



Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1 Math	2	7 Geometry	14
1.1 Matrix exponentiation	2	7.1 Point structure	14
1.2 Fast Fourier Transform	2	7.2 Lattice Points	14
		7.3 Convex Hull	14
		7.4 Line Intersegment	15
2 Graph	3	8 Algorithms	15
2.1 Dfs tree	3	8.1 Sparse table	15
2.2 Bellman Ford	3		
2.3 Floyd Warshall	4	9 Parser	16
2.4 2SAT	4	9.1 Parsing Functions	16
2.5 Dominator tree	5		
2.6 Kth Ancestor	5		
2.7 Topological Sort	6		
2.8 Dkistra	6		
2.9 Dinic Min cost	6		
2.10 Kosaraju	8		
2.11 Dinic	8		
3 Segtree	9		
3.1 Standard SegTree	9		
3.2 Persistent sum segment tree	10		
3.3 Set and update lazy seg	10		
3.4 Binary Indexed Tree	11		
3.5 Lazy SegTree	11		
4 Set	11		
4.1 Set	11		
4.2 Multiset	12		
4.3 Ordered Set	12		
5 Misc	12		
5.1 Template	12		
6 String	12		
6.1 String hash	12		
6.2 Suffix Array	13		
6.3 Z function	13		

1 Math

1.1 Matrix exponentiation

Complexity: $O(n \times n \times \log(b))$ to raise an $n \times n$ matrix to the power of b .

Computes powers of matrices efficiently.

```
1 // TITLE: Matrix exponentiation
2 // COMPLEXITY:  $O(n \times n \times \log(b))$  to raise an  $n \times n$  matrix
   to the power of  $b$ .
3 // DESCRIPTION: Computes powers of matrices
   efficiently.
4
5 struct Matrix {
6     vector<vi> m;
7     int r, c;
8
9     Matrix(vector<vi> mat) {
10         m = mat;
11         r = mat.size();
12         c = mat[0].size();
13     }
14
15     Matrix(int row, int col, bool ident=false) {
16         r = row; c = col;
17         m = vector<vi>(r, vi(c, 0));
18         if(ident) {
19             for(int i = 0; i < min(r, c); i++) {
20                 m[i][i] = 1;
21             }
22         }
23     }
24
25     Matrix operator*(const Matrix &o) const {
26         assert(c == o.r); // garantir que da pra
   multiplicar
27         vector<vi> res(r, vi(o.c, 0));
28
29         for(int i = 0; i < r; i++) {
30             for(int k = 0; k < c; k++) {
31                 for(int j = 0; j < o.c; j++) {
32                     res[i][j] = (res[i][j] + m[i][k]*
   o.m[k][j]) % MOD;
33                 }
34             }
35         }
36
37         return Matrix(res);
38     }
39 };
40
41 Matrix fpow(Matrix b, int e, int n) {
42     if(e == 0) return Matrix(n, n, true); //
   identidade
43     Matrix res = fexp(b, e/2, n);
44     res = (res * res);
45     if(e%2) res = (res * b);
46
47     return res;
48 }
```

1.2 Fast Fourier Transform

Complexity: $O(n \log(n))$

Multiply polynomials quickly

```
1 // TITLE: Fast Fourier Transform
2 // COMPLEXITY:  $O(n \log(n))$ 
3 // DESCRIPTION: Multiply polynomials quickly
4
```

```
5 typedef double ld;
6 typedef long long ll;
7
8 struct num{
9     ld x, y;
10     num() { x = y = 0; }
11     num(ld x, ld y) : x(x), y(y) {}
12 };
13
14 inline num operator+(num a, num b) { return num(a.x +
   b.x, a.y + b.y); }
15 inline num operator-(num a, num b) { return num(a.x -
   b.x, a.y - b.y); }
16 inline num operator*(num a, num b) { return num(a.x *
   b.x - a.y * b.y, a.x * b.y + a.y * b.x); }
17 inline num conj(num a) { return num(a.x, -a.y); }
18
19 int base = 1;
20 vector<num> roots = {{0, 0}, {1, 0}};
21 vector<int> rev = {0, 1};
22 const ld PI = acos(-1);
23
24 void ensure_base(int nbase){
25     if(nbase <= base)
26         return;
27
28     rev.resize(1 << nbase);
29     for(int i = 0; i < (1 << nbase); i++)
30         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (
   nbase - 1));
31
32     roots.resize(1 << nbase);
33
34     while(base < nbase){
35         ld angle = 2*PI / (1 << (base + 1));
36         for(int i = 1 << (base - 1); i < (1 << base);
   i++){
37             roots[i << 1] = roots[i];
38             ld angle_i = angle * (2 * i + 1 - (1 <<
   base));
39             roots[(i << 1) + 1] = num(cos(angle_i),
   sin(angle_i));
40             base++;
41         }
42     }
43 }
44
45 void fft(vector<num> &a, int n = -1){
46     if(n == -1)
47         n = a.size();
48
49     assert((n & (n-1)) == 0);
50     int zeros = __builtin_ctz(n);
51     ensure_base(zeros);
52     int shift = base - zeros;
53     for(int i = 0; i < n; i++)
54         if(i < (rev[i] >> shift))
55             swap(a[i], a[rev[i] >> shift]);
56
57     for(int k = 1; k < n; k <= 1)
58         for(int i = 0; i < n; i += 2 * k)
59             for(int j = 0; j < k; j++){
60                 num z = a[i+j+k] * roots[j+k];
61                 a[i+j+k] = a[i+j] - z;
62                 a[i+j] = a[i+j] + z;
63             }
64 }
65
66 vector<num> fa, fb;
67 vector<ll> multiply(vector<ll> &a, vector<ll> &b){
68     int need = a.size() + b.size() - 1;
69     int nbase = 0;
70     while((1 << nbase) < need) nbase++;
71 }
```

```

71     ensure_base(nbase);
72     int sz = 1 << nbase;
73     if(sz > (int) fa.size())
74         fa.resize(sz);
75
76     for(int i = 0; i < sz; i++){
77         int x = (i < (int) a.size() ? a[i] : 0);
78         int y = (i < (int) b.size() ? b[i] : 0);
79         fa[i] = num(x, y);
80     }
81     fft(fa, sz);
82     num r(0, -0.25 / sz);
83     for(int i = 0; i <= (sz >> 1); i++){
84         int j = (sz - i) & (sz - 1);
85         num z = (fa[j] * fa[j] - conj(fa[i] * fa[i]))
86             * r;
87         if(i != j) {
88             fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[
89             j])) * r;
90             fa[i] = z;
91         }
92         fft(fa, sz);
93         vector<ll> res(need);
94         for(int i = 0; i < need; i++)
95             res[i] = round(fa[i].x);
96     }
97 }
98
99
100 vector<ll> multiply_mod(vector<ll> &a, vector<ll> &b,
    int m, int eq = 0){
101     int need = a.size() + b.size() - 1;
102     int nbase = 0;
103     while((1 << nbase) < need) nbase++;
104     ensure_base(nbase);
105     int sz = 1 << nbase;
106     if(sz > (int) fa.size())
107         fa.resize(sz);
108
109     for(int i=0;i<(int)a.size();i++){
110         int x = (a[i] % m + m) % m;
111         fa[i] = num(x & ((1 << 15) - 1), x >> 15);
112     }
113     fill(fa.begin() + a.size(), fa.begin() + sz, num
114     {0, 0});
115     fft(fa, sz);
116     if(sz > (int) fb.size())
117         fb.resize(sz);
118     if(eq)
119         copy(fa.begin(), fa.begin() + sz, fb.begin())
120     ;
121     else{
122         for(int i = 0; i < (int) b.size(); i++){
123             int x = (b[i] % m + m) % m;
124             fb[i] = num(x & ((1 << 15) - 1), x >> 15)
125         ;
126     }
127     fill(fb.begin() + b.size(), fb.begin() + sz, num
128     {0, 0});
129     fft(fb, sz);
130 }
131 ld ratio = 0.25 / sz;
132 num r2(0, -1);
133 num r3(ratio, 0);
134 num r4(0, -ratio);
135 num r5(0, 1);
136 for(int i=0;i<=(sz >> 1);i++) {
137     int j = (sz - i) & (sz - 1);
138     num a1 = (fa[i] + conj(fa[j]));
139     num a2 = (fa[i] - conj(fa[j])) * r2;
140     num b1 = (fb[i] + conj(fb[j])) * r3;

```

```

137     num b2 = (fb[i] - conj(fb[j])) * r4;
138     if(i != j){
139         num c1 = (fa[j] + conj(fa[i]));
140         num c2 = (fa[j] - conj(fa[i])) * r2;
141         num d1 = (fb[j] + conj(fb[i])) * r3;
142         num d2 = (fb[j] - conj(fb[i])) * r4;
143         fa[i] = c1 * d1 + c2 * d2 * r5;
144         fb[i] = c1 * d2 + c2 * d1;
145     }
146     fa[j] = a1 * b1 + a2 * b2 * r5;
147     fb[j] = a1 * b2 + a2 * b1;
148 }
149 fft(fa, sz);
150 fft(fb, sz);
151 vector<ll> res(need);
152 for(int i=0;i<need;i++){
153     ll aa = round(fa[i].x);
154     ll bb = round(fb[i].x);
155     ll cc = round(fa[i].y);
156     res[i] = (aa + ((bb % m) << 15) + ((cc % m)
157     << 30)) % m;
158 }
159 return res;
160 }

```

2 Graph

2.1 Dfs tree

Complexity: $O(E + V)$

```

1 // TITLE: Dfs tree
2 // COMPLEXITY:  $O(E + V)$ 
3 // DESCRIPTION: Create dfs tree from graph
4
5 int desce[mxN], sobe[mxN];
6 int backedges[mxN], vis[mxN];
7 int pai[mxN], h[mxN];
8
9 void dfs(int a, int p) {
10     if(vis[a]) return;
11     pai[a] = p;
12     h[a] = h[p]+1;
13     vis[a] = 1;
14
15     for(auto b : g[a]) {
16         if (p == b) continue;
17         if (vis[b]) continue;
18         dfs(b, a);
19         backedges[a] += backedges[b];
20     }
21     for(auto b : g[a]) {
22         if(h[b] > h[a]+1)
23             desce[a]++;
24         else if(h[b] < h[a]-1)
25             sobe[a]++;
26     }
27     backedges[a] += sobe[a] - desce[a];
28 }

```

2.2 Bellman Ford

Complexity: $O(n * m)$ | $n = |nodes|$, $m = |edges|$

Finds shortest paths from a starting node to all nodes of the graph. Detects negative cycles, if they exist.

```

1 // TITLE: Bellman Ford
2 // COMPLEXITY:  $O(n * m)$  |  $n = |nodes|$ ,  $m = |edges|$ 

```

```

3 // DESCRIPTION: Finds shortest paths from a starting
  node to all nodes of the graph. Detects negative
  cycles, if they exist.
4
5 // a and b vertices, c cost
6 // [{a, b, c}, {a, b, c}]
7 vector<tuple<int, int, int>> edges;
8 int N;
9
10 void bellman_ford(int x){
11     for (int i = 0; i < N; i++){
12         dist[i] = oo;
13     }
14     dist[x] = 0;
15
16     for (int i = 0; i < N - 1; i++){
17         for (auto [a, b, c]: edges){
18             if (dist[a] == oo) continue;
19             dist[b] = min(dist[b], dist[a] + w);
20         }
21     }
22 }
23 // return true if has cycle
24 bool check_negative_cycle(int x){
25     for (int i = 0; i < N; i++){
26         dist[i] = oo;
27     }
28     dist[x] = 0;
29
30     for (int i = 0; i < N - 1; i++){
31         for (auto [a, b, c]: edges){
32             if (dist[a] == oo) continue;
33             dist[b] = min(dist[b], dist[a] + w);
34         }
35     }
36
37     for (auto [a, b, c]: edges){
38         if (dist[a] == oo) continue;
39         if (dist[a] + w < dist[b]){
40             return true;
41         };
42     }
43     return false;
44 }
45 '''

```

2.3 Floyd Warshall

Complexity: $O(V^3)$

Finds shortest distances between all pairs of vertices

```

1 // TITLE: Floyd Warshall
2 // COMPLEXITY:  $O(V^3)$ 
3 // DESCRIPTION: Finds shortest distances between all
  pairs of vertices
4
5 for(int k=0;k<n;k++) {
6
7     for(int i=0;i<n;i++) {
8         for(int j=0;j<n;j++) {
9             graph[i][j]=min(graph[i][j],
10                graph[i][k] + graph[k][j]);
11         }
12     }
13 }

```

2.4 2SAT

Complexity: $O(n+m)$, n = number of variables, m = number of conjunctions (ands).

Finds an assignment that makes a certain boolean formula true, or determines that such an assignment does not exist.

```

1 // TITLE: 2SAT
2 // COMPLEXITY:  $O(n+m)$ ,  $n$  = number of variables,  $m$  =
  number of conjunctions (ands).
3 // DESCRIPTION: Finds an assignment that makes a
  certain boolean formula true, or determines that
  such an assignment does not exist.
4
5 struct twosat {
6     vi vis, degin;
7     stack<int> tout;
8     vector<vi> g, gi, con, sccg;
9     vi repr, conv;
10    int gsize;
11    void dfs1(int a) {
12        if (vis[a]) return;
13        vis[a]=true;
14
15        for(auto& b : g[a]) {
16            dfs1(b);
17        }
18
19        tout.push(a);
20    }
21
22    void dfs2(int a, int orig) {
23        if (vis[a]) return;
24        vis[a]=true;
25
26        repr[a]=orig;
27        sccg[orig].pb(a);
28        for(auto& b : gi[a]) {
29            if (vis[b]) {
30                if (repr[b] != orig) {
31                    con[repr[b]].pb(orig);
32                    degin[orig]++;
33                }
34                continue;
35            }
36            dfs2(b, orig);
37        }
38    }
39 }
40
41 // if s1 = 1 and s2 = 1 this adds a \ / b to the
  graph
42 void addedge(int a, int s1,
43              int b, int s2) {
44     g[2*a+(!s1)].pb(2*b+s2);
45     gi[2*b+s2].pb(2*a+(!s1));
46
47     g[2*b+(!s2)].pb(2*a+s1);
48     gi[2*a+s1].pb(2*b+(!s2));
49 }
50
51 twosat(int nvars) {
52     gsize=2*nvars;
53     g.assign(gsize, vi());
54     gi.assign(gsize, vi());
55     con.assign(gsize, vi());
56     sccg.assign(gsize, vi());
57     repr.assign(gsize, -1);
58     vis.assign(gsize, 0);
59     degin.assign(gsize, 0);
60 }
61
62 // returns empty vector if the formula is not
  satisfiable.
63 vi run() {
64     vi vals(gsize/2, -1);
65     rep(i,0,gsize) dfs1(i);

```

```

66     vis.assign(gsize,0);
67     while(!tout.empty()) {
68         int cur = tout.top(); tout.pop();
69         if (vis[cur]) continue;
70         dfs2(cur,cur);
71         conv.pb(cur);
72     }
73
74     rep(i, 0, gsize/2) {
75         if (repr[2*i] == repr[2*i+1]) {
76             return {};
77         }
78     }
79
80     queue<int> q;
81     for(auto& v : conv) {
82         if (degin[v] == 0) q.push(v);
83     }
84
85     while(!q.empty()) {
86         int cur=q.front(); q.pop();
87         for(auto guy : sccg[cur]) {
88             int s = guy%2;
89             int idx = guy/2;
90             if (vals[idx] != -1) continue;
91             if (s) {
92                 vals[idx] = false;
93             } else {
94                 vals[idx]=true;
95             }
96         }
97         for (auto& b : con[cur]) {
98             if(--degin[b] == 0) q.push(b);
99         }
100     }
101
102     return vals;
103 }
104 };

```

2.5 Dominator tree

Complexity: $O(E + V)$

```

1 // TITLE: Dominator tree
2 // COMPLEXITY:  $O(E + V)$ 
3 // DESCRIPTION: Builds dominator tree
4
5 vector<int> g[mxN];
6 vector<int> S, gt[mxN], T[mxN];
7 int dsu[mxN], label[mxN];
8 int sdom[mxN], idom[mxN], id[mxN];
9 int dfs_time = 0;
10
11 vector<int> bucket[mxN];
12 vector<int> down[mxN];
13
14 void prep(int a)
15 {
16     S.pb(a);
17     id[a] = ++dfs_time;
18     label[a] = sdom[a] = dsu[a] = a;
19
20     for (auto b: g[a]) {
21         if (!id[b]) {
22             prep(b);
23             down[a].pb(b);
24         }
25         gt[b].pb(a);
26     }
27 }

```

```

28
29 int fnd(int a, int flag = 0)
30 {
31     if (a == dsu[a]) return a;
32     int p = fnd(dsu[a], 1);
33     int b = label[ dsu[a] ];
34     if (id [ sdom[b] ] < id[ sdom[ label[a] ] ]) {
35         label[a] = b;
36     }
37     dsu[a] = p;
38     return (flag ? p: label[a]);
39 }
40
41 void build_dominator_tree(int root)
42 {
43     prep(root);
44     reverse(all(S));
45
46     int w;
47     for (int a: S) {
48         for (int b: gt[a]) {
49             w = fnd(b);
50             if (id[ sdom[w] ] < id[ sdom[a] ]) {
51                 sdom[a] = sdom[w];
52             }
53         }
54         gt[a].clear();
55         if (a != root) {
56             bucket[ sdom[a] ].pb(a);
57         }
58         for (int b: bucket[a]) {
59             w = fnd(b);
60             if (sdom[w] == sdom[b]) {
61                 idom[b] = sdom[b];
62             }
63             else {
64                 idom[b] = w;
65             }
66         }
67         bucket[a].clear();
68         for (int b: down[a]) {
69             dsu[b] = a;
70         }
71         down[a].clear();
72     }
73     reverse(all(S));
74     for (int a: S) {
75         if (a != root) {
76             if (idom[a] != sdom[a]) {
77                 idom[a] = idom[ idom[a] ];
78             }
79             T[ idom[a] ].pb(a);
80         }
81     }
82     S.clear();
83 }

```

2.6 Kth Ancestor

Complexity: $O(n * \log(n))$

Preprocess, then find in $\log n$

```

1 // TITLE: Kth Ancestor
2 // COMPLEXITY:  $O(n * \log(n))$ 
3 // DESCRIPTION: Preprocess, then find in  $\log n$ 
4
5 const int LOG_N = 30;
6 int get_kth_ancestor(vector<vector<int>> & up, int v,
7     int k)
8 {
9     for (int j = 0; j < LOG_N; j++) {
10         if (k & ((int)1 << j)) {

```

```

10         v = up[v][j];
11     }
12 }
13 return v;
14 }
15
16 void solve()
17 {
18     vector<vector<int>> up(n, vector<int>(LOG_N));
19
20     for (int i = 0; i < n; i++) {
21         up[i][0] = parents[i];
22         for (int j = 1; j < LOG_N; j++) {
23             up[i][j] = up[up[i][j-1]][j-1];
24         }
25     }
26     cout << get_kth_ancestor(up, x, k) << endl;
27 }
28 }

```

2.7 Topological Sort

Complexity: $O(N + M)$, N : Vertices, M : Arestas

Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

```

1 // TITLE: Topological Sort
2 // COMPLEXITY:  $O(N + M)$ ,  $N$ : Vertices,  $M$ : Arestas
3 // DESCRIPTION: Retorna no do grapho em ordem
  topologica, se a quantidade de nos retornada nao
  for igual a quantidade de nos e impossivel
4
5 typedef vector<vector<int>> Adj_List;
6 typedef vector<int> Indegree_List; // How many nodes
  depend on him
7 typedef vector<int> Order_List;    // The order in
  which the nodes appears
8
9 Order_List kahn(Adj_List adj, Indegree_List indegree)
10 {
11     queue<int> q;
12     // priority_queue<int> q; // If you want in
  lexicografic order
13     for (int i = 0; i < indegree.size(); i++) {
14         if (indegree[i] == 0)
15             q.push(i);
16     }
17     vector<int> order;
18
19     while (not q.empty()) {
20         auto a = q.front();
21         q.pop();
22
23         order.push_back(a);
24         for (auto b: adj[a]) {
25             indegree[b]--;
26             if (indegree[b] == 0)
27                 q.push(b);
28         }
29     }
30     return order;
31 }
32
33 int32_t main()
34 {
35
36     Order_List = kahn(adj, indegree);
37     if (Order_List.size() != N) {
38         cout << "IMPOSSIBLE" << endl;
39     }
40     return 0;

```

```

41 }

```

2.8 Dkistra

Complexity: $O(E + V \cdot \log(V))$

```

1 // TITLE: Dkistra
2 // COMPLEXITY:  $O(E + V \cdot \log(V))$ 
3 // DESCRIPTION: Finds to shortest path from start
4
5 int dist[mxN];
6 bool vis[mxN];
7 vector<pair<int, int>> g[mxN];
8
9 void dikstra(int start)
10 {
11     fill(dist, dist + mxN, oo);
12     fill(vis, vis + mxN, 0);
13     priority_queue<pair<int, int>> q;
14     dist[start] = 0;
15     q.push({0, start});
16
17     while(!q.empty()) {
18         auto [d, a] = q.top();
19         q.pop();
20         if (vis[a]) continue;
21         vis[a] = true;
22         for (auto [b, w]: g[a]) {
23             if (dist[a] + w < dist[b]) {
24                 dist[b] = dist[a] + w;
25                 q.push({-dist[b], b});
26             }
27         }
28     }
29 }

```

2.9 Dinic Min cost

Complexity: $O(V \cdot V \cdot E)$, Bipartite is $O(\sqrt{V} \cdot E)$

Gives you the max_flow with the min cost

```

1 // TITLE: Dinic Min cost
2 // COMPLEXITY:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$ 
3 // DESCRIPTION: Gives you the max_flow with the min
  cost
4
5 // Edge structure
6 struct Edge
7 {
8     int from, to;
9     int flow, capacity;
10    int cost;
11
12    Edge(int from_, int to_, int flow_, int capacity_,
  int cost_)
13        : from(from_), to(to_), flow(flow_), capacity
  (capacity_), cost(cost_)
14    {}
15 };
16
17 struct Dinic
18 {
19     vector<vector<int>> graph;
20     vector<Edge> edges;
21     vector<int> dist;
22     vector<bool> inqueue;
23     int size;
24     int cost = 0;
25

```

```

26 Dinic(int n)
27 {
28     graph.resize(n);
29     dist.resize(n);
30     inqueue.resize(n);
31     size = n;
32     edges.clear();
33 }
34
35 void add_edge(int from, int to, int capacity, int
36 cost)
37 {
38     edges.emplace_back(from, to, 0, capacity,
39 cost);
40     graph[from].push_back(edges.size() - 1);
41     edges.emplace_back(to, from, 0, 0, -cost);
42     graph[to].push_back(edges.size() - 1);
43 }
44
45 int get_max_flow(int source, int sink)
46 {
47     int max_flow = 0;
48     vector<int> next(size);
49     while(spfa(source, sink)) {
50         next.assign(size, 0);
51         for (int f = dfs(source, sink, next, oo);
52             f != 0; f = dfs(source, sink, next, oo)) {
53             max_flow += f;
54         }
55     }
56     return max_flow;
57 }
58
59 bool spfa(int source, int sink)
60 {
61     dist.assign(size, oo);
62     inqueue.assign(size, false);
63     queue<int> q;
64     q.push(source);
65     dist[source] = 0;
66     inqueue[source] = true;
67
68     while(!q.empty()) {
69         int a = q.front();
70         q.pop();
71         inqueue[a] = false;
72
73         for (int & b: graph[a]) {
74             auto edge = edges[b];
75             int cap = edge.capacity - edge.flow;
76             if (cap > 0 && dist[edge.to] > dist[
77 edge.from] + edge.cost) {
78                 dist[edge.to] = dist[edge.from] +
79 edge.cost;
80                 if (not inqueue[edge.to]) {
81                     q.push(edge.to);
82                     inqueue[edge.to] = true;
83                 }
84             }
85         }
86     }
87     return dist[sink] != oo;
88 }
89
90 int dfs(int curr, int sink, vector<int> & next,
91 int flow)
92 {
93     if (curr == sink) return flow;
94     int num_edges = graph[curr].size();
95     for (; next[curr] < num_edges; next[curr]++)
96         int b = graph[curr][next[curr]];
97         auto & edge = edges[b];
98         auto & rev_edge = edges[b^1];
99         int cap = edge.capacity - edge.flow;
100         if (cap > 0 && (dist[edge.from] + edge.
101 cost == dist[edge.to])) {
102             int bottle_neck = dfs(edge.to, sink,
103 next, min(flow, cap));
104             if (bottle_neck > 0) {
105                 edge.flow += bottle_neck;
106                 rev_edge.flow -= bottle_neck;
107                 cost += edge.cost * bottle_neck;
108                 return bottle_neck;
109             }
110         }
111     }
112     return 0;
113 }
114
115 vector<pair<int, int>> mincut(int source, int
116 sink)
117 {
118     vector<pair<int, int>> cut;
119     spfa(source, sink);
120     for (auto & e: edges) {
121         if (e.flow == e.capacity && dist[e.from]
122 != oo && level[e.to] == oo && e.capacity > 0) {
123             cut.emplace_back(e.from, e.to);
124         }
125     }
126     return cut;
127 }
128
129 // Example on how to use
130 void solve()
131 {
132     int N = 10;
133
134     int source = 8;
135     int sink = 9;
136
137     Dinic flow(N);
138     flow.add_edge(8, 0, 4, 0);
139     flow.add_edge(8, 1, 3, 0);
140     flow.add_edge(8, 2, 2, 0);
141     flow.add_edge(8, 3, 1, 0);
142
143     flow.add_edge(0, 6, oo, 3);
144     flow.add_edge(0, 7, oo, 2);
145     flow.add_edge(0, 5, oo, 0);
146
147     flow.add_edge(1, 4, oo, 0);
148
149     flow.add_edge(4, 9, oo, 0);
150     flow.add_edge(5, 9, oo, 0);
151     flow.add_edge(6, 9, oo, 0);
152     flow.add_edge(7, 9, oo, 0);
153
154     int ans = flow.get_max_flow(source, sink);
155     debug(ans);
156     debug(flow.cost);
157 }
158
159 int32_t main()
160 {
161     solve();
162 }

```

2.10 Kosaraju

Complexity: $O(V+E)$

Find the strongly connected components of a graph

```
1 // TITLE: Kosaraju
2 // COMPLEXITY:  $O(V+E)$ 
3 // DESCRIPTION: Find the strongly connected
  components of a graph
4
5 int n,m;
6 vector<vi> g, gi, scc;
7 vi vis, order, p;
8
9 void dfs1(int a) {
10     if(vis[a]) return;
11     vis[a]=true;
12     for(auto& b:g[a]) {
13         dfs1(b);
14     }
15     order.pb(a);
16 }
17
18 void dfs2(int a, int orig) {
19     if (vis[a]) return;
20     vis[a]=true;
21     p[a]=orig;
22
23     for(auto& b:gi[a]) {
24         if (vis[b] && p[b] != orig)
25             scc[p[b]].pb(orig);
26         dfs2(b,orig);
27     }
28 }
29
30 void solve() {
31     cin>>n>>m;
32
33     g.assign(n, vi());
34     gi.assign(n, vi());
35     scc.assign(n, vi());
36     vis.assign(n, 0);
37     p.assign(n, 0);
38     rep(i, 0, m) {
39         int a,b;cin>>a>>b;a--;b--;
40         g[a].pb(b);
41         gi[b].pb(a);
42     }
43
44     rep(i,0,n)dfs1(i);
45     vis.assign(n,0);
46     for(int i=n-1; i>=0;i--) dfs2(order[i],order[i]);
47
48     vis.assign(n,0);
49 }
```

2.11 Dinic

Complexity: $O(V*V*E)$, Bipartite is $O(\sqrt{V} E)$

Dinic

```
1 // TITLE: Dinic
2 // COMPLEXITY:  $O(V*V*E)$ , Bipartite is  $O(\sqrt{V} E)$ 
3 // DESCRIPTION: Dinic
4
5 const int oo = 0x3f3f3f3f3f3f3f3f;
6 // Edge structure
7 struct Edge
8 {
9     int from, to;
10    int flow, capacity;
11 }
```

```
12    Edge(int from_, int to_, int flow_, int capacity_
13        )
14        : from(from_), to(to_), flow(flow_), capacity
15        (capacity_)
16    {}
17 };
18
19 struct Dinic
20 {
21     vector<vector<int>> graph;
22     vector<Edge> edges;
23     vector<int> level;
24     int size;
25
26     Dinic(int n)
27     {
28         graph.resize(n);
29         level.resize(n);
30         size = n;
31         edges.clear();
32     }
33
34     void add_edge(int from, int to, int capacity)
35     {
36         edges.emplace_back(from, to, 0, capacity);
37         graph[from].push_back(edges.size() - 1);
38
39         edges.emplace_back(to, from, 0, 0);
40         graph[to].push_back(edges.size() - 1);
41     }
42
43     int get_max_flow(int source, int sink)
44     {
45         int max_flow = 0;
46         vector<int> next(size);
47         while(bfs(source, sink)) {
48             next.assign(size, 0);
49             for (int f = dfs(source, sink, next, oo);
50                  f != 0; f = dfs(source, sink, next, oo)) {
51                 max_flow += f;
52             }
53         }
54         return max_flow;
55     }
56
57     bool bfs(int source, int sink)
58     {
59         level.assign(size, -1);
60         queue<int> q;
61         q.push(source);
62         level[source] = 0;
63
64         while(!q.empty()) {
65             int a = q.front();
66             q.pop();
67
68             for (int & b: graph[a]) {
69                 auto edge = edges[b];
70                 int cap = edge.capacity - edge.flow;
71                 if (cap > 0 && level[edge.to] == -1)
72                 {
73                     level[edge.to] = level[a] + 1;
74                     q.push(edge.to);
75                 }
76             }
77         }
78         return level[sink] != -1;
79     }
80
81     int dfs(int curr, int sink, vector<int> & next,
82             int flow)
83     {
84         if (curr == sink) return flow;
85     }
```



```

80         int num_edges = graph[curr].size();
81
82         for (; next[curr] < num_edges; next[curr]++)
83         {
84             int b = graph[curr][next[curr]];
85             auto & edge = edges[b];
86             auto & rev_edge = edges[b^1];
87
88             int cap = edge.capacity - edge.flow;
89             if (cap > 0 && (level[curr] + 1 == level[
edge.to])) {
90                 int bottle_neck = dfs(edge.to, sink,
next, min(flow, cap));
91                 if (bottle_neck > 0) {
92                     edge.flow += bottle_neck;
93                     rev_edge.flow -= bottle_neck;
94                     return bottle_neck;
95                 }
96             }
97             return 0;
98         }
99
100     vector<pair<int, int>> mincut(int source, int
sink)
101     {
102         vector<pair<int, int>> cut;
103         bfs(source, sink);
104         for (auto & e: edges) {
105             if (e.flow == e.capacity && level[e.from]
!= -1 && level[e.to] == -1 && e.capacity > 0) {
106                 cut.emplace_back(e.from, e.to);
107             }
108         }
109         return cut;
110     }
111 };
112
113 // Example on how to use
114 void solve()
115 {
116     int n, m;
117     cin >> n >> m;
118     int N = n + m + 2;
119
120     int source = N - 2;
121     int sink = N - 1;
122
123     Dinic flow(N);
124
125     for (int i = 0; i < n; i++) {
126         int q; cin >> q;
127         while(q--) {
128             int b; cin >> b;
129             flow.add_edge(i, n + b - 1, 1);
130         }
131     }
132     for (int i = 0; i < n; i++) {
133         flow.add_edge(source, i, 1);
134     }
135     for (int i = 0; i < m; i++) {
136         flow.add_edge(i + n, sink, 1);
137     }
138
139     cout << m - flow.get_max_flow(source, sink) <<
endl;
140
141     // Getting participant edges
142     for (auto & edge: flow.edges) {
143         if (edge.capacity == 0) continue; // This
means is a reverse edge
144         if (edge.from == source || edge.to == source)
continue;
145
146         if (edge.from == sink || edge.to == sink)
continue;
147         if (edge.flow == 0) continue; // Is not
participant
148
149         cout << edge.from + 1 << " " << edge.to - n +
1 << endl;
150     }
151 }

```

3 Segtree

3.1 Standard SegTree

Complexity: $O(\log(n))$ query and update
Sum segment tree with point update.

```

1 // TITLE: Standard SegTree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segment tree with point update.
4
5 using type = int;
6
7 type iden = 0;
8 vector<type> seg;
9 int segsize;
10
11 type func(type a, type b)
12 {
13     return a + b;
14 }
15
16 // query do intervalo [l, r)
17 type query(int l, int r, int no = 0, int lx = 0, int
rx = segsize)
18 {
19     // l lx rx r
20     if (r <= lx or rx <= l)
21         return iden;
22     if (l <= lx and rx <= r)
23         return seg[no];
24
25     int mid = lx + (rx - lx) / 2;
26     return func(query(l, r, 2 * no + 1, lx, mid),
query(l, r, 2 * no + 2, mid, rx));
27 }
28
29 void update(int dest, type val, int no = 0, int lx =
0, int rx = segsize)
30 {
31     if (dest < lx or dest >= rx)
32         return;
33     if (rx - lx == 1)
34     {
35         seg[no] = val;
36         return;
37     }
38
39     int mid = lx + (rx - lx) / 2;
40     update(dest, val, 2 * no + 1, lx, mid);
41     update(dest, val, 2 * no + 2, mid, rx);
42     seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
43 }
44
45 signed main()
46 {
47     ios_base::sync_with_stdio(0);
48     cin.tie(0);
49     cout.tie(0);
50     int n;
51

```

```

52     cin >> n;
53     segsize = n;
54     if (__builtin_popcount(n) != 1)
55     {
56         segsize = 1 + (int)log2(segsz);
57         segsize = 1 << segsize;
58     }
59     seg.assign(2 * segsize - 1, iden);
60
61     rep(i, 0, n)
62     {
63         int x;
64         cin >> x;
65         update(i, x);
66     }
67 }

```

3.2 Persistent sum segment tree

Complexity: $O(\log(n))$ query and update, $O(k \log(n))$ memory,
 n = number of elements, k = number of operations
Sum segment tree which preserves its history.

```

1 // TITLE: Persistent sum segment tree
2 // COMPLEXITY:  $O(\log(n))$  query and update,  $O(k \log(n))$  memory,  $n$  = number of elements,  $k$  = number of operations
3 // DESCRIPTION: Sum segment tree which preserves its history.
4
5 int segsize;
6
7 struct node {
8     int val;
9     int lx, rx;
10    node *l=0, *r=0;
11
12    node() {}
13    node(int val, int lx, int rx, node *l, node *r) : val(val), lx(lx), rx(rx), l(l), r(r) {}
14 };
15
16 node* build(vi& arr, int lx=0, int rx=segsz) {
17     if (rx - lx == 1) {
18         if (lx < (int)arr.size()) {
19             return new node(arr[lx], lx, rx, 0, 0);
20         }
21     }
22     return new node(0, lx, rx, 0, 0);
23 }
24
25 int mid = (lx+rx)/2;
26 auto nol = build(arr, lx, mid);
27 auto nor = build(arr, mid, rx);
28 return new node(nol->val + nor->val, lx, rx, nol, nor);
29 }
30
31 node* update(int idx, int val, node *no) {
32     if (idx < no->lx or idx >= no->rx) return no;
33     if (no->rx - no->lx == 1) {
34         return new node(val+no->val, no->lx, no->rx, no->l, no->r);
35     }
36
37     auto nol = update(idx, val, no->l);
38     auto nor = update(idx, val, no->r);
39     return new node(nol->val + nor->val, no->lx, no->rx, nol, nor);
40 }
41 }
42

```

```

43 int query(int l, int r, node *no) {
44     if (r <= no->lx or no->rx <= l) return 0;
45     if (l <= no->lx and no->rx <= r) return no->val;
46
47     return query(l, r, no->l) + query(l, r, no->r);
48 }

```

3.3 Set and update lazy seg

Complexity: $O(\log(n))$ query and update
Sum segtree with set and update

```

1 // TITLE: Set and update lazy seg
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segtree with set and update
4
5 vector<int> lazy, opvec;
6 vector<int> seg;
7
8 constexpr int SET = 30;
9 constexpr int ADD = 31;
10
11 int segsize;
12
13 void propagate(int no, int lx, int rx) {
14     if (lazy[no] == -1) return;
15
16     if (rx-lx == 1) {
17         if (opvec[no] == SET) seg[no] = lazy[no];
18         else seg[no] += lazy[no];
19
20         lazy[no] = -1;
21         opvec[no] = -1;
22         return;
23     }
24
25     if (opvec[no] == SET) {
26         seg[no] = (rx-lx) * lazy[no];
27         lazy[2*no+1] = lazy[no];
28         lazy[2*no+2] = lazy[no];
29
30         opvec[2*no+1] = SET;
31         opvec[2*no+2] = SET;
32
33         lazy[no] = -1;
34         opvec[no] = -1;
35         return;
36     }
37
38     seg[no] += (rx-lx) * lazy[no];
39     if (lazy[2*no+1] == -1) {
40         lazy[2*no+1] = 0;
41         opvec[2*no+1] = ADD;
42     }
43     if (lazy[2*no+2] == -1) {
44         lazy[2*no+2] = 0;
45         opvec[2*no+2] = ADD;
46     }
47     lazy[2*no+1] += lazy[no];
48     lazy[2*no+2] += lazy[no];
49
50     lazy[no] = -1;
51     opvec[no] = -1;
52 }
53
54 void update(int l, int r, int val, int op, int no=0, int lx=0, int rx=segsz) {
55     propagate(no, lx, rx);
56     if (r <= lx or l >= rx) return;
57     if (lx >= l and rx <= r) {
58         lazy[no] = val;
59         opvec[no] = op;

```

```

60     propagate(no, lx, rx);
61     return;
62 }
63
64 int mid = (rx+lx)/2;
65 update(1, r, val, op, 2*no+1, lx, mid);
66 update(1, r, val, op, 2*no+2, mid, rx);
67 seg[no] = seg[2*no+1]+seg[2*no+2];
68 }
69
70 int query(int l, int r, int no=0, int lx=0, int rx=
    segsize) {
71     propagate(no, lx, rx);
72     if (r <= lx or l >= rx) return 0;
73     if (lx >= l and rx <= r) return seg[no];
74
75     int mid = (rx+lx)/2;
76     return
77         query(1,r,2*no+1,lx,mid) +
78         query(1,r,2*no+2, mid, rx);
79 }

```

3.4 Binary Indexed Tree

Complexity: $O(\log(n))$ query and update

Range sum queries with point update. One-indexed.

```

1 // TITLE: Binary Indexed Tree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Range sum queries with point update.
  One-indexed.
4
5 struct BIT{
6     #define lowbit(x) ( x & -x )
7     int n;
8     vi b;
9
10    BIT( int n ) : n(n) , b(n+1 , 0){};
11    BIT( vi &c ){
12        n = c.size() , b = c;
13        for( int i = 1 , fa = i + lowbit(i) ; i <= n
14            ;
15            i ++ , fa = i + lowbit(i) )
16            if( fa <= n ) b[fa] += b[i];
17    }
18    void add( int i , int y ){
19        for( ; i <= n ; i += lowbit(i) ) b[i] += y;
20    }
21
22    int calc( int i ){
23        int sum = 0;
24        for( ; i ; i -= lowbit(i) ) sum += b[i];
25        return sum;
26    };
27 }

```

3.5 Lazy SegTree

Complexity: $O(\log(n))$ query and update

Sum segment tree with range sum update.

```

1 // TITLE: Lazy SegTree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segment tree with range sum
  update.
4 vector<int> seg, lazy;
5 int segsize;
6
7 // change 0s to -1s if update is
8 // set instead of add. also,

```

```

9 // remove the +=s
10 void prop(int no, int lx, int rx) {
11     if (lazy[no] == 0) return;
12
13     seg[no]+=(rx-lx)*lazy[no];
14     if(rx-lx>1) {
15         lazy[2*no+1] += lazy[no];
16         lazy[2*no+2] += lazy[no];
17     }
18
19     lazy[no]=0;
20 }
21
22 void update(int l, int r, int val,int no=0, int lx=0,
    int rx=segsize) {
23     // l r lx rx
24     prop(no, lx, rx);
25     if (r <= lx or rx <= l) return;
26     if (l <= lx and rx <= r) {
27         lazy[no]=val;
28         prop(no,lx,rx);
29         return;
30     }
31
32     int mid=lx+(rx-lx)/2;
33     update(1,r,val,2*no+1,lx,mid);
34     update(1,r,val,2*no+2,mid,rx);
35     seg[no] =seg[2*no+1]+seg[2*no+2];
36 }
37
38 int query(int l,int r,int no=0,int lx=0, int rx=
    segsize) {
39     prop(no,lx,rx);
40     if (r <= lx or rx <= l) return 0;
41     if (l <= lx and rx <= r) return seg[no];
42
43     int mid=lx+(rx-lx)/2;
44     return query(1,r,2*no+1, lx, mid)+
45         query(1,r,2*no+2,mid,rx);
46 }
47
48 signed main() {
49     ios_base::sync_with_stdio(0);cin.tie(0);cout.tie
    (0);
50
51     int n;cin>>n;
52     segsize=n;
53     if(__builtin_popcount(n) != 1) {
54         segsize=1+(int)log2(segsize);
55         segsize= 1<<segsize;
56     }
57
58     seg.assign(2*segsize-1, 0);
59     // use -1 instead of 0 if
60     // update is set instead of add
61     lazy.assign(2*segsize-1, 0);
62 }

```

4 Set

4.1 Set

Complexity: Insertion $\log(n)$

Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

```

1 // TITLE: Set
2 // COMPLEXITY: Insertion  $\log(n)$ 
3 // Description: Keeps elements sorted, remove
  duplicates, upper_bound, lower_bound, find, count

```

```

4
5 int main() {
6     set<int> set1;
7
8     set1.insert(1);          // O(log(n))
9     set1.erase(1);          // O(log(n))
10
11    set1.upper_bound(1);      // O(log(n))
12    set1.lower_bound(1);      // O(log(n))
13    set1.find(1);             // O(log(n))
14    set1.count(1);            // O(log(n))
15
16    set1.size();              // O(1)
17    set1.empty();             // O(1)
18
19    set1.clear()              // O(1)
20    return 0;
21 }

```

4.2 Multiset

Complexity: $O(\log(n))$

Same as set but you can have multiple elements with same values

```

1 // TITLE: Multiset
2 // COMPLEXITY: O(log(n))
3 // DESCRIPTION: Same as set but you can have multiple
  elements with same values
4
5 int main() {
6     multiset<int> set1;
7 }

```

4.3 Ordered Set

Complexity: $\log n$

Worst set with additional operations

```

1 // TITLE: Ordered Set
2 // COMPLEXITY: log n
3 // DESCRIPTION: Worst set with additional operations
4
5
6 #include <bits/extc++.h>
7 using namespace __gnu_pbds; // or pb_ds;
8 template<typename T, typename B = null_type>
9 using ordered_set = tree<T, B, less<T>, rb_tree_tag,
  tree_order_statistics_node_update>;
10
11 int32_t main() {
12     ordered_set<int> oset;
13
14     oset.insert(5);
15     oset.insert(1);
16     oset.insert(2);
17     // o_set = {1, 2, 5}
18     5 == *(oset.find_by_order(2)); // Like an array
  index
19     2 == oset.order_of_key(4); // How many elements
  are strictly less than 4
20 }

```

5 Misc

5.1 Template

Complexity: $O(1)$

Standard template for competitions

```

1 // TITLE: Template
2 // COMPLEXITY: O(1)
3 // DESCRIPTION: Standard template for competitions
4
5 #include <bits/stdc++.h>
6
7 #define int long long
8 #define endl '\n'
9 #define pb push_back
10 #define eb emplace_back
11 #define all(x) (x).begin(), (x).end()
12 #define rep(i, a, b) for(int i=(int)(a); i < (int)(b);
  i++)
13 #define debug(var) cout << #var << " : " << var <<
  endl
14 #define pii pair<int, int>
15 #define vi vector<int>
16
17 int MAX = 2e5;
18 int MOD=1e9+7;
19 int oo=0x3f3f3f3f3f3f3f3f;
20
21 using namespace std;
22
23 void solve()
24 {
25
26 }
27
28 signed main()
29 {
30     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie
  (0);
31     int t=1;
32     // cin>>t;
33     while(t--) solve();
34 }

```

6 String

6.1 String hash

Complexity: $O(n)$ preprocessing, $O(1)$ query

Computes the hash of arbitrary substrings of a given string s.

```

1 // TITLE: String hash
2 // COMPLEXITY: O(n) preprocessing, O(1) query
3 // DESCRIPTION: Computes the hash of arbitrary
  substrings of a given string s.
4 int m1, m2;
5 int n; string s;
6
7 struct Hash {
8     const int P = 31;
9     int n; string s;
10    vector<int> h, hi, p, p2, h2, hi2;
11    Hash() {}
12    Hash(string s):
13        s(s), n(s.size()), h(n), hi(n), p(n), h2(n), hi2(
  n), p2(n) {
14        for (int i=0; i<n; i++) p[i] = (i ? P*p[i-1]:1)
  % m1;

```

```

15     for (int i=0;i<n;i++) p2[i] = (i ? P*p2[i-1]:1) % m2;
16
17     for (int i=0;i<n;i++)
18         h[i] = (s[i] + (i ? h[i-1]:0) * P) % m1;
19     for (int i=0;i<n;i++)
20         h2[i] = (s[i] + (i ? h2[i-1]:0) * P) % m2;
21
22     for (int i=n-1;i>=0;i--)
23         hi[i] = (s[i] + (i+1<n ? hi[i+1]:0) * P) % m1;
24     for (int i=n-1;i>=0;i--)
25         hi2[i] = (s[i] + (i+1<n ? hi2[i+1]:0) * P) % m2;
26
27     int gethash(int l, int r) {
28         int hash = (h[r] - (l ? h[l-1]*p[r-l+1]:0) % m1 : 0);
29         int hash2 = (h2[r] - (l ? h2[l-1]*p2[r-l+1]:0) % m2 : 0);
30         hash = hash < 0 ? hash + m1 : hash;
31         hash2 = hash2 < 0 ? hash2 + m2 : hash2;
32         return (hash << 30) ^ hash2;
33     }
34     int gethashi(int l, int r) {
35         int hash = (hi[l] - (r+1 < n ? hi[r+1]*p[r-l+1]:0) % m1 : 0);
36         int hash2 = (hi2[l] - (r+1 < n ? hi2[r+1]*p2[r-l+1]:0) % m2 : 0);
37         hash = hash < 0 ? hash + m1 : hash;
38         hash2 = hash2 < 0 ? hash2 + m2 : hash2;
39         return (hash << 30) ^ hash2;
40     }
41 };
42
43 void solve()
44 {
45     srand(time(0));
46     m1 = rand()/10 + 1e9;
47     m2 = rand()/10 + 1e9;
48     Hash hasher(s);
49 }

```

```

22     int n=s.size();
23     vi p(n);
24     vi c(n);
25     {
26         vector<pair<char, int>> a(n);
27         rep(i,0,n) a[i]={s[i],i};
28         sort(all(a));
29
30         rep(i,0,n) p[i]=a[i].second;
31
32         c[p[0]]=0;
33         rep(i,1,n) {
34             if(s[p[i]] == s[p[i-1]]) {
35                 c[p[i]]=c[p[i-1]];
36             }
37             else c[p[i]]=c[p[i-1]]+1;
38         }
39     }
40
41     for(int k=0; (1<<k) < n; k++) {
42         rep(i, 0, n)
43             p[i] = (p[i] - (1<<k) + n) % n;
44
45         countingsort(p,c);
46
47         vi nc(n);
48         nc[p[0]]=0;
49         rep(i,1,n) {
50             pii prev = {c[p[i-1]], c[(p[i-1]+(1<<k))%n]};
51             pii cur = {c[p[i]], c[(p[i]+(1<<k))%n]};
52
53             if (prev == cur)
54                 nc[p[i]]=nc[p[i-1]];
55             else nc[p[i]]=nc[p[i-1]]+1;
56         }
57         c=nc;
58     }
59
60     return p;
61 }

```

6.2 Suffix Array

Complexity: $O(n \log(n))$, contains big constant (around 25).
Computes a sorted array of the suffixes of a string.

```

1 // TITLE: Suffix Array
2 // COMPLEXITY:  $O(n \log(n))$ , contains big constant (around 25).
3 // DESCRIPTION: Computes a sorted array of the suffixes of a string.
4
5 void countingsort(vi& p, vi& c) {
6     int n=p.size();
7     vi count(n,0);
8     rep(i,0,n) count[c[i]]++;
9
10    vi psum(n); psum[0]=0;
11    rep(i,1,n) psum[i]=psum[i-1]+count[i-1];
12
13    vi ans(n);
14    rep(i,0,n)
15        ans[psum[c[p[i]]]++]=p[i];
16
17    p = ans;
18 }
19
20 vi sfa(string s) {
21     s += "$";

```

6.3 Z function

Complexity: $O(n)$
 $z[i]$ = largest m such that $s[0..m]=s[i..i+m]$

```

1 // TITLE: Z function
2 // COMPLEXITY:  $O(n)$ 
3 // DESCRIPTION:  $z[i]$  = largest  $m$  such that  $s[0..m]=s[i..i+m]$ 
4
5 vector<int> Z(string s) {
6     int n = s.size();
7     vector<int> z(n);
8     int x = 0, y = 0;
9     for (int i = 1; i < n; i++) {
10         z[i] = max(0, min(z[i - x], y - i + 1));
11         while (i + z[i] < n and s[z[i]] == s[i + z[i]]) {
12             x = i; y = i + z[i]; z[i]++;
13         }
14     }
15     return z;
16 }

```

7 Geometry

7.1 Point structure

Complexity: Does not apply

Basic 2d point functionality

```
1 // TITLE: Point structure
2 // COMPLEXITY: Does not apply
3 // DESCRIPTION: Basic 2d point functionality
4
5 // Point/vector structure definition and sorting
6
7 #define T int
8 float EPS = 1e-6;
9 bool eq(T a, T b){ return abs(a-b)<=EPS; }
10
11 struct point{
12     T x, y;
13     point(t x=0, t y=0): x(x), y(y){}
14
15     point operator+(const point &o) const{ return {x
16 + o.x, y + o.y}; }
17     point operator-(const point &o) const{ return {x
18 - o.x, y + o.y}; }
19     point operator*(T k) const{ return {x*k, y*k}; }
20     point operator/(T k) const{ return {x/k, y/k}; }
21     T operator*(const point &o) const{ return x*o.x +
22 y*o.y; }
23     T operator^(const point &o) const{ return x*o.y -
24 y*o.x; }
25     bool operator<(const point &o) const{ return (eq(
26 x, o.x) ? y < o.y : x < o.x); }
27     bool operator==(const point &o) const{ return eq(
28 x, o.x) and eq(y, o.t); }
29
30     friend ostream& operator<<(ostream& os, point p){
31         return os << "(" << p.x << "," << p.y << ")";
32     }
33 };
34
35 int ret[2][2] = {{3, 2}, {4, 1}};
36 inline int quad(point p){
37     return ret[p.x >= 0][p.y >= 0];
38 }
39
40 bool comp(point a, point b){
41     int qa = quad(a), qb = quad(b);
42     return (qa == qb ? (a ^ b) > 0 : qa < qb);
43 }
44 }
```

7.2 Lattice Points

Complexity: N

Points with integer coordinate

```
1 // TITLE: Lattice Points
2 // COMPLEXITY: N
3 // DESCRIPTION: Points with integer coordinate
4
5 // Picks theorem
6 // A = area
7 // i = points_inside
8 // b = points in boundary including vertices
9 // A = i + b/2 - 1
10
11 void solve()
12 {
13     int n; cin >> n;
14     vector<Point> points(n);
15     for (int i = 0; i < n; i++) {
```

```
16         points[i].read();
17     }
18
19     // Calculatting points on boundary
20     int B = 0;
21     for (int i = 0; i < n; i++) {
22         int j = (i + 1) % n;
23         Point p = points[j] - points[i];
24         B += __gcd(abs(p.x), abs(p.y)); // Unsafe for 0
25     }
26     // Calculating Area
27     int a2 = 0;
28     for (int i = 0; i < n; i++) {
29         int j = (i + 1) % n;
30         a2 += points[i] * points[j];
31     }
32     a2 = abs(a2);
33     // Picks theorem
34     int I = (a2 - B + 2)/2;
35     cout << I << " " << B << endl;
36 }
```

7.3 Convex Hull

Complexity: N

Gives you the convex hull of a set of points

```
1 // TITLE: Convex Hull
2 // COMPLEXITY: N
3 // DESCRIPTION: Gives you the convex hull of a set of
4 points
5
6 struct Point
7 {
8     int x, y;
9
10     void read()
11     {
12         cin >> x >> y;
13     }
14
15     Point operator- (const Point & b) const
16     {
17         Point p;
18         p.x = x - b.x;
19         p.y = y - b.y;
20         return p;
21     }
22
23     void operator-= (const Point & b)
24     {
25         x -= b.x;
26         y -= b.y;
27     }
28
29     int operator* (const Point & b) const
30     {
31         return x * b.y - b.x * y;
32     }
33
34     bool operator< (const Point & b) const
35     {
36         return make_pair(x, y) < make_pair(b.x, b.y);
37     }
38 };
39
40 int triangle(const Point & a, const Point & b, const
41 Point & c)
42 {
43     return (b - a) * (c - a);
```

```

44 }
45
46 vector<Point> convex_hull(vector<Point> points)
47 {
48     vector<Point> hull;
49     sort(all(points));
50
51     for (int z = 0; z < 2; z++) {
52         int s = hull.size();
53         for (int i = 0; i < points.size(); i++) {
54             while(hull.size() >= s + 2) {
55                 auto a = hull.end()[-2];
56                 auto b = hull.end()[-1];
57                 if (triangle(a, b, points[i]) <= 0) {
58                     break;
59                 }
60                 hull.pop_back();
61             }
62             hull.push_back(points[i]);
63         }
64         hull.pop_back();
65         reverse(all(points));
66     }
67     return hull;
68 }

```

7.4 Line Intersegment

Complexity: $O(1)$

Check if two half segments intersect with which other

```

1 // TITLE: Line Intersegment
2 // COMPLEXITY:  $O(1)$ 
3 // DESCRIPTION: Check if two half segments intersect
  with which other
4
5 struct Point
6 {
7     int x, y;
8
9     void read()
10    {
11        cin >> x >> y;
12    }
13
14    Point operator- (const Point & b) const
15    {
16        Point p;
17        p.x = x - b.x;
18        p.y = y - b.y;
19        return p;
20    }
21
22    void operator-= (const Point & b)
23    {
24        x -= b.x;
25        y -= b.y;
26    }
27
28    int operator* (const Point & b) const
29    {
30        return x * b.y - b.x * y;
31    }
32 };
33
34 int triangle(const Point & a, const Point & b, const
  Point & c)
35 {
36     return (b - a) * (c - a);
37 }
38 }
39

```

```

40 bool intersect(const Point & p1, const Point & p2,
  const Point & p3, const Point & p4) {
41     bool ans = true;
42     int s1 = triangle(p1, p2, p3);
43     int s2 = triangle(p1, p2, p4);
44
45     if (s1 == 0 && s2 == 0) {
46         int a_min_x = min(p1.x, p2.x);
47         int a_max_x = max(p1.x, p2.x);
48         int a_min_y = min(p1.y, p2.y);
49         int a_max_y = max(p1.y, p2.y);
50
51         int b_min_x = min(p3.x, p4.x);
52         int b_max_x = max(p3.x, p4.x);
53         int b_min_y = min(p3.y, p4.y);
54         int b_max_y = max(p3.y, p4.y);
55         if (a_min_x > b_max_x || a_min_y > b_max_y) {
56             ans = false;
57         }
58         if (b_min_x > a_max_x || b_min_y > a_max_y) {
59             ans = false;
60         }
61         return ans;
62     }
63     int s3 = triangle(p3, p4, p1);
64     int s4 = triangle(p3, p4, p2);
65
66     if ((s1 < 0) && (s2 < 0)) ans = false;
67     if ((s1 > 0) && (s2 > 0)) ans = false;
68     if ((s3 < 0) && (s4 < 0)) ans = false;
69     if ((s3 > 0) && (s4 > 0)) ans = false;
70     return ans;
71 }

```

8 Algorithms

8.1 Sparse table

Complexity: $O(n \log(n))$ preprocessing, $O(1)$ query

Computes the minimum of a half open interval.

```

1 // TITLE: Sparse table
2 // COMPLEXITY:  $O(n \log(n))$  preprocessing,  $O(1)$  query
3 // DESCRIPTION: Computes the minimum of a half open
  interval.
4
5 struct sptable {
6     vector<vi> table;
7
8     int ilog(int x) {
9         return (__builtin_clzll(1ll) -
  __builtin_clzll(x));
10    }
11
12    sptable(vi& vals) {
13        int n = vals.size();
14        int ln = ilog(n)+1;
15        table.assign(ln, vi(n));
16
17        rep(i,0,n) table[0][i]=vals[i];
18
19        rep(k, 1, ln) {
20            rep(i,0,n) {
21                table[k][i] = min(table[k-1][i],
  table[k-1][min(i + (1<<(k-1)), n-1)])
22            }
23        }
24    }
25 }
26

```

```

27 // returns minimum of vals in range [a, b)
28 int getmin(int a, int b) {
29     int k = ilog(b-a);
30     return min(table[k][a], table[k][b-(1<<k)]);
31 }
32 };

```

9 Parser

9.1 Parsing Functions

Complexity:

```

1 // TITLE: Parsing Functions
2
3 vector<string> split_string(const string & s, const
    string & sep = " ") {
4     int w = sep.size();
5     vector<string> ans;
6     string curr;
7
8     auto add = [&](string a) {
9         if (a.size() > 0) {
10             ans.push_back(a);
11         }
12     };
13
14     for (int i = 0; i + w < s.size(); i++) {
15         if (s.substr(i, w) == sep) {
16             i += w-1;
17             add(curr);
18             curr.clear();
19             continue;

```

```

20         }
21         curr.push_back(s[i]);
22     }
23     add(curr);
24     return ans;
25 }
26
27 vector<int> parse_vector_int(string & s)
28 {
29     vector<int> nums;
30     for (string x: split_string(s)) {
31         nums.push_back(stoi(x));
32     }
33     return nums;
34 }
35
36 vector<float> parse_vector_float(string & s)
37 {
38     vector<float> nums;
39     for (string x: split_string(s)) {
40         nums.push_back(stof(x));
41     }
42     return nums;
43 }
44
45 void solve()
46 {
47     cin.ignore();
48     string s;
49     getline(cin, s);
50
51     auto nums = parse_vector_float(s);
52     for (auto x: nums) {
53         cout << x << endl;
54     }
55 }

```