(0, false, 1, true);
68     //    a v not b
```

```

68     solver.add_disjunction(0, true, 1, true);
// not a v not b
69     solver.add_disjunction(1, false, 2, false);
// b v c
70     solver.add_disjunction(0, false, 0, false);
// a v a
71     assert(solver.solve_2SAT() == true);
72     auto expected = vector<bool>(True, False,
True);
    assert(solver.assignment == expected);
73 }
74
75 };

```

Listing 29: 2-SAT implementation from cp-algorithms.com. Each component added is a expression of the form $a \vee b$, which is equivalent to $\neg a \Rightarrow b \wedge \neg b \Rightarrow a$ (if one of the variables is false, then the other one must be true). A directed graph is constructed based on these implication: For each x , there's two vertices v_x and $v_{\neg x}$. If there is an edge $a \Rightarrow b$, then there also is an edge $\neg b \Rightarrow \neg a$. For any x , if x is reachable from $\neg x$ and $\neg x$ is reachable from x , the problem has no solution. This means, each variable must be in a different SCC than their negative. This is verified by the method `solve_2SAT()`, which returns a boolean: `True` if it has a solution and `False` if it doesn't.

Giant Pizza How does a particular 2-SAT problem look like? Following is the statement for the problem CSES 1684 (Giant Pizza):

Uolevi's family is going to order a large pizza and eat it together. A total of n family members will join the order, and there are m possible toppings. The pizza may have any number of toppings. Each family member gives two

wishes concerning the toppings of the pizza. The wishes are of the form " $\text{topping } x \text{ is good/bad}$ ". Your task is to choose the toppings so that at least one wish from everybody becomes true (a good topping is included in the pizza or a bad topping is not included).

Input

The first input line has two integers n and m : the number of family members and toppings. The toppings are numbered $1, 2, \dots, m$. After this, there are n lines describing the wishes. Each line has two wishes of the form " $+x$ " (topping x is good) or " $-x$ " (topping x is bad).

Output

Print a line with m symbols: for each topping " $+$ " if it is included and " $-$ " if it is not included. You can print any valid solution. If there are no valid solutions, print "IMPOSSIBLE".

```

1 int main(){
2     fastIO();
3     int n = nxt(), m = nxt();
4     TwoSatSolver TwoSat(m);
5
6     for(i, 0, n){
7         char type1, type2;
8         int top1, top2;
9         cin >> type1 >> top1 >> type2 >> top2;
10
11        top1--; top2--;
12        TwoSat.add_disjunction(top1, type1 == '+' ?
13            top2, type2 == '+' : );
14
15        if(TwoSat.solve_2SAT()){
16            for(i, 0, m){

```

```

17     if(TwoSat.assignment[i])
18         cout << "+ ";
19     else
20         cout << "- ";
21     }
22 }
23 else cout << "IMPOSSIBLE";
24 return 0;
25 }
```

```

13     return;
14 }
15 for( ii move : dirs ) {
16     newI = i + move.fi; newJ = j + move.se;
17     if( newI >= 0 && newI < n && newJ >= 0 && newJ <
18         m )
19         traverse(newI, newJ);
20 }
21 for(i,0,n)
22     cin >> grid[i];
23 for(i,0,n) {
24     for(j,0,m) {
25         if( grid[i][j] == 'S' ) {
26             grid[i][j] = '.';
27             start.fi = i;
28             start.se = j;
29         }
30     }
31 }
32 traverse(start.fi, start.se);
33 cout << arns << endl;
```

Listing 30: Main method for solving CSES 1684 Giant Pizza using 2-SAT template.

5.12 *Bridges and point articulation

5.13 Flood Fill

```

1 vector<string> grid(n);
2 vii dirs = {{0,1},{0,-1},{1,0}, {-1,0}};
3 ii start;
4 int arns = 0;
5 function<void(int,int)> traverse = [&](int i, int j)
6     -> void {
7     if( grid[i][j] == '#' ) return;
8     int newI, newJ;
9     if( grid[i][j] != '.' ) arns += grid[i][j] - '0';
10    grid[i][j] = '#';
11    for( ii move : dirs ) {
12        newI = i + move.fi; newJ = j + move.se;
13        if( newI >= 0 && newI < n && newJ >= 0 && newJ <
14            m && grid[newI][newJ] == 'T' )
```

Listing 31: 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)

```

1  typedef array<int,3> iii;
2
3  vii dirs = {{1,0},{0,1},{-1,0},{0,-1}};
4  map<int,string> path = {{0,"D"},{1,"R"},{2,"U"},{3,"L"}};
5  int n, m;
6  string arns = "";
7  bool escaped = false;
8  cin >> n >> m;
9  vector<string> grid(n);
10 vvi times(n,vi(m,INF)), prev(n,vi(m,-1));
11 vector<vector<bool>> visited(n,vector<bool>(m,false));
12 queue<iii> q;
13 ii start, end;
14 for(int i = 0; i < n; i++) {

```

```

15    cin >> grid[i];
16    for(int j = 0; j < m; j++) {
17      if( grid[i][j] == 'M' ) {
18        q.push({i,j,0});
19        times[i][j] = 0;
20      }
21      else if( grid[i][j] == 'A' )
22        start = {i,j};
23    }
24  }
25  function<bool(int,int)> valid = [&](int I, int J) ->
26    bool {
27    return (I >= 0) and (I < n) and (J >= 0) and (J <
28      m) and (grid[I][J] != '#') and (times[I][J] ==
29      INF);
30  };
31  function<bool(int,int)> valid_player = [&](int I,
32    int J) -> bool {
33    return (I >= 0) and (I < n) and (J >= 0) and (J <
34      m) and (!visited[I][J]) and (grid[I][J] != '#');
35  };
36  function<bool(int,int)> is_border = [&](int I, int J
37    ) -> bool {
38    return I == 0 || I == n-1 || J == 0 || J == m-1;
39  };
40  // Corner cases
41  if( is_border(start.fi,start.se) ) {
42    cout << "YES" << endl << "0" << endl;
43}

```

```
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][
64             newJ] ) {
65             visited[newI][newJ] = true;
66             prev[newI][newJ] = i;
67             q.push({newI,newJ,w});
68             if( is_border(newI,newJ) ) {
69                 end.fi = newI;
70                 end.se = newJ;
71                 escaped = true;
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 32: 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 Algorithm

```

1 const ll INF = 1e17;
2 /**
3 * @brief Represents a directed edge in a flow
4 * network.
5 * @details Stores the edge's source, destination,
6 * capacity, and current flow.
7 *          Used in max-flow algorithms like Dinic
8 *          or Ford-Fulkerson. */
9 struct flowEdge {
10     int u; // Source node
11     int v; // Destination node
12     ll cap; // Maximum flow capacity of the edge
13     ll flow = 0; // Current flow through the edge (
14     // initially 0)
15     flowEdge( int u, int v, ll cap ) : u(u), v(v), cap
16     (cap) {};
17 };
18 /**
19 */

```

```

14     * @brief Implementation of Dinic's max-flow
15     * algorithm.
16     * @details Manages a flow network with BFS (Level
17     * Graph) and DFS (Blocking Flow) optimizations. */
18     struct Dinic {
19         vector<flowEdge> edges; // All edges in the flow
20         network (including reverse edges)
21         vector<vi> adj;
22         int n; // Total number of nodes in the graph
23         int s; // Source node
24         int t; // Sink node (destination of flow)
25         int id = 0; // Counter for edge indexing
26         vi level; // Stores the level (distance from 's')
27         of each node during BFS
28         vi next; // Optimization for DFS: tracks the next
29         edge to explore for each node
30         queue<int> q; // Queue for BFS traversal
31         /**
32          * @brief Constructs a Dinic solver for a flow
33          * network.
34          * @param n Number of nodes.
35          * @param s Source node.
36          * @param t Sink node. */
37         Dinic( int n, int s, int t ) : n(n), s(s), t(t) {
38             adj.resize(n); // Initialize adjacency list for
39             'n' nodes.
40             level.resize(n); // Prepare level array for BFS.
41         }
42     };
43 
```

```
34     next.resize(n); // Prepare next-edge array for
35     DFS.
36
37     fill(all(level), -1); // Mark all levels as
38     unvisited (-1).
39
40     level[s] = 0; // The source has level 0.
41     q.push(s); // Start BFS from the source.
42 }
43 /**
44 * @brief Adds a directed edge and its residual
45 * reverse edge to the flow network. */
46 void addEdge( int u, int v, ll cap ) {
47     edges.emplace_back(u,v,cap); // Original edge: u
48     -> v
49     edges.emplace_back(v,u,0); // Residual edge: v
50     -> u
51
52     adj[u].pb(id++);
53     adj[v].pb(id++);
54 }
55 /**
56 * @brief Performs BFS to construct the level
57 * graph (Layered Network) from source 's' to sink
58 * 't'.
59 * @details Assigns levels (minimum distances
60 * from 's') to all nodes and checks if 't' is
61 * reachable.
62 *
63 *          Levels are used to guide the DFS
64 * phase in Dinic's algorithm.
65
66     * @return bool True if the sink 't' is reachable
67     * (i.e., there exists an augmenting path), false
68     * otherwise. */
69     bool bfs() {
70         while( ! q.empty() ) {
71             int curr = q.front();
72             q.pop();
73
74             for( auto e : adj[curr] ) {
75                 if( edges[e].cap - edges[e].flow < 1 ) // Skip
76                 saturated edges (no residual capacity).
77                 continue;
78
79                 if( level[edges[e].v] != -1 ) // Skip
80                 already visited nodes (level assigned).
81                 continue;
82
83                 // Assign level to the neighbor node.
84                 level[edges[e].v] = level[edges[e].u] +
85                 1; // Next level = current + 1.
86
87                 q.push(edges[e].v); // Add neighbor to the
88                 queue for further BFS.
89             }
90         }
91
92         return level[t] != -1; // Return whether the
93         sink 't' was reached (level[t] != -1).
94     }
95
96 /**
97 * @brief Finds a blocking flow using DFS in the
98 * level graph constructed by BFS.
99 * @param u Current node being processed.
```

```
71     * @param flow Maximum flow that can be sent from
72     *          'u' to the sink 't'.
73
74     * @return ll The amount of flow successfully
75     *          sent to 't'. */
76
77     ll dfs( int u, ll flow ) {
78
79         if( flow == 0 ) // No remaining flow to send.
80             return 0;
81
82         if( u == t ) // Reached the sink; return
83             accumulated flow.
84
85         return flow;
86
87         // Explore edges from 'u' using 'next[u]' to
88         // avoid revisiting processed edges.
89
90         for( int& cid = next[u]; cid < sz(adj[u]); cid++ )
91             {
92
93                 int e = adj[u][cid]; // Index of the edge in
94                 // 'edges'.
95
96                 int v = edges[e].v; // Destination node of
97                 // the edge.
98
99                 // Skip invalid edges:
100
101                // 1. Not in the level graph (level[u] + 1 != level[v]). Just edges in exactly one level ahead (ensures shortest paths).
102
103                // 2. No residual capacity (cap - flow < 1).
104
105                if( level[edges[e].u] + 1 != level[v] || edges[e].cap - edges[e].flow < 1 )
106
107                    continue;
108
109                    ll f = dfs( v, min(flow, edges[e].cap - edges[e].flow) ); // Recursively send flow to 'v'.
110
111                    if( f == 0 ) // No flow could be sent via this
112                        edge.
113
114                        continue;
115
116                        // Update residual capacities:
117
118                        edges[e].flow += f; // Increase flow in
119                        // the original edge.
120
121                        edges[ e ^ 1 ].flow -= f; // Decrease flow in
122                        // the reverse edge. (All reverse edges have
123                        // distinct parity)
124
125                        return f; // Return the flow
126
127                        sent.
128
129
130                    }
131
132                    return 0; // No augmenting path found from 'u'.
133
134    }
135
136
137    /**
138     * @brief Computes the maximum flow from source 's' to sink 't' using Dinic's algorithm.
139     * @details Iterates through BFS and DFS phases
140     *          to find the maximum flow.
141
142     * Accumulates flow while there exists
143     * augmentation paths in the residual graph.
144
145     * Restart auxiliary structures for every new
146     * phase.
147
148     * @return ll The maximum flow value. */
149
150     ll maxFlow() {
151
152         ll flow = 0; // Tracks the total flow sent.
153
154         while( bfs() ) { // While there are augmenting
155
156             paths:
```

```
106     fill(all(next),0); // Reset 'next' for DFS.  
107     for( ll f = dfs(s,INF); f != 0ll; f = dfs(s,  
108         INF) ) // Send blocking flow in the level graph:  
109         flow += f;  
110         // Reset for next BFS phase:  
111         fill(all(level),-1);  
112         level[s] = 0;  
113         q.push(s);  
114     }  
115     return flow;  
116 }  
117 /**  
118 * @brief Finds edges belonging to the minimum  
119 cut after maxFlow().  
120 * @details First, it marks all the reachable  
121 nodes from 's' with an augmentation path after  
122 obtained the max flow  
123     and all the saturated edges coming out from  
124 any of the nodes who belong to the min-cut.  
125     For 'minCut()' to work, 'maxFlow()' must be  
126 first executed to get the min-cut.  
127     If only is needed the value, is enough  
128 returning the value of 'maxFlow()'.  
129     * @return vii List of edges (u, v) in the min-  
130 cut. Its size is the minimum number of 'roads' to  
131 close. */  
132 vii minCut() {  
133     vii ans;  
134     fill(all(level),-1); // Reset levels.  
135     level[s] = 0;         // Mark source as reachable  
136     .  
137     q.push(s);  
138     while( ! q.empty() ) { // BFS to mark nodes  
139         reachable from 's' in the residual graph.  
140         int curr = q.front();  
141         q.pop();  
142         for( int id = 0; id < sz(adj[curr]); id++ ) {  
143             // For every edge going out from 'curr'.  
144             int e = adj[curr][id];  
145             // If 'v' is has not been visited yet, and  
146             // the edge have residual capacity.  
147             if( level[edges[e].v] == -1 && edges[e].cap  
148 - edges[e].flow > 0 ) {  
149                 q.push(edges[e].v);  
150                 level[edges[e].v] = level[edges[e].u] + 1;  
151             }  
152         }  
153     }  
154     for( int i = 0; i < sz(level); i++ ) {  
155         if( level[i] != -1 ) {  
156             for( int id = 0; id < sz(adj[i]); id++ ) {  
157                 int e = adj[i][id];  
158                 if( level[edges[e].v] == -1 && edges[e].  
159 cap - edges[e].flow == 0 )  
160                     ans.emplace_back(edges[e].u,edges[e].v);  
161             }  
162         }  
163     }  
164 }
```

```
147     }
148 }
149 return ans;
150 }
151 /**
152 * @brief Reconstructs the maximum bipartite
153 matching after running 'maxFlow()'.
```

* @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
156     * @return vii List of matched pairs (boy, girl).
157     */
158 vii maximumMatching() {
159     vii ans;
160     fill(all(level), -1); // Reset levels.
161     level[s] = 0;          // Mark source as reachable
162     .
163     q.push(s);
164     while( ! q.empty() ) { // BFS to mark nodes
165         reachable via saturated edges with flow greater
166         than zero.
167         int curr = q.front();
168         q.pop();
169         for( int id = 0; id < sz(adj[curr]); id++ ) {
170             int e = adj[curr][id];
171             if( level[edges[e].v] == -1 && edges[e].cap
172 - edges[e].flow == 0 && edges[e].flow != 011 ) {
173                 q.push(edges[e].v);
174                 level[edges[e].v] = level[edges[e].u] + 1;
175             }
176         }
177         for( int i = 0; i < sz(level); i++ ) { // Collect original edges (boy -> girl) that are
178             saturated and have flow > 0.
179             if( level[i] != -1 ) {
180                 for( int id = 0; id < sz(adj[i]); id++ ) {
181                     int e = adj[i][id];
182                     if( edges[e].u != s && edges[e].v != t
183 && edges[e].cap - edges[e].flow == 0 && edges[e].
184 flow != 011 )
185                         ans.emplace_back(edges[e].u, edges[e].v
186 );
187                 }
188             }
189         }
190     }
191     return ans;
192 }
```

Listing 33: 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 $1, 2, \dots, n$. 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.

```

1 int main()
2 {
3     fastIO();
4
5     int n, m, u, v, w;
6
7     cin >> n >> m;
8
9     Dinic flow(n+1, 1, n); // size n+1 to fix 0-
```

indexed indexes, 1 is the source (server), 'n' is the sink (Kotivalo)

```

10
11     for(i, 0, m)
12     {
13         cin >> u >> v >> w;
14         flow.addEdge(u, v, w);
15     }
16
17     cout << flow.maxFlow() << endl;
18
19     return 0;
20 }
```

Listing 34: 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 $1, 2, \dots, n$. 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-
11        indexed indexes, 1 is the source (bank), 'n' is
12        the sink (harbor)
13
14    for(i, 0, m)
15    {
16        cin >> u >> v;
17        flow.addEdge(u, v, 1);
18        flow.addEdge(v, u, 1);
19
20    }
21
22    flow.maxFlow();
23    minCut = flow.minCut();
24
25    cout << (sz(minCut)/2) << endl;
26
27 }
```

```

23     for(int i = 0; i < sz(minCut); i += 2)
24         cout << minCut[i].fi << " " << minCut[i].se
25         << endl;
26
27     return 0;
28 }
```

Listing 35: 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 $1, 2, \dots, n$, and the girls are numbered $1, 2, \dots, m$.

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 36: 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 37: 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 (KMP)

```
1 // Longest Prefix-Suffix
2 vi compute_LPS(string s) {
3     size_t len = 0, i = 1, sz = s.size();
4     vi lps(sz, 0);
5     while(i < sz) {
6         if(s[i] == s[len])
7             lps[i++] = ++len;
8         else
9             if(len != 0)
10                 len = lps[len-1];
11             else
12                 lps[i++] = 0;
13     }
14     return lps;
```

```
15 }
16 // Get number of occurrences of a pattern p in a
17 string s
18 int kmp(string s, string p) {
19     vi lps = compute_LPS(p);
20     size_t n = s.size(), m = p.size(), i = 0, j = 0;
21     int cnt = 0;
22     while(i < n) {
23         if(p[j] == s[i])
24             j++, i++;
25     }
26     if(j == m) { // Full match
27         cnt++;
28         j = lps[j-1];
29     }
30     else if(i < n and p[j] != s[i]) { // Mismatch
31         after j matches
32         if(j != 0)
33             j = lps[j-1];
34         else
35             i++;
36     }
37 }
```

Listing 38: KMP algorithm for counting how many times a pattern appear into a string. Runs in $O(n + m)$.

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)

```

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 39: 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 $O(1)$.

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

$$n = \sqrt{2S + \frac{1}{4}} - \frac{1}{2} \quad (2)$$

```

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

Listing 40: 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
9         a = ( 1LL * a * a ) % MOD;
10
11        b >= 1;
12
13    }
14
15    return sol % MOD;
16
17 }
```

Listing 41: Applying binary exponentiation to a problem requiring $a^b \bmod(10^9 + 7)$ in $O(\log_2(b))$.

```
15     Matrix(int n, int m) {
16         mat.resize(n);
17         for(int i = 0; i < n; i++) {mat[i].resize(m);
18     }
19     }
20     int rows() const { return mat.size(); }
21     int cols() const { return mat[0].size(); }
22     void makeIdent() {
23         for(int i = 0; i < rows(); i++)
24             for(int j = 0; j < cols(); j++)
25                 mat[i][j] = (i == j ? 1 : 0);
26     }
27     Matrix operator*=(const Matrix &b) {
28         matmul(mat, b.mat);
29         return *this;
30     }
31     void print() {
32         for(int i = 0; i < rows(); i++) {
33             for(int j = 0; j < cols(); j++)
34                 cout << mat[i][j] << " ";
35             cout << endl;
36         }
37     }
38     Matrix operator*(const Matrix &b) { return Matrix
39         (*this) *= b; }
40 };
41 int main() {
42     Matrix<ll> A( {{1,1},{1,0}} );
43 }
```

9.1.8 Matrix Exponentiation (Linear Recurrency)

```
1 template <typename T> void matmul(vector<vector<T>>
2     &a, const vector<vector<T>>& b) {
3     size_t n = a.size(), m = a[0].size(), p = b[0].
4     size();
5     assert(m == b.size());
6     vector<vector<T>> c(n, vector<T>(p));
7     for(size_t i = 0; i < n; i++)
8         for(size_t j = 0; j < p; j++)
9             for(size_t k = 0; k < m; k++)
10                 c[i][j] = (c[i][j] + a[i][k] * b[k][j])
11                 % MOD;
12     a = c;
13 }
14 template <typename T> struct Matrix {
15     vector<vector<T>> mat;
16     Matrix() {}
17     Matrix(vector<vector<T>> a) { mat = a; }
```

```

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 42: 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

```

1 bool prime( int n ){
2     if( n == 2 )
3         return true;
4     if( n % 2 == 0 || n <= 1 )
5         return false;
```

```

6     for( int i = 3; i * i <= n; i += 2 )
7         if( ( n % i ) == 0 )
8             return false;
9     return true;
10 }
```

Listing 43: 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

```

1 void prime_factorization(vll& factorization, ll n) {
2     for(long long d = 2; d*d <= n; d++) {
3         while(n % d == 0) {
4             factorization.push_back(d);
5             n /= d;
6         }
7     }
8     if( n > 1 )
9         factorization.push_back(n);
10 }
```

Listing 44: 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

```

1 void sieve_of_eratosthenes(vector<bool>& is_prime,
2                             int n) {
3     is_prime.assign(n+1,true);
```

```

3     is_prime[0] = is_prime[1] = false;
4     for(int i = 2; i <= n; i++) {
5         if( is_prime[i] && (long long)i * i <= n ) {
6             for(int j = i*i; j <= n; j += i)
7                 is_prime[j] = false;
8         }
9     }
10 }
```

Listing 45: 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

```

1 ll sum_of_divisors(ll n) {
2     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 }
```

```

16     }
17 }
18 if(n > 1)
19     sum *= (1 + n);
20 return sum;
21 }
```

Listing 46: Calculates the sum of all divisors of number n . e.g.
 $\text{sum_of_divisors}(12) = 18$. 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 47: 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
13    return sol % MOD;
14}
15
16 void combi() {
17     fact[0] = inv[0] = 1;
18     for(i,1,MAXN+1) {
19         fact[i] = fact[i-1] * i % MOD;
20         inv[i] = binpow( fact[i], MOD - 2 );
21     }
22
23 nCr( ll n, ll r ) {
24     return fact[n] * inv[r] % MOD * inv[n-r] % MOD;
25 }
26 combi();
27 nCr(a,b);

```

Listing 48: 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 generalized to $O(n \cdot \log(n))$, $n \leq 10^6$.

9.2.2 Common combinatorics formulas

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

$$\sum_{k=0}^n \binom{n}{k} = 2^n \quad (4)$$

$$\sum_{k=0}^n \binom{n}{k} \binom{n}{n-k} = \binom{2n}{n} \quad (5)$$

$$\sum_{k=0}^n k \binom{n}{k} = n2^{n-1} \quad (6)$$

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

(8)

9.2.3 Stars and Bars

The number of ways to accomodate a binary string made of n and k elements.
The number of ways I have to accomodate n equal balls into k bags.

$$\binom{n+k-1}{k-1} = \binom{n+k-1}{n}$$

9.3 Probability

9.4 Computational Geometry

9.4.1 *Cross Product

9.4.2 *Convex Hull

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

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

10.3 Printable ASCII characters

32	whitespace	58	:	65	A	97	a
33	!	59	;	66	B	98	b
34	"	60	i	67	C	99	c
35	#	61	=	68	D	100	d
36	\$	62	l	69	E	101	e
37	%	63	?	70	F	102	f
38	&	64	@	71	G	103	g
39	'	91	[72	H	104	h
40	(92	\	73	I	105	i
41)	93]	74	J	106	j
42	*	94	\wedge	75	K	107	k
43	+	95	-	76	L	108	l
44	,	96	'	77	M	109	m
45	-	126	\sim	78	N	110	n
46	.			79	O	111	o
47	/			80	P	112	p
48	0			81	Q	113	q
49	1			82	R	114	r
50	2			83	S	115	s
51	3			84	T	116	t
52	4			85	U	117	u
53	5			86	V	118	v
54	6			87	W	119	w
55	7			88	X	120	x
56	8			89	Y	121	y
57	9			90	Z	122	z

Table 2: Code and symbol of printable ASCII characters.

10.4 Numbers bit representation

1	00000001	31	00011111	61	00111101	91	01011011	121	01111001
2	00000010	32	00100000	62	00111110	92	01011100	122	01111010
3	00000011	33	00100001	63	00111111	93	01011101	123	01111011
4	00000100	34	00100010	64	01000000	94	01011110	124	01111100
5	00000101	35	00100011	65	01000001	95	01011111	125	01111101
6	00000110	36	00100100	66	01000010	96	01100000	126	01111110
7	00000111	37	00100101	67	01000011	97	01100001	127	01111111
8	00001000	38	00100110	68	01000100	98	01100010	128	10000000
9	00001001	39	00100111	69	01000101	99	01100011	129	10000001
10	00001010	40	00101000	70	01000110	100	01100100	130	10000010
11	00001011	41	00101001	71	01000111	101	01100101	131	10000011
12	00001100	42	00101010	72	01001000	102	01100110	132	10000100
13	00001101	43	00101011	73	01001001	103	01100111	133	10000101
14	00001110	44	00101100	74	01001010	104	01101000	134	10000110
15	00001111	45	00101101	75	01001011	105	01101001	135	10000111
16	00010000	46	00101110	76	01001100	106	01101010	136	10001000
17	00010001	47	00101111	77	01001101	107	01101011	137	10001001
18	00010010	48	00110000	78	01001110	108	01101100	138	10001010
19	00010011	49	00110001	79	01001111	109	01101101	139	10001011
20	00010100	50	00110010	80	01010000	110	01101110	140	10001100
21	00010101	51	00110011	81	01010001	111	01101111	141	10001101
22	00010110	52	00110100	82	01010010	112	01110000	142	10001110
23	00010111	53	00110101	83	01010011	113	01110001	143	10001111
24	00011000	54	00110110	84	01010100	114	01110010	144	10010000
25	00011001	55	00110111	85	01010101	115	01110011	145	10010001
26	00011010	56	00111000	86	01010110	116	01110100	146	10010010
27	00011011	57	00111001	87	01010111	117	01110101	147	10010011
28	00011100	58	00111010	88	01011000	118	01110110	148	10010100
29	00011101	59	00111011	89	01011001	119	01110111	149	10010101
30	00011110	60	00111100	90	01011010	120	01111000	150	10010110

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

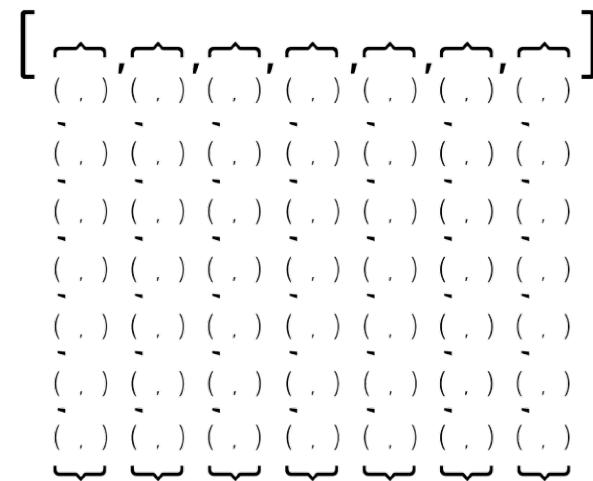


Figure 1: Visual representation of a vector of vector of pairs.

10.6 How all neighbours of a grid looks like

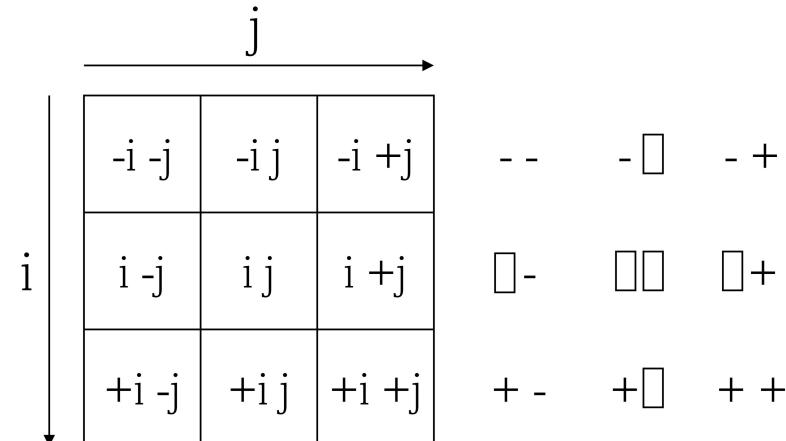


Figure 2: Visual representation of how all adjacent cells in a grid looks like.