

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



# ICPC 2026 Reference

*CPCFI UNAM*

>:)

RacsoFractal, zum, edwardsal17

## Contents

<b>1</b>	<b>Template</b>	<b>3</b>			
<b>2</b>	<b>C++ Sintaxis</b>	<b>3</b>			
2.1	Compilation sentences	3			
2.2	Custom comparators	3			
2.3	*STL DS Usage	4			
2.4	*Strings methods	4			
2.5	*Pragmas	4			
2.6	Bit Manipulation Cheat Sheet	4			
2.7	Comparing Floats	5			
2.8	Ceil	5			
<b>3</b>	<b>Miscellaneous</b>	<b>5</b>			
3.1	Binary Search	5			
3.1.1	*Parallel Binary Search	6			
3.1.2	*Ternary Search	6			
3.2	Kadane's Algorithm	6			
<b>4</b>	<b>Queries</b>	<b>7</b>			
4.1	Prefix Sum 2D	7			
4.2	*Sparse Table	7			
4.3	*Sqrt Decomposition	7			
4.4	*Fenwick Tree	7			
4.5	*Fenwick Tree 2D	7			
4.6	Segment Tree	7			
4.6.1	*2D Segment Tree	9			
4.6.2	*Persistent Segment Tree	9			
4.6.3	*Lazy Propagation	9			
4.7	Ordered Set	9			
4.7.1	Multi-Ordered Set	9			
4.8	*Treap	10			
4.9	*Trie	10			
<b>5</b>	<b>Graph Theory</b>	<b>10</b>			
5.1	Breadth-First Search (BFS)	10			
5.2	Deep-First Search (DFS)	12			
5.3	Shortest Path	13			
5.3.1	Dijkstra's Algorithm	13			
5.3.2	*Floyd-Warshall's Algorithm	14			
5.3.3	Bellman-Ford Algorithm	14			
5.4	Minimum Spanning Tree (MST)	15			
5.4.1	*Prim's Algorithm	15			
5.4.2	*Kruskal's Algorithm	15			
5.5	*Bipartite Checking	15			
5.6	*Negative Cycles	15			
5.7	Topological Sort	15			
5.8	*Disjoint Set Union (DSU)	16			
5.9	*Condensation Graph	16			
5.10	*Strongly Connected Components (SCC)	16			
5.11	*2-SAT	16			
5.12	*Bridges and point articulation	16			
5.13	Flood Fill	16			
5.14	Lava Flow (Multi-source BFS)	17			
5.15	MaxFlow	19			
5.15.1	Dinic's Algorihtm	19			
5.15.2	*Ford-Fulkerson Algorithm	26			
5.15.3	*Goldber-Tarjan Algorithm	26			
<b>6</b>	<b>Trees</b>	<b>26</b>			
6.1	Counting Childrens	26			
6.2	*Tree Diameter	27			
6.3	*Centroid Decomposition	27			
6.4	*Euler Tour	27			
6.5	*Lowest Common Ancestor (LCA)	27			
6.6	*Heavy-Light Decomposition (HLD)	27			
<b>7</b>	<b>Strings</b>	<b>27</b>			
7.1	Knuth-Morris-Pratt Algorithm (KMP)	27			
7.2	*Suffix Array	28			
7.3	*Rolling Hashing	28			
7.4	*Z Function	28			
7.5	*Aho-Corasick Algorithm	28			
<b>8</b>	<b>Dynamic Programming</b>	<b>28</b>			
8.1	*Coins	28			
8.2	*Longest Increasing Subsequence (LIS)	28			
8.3	*Edit Distance	28			
8.4	*Knapsack	28			
8.5	*SOS DP	28			

8.6	*Digit DP . . . . .	28
8.7	*Bitmask DP . . . . .	28
<b>9</b>	<b>Mathematics</b>	<b>28</b>
9.1	Number Theory . . . . .	28
9.1.1	Greatest Common Divisor (GCD) . . . . .	28
9.1.2	Gauss Sum . . . . .	28
9.1.3	*Modular Theory . . . . .	28
9.1.4	*Modulo Inverse . . . . .	28
9.1.5	*Fermat's Little Theorem . . . . .	28
9.1.6	*Chinese Remainder Theorem . . . . .	28
9.1.7	Binpow . . . . .	28
9.1.8	Matrix Exponentiation (Linear Recurrency) . . . . .	29
9.1.9	Prime checking . . . . .	30
9.1.10	Prime factorization . . . . .	30
9.1.11	Sieve of Eratosthenes . . . . .	30
9.1.12	Sum of Divisors . . . . .	31
9.2	Combinatorics . . . . .	31
9.2.1	Binomial Coefficients . . . . .	31
9.2.2	Common combinatorics formulas . . . . .	32
9.3	*Stars and Bars . . . . .	32
9.4	Probability . . . . .	32
9.5	Computational Geometry . . . . .	32
9.5.1	*Cross Product . . . . .	32
9.5.2	*Convex Hull . . . . .	32
9.6	*Fast Fourier Transform (FFT) . . . . .	32
<b>10</b>	<b>Appendix</b>	<b>32</b>
10.1	What to do against WA? . . . . .	32
10.2	Primitive sizes . . . . .	33
10.3	*ASCII table . . . . .	33
10.4	*Numbers bit representation . . . . .	33
10.5	How a <code>vector&lt;vector&lt;pair&lt;int,int&gt;&gt;&gt;</code> looks like . . . . .	33
10.6	How all neighbours of a grid looks like . . . . .	34

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

```

To compile and execute in Java:

```

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

```

To execute in Python (two options):

```

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

```

### 2.2 Custom comparators

```

1 // Using function
2 bool cmpFunction(const pair<int,int> &a, const pair<
    int,int> &b) {
3     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

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

- >> (Right shift): Shifts bits right.
- Get LSB:  $n \& -n$

**Basic Bit Tasks:**

- Get bit:  $(n \& (1 \ll i)) \neq 0$
- Set bit:  $n \mid (1 \ll i)$
- Clear bit:  $n \& \sim(1 \ll i)$
- Toggle bit:  $n \hat{=} (1 \ll i)$
- Clear LSB:  $n \& (n - 1)$

**Set Operations:**

- Subset check:  $(A \& B) == B$
- Set union:  $A \mid B$
- Set intersection:  $A \& B$
- Set difference:  $A \& \sim B$
- Toggle subset:  $A \hat{=} B$

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

## 2.7 Comparing Floats

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

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

## 2.8 Ceil

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

(Originally I thought  $\left\lceil \frac{a}{b} \right\rceil = \frac{a-1}{b} + 1$ , 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 4: A way to do ceil operation between integers without making explicit conversions. `a` and `b` are `int`, but the operation `a+b-1` can cause an overflow, so they must be casted into `long long` to avoid this. The result must be `int` anyway.

## 3 Miscellaneous

### 3.1 Binary Search

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

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

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

```

3     function<bool(int)> check = [&](int idx)->bool {
4         return idx < n && list[idx] <= target;
5     };
6     for( int i = 31; i >= 0; i-- ) {
7         if( check( ans + (1 << i) ) )
8             ans += 1 << i;
9     }
10    return list[ans] == target ? ans : -1;
11 }

```

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

```

1 int binary_search( vector<int>& list, int n, int
   target ) {
2     int left = 1, right = n + 1, mid;
3     while(right - left >= 1) {
4         mid = left + (right - left) / 2;
5         if( list[mid] >= target && target > list[mid -
6             1]) {
7             //----- TODO logic here -----//
8             break;
9         }
10        else
11            if( list[mid] < target )
12                left = mid + 1;
13        else
14            right = mid;

```

```

14     }
15 }

```

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

```

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

```

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

### 3.1.1 \*Parallel Binary Search

### 3.1.2 \*Ternary Search

## 3.2 Kadane's Algorithm

```

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

```

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

```

```

14     cout << arns << endl;
15 }

```

Listing 10: Construction and query 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;

```



```

16         int right = lft + 1;
17         update(lft, index, tl, mid, val);
18         update(right, index, mid+1, tr, val);
19         t[node] = min(t[lft], t[right]);
20     }
21 }
22 ll query(int node, int l, int r, int tl, int tr)
23 {
24     if( tl > r || tr < l ) return INF;
25     if( tl >= l and tr <= r ) return t[node];
26     else {
27         int mid = tl + ((tr-tl)>>1);
28         int lft = node << 1;
29         int right = lft + 1;
30         ll q1 = query(lft, l, r, tl, mid);
31         ll q2 = query(right, l, r, mid+1, tr);
32         return min(q1, q2);
33     }
34 void build(vi &v, int node, int tl, int tr) {
35     if( tl == tr ) t[node] = v[tl];
36     else {
37         int mid = tl + ((tr-tl)>>1);
38         int lft = node << 1;
39         int right = lft + 1;
40         build(v, lft, tl, mid);
41         build(v, right, mid+1, tr);
42         t[node] = min(t[lft], t[right]);

```

```

43     }
44 }
45 };
46 Segment_tree st(n);
47 st.build(v, 1, 0, n-1);
48 st.update(1, a-1, 0, n-1, b);
49 st.query(1, a-1, b-1, 0, n-1));

```

Listing 11: 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     }
18     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 12: Segment Tree for Dynamic Range **Sum** Queries. Zum's Implementation.

#### 4.6.1 \*2D Segment Tree

#### 4.6.2 \*Persistent Segment Tree

#### 4.6.3 \*Lazy Propagation

### 4.7 Ordered Set

```

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

```

Listing 13: 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_aset;
11 map<int,int> cuenta;
12 function<void(int)> insertar = [&](int val) -> void
    {
13     multi_aset.insert({val,++cuenta[val]});
14 };
15 function<void(int)> eliminar = [&](int val) -> void
    {
16     multi_aset.erase({val,cuenta[val]--});
17 };

```

Listing 14: Declaration of multi-aset 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 15: Iterative implementation of BFS graph traversal over a graph represented as a AdjacencyList on vector of vectors. BFS runs in  $O(|V| + |E|)$ .

```

1  int n, m;
2  string arns = "";
3  cin >> n >> m;
4  vector<vector<bool>> visited(n,vector<bool>(m,false)
    );
5  vector<string> path(n,string(m,'0'));
6  vector<string> grid(n);
7  vii dirs = {{0,1},{0,-1},{1,0},{-1,0}};
8  string commands = "LRUD";

```

```

9  queue<ii> q;
10 ii start, end, curr;
11 function<bool(int,int)> valid = [&](int i, int j) ->
    bool {
12     return ( i >= 0 && i < n && j >= 0 && j < m &&
        grid[i][j] != '#' && ! visited[i][j] );
13 };
14 fori(i,0,n) cin >> grid[i];
15 fori(i,0,n) {
16     fori(j,0,m)
17         if( grid[i][j] == 'A' ) {
18             visited[i][j] = true;
19             q.push( {i,j} );
20         }
21 }
22 while( ! q.empty() ) {
23     curr = q.front();
24     q.pop();
25     int i = curr.fi;
26     int j = curr.se;
27     if( grid[i][j] == 'B' ) {
28         end.fi = i;
29         end.se = j;
30         break;
31     }
32     int newI, newJ;
33     fori(I,0,4) {
34         newI = i + dirs[I].fi;

```

```

35         newJ = j + dirs[I].se;
36         if( valid(newI,newJ) ) {
37             visited[newI][newJ] = true;
38             q.push( {newI,newJ} );
39             path[newI][newJ] = commands[I];
40         }
41     }
42 }
43 while( path[ end.fi ][ end.se ] != '0' ) {
44     fori(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 16: 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    }
17 }

```

Listing 17: 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    }
17 }

```

Listing 18: 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 19: 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 20: 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 21: Iterative implementation of Dijkstra's Algorithm for shortest path over a graph represented as a AdjacencyList on vector of vectors. Returns a vector with the shortest path to every other vertex in the graph.  $O(|E| \times \log_2(|v|))$ . doesn't work with negative weights.

```

1 vvp11 graph(n+1,vp11());
2 vector<bool> visited(n+1,false);
3 function<vll(int)> dijkstra = [&](int source) -> vll

```

```

    {
4   vll dist(n+1, INF);
5   priority_queue<pll, vpll, greater<pll>> pq;
6   dist[source] = 0;
7   pq.push({0, source});
8   while( ! pq.empty() ) {
9       ll d = pq.top().fi;
10      ll u = pq.top().se;
11      pq.pop();
12      if( d > dist[u] ) continue;
13      for(pll edge : graph[u]) {
14          ll v = edge.fi;
15          ll w = edge.se;
16          if( dist[u] + w < dist[v] ) {
17              dist[v] = dist[u] + w;
18              pq.push({dist[v], v});
19          }
20      }
21  }
22  return dist;
23 };

```

Listing 22: Iterative implementation of Dijkstra's Algorithm as a Lambda Function for shortest path over a graph represented as a AdjacencyList on vector of vectors. Returns a vector with the shortest path to every other vertex in the graph.  $O(|E| \times \log_2(|V|))$ . Doesn't work with negative weights.

### 5.3.2 \*Floyd-Warshall's Algorithm

### 5.3.3 Bellman-Ford Algorithm

```

1  int V, E;
2  cin >> V >> E;
3  vvi edges(E, vi(3, 0));
4
5  for(int i = 1; i <= E; i++)
6      cin >> edges[i][0] >> edges[i][1] >> edges[i][2];
7
8  function<vi(int)> bellman_ford = [&](int src) -> vi
9  {
10     vi dist(V, INF);
11     dist[src] = 0;
12     for(int i = 0; i < V; i++)
13     {
14         for( vi edge : edges )
15         {
16             int u = edge[0];
17             int v = edge[1];
18             int w = edge[2];
19             if( dist[u] != INF and dist[u] + w < dist[v]
20         )
21         {
22             if( i == V-1 )
23                 return = {-1};
24             dist[v] = dist[u] + w;
25         }
26     }
27 }

```

```

25     }
26 }
27 return dist;
28 };

```

Listing 23: 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 24: Recursive toposort implementation for unweighted DAG through vvi with DFS with inverted postorder. Runs in  $O(|V| \times |E|)$ .

```

1 vi indegree(V+1,0);
2 vvi graph(V+1,vi());
3 vector<bool> visited(V+1,false);
4 fori(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 25: Kahn's Algorithm for Topological Sorting using BFS and indegree vertex analysis (nodes in a cycle will never have indegree zero). Works over unweighted directed graphs containing cycles through vvi. Runs in  $O(|V| \times |E|)$ .

## 5.8 \*Disjoint Set Union (DSU)

## 5.9 \*Condensation Graph

## 5.10 \*Strongly Connected Components (SCC)

## 5.11 \*2-SAT

## 5.12 \*Bridges and point articulation

## 5.13 Flood Fill

```

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)
    -> void {
6     if( grid[i][j] == '#' ) return;
7     int newI, newJ;

```

```

8     if( grid[i][j] != '.' ) arns += grid[i][j] - '0';
9     grid[i][j] = '#';
10    for( ii move : dirs ) {
11        newI = i + move.fi; newJ = j + move.se;
12        if( newI >= 0 && newI < n && newJ >= 0 && newJ <
            m && grid[newI][newJ] == 'T' )
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 <
            m )
18            traverse(newI, newJ);
19    }
20 };
21 fori(i,0,n)
22     cin >> grid[i];
23 fori(i,0,n) {
24     fori(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 26: 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) ->
    bool {
26     return (I >= 0) and (I < n) and (J >= 0) and (J <
        m) and (grid[I][J] != '#') and (times[I][J] ==
        INF);
27 };
28 function<bool(int,int)> valid_player = [&](int I,
    int J) -> bool {
29     return (I >= 0) and (I < n) and (J >= 0) and (J <
        m) and (!visited[I][J]) and (grid[I][J] != '#');
30 };
31 function<bool(int,int)> is_border = [&](int I, int J
    ) -> bool {
32     return I == 0 || I == n-1 || J == 0 || J == m-1;
33 };
34 // Corner cases
35 if( is_border(start.fi,start.se) ) {
36     cout << "YES" << endl << "0" << endl;

```

```

37     return 0;
38 }
39 // Multi-Source BFS
40 while( ! q.empty() ) {
41     iii u = q.front();
42     q.pop();
43     for(ii dir : dirs) {
44         int newI = u[0] + dir.fi;
45         int newJ = u[1] + dir.se;
46         int w = u[2] + 1;
47         if( valid(newI,newJ) ) {
48             times[newI][newJ] = w;
49             q.push({newI,newJ,w});
50         }
51     }
52 }
53 // Player BFS
54 q.push({start.fi,start.se,0});
55 visited[start.fi][start.se] = true;
56 while( ! q.empty() and !escaped ) {
57     iii u = q.front();
58     q.pop();
59     for(int i = 0; i < 4; i++) {
60         int newI = u[0] + dirs[i].fi;
61         int newJ = u[1] + dirs[i].se;
62         int w = u[2] + 1;
63         if( valid_player(newI,newJ) and w < times[newI][
newJ] ) {

```

```

64         visited[newI][newJ] = true;
65         prev[newI][newJ] = i;
66         q.push({newI,newJ,w});
67         if( is_border(newI,newJ) ) {
68             end.fi = newI;
69             end.se = newJ;
70             escaped = true;
71             break;
72         }
73     }
74 }
75 }
76 if( !escaped ) {
77     cout << "NO" << endl;
78     return 0;
79 }
80 // Path reconstruction
81 cout << "YES" << endl;
82 int i = end.fi;
83 int j = end.se;
84 while( prev[i][j] != -1 ) {
85     int oldI = i;
86     arns += path[ prev[i][j] ];
87     i -= dirs[prev[i][j]].fi;
88     j -= dirs[prev[oldI][j]].se;
89 }
90 reverse(all(arns));
91 cout << sz(arns) << endl << arns << endl;

```

Listing 27: 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
      network.
4   * @details Stores the edge's source, destination,
      capacity, and current flow.
5   *          Used in max-flow algorithms like Dinic
      or Ford-Fulkerson. */
6  struct flowEdge {
7      int u; // Source node
8      int v; // Destination node
9      ll cap; // Maximum flow capacity of the edge
10     ll flow = 0; // Current flow through the edge (
        initially 0)
11     flowEdge( int u, int v, ll cap ) : u(u), v(v), cap
        (cap) {};
12 };
13 /**
```

```

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

```

34     next.resize(n); // Prepare next-edge array for
      DFS.
35     fill(all(level),-1); // Mark all levels as
      unvisited (-1).
36     level[s] = 0; // The source has level 0.
37     q.push(s); // Start BFS from the source.
38 }
39 /**
40  * @brief Adds a directed edge and its residual
      reverse edge to the flow network. */
41 void addEdge( int u, int v, ll cap ) {
42     edges.emplace_back(u,v,cap); // Original edge: u
      -> v
43     edges.emplace_back(v,u,0); // Residual edge: v
      -> u
44     adj[u].pb(id++);
45     adj[v].pb(id++);
46 }
47 /**
48  * @brief Performs BFS to construct the level
      graph (Layered Network) from source 's' to sink '
      t'.
49  * @details Assigns levels (minimum distances
      from 's') to all nodes and checks if 't' is
      reachable.
50  *           Levels are used to guide the DFS
      phase in Dinic's algorithm.

```

```

51     * @return bool True if the sink 't' is reachable
      (i.e., there exists an augmenting path), false
      otherwise. */
52 bool bfs() {
53     while( ! q.empty() ) {
54         int curr = q.front();
55         q.pop();
56         for( auto e : adj[curr] ) {
57             if( edges[e].cap - edges[e].flow < 1 ) //
      Skip saturated edges (no residual capacity).
58                 continue;
59             if( level[ edges[e].v ] != -1 ) // Skip
      already visited nodes (level assigned).
60                 continue;
61             // Assign level to the neighbor node.
62             level[ edges[e].v ] = level[ edges[e].u ] +
      1; // Next level = current + 1.
63             q.push( edges[e].v ); // Add neighbor to the
      queue for further BFS.
64         }
65     }
66     return level[t] != -1; // Return whether the
      sink 't' was reached (level[t] != -1).
67 }
68 /**
69  * @brief Finds a blocking flow using DFS in the
      level graph constructed by BFS.
70  * @param u Current node being processed.

```

```

71  * @param flow Maximum flow that can be sent from
    'u' to the sink 't'.
72  * @return ll The amount of flow successfully
    sent to 't'. */
73  ll dfs( int u, ll flow ) {
74      if( flow == 0 ) // No remaining flow to send.
75          return 0;
76      if( u == t )    // Reached the sink; return
    accumulated flow.
77          return flow;
78      // Explore edges from 'u' using 'next[u]' to
    avoid revisiting processed edges.
79      for( int& cid = next[u]; cid < sz(adj[u]); cid++
    ) {
80          int e = adj[u][cid]; // Index of the edge in '
    edges'.
81          int v = edges[e].v;  // Destination node of
    the edge.
82          // Skip invalid edges:
83          // 1. Not in the level graph (level[u] + 1 !=
    level[v]). Just edges in exactly one level ahead (
    ensures shortest paths).
84          // 2. No residual capacity (cap - flow < 1).
85          if( level[edges[e].u] + 1 != level[v] || edges
    [e].cap - edges[e].flow < 1 )
86              continue;
87          ll f = dfs( v, min(flow, edges[e].cap - edges[
    e].flow ) ); // Recursively send flow to 'v'.

88      if( f == 0 ) // No flow could be sent via this
    edge.
89          continue;
90      // Update residual capacities:
91      edges[e].flow += f;          // Increase flow in
    the original edge.
92      edges[ e ^ 1 ].flow -= f; // Decrease flow in
    the reverse edge. (All reverse edges have
    distinct parity)
93      return f;                  // Return the flow
    sent.
94  }
95  return 0; // No augmenting path found from 'u'.
96  }
97  /**
98   * @brief Computes the maximum flow from source '
    s' to sink 't' using Dinic's algorithm.
99   * @details Iterates through BFS and DFS phases
    to find the maximum flow.
100      Accumulates flow while there exists
    augmentation paths in the residual graph.
101      Restart auxiliary structures for every new
    phase.
102   * @return ll The maximum flow value. */
103  ll maxFlow() {
104      ll flow = 0; // Tracks the total flow sent.
105      while( bfs() ) { // While there are augmenting
    paths:

```

```

106     fill(all(next),0); // Reset 'next' for DFS.
107     for( ll f = dfs(s,INF); f != 0ll; f = dfs(s,
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  * @brief Finds edges belonging to the minimum
118  * cut after maxFlow().
119  * @details First, it marks all the reachable
120  * nodes from 's' with an augmentation path after
121  * obtained the max flow
122  * and all the saturated edges coming out from
123  * any of the nodes who belong to the min-cut.
124  * For 'minCut()' to work, 'maxFlow()' must be
125  * first executed to get the min-cut.
126  * If only is needed the value, is enough
127  * returning the value of 'maxFlow()'.
128  * @return vii List of edges (u, v) in the min-
129  * cut. Its size is the minimum number of 'roads' to
130  * close. */
131 vii minCut() {
132     vii ans;

```

```

125     fill(all(level),-1); // Reset levels.
126     level[s] = 0;          // Mark source as reachable
127     .
128     q.push(s);
129     while( ! q.empty() ) { // BFS to mark nodes
130         reachable from 's' in the residual graph.
131         int curr = q.front();
132         q.pop();
133         for( int id = 0; id < sz(adj[curr]); id++ ) {
134             // For every edge going out from 'curr'.
135             int e = adj[curr][id];
136             // If 'v' is has not been visited yet, and
137             the edge have residual capacity.
138             if( level[edges[e].v] == -1 && edges[e].cap
139                 - edges[e].flow > 0 ) {
140                 q.push(edges[e].v);
141                 level[edges[e].v] = level[edges[e].u] + 1;
142             }
143         }
144     }
145     for( int i = 0; i < sz(level); i++ ) {
146         if( level[i] != -1 ) {
147             for( int id = 0; id < sz(adj[i]); id++ ) {
148                 int e = adj[i][id];
149                 if( level[edges[e].v] == -1 && edges[e].
150                     cap - edges[e].flow == 0 )
151                     ans.emplace_back(edges[e].u, edges[e].v);
152             }
153         }
154     }

```

```

147     }
148 }
149 return ans;
150 }
151 /**
152  * @brief Reconstructs the maximum bipartite
153  * matching after running 'maxFlow()'.
154  * @details Every edge that belong to the
155  * original graph and have flow greater than zero,
156  * belongs to the matching.
157  * For 'maximumMatching()' to work, 'maxFlow()'
158  * must be called first.
159  * @return vii List of matched pairs (boy, girl).
160  */
161 vii maximumMatching() {
162     vii ans;
163     fill(all(level), -1); // Reset levels.
164     level[s] = 0;         // Mark source as reachable
165     .
166     q.push(s);
167     while( ! q.empty() ) { // BFS to mark nodes
168         reachable via saturated edges with flow greater
169         than zero.
170         int curr = q.front();
171         q.pop();
172         for( int id = 0; id < sz(adj[curr]); id++ ) {
173             int e = adj[curr][id];

```

```

174             // If 'v' has not been visited yet, the edge
175             is saturated and have flow greater than zero.
176             if( level[edges[e].v] == -1 && edges[e].cap
177             - edges[e].flow == 0 && edges[e].flow != 0 ) {
178                 q.push(edges[e].v);
179                 level[edges[e].v] = level[edges[e].u] + 1;
180             }
181         }
182     }
183     for( int i = 0; i < sz(level); i++ ) { //
184     Collect original edges (boy -> girl) that are
185     saturated and have flow > 0.
186     if( level[i] != -1 ) {
187         for( int id = 0; id < sz(adj[i]); id++ ) {
188             int e = adj[i][id];
189             if( edges[e].u != s && edges[e].v != t
190             && edges[e].cap - edges[e].flow == 0 && edges[e].
191             flow != 0 )
192                 ans.emplace_back(edges[e].u, edges[e].v
193             );
194         }
195     }
196     }
197     return ans;
198 }

```



Listing 28: 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     fori(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 29: 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-
        indexed indexes, 1 is the source (bank), 'n' is
        the sink (harbor)
11
12     for(i,0,m)
13     {
14         cin >> u >> v;
15         flow.addEdge(u,v,1);
16         flow.addEdge(v,u,1);
17     }
18
19     flow.maxFlow();
20     minCut = flow.minCut();
21
22     cout << (sz(minCut)/2) << endl;

```

```

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

```

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

Listing 33: 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 34: 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 35: Implementation of the Gauss Sum.

### 9.1.3 \*Modular Theory

### 9.1.4 \*Modulo Inverse

### 9.1.5 \*Fermat's Little Theorem

### 9.1.6 \*Chinese Remainder Theorem

### 9.1.7 Binpow

```

1 const int MOD = 1e9+7;
2 int binpow( long long a, long long b ) { // a^b
3     long long sol = 1;
4     a %= MOD;
5     while( b > 0 ) {
6         if( b & 1 )
```

```

7         sol = ( 1LL * sol * a ) % MOD;
8         a = ( 1LL * a * a ) % MOD;
9         b >= 1;
10    }
11    return sol % MOD;
12 }

```

Listing 36: Applying binary exponentiation to a problem requiring  $a^b \bmod (10^9 + 7)$  in  $O(\log_2(b))$ .

### 9.1.8 Matrix Exponentiation (Linear Recurrency)

```

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

```

```

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

```

```

41 Matrix<ll> ini(2,1);
42 ini.mat[0][0] = 0;
43 ini.mat[1][0] = 1;
44 Matrix<ll> iden(2,2);
45 iden.makeIden();
46 ll n;
47 cin >> n;
48 while(n > 0) {
49     if( n & 1 ) iden *= A;
50     A *= A;
51     n >>= 1;
52 }
53 Matrix<ll> res = iden * ini;
54 cout << res.mat[0][0] << endl;
55 return 0;
56 }

```

Listing 37: 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 38: 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 39: 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 40: 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 }
```

Listing 41: Calculates the sum of all divisors of number  $n$ . e.g.  $sum\_of\_divisors(12) = 18$ . Runs in  $O(\sqrt{n})$ .

```

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

Listing 42: Calculates the sum of all divisors of all numbers from 1 to  $n$ . Runs in  $O(n \log(n))$ .

## 9.2 Combinatorics

### 9.2.1 Binomial Coefficients

```

1 const int MAXN = 1e6+1;
2 vll fact(MAXN+1), inv(MAXN+1);
3 int binpow( ll a, ll b ) { // a^b
4     ll sol = 1;
5     a %= MOD;
6     while( b > 0 ) {

```



```

7         if( b & 1 )
8             sol = ( 1LL * sol * a ) % MOD;
9         a = ( 1LL * a * a ) % MOD;
10        b >>= 1;
11    }
12    return sol % MOD;
13 }
14 void combi() {
15     fact[0] = inv[0] = 1;
16     fori(i,1,MAXN+1) {
17         fact[i] = fact[i-1] * i % MOD;
18         inv[i] = binpow( fact[i], MOD - 2 );
19     }
20 }
21 ll nCr( ll n, ll r ) {
22     return fact[n] * inv[r] % MOD * inv[n-r] % MOD;
23 }
24 combi();
25 nCr(a,b);

```

Listing 43: 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

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

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

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

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

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

$$(8)$$

## 9.3 \*Stars and Bars

## 9.4 Probability

## 9.5 Computational Geometry

### 9.5.1 \*Cross Product

### 9.5.2 \*Convex Hull

## 9.6 \*Fast Fourier Transform (FFT)

# 10 Appendix

## 10.1 What to do against WA?

1. Have you done the correct complexity analysis?
2. Have you understood well the statement?
3. Have you corroborated yet the trivial test cases?
4. Have you checked all the corner cases?
5. Have you proposed a lot of non-trivial test cases?

6. Isn't there any possibility of overflow? (Multiplying two `int` needs to be fitted into a `long long`)
7. Have you done a desktop test?
8. Have you read 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	$1.1 \times 10^{-38}$	$3.4 \times 10^{38}$
<code>double</code>	8	$2.2 \times 10^{-308}$	$1.7 \times 10^{308}$
<code>long double</code>	12	$3.3 \times 10^{-4932}$	$1.1 \times 10^{4932}$

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

## 10.3 \*ASCII table

## 10.4 \*Numbers bit representation

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

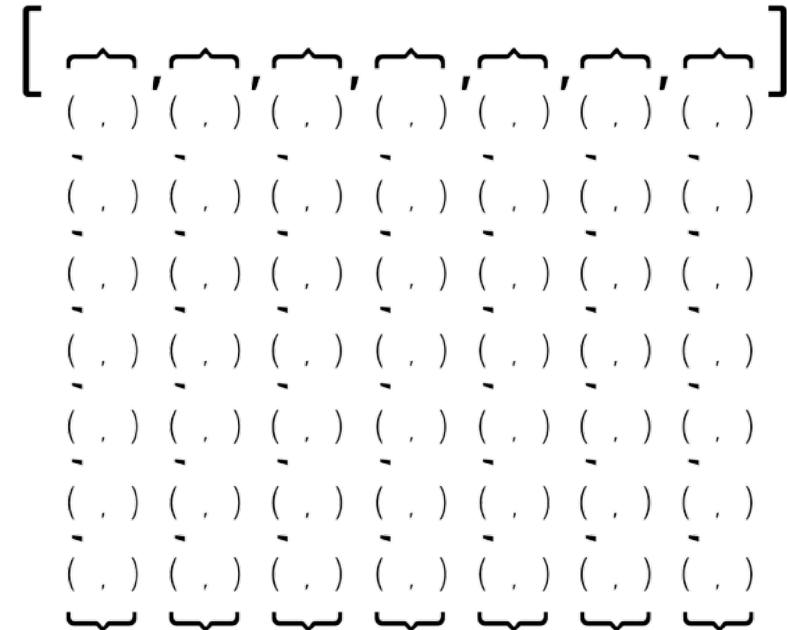


Figure 1: Visual representation of a vector of vector of pairs.

## 10.6 How all neighbours of a grid looks like

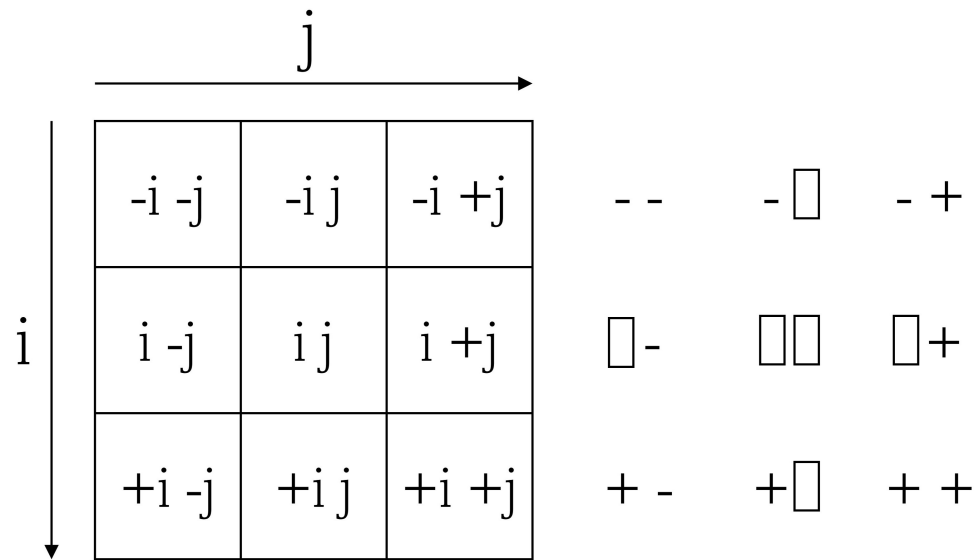


Figure 2: Visual representation of how all adjacent cells in a grid looks like.