

# UNAM

## Contents

<b>1 Data structures</b>	<b>2</b>	3.4 Binomial Coefficient . . . . .	19
1.1 Fenwick Tree . . . . .	2	3.5 Fibonacci (Logn) . . . . .	20
1.2 Segment Tree . . . . .	2	3.6 GCD . . . . .	20
1.3 Segment tree(Lazy) . . . . .	2	3.7 Sieve $O(n \log(\log(n)))$ . . . . .	20
1.4 Segment tree types . . . . .	3	3.8 Inverse Modulo . . . . .	20
1.5 DSU . . . . .	4	3.9 Divisors . . . . .	21
1.6 DSU with deletion . . . . .	4	3.10 NTT . . . . .	21
1.7 BIT . . . . .	6	3.11 FFT . . . . .	22
1.8 BIT2D . . . . .	6	3.12 Matrix Exponentiation . . . . .	23
1.9 Oset . . . . .	6	3.13 Mobius . . . . .	23
1.10 Multiset . . . . .	6	3.14 Phi . . . . .	24
1.11 Treap . . . . .	7	3.15 Prefix sum Phi . . . . .	24
1.12 Treap with inversion . . . . .	8	3.16 Identities . . . . .	24
1.13 HLD . . . . .	8	3.17 Burnside's Lemma . . . . .	25
<b>2 Graphs</b>	<b>9</b>	3.18 Recursion . . . . .	25
2.1 BFS . . . . .	9	3.19 Theorems . . . . .	25
2.2 DFS . . . . .	9	<b>4 Game Theory</b>	<b>25</b>
2.3 Bellman Ford . . . . .	10	4.1 Sprague-Grundy theorem . . . . .	25
2.4 Dijkstra . . . . .	10	<b>5 Strings</b>	<b>25</b>
2.5 Floyd Warshal . . . . .	10	5.1 Manacher . . . . .	25
2.6 Strongly Connected Components . . . . .	10	5.2 Suffix Array $O(n \log(n))$ . . . . .	26
2.7 Condensation Graph . . . . .	11	5.3 Z Function $O(n)$ . . . . .	26
2.8 Bridges Articulation . . . . .	11	5.4 KMP . . . . .	27
2.9 LCA . . . . .	12	<b>6 DP Optimization</b>	<b>28</b>
2.10 2sat . . . . .	13	6.1 Convex Hull Trick . . . . .	28
2.11 Bipartite check . . . . .	14	6.2 Divide and Conquer DP . . . . .	28
2.12 LCA with RMQ . . . . .	14	<b>7 Other</b>	<b>29</b>
2.13 Find Centroid . . . . .	16	7.1 Binary Search . . . . .	29
2.14 Centroid Decomposition . . . . .	16	7.2 BinSearch with doubles . . . . .	29
2.15 Koenig . . . . .	17	7.3 Ternary Search . . . . .	29
2.16 Max Bipartite Matching . . . . .	18	7.4 LIS DP . . . . .	29
<b>3 Math</b>	<b>19</b>	7.5 Random numbers . . . . .	29
3.1 Binary Exponentiation . . . . .	19	7.6 XOR basis . . . . .	30
3.2 Binom small numbers without mod . . . . .	19	7.7 Bitsets . . . . .	30
3.3 Factorial . . . . .	19	7.8 All permutations . . . . .	30
		7.9 Print doubles . . . . .	30
		<b>8 Flows</b>	<b>30</b>
		8.1 Maximum Flow $O(V * E * E)$ . . . . .	30

8.2	Maximum Flow $O(V*V*E)$ . . . . .	31
8.3	Maximum matching $O(E\sqrt{V})$ . . . . .	32
8.4	Minimum Cost Maximum Flow . . . . .	33
8.5	Dinic . . . . .	34
<b>9</b>	<b>Geometry</b> . . . . .	<b>35</b>
9.1	Point struct . . . . .	35
9.2	Sort points . . . . .	36
<b>10</b>	<b>Compile</b> . . . . .	<b>36</b>
10.1	Template . . . . .	36
10.2	Compile . . . . .	36

## 1 Data structures

### 1.1 Fenwick Tree

```

1 ll fwsum(ll k){
2     ll s = 0;
3     while(k >= 1){
4         s += tr[k];
5         k -= k & (-k);
6     }
7     return s;
8 }
9
10 void fwadd(ll k, ll x, int n){
11     while(k <= n){
12         tr[k] += x;
13         k += k & (-k);
14     }
15 }
```

### 1.2 Segment Tree

```

1 int x[400000] = {1000000001};
2 int n;
3 void update(int a, int b)
4 {
5     a += n;
6     x[a] = b;
7     for (a /= 2; a >= 1; a /= 2)
8     {
```

```

9         x[a] = min(x[a * 2], x[a * 2 + 1]);
10    }
11 }
12 int find(int a, int b)
13 {
14     a += n;
15     b += n;
16     int s = 1000000000;
17     while (a <= b)
18     {
19         if (a % 2 == 1)
20             s = min(s, x[a++]);
21         if (b % 2 == 0)
22             s = min(s, x[b--]);
23         a /= 2;
24         b /= 2;
25     }
26     return s;
27 }
```

### 1.3 Segment tree(Lazy)

```

1 template <typename num_t>
2 struct segtree {
3     int n, depth;
4     vector<num_t> tree, lazy;
5
6     void init(int s, long long* arr) {
7         n = s;
8         tree = vector<num_t>(4 * s, 0);
9         lazy = vector<num_t>(4 * s, 0);
10        init(0, 0, n - 1, arr);
11    }
12
13    num_t init(int i, int l, int r, long long* arr) {
14        if (l == r) return tree[i] = arr[l];
15
16        int mid = (l + r) / 2;
17        num_t a = init(2 * i + 1, l, mid, arr),
18              b = init(2 * i + 2, mid + 1, r, arr);
19        return tree[i] = a.op(b);
20    }
21 }
```

```

22 void update(int l, int r, num_t v) {
23     if (l > r) return;
24     update(0, 0, n - 1, l, r, v);
25 }
26
27 num_t update(int i, int tl, int tr, int ql, int qr, num_t v) {
28     eval_lazy(i, tl, tr);
29
30     if (tr < ql || qr < tl) return tree[i];
31     if (ql <= tl && tr <= qr) {
32         lazy[i] = lazy[i].val + v.val;
33         eval_lazy(i, tl, tr);
34         return tree[i];
35     }
36
37     int mid = (tl + tr) / 2;
38     num_t a = update(2 * i + 1, tl, mid, ql, qr, v),
39             b = update(2 * i + 2, mid + 1, tr, ql, qr, v);
40     return tree[i] = a.op(b);
41 }
42
43 num_t query(int l, int r) {
44     if (l > r) return num_t::null_v;
45     return query(0, 0, n-1, l, r);
46 }
47
48 // int get_first(int v, int tl, int tr, int l, int r, int x) {
49 //     eval_lazy(0, tl, tr);
50 //     if (tl > r || tr < l) return -1;
51 //     if (tree[v].val < x) return -1;
52
53 //     if (tl == tr) return tl;
54
55 //     int tm = tl + (tr - tl) / 2;
56 //     int left = get_first(2*v+1, tl, tm, l, r, x);
57 //     if (left != -1) return left;
58 //     return get_first(2*v+2, tm+1, tr, l, r, x);
59 // }
60
61 num_t query(int i, int tl, int tr, int ql, int qr) {
62     eval_lazy(i, tl, tr);
63
64     if (ql <= tl && tr <= qr) return tree[i];

```

```

65     if (tr < ql || qr < tl) return num_t::null_v;
66
67     int mid = (tl + tr) / 2;
68     num_t a = query(2 * i + 1, tl, mid, ql, qr),
69             b = query(2 * i + 2, mid + 1, tr, ql, qr);
70     return a.op(b);
71 }
72
73 void eval_lazy(int i, int l, int r) {
74     tree[i] = tree[i].lazy_op(lazy[i], (r - l + 1));
75     if (l != r) {
76         lazy[i * 2 + 1] = lazy[i].val + lazy[i * 2 + 1].val;
77         lazy[i * 2 + 2] = lazy[i].val + lazy[i * 2 + 2].val;
78     }
79
80     lazy[i] = num_t();
81 }
82 };

```

## 1.4 Segment tree types

```

1 struct max_t {
2     long long val;
3     static const long long null_v = -9223372036854775807LL;
4
5     max_t(): val(0) {}
6     max_t(long long v): val(v) {}
7
8     max_t op(max_t& other) {
9         return max_t(max(val, other.val));
10    }
11
12    max_t lazy_op(max_t& v, int size) {
13        return max_t(val + v.val);
14    }
15 };
16
17
18 struct min_t {
19     long long val;
20     static const long long null_v = 9223372036854775807LL;
21
22    min_t(): val(0) {}

```

```

23 min_t(long long v): val(v) {}
24
25 min_t op(min_t& other) {
26     return min_t(min(val, other.val));
27 }
28
29 min_t lazy_op(min_t& v, int size) {
30     return min_t(val + v.val);
31 }
32 };
33
34
35 struct sum_t {
36     long long val;
37     static const long long null_v = 0;
38
39     sum_t(): val(0) {}
40     sum_t(long long v): val(v) {}
41
42     sum_t op(sum_t& other) {
43         return sum_t(val + other.val);
44     }
45
46     sum_t lazy_op(sum_t& v, int size) {
47         return sum_t(val + v.val * size);
48     }
49 };

```

## 1.5 DSU

```

1 class DisjointSets {
2     private:
3     vector<int> parents;
4     vector<int> sizes;
5
6     public:
7     DisjointSets(int size) : parents(size), sizes(size, 1) {
8         for (int i = 0; i < size; i++) { parents[i] = i; }
9     }
10
11     /** @return the "representative" node in x's component */
12     int find(int x) {
13         return parents[x] == x ? x : (parents[x] = find(parents[x]));

```

```

14     }
15
16     /** @return whether the merge changed connectivity */
17     bool unite(int x, int y) {
18         int x_root = find(x);
19         int y_root = find(y);
20         if (x_root == y_root) { return false; }
21
22         if (sizes[x_root] < sizes[y_root]) { swap(x_root, y_root); }
23         sizes[x_root] += sizes[y_root];
24         parents[y_root] = x_root;
25         return true;
26     }
27
28     /** @return whether x and y are in the same connected component */
29     bool connected(int x, int y) { return find(x) == find(y); }
30 };
31 /**recibe valores de un grafo cero indexado, retorna valores uno indexado

```

## 1.6 DSU with deletion

```

1 struct dsu_save {
2     int v, rnk, u, rnku;
3
4     dsu_save() {}
5
6     dsu_save(int _v, int _rnk, int _u, int _rnk)
7         : v(_v), rnk(_rnk), u(_u), rnku(_rnk) {}
8 };
9
10 struct dsu_with_rollbacks {
11     vector<int> p, rnk;
12     int comps;
13     stack<dsu_save> op;
14
15     dsu_with_rollbacks() {}
16
17     dsu_with_rollbacks(int n) {
18         p.resize(n);
19         rnk.resize(n);
20         for (int i = 0; i < n; i++) {
21             p[i] = i;
22             rnk[i] = 0;

```

```

23     }
24     comps = n;
25 }
26
27 int find_set(int v) {
28     return (v == p[v]) ? v : find_set(p[v]);
29 }
30
31 bool unite(int v, int u) {
32     v = find_set(v);
33     u = find_set(u);
34     if (v == u)
35         return false;
36     comps--;
37     if (rnk[v] > rnk[u])
38         swap(v, u);
39     op.push(dsu_save(v, rnk[v], u, rnk[u]));
40     p[v] = u;
41     if (rnk[u] == rnk[v])
42         rnk[u]++;
43     return true;
44 }
45
46 void rollback() {
47     if (op.empty())
48         return;
49     dsu_save x = op.top();
50     op.pop();
51     comps++;
52     p[x.v] = x.v;
53     rnk[x.v] = x.rnk;
54     p[x.u] = x.u;
55     rnk[x.u] = x.rnk;
56 }
57 };
58
59 struct query {
60     int v, u;
61     bool united;
62     query(int _v, int _u) : v(_v), u(_u) {
63     }
64 };
65

```

```

66 struct QueryTree {
67     vector<vector<query>> t;
68     dsu_with_rollback dsu;
69     int T;
70
71     QueryTree() {}
72
73     QueryTree(int _T, int n) : T(_T) {
74         dsu = dsu_with_rollback(n);
75         t.resize(4 * T + 4);
76     }
77
78     void add_to_tree(int v, int l, int r, int ul, int ur, query& q) {
79         if (ul > ur)
80             return;
81         if (l == ul && r == ur) {
82             t[v].push_back(q);
83             return;
84         }
85         int mid = (l + r) / 2;
86         add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
87         add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
88     }
89     // nodes to unite in query, interval of time when nodes are
90     // connected
91     void add_query(query q, int l, int r) {
92         add_to_tree(1, 0, T - 1, l, r, q);
93     }
94
95     void dfs(int v, int l, int r, vector<int>& ans) {
96         for (query& q : t[v]) {
97             q.united = dsu.unite(q.v, q.u);
98         }
99         if (l == r)
100             ans[l] = dsu.comps;
101         else {
102             int mid = (l + r) / 2;
103             dfs(2 * v, l, mid, ans);
104             dfs(2 * v + 1, mid + 1, r, ans);
105         }
106         for (query q : t[v]) {
107             if (q.united)
108                 dsu.rollback();
109         }
110     }
111 };

```

```

108     }
109 }
110
111 vector<int> solve() {
112     vector<int> ans(T);
113     dfs(1, 0, T - 1, ans);
114     return ans;
115 }
116 }; // when using map of pairs be careful with order as m[{a, b}] != m[{b
    , a}]

```

## 1.7 BIT

```

1 #define MAXN 10000
2 int bit[MAXN];
3 void update(int x, int val){
4     for(; x < MAXN; x+=x&-x)
5         bit[x] += val;
6 }
7 int get(int x){
8     int ans = 0;
9     for(; x; x-=x&-x)
10        ans += bit[x];
11     return ans;
12 }

```

## 1.8 BIT2D

```

1 #define MAXN 1000
2 int bit[MAXN][MAXN];
3
4 void update(int x, int y, int val){
5     for(; x < MAXN; x+=x&-x)
6         for(int j = y; j < MAXN; j+=j&-j)
7             bit[x][j] += val;
8 }
9
10 int get(int x, int y){
11     int ans = 0;
12     for(; x; x-=x&-x)
13         for(int j = y; j; j-=j&-j)
14             ans += bit[x][j];
15     return ans;
16 }

```

```

17
18 int get(int x1, int y1, int x2, int y2){
19     return get(x2, y2) - get(x1-1, y2) - get(x2, y1-1) + get(x1-1, y1-1);
20 }

```

## 1.9 Oset

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T> using oset = __gnu_pbds::tree<
5     T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
6     __gnu_pbds::tree_order_statistics_node_update
7 >;
8 // order_of_key() primero mayor o igual;
9 // find_by_order() apuntador al elemento k;
10 // oset<pair<int,int>> os;
11 // os.insert({1,2});
12 // os.insert({2,3});
13 // os.insert({5,6});
14 // ll k=os.order_of_key({2,0});
15 // cout<<k<<endl; // 1
16 // pair<int,int> p=os.find_by_order(k);
17 // cout<<p.f<<" "<<p.s<<endl; // 2 3
18 // os.erase(p);
19 // p=os.find_by_order(k);
20 // cout<<p.f<<" "<<p.s<<endl; // 5 6

```

## 1.10 Multioset

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T> using oset = __gnu_pbds::tree<
5     T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
6     __gnu_pbds::tree_order_statistics_node_update
7 >;
8
9 //en main
10
11 oset<pair<int,int>> name;
12 map<int,int> cuenta;
13 function<void(int)> meter = [&] (int val) {
14     name.insert({val,++cuenta[val]});

```

```

15     };
16     auto quitar = [&] (int val) {
17         name.erase({val, cuenta[val]--});
18     };
19
20 meter(x);
21 quitar(y);
22 multiset.order_of_key({y+1,-1})-multiset.order_of_key({x,0})

```

### 1.11 Treap

```

1 struct Node {
2     Node *l = 0, *r = 0;
3     int val, y, c = 1;
4     Node(int val) : val(val), y(rand()) {}
5     void recalc();
6 };
7
8 int cnt(Node* n) { return n ? n->c : 0; }
9 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
10
11 template<class F> void each(Node* n, F f) {
12     if (n) { each(n->l, f); f(n->val); each(n->r, f); }
13 }
14
15 pair<Node*, Node*> split(Node* n, int k) {
16     if (!n) return {};
17     if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
18         auto pa = split(n->l, k);
19         n->l = pa.second;
20         n->recalc();
21         return {pa.first, n};
22     } else {
23         auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
24         n->r = pa.first;
25         n->recalc();
26         return {n, pa.second};
27     }
28 }
29
30 Node* merge(Node* l, Node* r) {
31     if (!l) return r;
32     if (!r) return l;

```

```

33     if (l->y > r->y) {
34         l->r = merge(l->r, r);
35         l->recalc();
36         return l;
37     } else {
38         r->l = merge(l, r->l);
39         r->recalc();
40         return r;
41     }
42 }
43
44 Node* ins(Node* t, Node* n, int pos) {
45     auto pa = split(t, pos);
46     return merge(merge(pa.first, n), pa.second);
47 }
48
49 // Example application: move the range [l, r) to index k
50 void move(Node*& t, int l, int r, int k) {
51     Node *a, *b, *c;
52     tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
53     if (k <= l) t = merge(ins(a, b, k), c);
54     else t = merge(a, ins(c, b, k - r));
55 }
56
57 // Usage
58 // create treap
59 // Node* name=nullptr;
60 // insert element
61 // name=ins(name, new Node(val), pos);
62 // Node* x = new Node(val);
63 // name = ins(name, x, pos);
64 // merge two treaps (name before x)
65 // name=merge(name, x);
66 // split treap (this will split treap in two treaps,
67 // first with elements [0, pos) and second with elements [pos, n))
68 // pa will be pair of two treaps
69 // auto pa = split(name, pos);
70 // move range [l, r) to index k
71 // move(name, l, r, k);
72 // iterate over treap
73 // each(name, [&](int val) {
74 //     cout << val << ' ';
75 // });

```

## 1.12 Treap with inversion

```

1 struct Node {
2     Node *l = 0, *r = 0;
3     int val, y, c = 1;
4     bool rev = 0;
5     Node(int val) : val(val), y(rand()) {}
6     void recalc();
7     void push();
8 };
9
10 int cnt(Node* n) { return n ? n->c : 0; }
11 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
12 void Node::push() {
13     if (rev) {
14         rev = 0;
15         swap(l, r);
16         if (l) l->rev ^= 1;
17         if (r) r->rev ^= 1;
18     }
19 }
20
21 template<class F> void each(Node* n, F f) {
22     if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
23 }
24
25 pair<Node*, Node*> split(Node* n, int k) {
26     if (!n) return {};
27     n->push();
28     if (cnt(n->l) >= k) {
29         auto pa = split(n->l, k);
30         n->l = pa.second;
31         n->recalc();
32         return {pa.first, n};
33     } else {
34         auto pa = split(n->r, k - cnt(n->l) - 1);
35         n->r = pa.first;
36         n->recalc();
37         return {n, pa.second};
38     }
39 }
40
41 Node* merge(Node* l, Node* r) {

```

```

42     if (!l) return r;
43     if (!r) return l;
44     l->push();
45     r->push();
46     if (l->y > r->y) {
47         l->r = merge(l->r, r);
48         l->recalc();
49         return l;
50     } else {
51         r->l = merge(l, r->l);
52         r->recalc();
53         return r;
54     }
55 }
56
57 Node* ins(Node* t, Node* n, int pos) {
58     auto pa = split(t, pos);
59     return merge(merge(pa.first, n), pa.second);
60 }
61
62 // Example application: reverse the range [l, r]
63 void reverse(Node*& t, int l, int r) {
64     Node *a, *b, *c;
65     tie(a,b) = split(t, l);
66     tie(b,c) = split(b, r - l + 1);
67     b->rev ^= 1;
68     t = merge(merge(a, b), c);
69 }
70
71 void move(Node*& t, int l, int r, int k) {
72     Node *a, *b, *c;
73     tie(a,b) = split(t, l);
74     tie(b,c) = split(b, r - l);
75     if (k <= l) t = merge(ins(a, b, k), c);
76     else t = merge(a, ins(c, b, k - r));
77 }

```

## 1.13 HLD

```

1 vector<int> g[N];
2 int par[N][LG + 1], dep[N], sz[N];
3 void dfs(int u, int p = 0) {
4     par[u][0] = p;

```



```

5   dep[u] = dep[p] + 1;
6   sz[u] = 1;
7   for (int i = 1; i <= LG; i++) par[u][i] = par[par[u][i - 1]][i - 1];
8   if (p) g[u].erase(find(g[u].begin(), g[u].end(), p));
9   for (auto &v : g[u]) if (v != p) {
10      dfs(v, u);
11      sz[u] += sz[v];
12      if (sz[v] > sz[g[u][0]]) swap(v, g[u][0]);
13   }
14 }
15 int lca(int u, int v) {
16   if (dep[u] < dep[v]) swap(u, v);
17   for (int k = LG; k >= 0; k--) if (dep[par[u][k]] >= dep[v]) u = par[u][k];
18   if (u == v) return u;
19   for (int k = LG; k >= 0; k--) if (par[u][k] != par[v][k]) u = par[u][k], v = par[v][k];
20   return par[u][0];
21 }
22 int kth(int u, int k) {
23   assert(k >= 0);
24   for (int i = 0; i <= LG; i++) if (k & (1 << i)) u = par[u][i];
25   return u;
26 }
27 int T, head[N], st[N], en[N];
28 void dfs_hld(int u) {
29   st[u] = ++T;
30   for (auto v : g[u]) {
31     head[v] = (v == g[u][0] ? head[u] : v);
32     dfs_hld(v);
33   }
34   en[u] = T;
35 }
36 int n;
37 //verify that the operation is the correct one.
38 int query_up(int u, int v) {
39   int ans = -inf;
40   while (head[u] != head[v]) {
41     ans = max(ans, t.query(1, 1, n, st[head[u]], st[u]));
42     u = par[head[u]][0];
43   }
44   ans = max(ans, t.query(1, 1, n, st[v], st[u]));
45   return ans;

```

```

46 }
47 int query(int u, int v) {
48   int l = lca(u, v);
49   int ans = query_up(u, l);
50   if (v != l) ans = max(ans, query_up(v, kth(v, dep[v] - dep[l] - 1)));
51   return ans;
52 }

```

## 2 Graphs

### 2.1 BFS

```

1 queue<int> q;
2 q.push(x);
3 vis[x]=1;
4 dis[x]=0;
5 while(!q.empty()){
6   int s=q.front();
7   q.pop();
8   for(auto u:adj[s]){
9     if(!vis[u]){
10      vis[u]=1;
11      dis[u]=dis[s]+1;
12      q.push(u);
13    }
14  }
15 }

```

### 2.2 DFS

```

1 vector<vector<int>>> adj;
2 vector<bool> vis;
3 void dfs(int a){
4   vis[a]=1;
5   for(auto u:adj[a]){
6     if(!vis[u]){
7       dfs(u);
8     }
9   }
10 }
11
12 //inside solve()
13

```

```

14 adj.assign(n,vector<int>());
15 vis.assign(n,0);

```

## 2.3 Bellman Ford

```

1 vector <tuple <ll, ll, ll>> edges(m);
2 for(int i = 0; i < m; i++){
3     ll a, b, c; cin >> a >> b >> c;
4     edges[i] ={a, b, c};
5 }
6
7 ll distance[n + 1];
8 for(int i = 0; i <= n; i++) distance[i] = INF;
9 distance[1] = 0;
10 for(int i = 1; i <= n - 1; i++){
11     bool change = false;
12     for(auto e: edges){
13         ll a, b, w;
14         tie(a, b, w) = e;
15         ll temp = distance[b];
16         distance[b] = min(distance[b], distance[a] + w);
17         if(temp != distance[b]) change = true;
18     }
19     if(!change) break;
20 }

```

## 2.4 Dijkstra

```

1 vector<vector<pair<ll,ll>>> g; // u->[(v,cost)]
2 vector<ll> dist;
3 int n; // be careful with redeclaration of n, works 0 indexed
4 void dijkstra(int x){
5     dist.assign(n,-1);
6     priority_queue<pair<ll,int> > q;
7     dist[x]=0;
8     q.push({0,x});
9     while(!q.empty()){
10         x=q.top().second;
11         ll c=-q.top().first;
12         q.pop();
13         if(dist[x]!=c)continue;
14         for(int i=0;i<g[x].size();i++){
15             int y=g[x][i].first;
16             ll c=g[x][i].second;

```

```

17         if(dist[y]<0||dist[x]+c<dist[y])
18             dist[y]=dist[x]+c,q.push({-dist[y],y});
19     }
20 }
21 }
22
23 void solve(){
24     g.assign(n,vector<pair<ll,ll>>());
25 }

```

## 2.5 Floyd Warshal

```

1 ll distances[n + 1][n + 1];
2 for(int i = 1; i <= n; i++){
3     for(int j = 1; j <= n; j++){
4         if(i == j) distances[i][j] = 0;
5         else if(adj[i][j]) distances[i][j] = adj[i][j];
6         else distances[i][j] = INF;
7     }
8 }
9
10 for(int k = 1; k <= n; k++){
11     for(int i = 1; i <= n; i++){
12         for(int j = 1; j <= n; j++){
13             distances[i][j] = min(distances[i][j], distances[i][k] + distances
14                                     [k][j]);
15         }
16     }
17 }

```

## 2.6 Strongly Connected Components

```

1 vector<vector<int>> adj,adjr;
2 vector<bool> vis;
3 vector<int> order,comp;
4 void dfs(int a){
5     vis[a]=1;
6     for(auto u:adj[a]){
7         if(!vis[u]){
8             dfs(u);
9         }
10     }
11 }
12 order.pb(a);

```

```

13 }
14 void dfsr(int a,int k){
15     vis[a]=1;
16     comp[a]=k;
17     for(auto u:adjr[a]){
18         if(!vis[u]){
19             dfsr(u,k);
20         }
21     }
22 }
23
24 void solve() {
25     int n,m;cin>>n>>m;
26     adj.assing(n,vector<int>());
27     adjr.assing(n,vector<int>());
28     comp.resize(n);
29     for(int i=0;i<m;i++){
30         int a,b;cin>>a>>b;a--;b--;
31         adj[a].pb(b);
32         adjr[b].pb(a);
33     }
34     vis.assign(n,0);
35     for(int i=0;i<n;i++){
36         if(!vis[i])dfs(i);
37     }
38     vis.assign(n,0);
39     int c=0;
40     for(int i=n-1;i>=0;i--){
41         if(!vis[order[i]]){
42             dfsr(order[i],c);
43             c++;
44         }
45     }
46 }
47 }

```

## 2.7 Condensation Graph

```

1 //after scc
2 vector<vector<int>> adj_scc;
3
4 void cndstn(int c){
5     adj_scc.assign(c,vector<int>());

```

```

6     for(ll i=0;i<n;i++){
7         for(auto u:adj[i]){
8             if(comp[u]!=comp[i]){
9                 adj_scc[comp[i]].pb(comp[u]);
10            }
11        }
12    }
13 }

```

## 2.8 Bridges Articulation

```

1 int n;
2 vector<vector<int>> adj;
3
4 vector<bool> visited;
5 vector<int> tin, low;
6 int timer;
7
8 void dfs(int v, int p = -1) {
9     visited[v] = true;
10    tin[v] = low[v] = timer++;
11    for (int to : adj[v]) {
12        if (to == p) continue;
13        if (visited[to]) {
14            low[v] = min(low[v], tin[to]);
15        } else {
16            dfs(to, v);
17            low[v] = min(low[v], low[to]);
18            //bridge
19            //if (low[to] > tin[v]){
20            //articulation
21            //if (low[to] >= tin[v] && p!=-1){
22        }
23    }
24 }
25
26
27 void find_bridges() {
28     timer = 0;
29     visited.assign(n, false);
30     tin.assign(n, -1);
31     low.assign(n, -1);
32     for (int i = 0; i < n; ++i) {

```

```

33     if (!visited[i])
34         dfs(i);
35     }
36 }

```

## 2.9 LCA

```

1  class Tree
2  {
3  public:
4      const int root = 0;
5
6      const vector<vector<int>> &adj;
7      const int log2dist;
8      vector<int> par;
9      vector<vector<int>> pow2ends;
10     vector<int> depth;
11     /** works with 0 indexed graph ** /
12     /** check if graph is connected, if not, just iterate trough all nodes
13         **/
14     /** use DFS to calculate the depths and parents of each node */
15     void process(int at, int prev)
16     {
17         depth[at] = depth[prev] + 1;
18         for (int n : adj[at])
19         {
20             if (n != prev)
21             {
22                 process(n, at);
23                 par[n] = at;
24             }
25         }
26     }
27 public:
28     Tree(const vector<vector<int>> &adj)
29         : adj(adj), log2dist(std::ceil(std::log2(adj.size()))), par(adj.
30             size()),
31             pow2ends(par.size(), vector<int>(log2dist + 1)), depth(adj.size
32             ())
33     {
34         par[root] = depth[root] = -1;
35         process(root, root);

```

```

34     for (int n = 0; n < par.size(); n++)
35     {
36         pow2ends[n][0] = par[n];
37     }
38     for (int p = 1; p <= log2dist; p++)
39     {
40         for (int n = 0; n < par.size(); n++)
41         {
42             int halfway = pow2ends[n][p - 1];
43             if (halfway == -1)
44             {
45                 pow2ends[n][p] = -1;
46             }
47             else
48             {
49                 pow2ends[n][p] = pow2ends[halfway][p - 1];
50             }
51         }
52     }
53 }
54
55 /** @return the kth parent of node n */
56 int kth_parent(int n, int k)
57 {
58     if (k > par.size())
59     {
60         return -1;
61     }
62     int at = n;
63     for (int pow = 0; pow <= log2dist; pow++)
64     {
65         if ((k & (1 << pow)) != 0)
66         {
67             at = pow2ends[at][pow];
68             if (at == -1)
69             {
70                 break;
71             }
72         }
73     }
74     return at;
75 }
76

```

```

77
78  /** @return the LCA of nodes n1 and n2 */
79  int lca(int n1, int n2)
80  {
81      if (depth[n1] < depth[n2])
82      {
83          return lca(n2, n1);
84      }
85      // lift n1 up to the same height as n2
86      n1 = kth_parent(n1, depth[n1] - depth[n2]);
87      if (n1 == n2)
88      {
89          return n2; // in this case, n2 is a direct ancestor of n1
90      }
91
92      // move the nodes up as long as they don't meet
93      for (int i = log2dist; i >= 0; i--)
94      {
95          if (pow2ends[n1][i] != pow2ends[n2][i])
96          {
97              n1 = pow2ends[n1][i];
98              n2 = pow2ends[n2][i];
99          }
100      }
101      // at this point, the lca will be the parent of either node
102      return pow2ends[n1][0];
103  }
104  };

```

## 2.10 2sat

```

1  #include<bits/stdc++.h>
2  using namespace std;
3
4  const int N = 3e5 + 9;
5
6  /*
7  zero Indexed
8  we have vars variables
9  F=(x_0 XXX y_0) and (x_1 XXX y_1) and ... (x_{vars-1} XXX y_{vars-1})
10 here {x_i,y_i} are variables
11 and XXX belongs to {OR,XOR}
12 is there any assignment of variables such that F=true

```

```

13  */
14  struct twosat {
15      int n; // total size combining +, -. must be even.
16      vector< vector<int> > g, gt;
17      vector<bool> vis, res;
18      vector<int> comp;
19      stack<int> ts;
20      twosat(int vars = 0) {
21          n = vars << 1;
22          g.resize(n);
23          gt.resize(n);
24      }
25
26      //zero indexed, be careful
27      //if you want to force variable a to be true in OR or XOR combination
28      //add addOR (a,1,a,1);
29      //if you want to force variable a to be false in OR or XOR combination
30      //add addOR (a,0,a,0);
31
32      //(x_a or (not x_b)) -> af=1,bf=0
33      void addOR(int a, bool af, int b, bool bf) {
34          a += a + (af ^ 1);
35          b += b + (bf ^ 1);
36          g[a ^ 1].push_back(b); // !a => b
37          g[b ^ 1].push_back(a); // !b => a
38          gt[b].push_back(a ^ 1);
39          gt[a].push_back(b ^ 1);
40      }
41
42      //(!x_a xor !x_b) -> af=0, bf=0
43      void addXOR(int a, bool af, int b, bool bf) {
44          addOR(a, af, b, bf);
45          addOR(a, !af, b, !bf);
46      }
47
48      void _add(int a, bool af, int b, bool bf) {
49          a += a + (af ^ 1);
50          b += b + (bf ^ 1);
51          g[a].push_back(b);
52          gt[b].push_back(a);
53      }
54
55      //add this type of condition ->
56      //add(a,af,b,bf) means if a is af then b must need to be bf
57      void add(int a, bool af, int b, bool bf) {
58          _add(a, af, b, bf);
59      }

```

```

56     _add(b, !bf, a, !af);
57 }
58 void dfs1(int u) {
59     vis[u] = true;
60     for(int v : g[u]) if(!vis[v]) dfs1(v);
61     ts.push(u);
62 }
63 void dfs2(int u, int c) {
64     comp[u] = c;
65     for(int v : gt[u]) if(comp[v] == -1) dfs2(v, c);
66 }
67 bool ok() {
68     vis.resize(n, false);
69     for(int i = 0; i < n; ++i) if(!vis[i]) dfs1(i);
70     int scc = 0;
71     comp.resize(n, -1);
72     while(!ts.empty()) {
73         int u = ts.top();
74         ts.pop();
75         if(comp[u] == -1) dfs2(u, scc++);
76     }
77     res.resize(n / 2);
78     for(int i = 0; i < n; i += 2) {
79         if(comp[i] == comp[i + 1]) return false;
80         res[i / 2] = (comp[i] > comp[i + 1]);
81     }
82     return true;
83 }
84 };
85
86 int main() {
87     int n, m; cin >> n >> m;
88     twosat ts(n);
89     for(int i = 0; i < m; i++){
90         int u, v, k; cin >> u >> v >> k;
91         --u; --v;
92         if(k) ts.add(u, 0, v, 0), ts.add(u, 1, v, 1), ts.add(v, 0, u, 0), ts
            .add(v, 1, u, 1);
93         else ts.add(u, 0, v, 1), ts.add(u, 1, v, 0), ts.add(v, 0, u, 1), ts
            .add(v, 1, u, 0);
94     }
95     int k = ts.ok();
96     if(!k) cout<<"Impossible\n";

```

```

97     else {
98         vector<int> v;
99         for(int i = 0; i < n; i++) if(ts.res[i]) v.push_back(i);
100         cout << (int)v.size() << '\n';
101         for(auto x: v) cout << x + 1 << ' ';
102         cout << '\n';
103     }
104     return 0;
105 }

```

## 2.11 Bipartite check

```

1  int n;
2  vector<vector<int>> adj;
3
4  vector<int> side(n, -1);
5  bool is_bipartite = true;
6  queue<int> q;
7  for (int st = 0; st < n; ++st) {
8      if (side[st] == -1) {
9          q.push(st);
10         side[st] = 0;
11         while (!q.empty()) {
12             int v = q.front();
13             q.pop();
14             for (int u : adj[v]) {
15                 if (side[u] == -1) {
16                     side[u] = side[v] ^ 1;
17                     q.push(u);
18                 } else {
19                     is_bipartite &= side[u] != side[v];
20                 }
21             }
22         }
23     }
24 }
25
26 cout << (is_bipartite ? "YES" : "NO") << endl;

```

## 2.12 LCA with RMQ

```

1  class Tree //if lca needed just copy lca and kth_parent functions from
   | lca.cpp or change return from max_edge_cost function
2  {

```

```

3 private:
4     const int root = 0;
5
6     const vector<vector<int>> &adj;
7     const vector <vector<int>> &weight;
8     const int log2dist;
9     vector<int> par;
10    vector<int> edgepar;
11    vector<vector<int>> pow2ends;
12    vector<vector<int>> maxedge;
13    vector<int> depth;
14
15    /** use DFS to calculate the depths and parents of each node */
16    void process(int at, int prev)
17    {
18        int cont = 0;
19        depth[at] = depth[prev] + 1;
20        for (int n : adj[at])
21        {
22            if (n != prev)
23            {
24                process(n, at);
25                par[n] = at;
26                edgepar[n] = weight[at][cont];
27            }
28            cont++;
29        }
30    }
31
32 public:
33    Tree(const vector<vector<int>> &adj, const vector <vector<int>> &
        weight)
34        : adj(adj), weight(weight), log2dist(std::ceil(std::log2(adj.size()
        )), par(adj.size()), edgepar(adj.size()),
35        pow2ends(par.size(), vector<int>(log2dist + 1)), maxedge(par.
        size(), vector <int>(log2dist + 1)), depth(adj.size())
36    {
37        par[root] = depth[root] = -1;
38        edgepar[root] = 0;
39        process(root, root);
40
41        for (int n = 0; n < par.size(); n++)
42        {

```

```

43        pow2ends[n][0] = par[n];
44        maxedge[n][0] = edgepar[n];
45    }
46    for (int p = 1; p <= log2dist; p++)
47    {
48        for (int n = 0; n < par.size(); n++)
49        {
50            int halfway = pow2ends[n][p - 1];
51            if (halfway == -1)
52            {
53                pow2ends[n][p] = -1;
54                maxedge[n][p] = -1;
55            }
56            else
57            {
58                pow2ends[n][p] = pow2ends[halfway][p - 1];
59                maxedge[n][p] = max(maxedge[n][p - 1], maxedge[halfway][p -
60                1]);
61            }
62        }
63    }
64
65    /** @return the kth parent of node n with max edge cost */
66    pair <int, int> kth_parent_with_max_edge(int n, int k)
67    {
68        if (k > par.size())
69        {
70            pair <int, int> temp;
71            temp.first = -1;
72            temp.second = 0;
73            return temp;
74        }
75        int at = n, maxcost = 0;
76        for (int pot = 0; pot <= log2dist; pot++)
77        {
78            if ((k & (1 << pot)) != 0)
79            {
80                maxcost = max(maxedge[at][pot], maxcost);
81                at = pow2ends[at][pot];
82                if (at == -1)
83                {
84                    break;

```

```

85     }
86     }
87 }
88 pair <int, int> ans;
89 ans.first = at;
90 ans.second = maxcost;
91 return ans;
92 }
93
94
95 int max_edge_cost(int n1, int n2)
96 {
97     if (depth[n1] < depth[n2])
98     {
99         return max_edge_cost(n2, n1);
100     }
101     int maxcost;
102     // lift n1 up to the same height as n2 and find max edge of lifting
103     pair <int, int> temp = kth_parent_with_max_edge(n1, depth[n1] -
104         depth[n2]);
105     n1 = temp.first;
106     maxcost = temp.second;
107     if (n1 == n2)
108     {
109         return maxcost; // in this case, n2 is a direct ancestor of n1,
110         return maxcost
111     }
112
113     // move the nodes up as long as they don't meet
114     for (int i = log2dist; i >= 0; i--)
115     {
116         if (pow2ends[n1][i] != pow2ends[n2][i])
117         {
118             maxcost = max(maxcost, maxedge[n1][i]);
119             n1 = pow2ends[n1][i];
120             maxcost = max(maxcost, maxedge[n2][i]);
121             n2 = pow2ends[n2][i];
122         }
123     }
124     maxcost = max(maxcost, maxedge[n1][0]);
125     maxcost = max(maxcost, maxedge[n2][0]);
126     // at this point, the lca will be the parent of either node
127     return maxcost;

```

```

126     }
127 };

```

## 2.13 Find Centroid

```

1  const int maxn = 200010;
2
3  int n;
4  vector<int> adj[maxn];
5  int subtree_size[maxn];
6
7  int get_subtree_size(int node, int parent = -1) {
8      int &res = subtree_size[node];
9      res = 1;
10     for (int i : adj[node]) {
11         if (i == parent) { continue; }
12         res += get_subtree_size(i, node);
13     }
14     return res;
15 }
16
17 int get_centroid(int node, int parent = -1) {
18     for (int i : adj[node]) {
19         if (i == parent) { continue; }
20
21         if (subtree_size[i] * 2 > n) { return get_centroid(i, node); }
22     }
23     return node;
24 }
25
26 // Usage:
27 // get_subtree_size(0) to calculate subtree sizes
28 // get_centroid(0) to get the centroid of the tree

```

## 2.14 Centroid Decomposition

```

1  vector<vector<int>> adj;
2  vector<bool> is_removed;
3  vector<int> subtree_size;
4  // probably you want to add a parent array here
5
6
7  /** DFS to calculate the size of the subtree rooted at 'node' */
8  int get_subtree_size(int node, int parent = -1) {

```



```

9  subtree_size[node] = 1;
10 for (int child : adj[node]) {
11     if (child == parent || is_removed[child]) { continue; }
12     subtree_size[node] += get_subtree_size(child, node);
13 }
14 return subtree_size[node];
15 }
16
17 /**
18  * Returns a centroid (a tree may have two centroids) of the subtree
19  * containing node 'node' after node removals
20  * @param node current node
21  * @param tree_size size of current subtree after node removals
22  * @param parent parent of u
23  * @return first centroid found
24  */
25 int get_centroid(int node, int tree_size, int parent = -1) {
26     for (int child : adj[node]) {
27         if (child == parent || is_removed[child]) { continue; }
28         if (subtree_size[child] * 2 > tree_size) {
29             return get_centroid(child, tree_size, node);
30         }
31     }
32     return node;
33 }
34
35 /** Build up the centroid decomposition recursively */
36 void build_centroid_decomp(int node = 0, int parent = -1) {
37     int centroid = get_centroid(node, get_subtree_size(node));
38     // add parent array here
39
40     // do something
41
42     is_removed[centroid] = true;
43
44     for (int child : adj[centroid]) {
45         if (is_removed[child]) { continue; }
46         build_centroid_decomp(child, centroid);
47     }
48 }

```

## 2.15 Koenig

```

1  # ADD Maximum_Bipartite_Matching in python
2
3
4  def augment(u, bigraph, visit, timestamp, match):
5      """find augmenting path starting from u, by recursive DFS"""
6      for v in bigraph[u]:
7          if visit[v] < timestamp:
8              visit[v] = timestamp
9              if match[v] is None or augment(match[v], bigraph,
10                                             visit, timestamp, match):
11                  match[v] = u      # found an augmenting path
12              return True
13      return False
14
15
16  def max_bipartite_matching(bigraph):
17      """Bipartite maximum matching
18
19      param bigraph: adjacency list, index = vertex in U,
20      value = neighbor list in V
21      comment: U and V can have different cardinalities
22      returns: matching list, match[v] = u iff (u,v) in matching
23      complexity: O(|V|*|E|)
24      """
25      nU = len(bigraph)      # nU = cardinality of U, nV = card. of V
26      nV = max(max(adjlist, default=-1) for adjlist in bigraph) + 1
27      match = [None] * nV
28      visit = [-1] * nV      # timestamp of last visit
29      for u in range(nU):
30          augment(u, bigraph, visit, u, match)
31      return match
32
33
34
35  def alternate(u, bigraph, visitU, visitV, matchV):
36      """extend alternating tree from free vertex u.
37      visitU, visitV marks all vertices covered by the tree.
38      """
39      visitU[u] = True
40      for v in bigraph[u]:
41          if not visitV[v]:
42              visitV[v] = True
43              assert matchV[v] is not None      # otherwise match not maximum

```

```

44     alternate(matchV[v], bigraph, visitU, visitV, matchV)
45
46 #
47 #####
48 # Here starts Koenig algorithm
49 def koenig(bigraph):
50     """Bipartite minimum vertex cover by Koenig's theorem
51
52     param bigraph: adjacency list, index = vertex in U,
53     value = neighbor list in V
54     assumption: U = V = {0, 1, 2, ..., n-1} for n = len(bigraph)
55     returns: boolean table for U, boolean table for V
56     comment: selected vertices form a minimum vertex cover,
57     i.e. every edge is adjacent to at least one selected
58     vertex
59     and number of selected vertices is minimum
60     complexity: O(|V| * |E|)
61     """
62     V = range(len(bigraph))
63     matchV = max_bipartite_matching(bigraph)
64     matchU = [None for u in V]
65     for v in V:
66         # -- build the mapping from U to V
67         if matchV[v] is not None:
68             matchU[matchV[v]] = v
69     visitU = [False for u in V]
70     visitV = [False for v in V]
71     # -- build max alternating forest
72     for u in V:
73         if matchU[u] is None:
74             # -- starting with free vertices in
75             # U
76             alternate(u, bigraph, visitU, visitV, matchV)
77     inverse = [not b for b in visitU]
78     return (inverse, visitV)

```

## 2.16 Max Bipartite Matching

```

1  int n, k;
2  vector<vector<int>> g;
3  vector<int> mt;
4  vector<bool> used;

```

```

7  bool try_kuhn(int v) {
8      if (used[v])
9          return false;
10     used[v] = true;
11     for (int to : g[v]) {
12         if (mt[to] == -1 || try_kuhn(mt[to])) {
13             mt[to] = v;
14             return true;
15         }
16     }
17     return false;
18 }
19
20 int main() {
21     //... reading the graph ...
22
23     mt.assign(k, -1);
24     for (int v = 0; v < n; ++v) {
25         used.assign(n, false);
26         try_kuhn(v);
27     }
28
29     for (int i = 0; i < k; ++i)
30         if (mt[i] != -1)
31             printf("%d %d\n", mt[i] + 1, i + 1);
32 }

```

```

1
2 def augment(u, bigraph, visit, timestamp, match):
3     """find augmenting path starting from u, by recursive DFS"""
4     for v in bigraph[u]:
5         if visit[v] < timestamp:
6             visit[v] = timestamp
7             if match[v] is None or augment(match[v], bigraph,
8                 visit, timestamp, match):
9                 match[v] = u
10                # found an augmenting path
11                return True
12    return False

```

```

14 def max_bipartite_matching(bigraph):
15     """Bipartite maximum matching
16

```

```

17  """param_bigraph: adjacency_list, index_=vertex_in_U,
18  value_=neighbor_list_in_V
19  """comment: U and V can have different cardinalities
20  """returns: matching_list, match[v]_=u iff (u,v) in matching
21  """complexity: O(|V|*|E|)
22  """
23  nU = len(bigraph) # nU = cardinality of U, nV = card. of V
24  nV = max(max(adjlist, default=-1) for adjlist in bigraph) + 1
25  match = [None] * nV
26  visit = [-1] * nV # timestamp of last visit
27  for u in range(nU):
28      augment(u, bigraph, visit, u, match)
29  return match
30
31
32
33 def alternate(u, bigraph, visitU, visitV, matchV):
34     """extend alternating tree from free vertex u.
35     visitU, visitV marks all vertices covered by the tree.
36     """
37     visitU[u] = True
38     for v in bigraph[u]:
39         if not visitV[v]:
40             visitV[v] = True
41             assert matchV[v] is not None # otherwise match not maximum
42             alternate(matchV[v], bigraph, visitU, visitV, matchV)

```

## 3 Math

### 3.1 Binary Exponentiation

```

1  long long binexp(long long a, long long b)
2  {
3      long long res = 1;
4      while (b > 0)
5      {
6          if (b & 1)
7              res = res * a;
8          a = a * a;
9          b >>= 1;
10     }
11     return res;
12 }

```

### 3.2 Binom small numbers without mod

```

1  double bin[501][501];
2  void build(){
3      bin[0][0] = 1;
4      for(int k = 0; k < 501; k++){
5          for(int n = k; n < 501; n++){
6              if(k == 0){
7                  bin[n][k] = 1;
8                  continue;
9              }
10             if(n == k){
11                 bin[n][k] = 1;
12                 continue;
13             }
14             bin[n][k] = bin[n - 1][k - 1] + bin[n - 1][k];
15         }
16     }
17 }

```

### 3.3 Factorial

```

1  ll vals[1000001];
2  void fi(){
3      vals[0]=1;
4      vals[1]=1;
5      for(int i=2;i<1000001;i++){
6          vals[i]=i*vals[i-1];
7          vals[i]%=MOD;
8      }
9  }

```

### 3.4 Binomial Coefficient

```

1  ll binom(ll a,ll b){
2      if(b>a)return 0;
3      ll ans=vals[a];
4      ans*=inverse(vals[b]);
5      ans%=MOD;
6      ans*=inverse(vals[a-b]);
7      ans%=MOD;
8      return ans;
9  }

```

### 3.5 Fibonacci (Logn)

```

1 void fib(ll n, ll&x, ll&y){
2     if(n==0){
3         x = 0;
4         y = 1;
5         return ;
6     }
7
8     if(n&1){
9         fib(n-1, y, x);
10        y=(y+x)%MOD;
11    }else{
12        ll a, b;
13        fib(n>>1, a, b);
14        y = (a*a+b*b)%MOD;
15        x = (a*b + a*(b-a+MOD))%MOD;
16    }
17 }
18
19 // Usage
20 // ll x, y;
21 // fib(10, x, y);
22 // cout << x << " " << y << endl;
23 // This will output 55 89

```

### 3.6 GCD

```

1 ll gcd(ll a,ll b){
2     if(a==0)return b;
3     return gcd(b%a,a);
4 }

```

### 3.7 Sieve $O(n\log(\log(n)))$

```

1 const int kMaxV = 1e6;
2
3 int sieve[kMaxV + 1];
4
5 void primes()
6 {
7     for (int i = 4; i <= kMaxV; i += 2)
8         sieve[i] = 2;
9     for (int i = 3; i <= kMaxV / i; i += 2)

```

```

10     {
11         if (sieve[i])
12             continue;
13         for (int j = i * i; j <= kMaxV; j += i)
14             sieve[j] = i;
15     }
16 }
17
18 vector<int> PrimeFactors(int x)
19 {
20     if (x == 1)
21         return {};
22
23     unordered_set<int> primes;
24     while (sieve[x])
25     {
26         primes.insert(sieve[x]);
27         x /= sieve[x];
28     }
29     primes.insert(x);
30     return {primes.begin(), primes.end()};
31 }

```

### 3.8 Inverse Modulo

```

1 ll inverse(ll a,ll b=MOD,ll n=1,ll m=0){
2     if(a==1){
3         return n;
4     }
5     if(a<b){
6         long long x=b/a;
7         m+=(x*n);
8         m=m%MOD;
9         b=b%a;
10        return inverse(a,b,n,m);
11    }
12    else if(b==1){
13        return(MOD-m);
14    }
15    else{
16        long long x=a/b;
17        n+=(x*m);
18        n=n%MOD;

```

```

19     a=a%b;
20     return inverse(a,b,n,m);
21 }
22 }
23
24 // other inverse
25 ll inverse(ll a){
26     return binexp(a,MOD-2);
27 }

```

### 3.9 Divisors

```

1 long long numberOfDivisors(long long num)
2 {
3     long long total = 1;
4     for (int i = 2; (long long)i * i <= num; i++)
5     {
6         if (num % i == 0)
7         {
8             int e = 0;
9             do
10             {
11                 e++;
12                 num /= i;
13             } while (num % i == 0);
14             total *= e + 1;
15         }
16     }
17     if (num > 1)
18     {
19         total *= 2;
20     }
21     return total;
22 }
23
24 long long SumOfDivisors(long long num)
25 {
26     long long total = 1;
27
28     for (int i = 2; (long long)i * i <= num; i++)
29     {
30         if (num % i == 0)
31         {

```

```

32         int e = 0;
33         do
34         {
35             e++;
36             num /= i;
37         } while (num % i == 0);
38
39         long long sum = 0, pow = 1;
40         do
41         {
42             sum += pow;
43             pow *= i;
44         } while (e-- > 0);
45         total *= sum;
46     }
47 }
48 if (num > 1)
49 {
50     total *= (1 + num);
51 }
52 return total;
53 }

```

### 3.10 NTT

```

1 // number theory transform
2
3 const int MOD = 998244353, ROOT = 3;
4 // const int MOD = 7340033, ROOT = 5;
5 // const int MOD = 167772161, ROOT = 3;
6 // const int MOD = 469762049, ROOT = 3;
7
8 int power(int base, int exp) {
9     int res = 1;
10    while (exp) {
11        if (exp % 2) res = 1LL * res * base % MOD;
12        base = 1LL * base * base % MOD;
13        exp /= 2;
14    }
15    return res;
16 }
17
18 void ntt(vector<int>& a, bool invert) {

```

```

19 int n = a.size();
20 for (int i = 1, j = 0; i < n; i++) {
21     int bit = n >> 1;
22     for (; j & bit; bit >>= 1) j ^= bit;
23     j ^= bit;
24     if (i < j) swap(a[i], a[j]);
25 }
26 for (int len = 2; len <= n; len <= 1) {
27     int wlen = power(ROOT, (MOD - 1) / len);
28     if (invert) wlen = power(wlen, MOD - 2);
29     for (int i = 0; i < n; i += len) {
30         int w = 1;
31         for (int j = 0; j < len / 2; j++) {
32             int u = a[i + j], v = 1LL * a[i + j + len / 2] * w % MOD;
33             a[i + j] = u + v < MOD ? u + v : u + v - MOD;
34             a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + MOD;
35             w = 1LL * w * wlen % MOD;
36         }
37     }
38 }
39 if (invert) {
40     int n_inv = power(n, MOD - 2);
41     for (int& x : a) x = 1LL * x * n_inv % MOD;
42 }
43 }
44
45 vector<int> multiply(vector<int>& a, vector<int>& b) {
46     int n = 1;
47     while (n < a.size() + b.size()) n <= 1;
48     a.resize(n), b.resize(n);
49     ntt(a, false), ntt(b, false);
50     for (int i = 0; i < n; i++) a[i] = 1LL * a[i] * b[i] % MOD;
51     ntt(a, true);
52     return a;
53 }
54 // usage
55 // vector<int> a = {1, 2, 3}, b = {4, 5, 6};
56 // vector<int> c = multiply(a, b);
57 // for (int x : c) cout << x << " ";

```

### 3.11 FFT

```
1 | const int N = 3e5 + 9;
```

```

2
3 const double PI = acos(-1);
4 struct base {
5     double a, b;
6     base(double a = 0, double b = 0) : a(a), b(b) {}
7     const base operator + (const base &c) const
8         { return base(a + c.a, b + c.b); }
9     const base operator - (const base &c) const
10        { return base(a - c.a, b - c.b); }
11     const base operator * (const base &c) const
12        { return base(a * c.a - b * c.b, a * c.b + b * c.a); }
13 };
14 void fft(vector<base> &p, bool inv = 0) {
15     int n = p.size(), i = 0;
16     for(int j = 1; j < n - 1; ++j) {
17         for(int k = n >> 1; k > (i ^= k); k >>= 1);
18         if(j < i) swap(p[i], p[j]);
19     }
20     for(int l = 1, m; (m = 1 << 1) <= n; l <= 1) {
21         double ang = 2 * PI / m;
22         base wn = base(cos(ang), (inv ? 1. : -1.) * sin(ang)), w;
23         for(int i = 0, j, k; i < n; i += m) {
24             for(w = base(1, 0), j = i, k = i + 1; j < k; ++j, w = w * wn) {
25                 base t = w * p[j + 1];
26                 p[j + 1] = p[j] - t;
27                 p[j] = p[j] + t;
28             }
29         }
30     }
31     if(inv) for(int i = 0; i < n; ++i) p[i].a /= n, p[i].b /= n;
32 }
33 vector<long long> multiply(vector<int> &a, vector<int> &b) {
34     int n = a.size(), m = b.size(), t = n + m - 1, sz = 1;
35     while(sz < t) sz <= 1;
36     vector<base> x(sz), y(sz), z(sz);
37     for(int i = 0 ; i < sz; ++i) {
38         x[i] = i < (int)a.size() ? base(a[i], 0) : base(0, 0);
39         y[i] = i < (int)b.size() ? base(b[i], 0) : base(0, 0);
40     }
41     fft(x), fft(y);
42     for(int i = 0; i < sz; ++i) z[i] = x[i] * y[i];
43     fft(z, 1);
44     vector<long long> ret(sz);

```

```

45 for(int i = 0; i < sz; ++i) ret[i] = (long long) round(z[i].a);
46 while((int)ret.size() > 1 && ret.back() == 0) ret.pop_back();
47 return ret;
48 }
49 // usage
50 // vector<int> a = {1, 2, 3}, b = {4, 5, 6};
51 // vector<long long> c = multiply(a, b);

```

### 3.12 Matrix Exponentiation

```

1 struct Mat {
2     int n, m;
3     vector<vector<int>> a;
4     Mat() { }
5     Mat(int _n, int _m) {n = _n; m = _m; a.assign(n, vector<int>(m, 0)); }
6     Mat(vector< vector<int> > v) { n = v.size(); m = n ? v[0].size() : 0;
7         a = v; }
8     inline void make_unit() {
9         assert(n == m);
10        for (int i = 0; i < n; i++) {
11            for (int j = 0; j < n; j++) a[i][j] = i == j;
12        }
13    inline Mat operator + (const Mat &b) {
14        assert(n == b.n && m == b.m);
15        Mat ans = Mat(n, m);
16        for(int i = 0; i < n; i++) {
17            for(int j = 0; j < m; j++) {
18                ans.a[i][j] = (a[i][j] + b.a[i][j]) % mod;
19            }
20        }
21        return ans;
22    }
23    inline Mat operator - (const Mat &b) {
24        assert(n == b.n && m == b.m);
25        Mat ans = Mat(n, m);
26        for(int i = 0; i < n; i++) {
27            for(int j = 0; j < m; j++) {
28                ans.a[i][j] = (a[i][j] - b.a[i][j] + mod) % mod;
29            }
30        }
31        return ans;
32    }

```

```

33 inline Mat operator * (const Mat &b) {
34     assert(m == b.n);
35     Mat ans = Mat(n, b.m);
36     for(int i = 0; i < n; i++) {
37         for(int j = 0; j < b.m; j++) {
38             for(int k = 0; k < m; k++) {
39                 ans.a[i][j] = (ans.a[i][j] + 1LL * a[i][k] * b.a[k][j] % mod)
40                     % mod;
41             }
42         }
43     }
44     return ans;
45 }
46 inline Mat pow(long long k) {
47     assert(n == m);
48     Mat ans(n, n), t = a; ans.make_unit();
49     while (k) {
50         if (k & 1) ans = ans * t;
51         t = t * t;
52         k >>= 1;
53     }
54     return ans;
55 }
56 inline Mat& operator += (const Mat& b) { return *this = (*this) + b; }
57 inline Mat& operator -= (const Mat& b) { return *this = (*this) - b; }
58 inline Mat& operator *= (const Mat& b) { return *this = (*this) * b; }
59 inline bool operator == (const Mat& b) { return a == b.a; }
60 inline bool operator != (const Mat& b) { return a != b.a; }
61 };
62 // Usage
63 // Mat a(n, n);
64 // Mat b(n, n);
65 // Mat c = a * b;
66 // Mat d = a + b;
67 // Mat e = a - b;
68 // Mat f = a.pow(k);
69 // a.a[i][j] = x;

```

### 3.13 Mobius

```

1 int mob[N];
2 void mobius() {

```

```

3  mob[1] = 1;
4  for (int i = 2; i < N; i++){
5      mob[i]--;
6      for (int j = i + i; j < N; j += i) {
7          mob[j] -= mob[i];
8      }
9  }
10 }
```

### 3.14 Phi

```

1  void phi_1_to_n(int n) {
2      vector<int> phi(n + 1);
3      phi[0] = 0;
4      phi[1] = 1;
5      for (int i = 2; i <= n; i++)
6          phi[i] = i - 1;
7
8      for (int i = 2; i <= n; i++)
9          for (int j = 2 * i; j <= n; j += i)
10             phi[j] -= phi[i];
11 }
```

### 3.15 Prefix sum Phi

```

1  vector<ll> sieve(kMaxV + 1,0);
2  vector<ll> phi(kMaxV + 1,0);
3
4  void primes()
5  {
6      phi[1]=1;
7      vector<ll> pr;
8      for(int i=2;i<kMaxV;i++){
9          if(sieve[i]==0){
10             sieve[i]=i;
11             pr.pb(i);
12             phi[i]=i-1;
13         }
14         for(auto p:pr){
15             if(p>sieve[i] || i*p>kMaxV)break;
16             sieve[i*p]=p;
17             phi[i*p]=(p==sieve[i]?p:p-1)*phi[i];
18         }
19     }
```

```

20  for(int i=1;i<kMaxV;i++){
21      phi[i]+=phi[i-1];
22      phi[i]%=MOD;
23  }
24 }
25
26 map<ll,ll> m;
27 ll PHI(ll a){
28     if(a<kMaxV)return phi[a];
29     if(m.count(a))return m[a];
30     // if(a<3)return 1;
31     m[a]=(((a%MOD)*((a+1)%MOD))%MOD)*inverse(2));
32     m[a]%=MOD;
33     long long i=2;
34     while(i<=a){
35         long long j=a/i;
36         j=a/j;
37         m[a]+=MOD;
38         m[a]-=((j-i+1)*PHI(a/i))%MOD;
39         m[a]%=MOD;
40         i=j+1;
41     }
42     m[a]%=MOD;
43     return m[a];
44 }
```

### 3.16 Identities

$$C_n = \frac{2(2n-1)}{n+1} C_{n-1}$$

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

$$\sigma(n) = O(\log(\log(n))) \text{ (number of divisors of } n)$$

$$F_{2n+1} = F_n^2 + F_{n+1}^2$$

$$F_{2n} = F_{n+1}^2 - F_{n-1}^2$$

$$\sum_{i=1}^n F_i = F_{n+2} - 1$$

$$F_{n+i}F_{n+j} - F_nF_{n+i+j} = (-1)^n F_i F_j$$

$$\text{(Möbius Inv. Formula)} \mu(p^k) = [k=0] - [k=1] \text{ Let } g(n) = \sum_{d|n} f(d), \text{ then}$$

$$f(n) = \sum_{d|n} g(d) \mu\left(\frac{n}{d}\right). \text{ (Dirichlet Convolution) Let } f, g \text{ be arithmetic functions, then}$$

$$(f * g)(n) = \sum_{d|n} f(d)g\left(\frac{n}{d}\right). \text{ If } f, g \text{ are multiplicative, then so is } f * g.$$

$$n = \sum_{d|n} \phi(d)$$

$$\text{Lucas' Theorem: } \binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p} \text{ where } m = \sum_{i=0}^k m_i p^i \text{ and}$$

$$n = \sum_{i=0}^k n_i p^i.$$



### 3.17 Burnside's Lemma

Dado un grupo  $G$  de permutaciones y un conjunto  $X$  de  $n$  elementos, el número de órbitas de  $X$  bajo la acción de  $G$  es igual al promedio del número de puntos fijos de las permutaciones en  $G$ .

Formalmente, el número de órbitas es  $\frac{1}{|G|} \sum_{g \in G} f(g)$  donde  $f(g)$  es el número de puntos fijos de  $g$ .

Ejemplo: Dado un collar con  $n$  cuentas y 2 colores, el número de collares diferentes que se pueden formar es  $\frac{1}{n} \sum_{i=0}^n f(i)$  donde  $f(i)$  es el número de collares que quedan fijos bajo una rotación de  $i$  posiciones.

Para contar el número de collares que quedan fijos bajo una rotación de  $i$  posiciones, se puede usar la fórmula  $f(i) = 2^{\gcd(i,n)}$ .

Para un collar de  $n$  cuentas y  $k$  colores, el número de collares diferentes que se pueden formar es  $\frac{1}{n} \sum_{i=0}^n k^{\gcd(i,n)}$

Ejemplo: Dado un cubo con 6 caras y  $k$  colores, el número de cubos diferentes que se pueden formar es  $\frac{1}{24} \sum_{i=0}^{24} f(i)$  donde  $f(i)$  es el número de cubos que quedan fijos bajo una rotación de  $i$  posiciones. Esta formula es igual a  $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$

### 3.18 Recursion

Sea  $f(n) = \sum_{i=1}^k a_i f(n-i)$  entonces podemos considerar la matriz:

$$\begin{bmatrix} f(n) \\ f(n-1) \\ \vdots \\ f(n-k+1) \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \\ \vdots \\ f(n-k) \end{bmatrix}$$

De aqui podemos calcular  $f(n)$  con exponenciación de matrices.

$$\begin{bmatrix} f(n) \\ f(n-1) \\ \vdots \\ f(n-k+1) \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}^{n-k} \begin{bmatrix} f(k) \\ f(k-1) \\ \vdots \\ f(1) \end{bmatrix}$$

### 3.19 Theorems

**Koeing's Theorem:** La cardinalidad del emparejamiento maximo de una grafica bipartita es igual al minimum vertex cover.

**Hall's Theorem:** Una grafica bipartita  $G$  tiene un emparejamiento que cubre todos los nodos de  $G$  si y solo si para todo subconjunto  $S$  de nodos de  $G$ , el número de vecinos de  $S$  es mayor o igual a  $|S|$ .

## 4 Game Theory

### 4.1 Sprague-Grundy theorem

<https://codeforces.com/blog/entry/66040> Dado un juego con pilas  $p_1, p_2, \dots, p_n$  sea  $g(p)$  el nimer de la pila  $p$ , entonces el nimber del juego es  $g(p_1) \oplus g(p_2) \oplus \dots \oplus g(p_n)$ . Para calcular el nimber de una pila, se puede usar la fórmula  $g(r) = \text{mex}(\{g(r_1), g(r_2), \dots, g(r_k)\})$  donde  $r_1, r_2, \dots, r_k$  son los posibles estados a los que se puede llegar desde  $r$  y  $g(r) = 0$  si  $r$  es un estado perdedor.

## 5 Strings

### 5.1 Manacher

```
1 // Number of palindromes centered at each position
2
3 vector<int> manacher_odd(string s)
4 {
5     int n = s.size();
6     s = "$" + s + "^";
7     vector<int> p(n + 2);
8     int l = 1, r = 1;
9     for (int i = 1; i <= n; i++)
10     {
11         p[i] = max(0, min(r - i, p[l + (r - i)]));
12         while (s[i - p[i]] == s[i + p[i]])
13         {
14             p[i]++;
15         }
16         if (i + p[i] > r)
17         {
18             l = i - p[i], r = i + p[i];
19         }
20     }
21     return vector<int>(begin(p) + 1, end(p) - 1);
22 }
23 vector<int> manacher(string s)
24 {
25     string t;
26     for (auto c : s)
27     {
28         t += string("#") + c;
29     }
```

```

30 auto res = manacher_odd(t + "#");
31 return vector<int>(begin(res) + 1, end(res) - 1);
32 }
33
34 // usage
35 // vector<int> p = manacher("abacaba");
36 // this will return {2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2}
37 // vector<int> p = manacher("abaaba");
38 // this will return {2, 1, 4, 1, 2, 7, 2, 1, 4, 1, 2}

```

## 5.2 Suffix Array $O(n \log(n))$

```

1 using i64 = long long;
2
3 struct SuffixArray
4 {
5     int n;
6     std::vector<int> sa, rk, lc;
7     SuffixArray(const std::string &s)
8     {
9         n = s.length();
10        sa.resize(n);
11        lc.resize(n - 1);
12        rk.resize(n);
13        std::iota(sa.begin(), sa.end(), 0);
14        std::sort(sa.begin(), sa.end(), [&](int a, int b)
15            { return s[a] < s[b]; });
16        rk[sa[0]] = 0;
17        for (int i = 1; i < n; ++i)
18            rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
19        int k = 1;
20        std::vector<int> tmp, cnt(n);
21        tmp.reserve(n);
22        while (rk[sa[n - 1]] < n - 1)
23        {
24            tmp.clear();
25            for (int i = 0; i < k; ++i)
26                tmp.push_back(n - k + i);
27            for (auto i : sa)
28                if (i >= k)
29                    tmp.push_back(i - k);
30            std::fill(cnt.begin(), cnt.end(), 0);
31            for (int i = 0; i < n; ++i)

```

```

32            ++cnt[rk[i]];
33            for (int i = 1; i < n; ++i)
34                cnt[i] += cnt[i - 1];
35            for (int i = n - 1; i >= 0; --i)
36                sa[--cnt[rk[tmp[i]]]] = tmp[i];
37            std::swap(rk, tmp);
38            rk[sa[0]] = 0;
39            for (int i = 1; i < n; ++i)
40                rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i - 1] + k == n || tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
41            k *= 2;
42        }
43        for (int i = 0, j = 0; i < n; ++i)
44        {
45            if (rk[i] == 0)
46            {
47                j = 0;
48            }
49            else
50            {
51                for (j -= j > 0; i + j < n && sa[rk[i] - 1] + j < n && s[i + j] == s[sa[rk[i] - 1] + j];)
52                    ++j;
53                lc[rk[i] - 1] = j;
54            }
55        }
56    }
57 };

```

## 5.3 Z Function $O(n)$

```

1 // Mayor x tal que el prefijo de s de tamaño x es igual al prefijo
2 //del sufijo que empieza en la posición i y tiene tamaño x
3
4 vector<int> z_function(string s) {
5     int n = s.size();
6     vector<int> z(n);
7     int l = 0, r = 0;
8     for(int i = 1; i < n; i++) {
9         if(i < r) {
10             z[i] = min(r - i, z[i - l]);
11         }
12         while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {

```

```

13     z[i]++;
14 }
15 if(i + z[i] > r) {
16     l = i;
17     r = i + z[i];
18 }
19 }
20 return z;
21 }
22
23 // usage
24 // vector<int> z = z_function("abacaba");
25 // this will return {0, 0, 1, 0, 3, 0, 1}
26 // vector<int> z = z_function("aaaaa");
27 // this will return {0, 4, 3, 2, 1}
28 // vector<int> z = z_function("aaabaab");
29 // this will return {0, 2, 1, 0, 2, 1, 0}

```

## 5.4 KMP

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 const int N = 3e5 + 9;
5
6 // returns the longest proper prefix array of pattern p
7 // where lps[i]=longest proper prefix which is also suffix of p[0...i]
8 vector<int> build_lps(string p) {
9     int sz = p.size();
10    vector<int> lps;
11    lps.assign(sz + 1, 0);
12    int j = 0;
13    lps[0] = 0;
14    for(int i = 1; i < sz; i++) {
15        while(j >= 0 && p[i] != p[j]) {
16            if(j >= 1) j = lps[j - 1];
17            else j = -1;
18        }
19        j++;
20        lps[i] = j;
21    }
22    return lps;
23 }

```

```

24 vector<int>ans;
25 // returns matches in vector ans in 0-indexed
26 void kmp(vector<int> lps, string s, string p) {
27     int psz = p.size(), sz = s.size();
28     int j = 0;
29     for(int i = 0; i < sz; i++) {
30         while(j >= 0 && p[j] != s[i])
31             if(j >= 1) j = lps[j - 1];
32             else j = -1;
33         j++;
34         if(j == psz) {
35             j = lps[j - 1];
36             // pattern found in string s at position i-psz+1
37             ans.push_back(i - psz + 1);
38         }
39         // after each loop we have j=longest common suffix of s[0..i] which
40         // is also prefix of p
41     }
42 }
43
44 int main() {
45     int i, j, k, n, m, t;
46     cin >> t;
47     while(t--) {
48         string s, p;
49         cin >> s >> p;
50         vector<int>lps = build_lps(p);
51         kmp(lps, s, p);
52         if(ans.empty()) cout << "Not_Found\n";
53         else {
54             cout << ans.size() << endl;
55             for(auto x : ans) cout << x << ' ';
56             cout << endl;
57         }
58         ans.clear();
59         cout << endl;
60     }
61     return 0;
62 }

```

## 6 DP Optimization

### 6.1 Convex Hull Trick

```

1 const ll is_query = -(1LL << 62);
2 struct line
3 {
4     ll m, b;
5     mutable function<const line *(>> succ;
6     bool operator<(const line &rhs) const
7     {
8         if (rhs.b != is_query)
9             return m < rhs.m;
10        const line *s = succ();
11        if (!s)
12            return 0;
13        ll x = rhs.m;
14        return b - s->b < (s->m - m) * x;
15    }
16 };
17
18 struct dynamic_hull : public multiset<line>
19 { // will maintain upper hull for maximum
20     const ll inf = LLONG_MAX;
21     bool bad(iterator y)
22     {
23         auto z = next(y);
24         if (y == begin())
25         {
26             if (z == end())
27                 return 0;
28             return y->m == z->m && y->b <= z->b;
29         }
30         auto x = prev(y);
31         if (z == end())
32             return y->m == x->m && y->b <= x->b;
33
34         /* compare two lines by slope, make sure denominator is not 0 */
35         ll v1 = (x->b - y->b);
36         if (y->m == x->m)
37             v1 = x->b > y->b ? inf : -inf;
38         else
39             v1 /= (y->m - x->m);

```

```

40         ll v2 = (y->b - z->b);
41         if (z->m == y->m)
42             v2 = y->b > z->b ? inf : -inf;
43         else
44             v2 /= (z->m - y->m);
45         return v1 >= v2;
46     }
47     void insert_line(ll m, ll b)
48     {
49         auto y = insert({m, b});
50         y->succ = [=]
51         { return next(y) == end() ? 0 : &*next(y); };
52         if (bad(y))
53         {
54             erase(y);
55             return;
56         }
57         while (next(y) != end() && bad(next(y)))
58             erase(next(y));
59         while (y != begin() && bad(prev(y)))
60             erase(prev(y));
61     }
62     ll eval(ll x)
63     {
64         auto l = *lower_bound((line){x, is_query});
65         return l.m * x + l.b;
66     }
67 }; // gives max, for min insert_line(*-1, *-1) and eval()*-1

```

### 6.2 Divide and Conquer DP

```

1 int m, n;
2 vector<long long> dp_before, dp_cur;
3
4 long long C(int i, int j);
5
6 // compute dp_cur[l], ... dp_cur[r] (inclusive)
7 void compute(int l, int r, int optl, int optl) {
8     if (l > r)
9         return;
10
11     int mid = (l + r) >> 1;
12     pair<long long, int> best = {LLONG_MAX, -1};

```

```

13
14     for (int k = optl; k <= min(mid, optr); k++) {
15         best = min(best, {(k ? dp_before[k - 1] : 0) + C(k, mid), k});
16     }
17
18     dp_cur[mid] = best.first;
19     int opt = best.second;
20
21     compute(l, mid - 1, optl, opt);
22     compute(mid + 1, r, opt, optr);
23 }
24
25 long long solve() {
26     dp_before.assign(n,0);
27     dp_cur.assign(n,0);
28
29     for (int i = 0; i < n; i++)
30         dp_before[i] = C(0, i);
31
32     for (int i = 1; i < m; i++) {
33         compute(0, n - 1, 0, n - 1);
34         dp_before = dp_cur;
35     }
36
37     return dp_before[n - 1];
38 }

```

## 7 Other

### 7.1 Binary Search

```

1 int l = 1, r = n + 1;
2
3 while(r - l > 1){
4     int mid = l + (r - l) / 2;
5     bool check = true;
6
7     //proceso
8
9     if(check) l = mid;
10    else r = mid;
11 }

```

### 7.2 BinSearch with doubles

```

1 for(int i = 0; i < 200 && r - l > 1e-9; i++){
2     double mid = l + (r - l) / (double)2;
3     if(check(mid, obj, scores)) l = mid;
4     else r = mid;
5 }

```

### 7.3 Ternary Search

```

1 // ternary search no funciona si existe
2 // a[i]==a[i+1];
3 double ternary_search(double l, double r) {
4     double eps = 1e-9;           //set the error limit here
5     while (r - l > eps) {
6         double m1 = l + (r - l) / 3;
7         double m2 = r - (r - l) / 3;
8         double f1 = f(m1);       //evaluates the function at m1
9         double f2 = f(m2);       //evaluates the function at m2
10        if (f1 < f2)
11            l = m1;
12        else
13            r = m2;
14    }
15    return f(l);                  //return the maximum of f(x) in [l,
16    r]
17 }

```

### 7.4 LIS DP

```

1 vector<int> v;
2 v.pb(a[0]);
3 for(int i=1;i<n;i++){
4     if(a[i]>v[v.size()-1]){
5         v.pb(a[i]);
6     }
7     else{
8         int b=lower_bound(v.begin(),v.end(),a[i])-v.begin();
9         v[b]=a[i];
10    }
11 }

```

### 7.5 Random numbers

```

1 | mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
2 | mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

```

## 7.6 XOR basis

```

1 | vector<ll> basis;
2 | void add(ll x)
3 | {
4 |     for (int i = 0; i < basis.size(); i++)
5 |     {
6 |         x = min(x, x ^ basis[i]);
7 |     }
8 |     if (x != 0)
9 |     {
10 |         basis.pb(x);
11 |     }
12 | }

```

## 7.7 Bitsets

```

1 | // count the number of set bits in an integer
2 | #pragma GCC target("popcnt")
3 | (int) __builtin_popcount(x);
4 | (int) __builtin_popcountll(x);
5 |
6 | // declare bitset
7 | bitset<64> b;
8 |
9 | // count set bits in bitset
10 | b.count();

```

## 7.8 All permutations

```

1 | sort(v.begin(), v.end());
2 | while(next_permutation(v.begin(), v.end())){
3 |     for(auto u:v){
4 |         cout<<u<<" ";
5 |     }
6 |     cout<<endl;
7 | }
8 |
9 | string s="asdfassd";
10 | sort(s.begin(), s.end());
11 | while(next_permutation(s.begin(), s.end())){

```

```

12 |     cout<<s<<endl;
13 | }

```

## 7.9 Print doubles

```

1 | cout<<fixed<<setprecision(10);

```

# 8 Flows

## 8.1 Maximum Flow $O(V \cdot E^2)$

```

1 | long long max_flow(vector<vector<int>> adj, vector<vector<long long>>
   |     capacity,
2 |                     int source, int sink)
3 | {
4 |     int n = adj.size();
5 |     vector<int> parent(n, -1);
6 |     // Find a way from the source to sink on a path with non-negative
   |     capacities
7 |     auto reachable = [&]() -> bool
8 |     {
9 |         queue<int> q;
10 |         q.push(source);
11 |         while (!q.empty())
12 |         {
13 |             int node = q.front();
14 |             q.pop();
15 |             for (int son : adj[node])
16 |             {
17 |                 long long w = capacity[node][son];
18 |                 if (w <= 0 || parent[son] != -1)
19 |                     continue;
20 |                 parent[son] = node;
21 |                 q.push(son);
22 |             }
23 |         }
24 |         return parent[sink] != -1;
25 |     };
26 |
27 |     long long flow = 0;
28 |     // While there is a way from source to sink with non-negative
   |     capacities
29 |     while (reachable())

```

```

30 {
31     int node = sink;
32     // The minimum capacity on the path from source to sink
33     long long curr_flow = LLONG_MAX;
34     while (node != source)
35     {
36         curr_flow = min(curr_flow, capacity[parent[node]][node]);
37         node = parent[node];
38     }
39     node = sink;
40     while (node != source)
41     {
42         // Subtract the capacity from capacity edges
43         capacity[parent[node]][node] -= curr_flow;
44         // Add the current flow to flow backedges
45         capacity[node][parent[node]] += curr_flow;
46         node = parent[node];
47     }
48     flow += curr_flow;
49     fill(parent.begin(), parent.end(), -1);
50 }
51
52 return flow;
53 }
54
55 //vector<vector<long long>> capacity(n, vector<long long>(n));
56 //vector<vector<int>> adj(n);
57 //adj[a].push_back(b);
58 //adj[b].push_back(a);
59 //capacity[a][b] += c;

```

## 8.2 Maximum Flow $O(V \cdot V \cdot E)$

```

1 const int inf = 1000000000;
2
3 int n;
4 vector<vector<int>> capacity, flow;
5 vector<int> height, excess, seen;
6 queue<int> excess_vertices;
7
8 void push(int u, int v) {

```

```

9     int d = min(excess[u], capacity[u][v] - flow[u][v]);
10    flow[u][v] += d;
11    flow[v][u] -= d;
12    excess[u] -= d;
13    excess[v] += d;
14    if (d && excess[v] == d)
15        excess_vertices.push(v);
16 }
17
18 void relabel(int u) {
19     int d = inf;
20     for (int i = 0; i < n; i++) {
21         if (capacity[u][i] - flow[u][i] > 0)
22             d = min(d, height[i]);
23     }
24     if (d < inf)
25         height[u] = d + 1;
26 }
27
28 void discharge(int u) {
29     while (excess[u] > 0) {
30         if (seen[u] < n) {
31             int v = seen[u];
32             if (capacity[u][v] - flow[u][v] > 0 && height[u] > height[v])
33                 push(u, v);
34             else
35                 seen[u]++;
36         } else {
37             relabel(u);
38             seen[u] = 0;
39         }
40     }
41 }
42
43 int max_flow(int s, int t) {
44     height.assign(n, 0);
45     height[s] = n;
46     flow.assign(n, vector<int>(n, 0));
47     excess.assign(n, 0);
48     excess[s] = inf;
49     for (int i = 0; i < n; i++) {
50         if (i != s)
51             push(s, i);

```

```

52 }
53 seen.assign(n, 0);
54
55 while (!excess_vertices.empty()) {
56     int u = excess_vertices.front();
57     excess_vertices.pop();
58     if (u != s && u != t)
59         discharge(u);
60 }
61
62 int max_flow = 0;
63 for (int i = 0; i < n; i++)
64     max_flow += flow[i][t];
65 return max_flow;
66 }

```

### 8.3 Maximum matching $O(E\sqrt{V})$

```

1 // maximum matching in bipartite graph
2 vector<int> match, dist;
3 vector<vector<int>> g;
4 int n, m, k;
5 bool bfs()
6 {
7     queue<int> q;
8     // The alternating path starts with unmatched nodes
9     for (int node = 1; node <= n; node++)
10     {
11         if (!match[node])
12         {
13             q.push(node);
14             dist[node] = 0;
15         }
16         else
17         {
18             dist[node] = INF;
19         }
20     }
21
22     dist[0] = INF;
23
24     while (!q.empty())
25     {

```

```

26     int node = q.front();
27     q.pop();
28     if (dist[node] >= dist[0])
29     {
30         continue;
31     }
32     for (int son : g[node])
33     {
34         // If the match of son is matched
35         if (dist[match[son]] == INF)
36         {
37             dist[match[son]] = dist[node] + 1;
38             q.push(match[son]);
39         }
40     }
41 }
42 // Returns true if an alternating path has been found
43 return dist[0] != INF;
44 }
45
46 // Returns true if an augmenting path has been found starting from
47 // vertex node
48 bool dfs(int node)
49 {
50     if (node == 0)
51     {
52         return true;
53     }
54     for (int son : g[node])
55     {
56         if (dist[match[son]] == dist[node] + 1 && dfs(match[son]))
57         {
58             match[node] = son;
59             match[son] = node;
60             return true;
61         }
62     }
63     dist[node] = INF;
64     return false;
65 }
66
67 int hopcroft_karp()
68 {

```



```

68 int cnt = 0;
69 // While there is an alternating path
70 while (bfs())
71 {
72     for (int node = 1; node <= n; node++)
73     {
74         // If node is unmatched but we can match it using an augmenting
           path
75         if (!match[node] && dfs(node))
76         {
77             cnt++;
78         }
79     }
80 }
81 return cnt;
82 }
83 // usage
84 // n numero de puntos en la izquierda
85 // m numero de puntos en la derecha
86 // las aristas se guardan en g
87 // los puntos estan 1 indexados
88 // el punto 1 de m es el punto n+1 de g
89 // hopcroft_karp() devuelve el tamaño del máximo matching
90 // match contiene el match de cada punto
91 // si match de i es 0, entonces i no está matcheado
92 //
93 // https://judge.yosupo.jp/submission/247277

```

#### 8.4 Minimum Cost Maximum Flow

```

1 // dado un acomodo de flujos con costos
2 // devuelve el costo mínimo para un flujo especificado
3
4 struct Edge
5 {
6     int from, to, capacity, cost;
7     Edge(int _from, int _to, int _capacity, int _cost)
8     {
9         from = _from;
10        to = _to;
11        capacity = _capacity;
12        cost = _cost;
13    }

```

```

14 };
15
16 vector<vector<int>> adj, cost, capacity;
17
18 const int INF = 1e9;
19
20 void shortest_paths(int n, int v0, vector<int> &d, vector<int> &p)
21 {
22     d.assign(n, INF);
23     d[v0] = 0;
24     vector<bool> inq(n, false);
25     queue<int> q;
26     q.push(v0);
27     p.assign(n, -1);
28
29     while (!q.empty())
30     {
31         int u = q.front();
32         q.pop();
33         inq[u] = false;
34         for (int v : adj[u])
35         {
36             if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v])
37             {
38                 d[v] = d[u] + cost[u][v];
39                 p[v] = u;
40                 if (!inq[v])
41                 {
42                     inq[v] = true;
43                     q.push(v);
44                 }
45             }
46         }
47     }
48 }
49
50 int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t)
51 {
52     adj.assign(N, vector<int>());
53     cost.assign(N, vector<int>(N, 0));
54     capacity.assign(N, vector<int>(N, 0));
55     for (Edge e : edges)
56     {

```

```

57     adj[e.from].push_back(e.to);
58     adj[e.to].push_back(e.from);
59     cost[e.from][e.to] = e.cost;
60     cost[e.to][e.from] = -e.cost;
61     capacity[e.from][e.to] = e.capacity;
62 }
63
64 int flow = 0;
65 int cost = 0;
66 vector<int> d, p;
67 while (flow < K)
68 {
69     shortest_paths(N, s, d, p);
70     if (d[t] == INF)
71         break;
72
73     // find max flow on that path
74     int f = K - flow;
75     int cur = t;
76     while (cur != s)
77     {
78         f = min(f, capacity[p[cur]][cur]);
79         cur = p[cur];
80     }
81
82     // apply flow
83     flow += f;
84     cost += f * d[t];
85     cur = t;
86     while (cur != s)
87     {
88         capacity[p[cur]][cur] -= f;
89         capacity[cur][p[cur]] += f;
90         cur = p[cur];
91     }
92 }
93
94 if (flow < K)
95     return -1;
96 else
97     return cost;
98 }

```

## 8.5 Dinic

```

1 // Si en el grafo todos los vertices distintos
2 // de s y t cumplen que solo tienen una arista
3 // de entrada o una de salida la y dicha arista
4 // tiene capacidad 1 entonces la complejidad es
5 //  $O(E \sqrt{v})$ 
6
7 // si todas las aristas tienen capacidad 1
8 // el algoritmo tiene complejidad  $O(E \sqrt{E})$ 
9
10 struct FlowEdge {
11     int v, u;
12     long long cap, flow = 0;
13     FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
14 };
15
16 struct Dinic {
17     const long long flow_inf = 1e18;
18     vector<FlowEdge> edges;
19     vector<vector<int>> adj;
20     int n, m = 0;
21     int s, t;
22     vector<int> level, ptr;
23     queue<int> q;
24
25     Dinic(int n, int s, int t) : n(n), s(s), t(t) {
26         adj.resize(n);
27         level.resize(n);
28         ptr.resize(n);
29     }
30
31     void add_edge(int v, int u, long long cap) {
32         edges.emplace_back(v, u, cap);
33         edges.emplace_back(u, v, 0);
34         adj[v].push_back(m);
35         adj[u].push_back(m + 1);
36         m += 2;
37     }
38
39     bool bfs() {
40         while (!q.empty()) {
41             int v = q.front();

```

```

42     q.pop();
43     for (int id : adj[v]) {
44         if (edges[id].cap - edges[id].flow < 1)
45             continue;
46         if (level[edges[id].u] != -1)
47             continue;
48         level[edges[id].u] = level[v] + 1;
49         q.push(edges[id].u);
50     }
51 }
52 return level[t] != -1;
53 }
54
55 long long dfs(int v, long long pushed) {
56     if (pushed == 0)
57         return 0;
58     if (v == t)
59         return pushed;
60     for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
61         int id = adj[v][cid];
62         int u = edges[id].u;
63         if (level[v] + 1 != level[u] || edges[id].cap - edges[id].
64             flow < 1)
65             continue;
66         long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].
67             flow));
68         if (tr == 0)
69             continue;
70         edges[id].flow += tr;
71         edges[id ^ 1].flow -= tr;
72         return tr;
73     }
74     return 0;
75 }
76
77 long long flow() {
78     long long f = 0;
79     while (true) {
80         fill(level.begin(), level.end(), -1);
81         level[s] = 0;
82         q.push(s);
83         if (!bfs())
84             break;

```

```

83         fill(ptr.begin(), ptr.end(), 0);
84         while (long long pushed = dfs(s, flow_inf)) {
85             f += pushed;
86         }
87     }
88     return f;
89 }
90 };

```

## 9 Geometry

### 9.1 Point struct

```

1 struct Point{
2     ll x,y;
3
4     Point () : x(), y() {}
5     Point (ll _x, ll _y) : x(_x), y(_y){}
6
7     Point operator + (const Point &a) const {
8         return Point(x+a.x,y+a.y);
9     }
10
11     Point operator - (const Point &a) const {
12         return Point(x-a.x, y-a.y);
13     }
14
15     // dot product
16     // positivo si el angulo entre los vectores es agudo
17     // 0 si son perpendiculares
18     // negativo si el angulo es obtuso
19     ll operator % (const Point &a) const {
20         return x*a.x+y*a.y;
21     }
22
23     // cross product
24     // positivo si el segundo esta en sentido antihorario
25     // 0 si el angulo es 180
26     // negativo si el segundo esta en sentido horario
27     ll operator * (const Point &a) const {
28         return x*a.y - y * a.x;
29     }
30 };

```

## 9.2 Sort points

```
1 // This comparator sorts the points clockwise
2 // starting from the first quarter
3
4 bool getQ(Point a){
5     if(a.y!=0){
6         if(a.y>0)return 0;
7         return 1;
8     }
9     if(a.x>0)return 0;
10    return 1;
11 }
12 bool comp(Point a, Point b){
13     if(getQ(a)!=getQ(b))return getQ(a)<getQ(b);
14     return a*b>0;
15 }
```

# 10 Compile

## 10.1 Template

```
1 #include <bits/stdc++.h>
2 #pragma GCC optimize("O3,unroll-loops")
3 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
4 using namespace std;
5 #define pb push_back
6 #define ll long long
7 #define s second
8 #define f first
9 #define MOD 1000000007
10 #define INF 1000000000000000
11
12 void solve(){
13
14 }
15
16 int main() {
17     ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);
18     int t;cin>>t;for(int T=0;T<t;T++)
19         solve();
20 }
```

## 10.2 Compile

```
1 g++-13 nombre.cpp -o nombre (compilar)
2 ./nombre (ejecutar)
```