

Competitive Programming Algorithms and Topics

Ubiratan Neto

1	Template	1
1.1	Template Code	1
2	Data Structures	2
2.1	Dynamic Segment Tree	2
2.2	Segment Tree	2
2.3	Fenwick Tree	3
2.4	Trie	4
2.5	STL Ordered Set	4
2.6	Convex Hull Trick	4
2.7	Lichao Segment Tree - Convex Hull Trick	5
2.8	Lichao Segment Tree - Convex Hull Trick - Double Type	5
3	Uncategorized	6
3.1	Coordinate Compression	6
3.2	Longest Increasing Subsequence	6
3.3	Inversion Count - Merge Sort	6
3.4	Mo's Decomposition	7
4	Geometry	7
4.1	2D Structures	7
4.2	2D Geometry Functions	9
4.3	Convex Hull - Monotone Chain Algorithm	13
5	Graphs	14
5.1	Dynammic Connectivity - connected(u,v) query	14
5.2	Bellman Ford Shortest Path	15
5.3	Eulerian Circuits/Paths	16
5.4	Kosaraju SCC	16
5.5	Centroid Decomposition	17
5.6	Floyd Warshall Shortest Path	17
5.7	Tarjan's Bridge/Articulations Algorithm	18
5.8	Max Flow Dinic's Algorithm	18
5.9	HLD	19
5.10	LCA	20

6	Math and Number Theory	21
6.1	Diophantine Equations + CRT	21
6.2	Discrete Logarithm - Shanks Baby-Step Giant-Step	21
6.3	Binomial Coefficient DP	22
6.4	Erathostenes Sieve + Logn Prime Factorization	22
6.5	Segmented Sieve	22
6.6	Matrix Exponentiation	23
6.7	Fast Fourier Transform - Recursive and Iterative	23
7	String Algorithms	24
7.1	KMP Failure Function + String Matching	24
7.2	Z-Function	25
7.3	Suffix Array + Linear Sort	25
7.4	Rolling Hash	26

1. Template

1.1. Template Code

```

1 #include <bits/stdc++.h>
2
3 #define int long long
4 #define double long double
5 #define ff first
6 #define ss second
7 #define endl '\n'
8 #define ii pair<int, int>
9 #define DESYNC ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0)
10 #define pb push_back
11 #define vi vector<int>
12 #define vii vector<ii>
13 #define EPS 1e-9
14 #define INF 1e18
15 #define ROOT 1
16 #define M 1000000007
17 const double PI = acos(-1);
18
19 using namespace std;
20
21 inline int mod(int n, int m){ int ret = n%m; if(ret < 0) ret += m; return ret; }
22
23 int gcd(int a, int b){
24     if(a == 0 || b == 0) return 0;
25     else return abs(__gcd(a,b));
26 }
27
28 int32_t main(){
29     DESYNC;
30
31 }
```

2. Data Structures

2.1. Dynamic Segment Tree

```

1 namespace DynamicSegmentTree{
2
3     struct node {
4         node *left, *right;
5         //attributes of node
6         node() {
7             //initialize attributes
8             left = NULL;
9             right = NULL;
10        }
11    };
12
13    void combine(node *ans, node *left, node *right){
14        //combine operation
15    }
16
17    void propagate(node * root, int l, int r){
18        //check if exists lazy
19
20        //apply lazy on node
21
22        //propagate
23        if(!root->left) root->left = new node();
24        if(!root->right) root->right = new node();
25
26        if(l != r){
27            //propagate operation
28        }
29
30        //reset lazy
31    }
32
33    void build(node *root, int l, int r){
34        if(l == r){
35            //leaf operation
36            return;
37        }
38        int m = (l+r) >> 1;
39        if(!root->left) root->left = new node();
40        if(!root->right) root->right = new node();
41        build(root->left, l, m);
42        build(root->right, m+1, r);
43        combine(root, root->left, root->right);
44    }
45
46    void update(node *root, int l, int r, int a, int b, int val){
47        propagate(root, l, r);
48        if(l == a && r == b){
49            //do lazy operation
50            return;
51        }
52        int m = (l+r) >> 1;
53        if(!root->left) root->left = new node();
54        if(!root->right) root->right = new node();
55        if(b <= m) update(root->left, l, m, a, b, val);
56        else if(m < a) update(root->right, m+1, r, a, b, val);
57        else {
58            update(root->left, l, m, a, m, val);
59            update(root->right, m+1, r, m+1, b, val);
60        }
61        propagate(root, l, r);

```

```

62        propagate(root->left, l, m);
63        propagate(root->right, m+1, r);
64        combine(root, root->left, root->right);
65    }
66
67    node* query(node *root, int l, int r, int a, int b){
68        propagate(root, l, r);
69        if(l == a && r == b){
70            return root;
71        }
72        int m = (l+r) >> 1;
73        if(!root->left) root->left = new node();
74        if(!root->right) root->right = new node();
75        if(b <= m) return query(root->left, l, m, a, b);
76        else if(m < a) return query(root->right, m+1, r, a, b);
77        node *left = query(root->left, l, m, a, m);
78        node *right = query(root->right, m+1, r, m+1, b);
79        node *ans = new node();
80        combine(ans, left, right);
81        return ans;
82    }
83
84 }

```

2.2. Segment Tree

```

1 namespace SegmentTree{
2
3     struct node{
4         //attributes of node
5         int lazy = 0;
6         node() {}
7     };
8
9     struct Tree{
10        vector<node> st;
11        Tree() {}
12
13        Tree(int n){
14            st.resize(4*n);
15        }
16
17        node combine(node a, node b){
18            node res;
19            //combine operations
20            return res;
21        }
22
23        void propagate(int cur, int l, int r){
24            //return if there is no update
25            //update tree using lazy node
26            if(l != r){
27                //propagate for left and right child
28            }
29            //reset lazy node
30        }
31
32        void build(int cur, int l, int r){
33            if(l == r){
34                //leaf operation
35                return;
36            }
37
38            int m = (l+r)>>1;

```

```

39     build(2*cur, l, m);
40     build(2*cur + 1, m+1, r);
41     st[cur] = combine(st[2*cur], st[2*cur+1]);
42
43 }
44
45 void range_update(int cur, int l, int r, int a, int b, long long val){
46     propagate(cur, l, r);
47     if(l == a && r == b){
48         //lazy operation using val
49         return;
50     }
51
52     int m = (l+r)/2;
53
54     if(b <= m) range_update(2*cur, l, m, a, b, val);
55     else if(m < a) range_update(2*cur+1, m+1, r, a, b, val);
56     else {
57         range_update(2*cur, l, m, a, m, val);
58         range_update(2*cur+1, m+1, r, m+1, b, val);
59     }
60
61     propagate(cur, l, r);
62     propagate(2*cur, l, m);
63     propagate(2*cur+1, m+1, r);
64     st[cur] = combine(st[2*cur], st[2*cur+1]);
65 }
66
67 node query(int cur, int l, int r, int a, int b){
68     propagate(cur, l, r);
69     if(l == a && r == b) return st[cur];
70
71     int m = (l+r)/2;
72     if(b <= m) return query(2*cur, l, m, a, b);
73     else if(m < a) return query(2*cur+1, m+1, r, a, b);
74     else {
75         node left = query(2*cur, l, m, a, m);
76         node right = query(2*cur+1, m+1, r, m+1, b);
77         node ans = combine(left, right);
78         return ans;
79     }
80 }
81 };
82
83 }

```

2.3. Fenwick Tree

```

1 struct BIT {
2
3     vector<int> bit;
4
5     BIT() {}
6
7     int n;
8
9     BIT(int n) {
10         this->n = n;
11         bit.resize(n+1);
12     }
13
14     void update(int idx, int val){
15         for(int i = idx; i <= n; i += i&-i){
16             bit[i] += val;
17         }
18     }
19
20     int prefix_query(int idx){
21         int ans = 0;
22         for(int i = idx; i > 0; i -= i&-i){
23             ans += bit[i];
24         }
25         return ans;
26     }
27
28     int query(int l, int r){
29         return prefix_query(r) - prefix_query(l-1);
30     }
31
32     //int bit 0-1 it finds the index of k-th element active
33     int kth(int k) {
34         int cur = 0;
35         int acc = 0;
36         for(int i = 19; i >= 0; i--) {
37             if(cur + (1<<i) > n) continue;
38             if(acc + bit[cur + (1<<i)] < k) {
39                 cur += (1<<i);
40                 acc += bit[cur];
41             }
42         }
43         return ++cur;
44     }
45 };
46

```

2.4. Trie

```

1 namespace Trie{
2
3     struct node {
4         node *adj[SIZE_NODE];
5         node(){
6             for(int i=0; i<SIZE_NODE; i++) adj[i] = NULL;
7         }
8     };
9
10    struct Tree{
11
12        node *t;
13
14        Tree(){
15            t = new node();
16        }
17
18        void add(){
19            node *cur = t;
20        }
21
22
23        int query(){
24            node *cur = t;
25        }
26
27        void remove(){
28            node *cur = t;
29        }
30
31    };
32
33 }

```

2.5. STL Ordered Set

```

1 //INCLUDES
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4
5 //NAMESPACE
6 using namespace __gnu_pbds;
7
8 typedef tree<
9 int, //change for pair<int,int> to use like multiset
10 null_type,
11 less<int>, //change for pair<int,int> to use like multiset
12 rb_tree_tag,
13 tree_order_statistics_node_update>
14 ordered_set;
15
16 //int differ = 0; for multiset
17
18 //ordered_set myset; //declares a stl ordered set
19 //myset.insert(1); //inserts
20 //myset.insert(make_pair(1, differ++)); //insertion for multiset
21 //myset.find_by_order(k)//returns an iterator to the k-th element (or
    returns the end)
22 //myset.order_of_key(x)//returns the number of elements strictly less than x
23 //myset.order_of_key(myset.lower_bound(make_pair(x, 0))) //for multisets

```

2.6. Convex Hull Trick

```

1 struct ConvexHullTrick {
2     //max cht, suppose lines are added in crescent order of a
3     vector<line> cht;
4     ConvexHullTrick() {}
5     struct line{
6         int id, a, b;
7         line() {}
8         line(int id, int a, int b) : id(id), a(a), b(b) {}
9     };
10    bool cmp(const line & a, const line & b){
11        return (a.a < b.a || (a.a == b.a && a.b > b.b));
12    }
13    bool remove(line & a, line & b, line & c){
14        if((a.a - c.a)*(c.b - b.b) <= (b.a - c.a)*(c.b - a.b)) return true;
15        else return false;
16    }
17
18    void add(line & v){
19        if(cht.empty()){
20            cht.push_back(v);
21        }
22        else {
23            if(cht.back().a == v.a) return;
24            while(cht.size() > 1 && remove(cht[cht.size()-2], cht.back(), v)){
25                cht.pop_back();
26            }
27            cht.push_back(v);
28        }
29    }
30
31    void preprocess_cht(vector< line > & v){
32        sort(v.begin(), v.end(), cmp);
33        cht.clear();
34        for(int i=0; i<v.size(); i++){
35            add(v[i]);
36        }
37    }
38
39    int f(int i, int x){
40        return cht[i].a*x + cht[i].b;
41    }
42
43    //return line index
44    int query(int x){
45        if(cht.size() == 0) return -1;
46        if(cht.size() == 1) return 0;
47        int l = 0, r = cht.size()-2;
48        int ans= cht.size()-1;
49        while(l <= r){
50            int m = (l+r)/2;
51            int y1 = f(m, x);
52            int y2 = f(m+1, x);
53            if(y1 >= y2){
54                ans = m;
55                r = m-1;
56            }
57            else l = m+1;
58        }
59        return ans;
60    }
61
62 };

```

2.7. Lichao Segment Tree - Convex Hull Trick

```

1 namespace Lichao{
2     //min lichao tree
3
4     struct line {
5         int a, b;
6         line() {
7             a = 0;
8             b = INF;
9         }
10        line(int a, int b) : a(a), b(b) {}
11        int eval(int x){
12            return a*x + b;
13        }
14    };
15
16    struct node {
17        node * left, * right;
18        line ln;
19        node(){
20            left = NULL;
21            right = NULL;
22        }
23    };
24
25    struct Tree {
26
27        node * root;
28
29        Tree() {
30            root = new node();
31        }
32
33        void add(node * root, int l, int r, line ln){
34            if(!root->left) root->left = new node();
35            if(!root->right) root->right = new node();
36            int m = (l+r)>>1;
37            bool left = ln.eval(l) < (root->ln).eval(l);
38            bool mid = ln.eval(m) < (root->ln).eval(m);
39
40            if(mid){
41                swap(root->ln, ln);
42            }
43
44            if(l == r) return;
45            else if(left != mid) add(root->left, l, m, ln);
46            else add(root->right, m+1, r, ln);
47        }
48
49        int query(node * root, int l, int r, int x){
50            if(!root->left) root->left = new node();
51            if(!root->right) root->right = new node();
52            int m = (l+r)>>1;
53            if(l == r) return (root->ln).eval(x);
54            else if(x < m) return min((root->ln).eval(x), query(root->left, l, m,
55 x));
56            else return min((root->ln).eval(x), query(root->right, m+1, r, x));
57        }
58    };
59
60 }

```

2.8. Lichao Segment Tree - Convex Hull Trick - Double Type

```

1 namespace Lichao{
2     //min lichao tree working with doubles
3
4     struct line {
5         double a, b;
6         line() {
7             a = 0;
8             b = INF;
9         }
10        line(double a, double b) : a(a), b(b) {}
11        double eval(double x){
12            return a*x + b;
13        }
14    };
15
16    struct node {
17        node * left, * right;
18        line ln;
19        node(){
20            left = NULL;
21            right = NULL;
22        }
23    };
24
25    struct Tree {
26
27        node * root;
28
29        Tree() {
30            root = new node();
31        }
32
33        void add(node * root, double l, double r, line ln){
34            if(!root->left) root->left = new node();
35            if(!root->right) root->right = new node();
36            double m = (l+r)/2.;
37            bool left = ln.eval(l) < (root->ln).eval(l);
38            bool mid = ln.eval(m) < (root->ln).eval(m);
39
40            if(mid){
41                swap(root->ln, ln);
42            }
43
44            if(abs(r-l) <= 1e-9) return;
45            else if(left != mid) add(root->left, l, m, ln);
46            else add(root->right, m, r, ln);
47        }
48
49        double query(node * root, double l, double r, double x){
50            if(!root->left) root->left = new node();
51            if(!root->right) root->right = new node();
52            double m = (l+r)/2.;
53            if(abs(r-l) <= 1e-9) return (root->ln).eval(x);
54            else if(x < m) return min((root->ln).eval(x), query(root->left, l, m,
55 x));
56            else return min((root->ln).eval(x), query(root->right, m, r, x));
57        }
58    };
59
60 }

```

3. Uncategorized

3.1. Coordinate Compression

```

1 struct Compressor {
2
3     vector<int> value;
4
5     Compressor() {}
6
7     Compressor(int n){
8         value.resize(n);
9     }
10
11     void compress(vector<int> & v){
12         vector<int> tmp;
13         set<int> s;
14         for(int i=0; i<v.size(); i++) s.insert(v[i]);
15         for(int x : s) tmp.pb(x);
16         for(int i=0; i<v.size(); i++){
17             int idx = lower_bound(tmp.begin(), tmp.end(), v[i]) - tmp.begin();
18             value[idx] = v[i];
19             v[i] = idx;
20         }
21     }
22
23 } compressor;

```

3.2. Longest Increasing Subsequence

```

1 /* Use upper_bound to swap to longest non decreasing subsequence */
2
3 struct LIS{
4
5     vector<int> seq;
6
7     LIS() {}
8
9     LIS(int n){
10         seq.resize(n+1);
11     }
12
13     void calculate(vector<int> & v){
14         int n = v.size();
15         for(int i=1; i<=n; i++) seq[i] = INT_MAX;
16         seq[0] = INT_MIN;
17         for(int i=0; i<n; i++){
18             int index = lower_bound(seq.begin(), seq.end(), v[i]) - seq.begin();
19             index--;
20             seq[index+1] = min(seq[index+1], v[i]);
21         }
22     }
23
24 };

```

3.3. Inversion Count - Merge Sort

```

1 int mergesort_count(vector<int> & v){
2     vector<int> a,b;
3     if(v.size() == 1) return 0;
4     for(int i=0; i<v.size()/2; i++) a.push_back(v[i]);
5     for(int i=v.size()/2; i<v.size(); i++) b.push_back(v[i]);
6     int ans = 0;
7     ans += mergesort_count(a);
8     ans += mergesort_count(b);
9     a.push_back(LLONG_MAX);
10    b.push_back(LLONG_MAX);
11    int x = 0, y = 0;
12    for(int i=0; i<v.size(); i++){
13        if(a[x] <= b[y]){
14            v[i] = a[x++];
15        }
16        else {
17            v[i] = b[y++];
18            ans += a.size() - x - 1;
19        }
20    }
21    return ans;
22 }

```

3.4. Mo's Decomposition

```

1 namespace Mos {
2
3     int sqr;
4
5     struct query{
6         int id, l, r, ans;
7         bool operator<(const query & b) const {
8             if(l/sqr != b.l/sqr) return l/sqr < b.l/sqr;
9             return (l/sqr) % 2 ? r > b.r : r < b.r;
10        }
11    };
12
13    struct QueryDecomposition {
14
15        vector<query> q;
16
17        QueryDecomposition(int n, int nq){
18            q.resize(nq);
19            sqr = (int)sqrt(n);
20        }
21
22        void read(){
23
24        }
25
26        void add(int idx){
27
28        }
29
30        void remove(int idx){
31
32        }
33
34        int answer_query(){
35
36        }
37
38        void calculate(){
39            sort(q.begin(), q.end());
40            int l = 0, r = -1;
41            for(int i=0; i<q.size(); i++){
42                while(q[i].l < l) add(--l);
43                while(r < q[i].r) add(++r);
44                while(q[i].l > l) remove(l++);
45                while(r > q[i].r) remove(r--);
46                q[i].ans = answer_query();
47            }
48        }
49
50        void print(){
51            sort(q.begin(), q.end(), [](const query & a, const query & b){
52                return a.id < b.id;
53            });
54
55            for(query x : q){
56                cout << x.ans << endl;
57            }
58        }
59    };
60
61 }

```

4. Geometry

4.1. 2D Structures

```

1 ////////////////////////////////////////////////////////////////// Geometry Structures
2 //////////////////////////////////////////////////////////////////
3 namespace Geo2D {
4
5     struct Point {
6         int x,y;
7
8         Point(){
9             x = 0;
10            y = 0;
11        }
12
13        Point(int x, int y) : x(x), y(y) {}
14
15        Point(Point a, Point b){
16            x = b.x - a.x;
17            y = b.y - a.y;
18        }
19
20        Point operator+(const Point b) const{
21            return Point(x + b.x, y + b.y);
22        }
23
24        Point operator-(const Point b) const{
25            return Point(x - b.x, y - b.y);
26        }
27
28        int operator*(const Point b) const{
29            return (x*b.x + y*b.y);
30        }
31
32        int operator^(const Point b) const{
33            return x*b.y - y*b.x;
34        }
35
36        Point scale(int n){
37            return Point(x*n, y*n);
38        }
39
40        void operator=(const Point b) {
41            x = b.x;
42            y = b.y;
43        }
44
45        bool operator==(const Point b){
46            return x == b.x && y == b.y;
47        }
48
49        double distanceTo(Point b){
50            return sqrt((x - b.x)*(x - b.x) + (y - b.y)*(y - b.y));
51        }
52
53        int squareDistanceTo(Point b){
54            return (x - b.x)*(x - b.x) + (y - b.y)*(y - b.y);
55        }
56
57        bool operator<(const Point & p) const{
58            return tie(x,y) < tie(p.x, p.y);
59        }
60    }

```

```

61 double size(){
62     return sqrt(x*x + y*y);
63 }
64
65 int squareSize(){
66     return x*x + y*y;
67 }
68
69 //Only with double type
70 Point normalize(){
71     return Point((double)x/size(), (double)y/size());
72 }
73
74 void rotate(double ang){
75     double xx = x, yy = y;
76     x = xx*cos(ang) + yy*-sin(ang);
77     y = xx*sin(ang) + yy*cos(ang);
78 }
79
80 };
81
82 struct Line {
83     Point p, q;
84     Point v;
85     Point normal;
86
87     int a,b,c;
88
89     Line() {
90         p = Point();
91         q = Point();
92         v = Point();
93         normal = Point();
94         a = 0;
95         b = 0;
96         c = 0;
97     }
98
99     Line(int aa, int bb, int cc){
100         a = aa;
101         b = bb;
102         c = cc;
103         normal = Point(a,b);
104         v = Point(-normal.y, normal.x);
105         p = Point();
106         q = Point();
107     }
108
109     void operator=(const Line l){
110         a = l.a;
111         b = l.b;
112         c = l.c;
113         p = l.p;
114         q = l.q;
115         v = l.v;
116         normal = l.normal;
117     }
118
119     Line(Point r, Point s){
120         p = r;
121         q = s;
122         v = Point(r, s);
123         normal = Point(-v.y, v.x);
124         a = -v.y;
125         b = v.x;

```

```

126     c = -(a*p.x + b*p.y);
127 }
128
129 void flip_sign(){
130     a = -a, b = -b, c = -c;
131 }
132
133 void normalize(){
134     if(a < 0) flip_sign();
135     else if(a == 0 && b < 0) flip_sign();
136     else if(a == 0 && b == 0 && c < 0) flip_sign();
137     int g = max(a, max(b,c));
138     if(a != 0) g = gcd(g, a); if(b != 0) g = gcd(g,b); if(c != 0) g =
gcd(g,c);
139     if(g > 0) a/=g, b/=g, c/=g;
140 }
141
142 bool operator<(const Line & l) const{
143     return tie(a,b,c) < tie(l.a, l.b, l.c);
144 }
145
146 };
147
148 struct Circle{
149     Point c;
150     double r;
151     Circle() {}
152     Circle(Point center, double radius) : c(center), r(radius) {}
153
154     bool operator=(Circle circ){
155         c = circ.c;
156         r = circ.r;
157     }
158
159     pair<Point, Point> getTangentPoints(Point p){
160         //p needs to be outside the circle
161         double d = p.distanceTo(c);
162         double ang = asin(1.*r/d);
163         Point v1(p, c);
164         v1.rotate(ang);
165         Point v2(p, c);
166         v2.rotate(-ang);
167         v1 = v1.scale(sqrt(d*d - r*r)/d);
168         v2 = v2.scale(sqrt(d*d - r*r)/d);
169         Point p1(v1.x + p.x, v1.y + p.y);
170         Point p2(v2.x + p.x, v2.y + p.y);
171         return make_pair(p1,p2);
172     }
173
174     double sectorArea(double ang){
175         return (ang*r*r)/2.;
176     }
177
178     double arcLength(double ang){
179         return ang*r;
180     }
181
182     double sectorArea(Point p1, Point p2){
183         double h = p1.distanceTo(p2);
184         double ang = acos(1. - h*h/r*r);
185         return sectorArea(ang);
186     }
187
188     double arcLength(Point p1, Point p2){
189         double h = p1.distanceTo(p2);

```



```

190     double ang = acos(1. - (h*h)/(2*r*r));
191     return arcLength(ang);
192 }
193
194 bool inside(const Point & p){
195     if(Point(c,p).size() + EPS < r) return true;
196     else if(r + EPS < Point(c,p).size()) return false;
197     else return true;
198 }
199
200 };
201
202 }
203
204 ////////////////////////////////////////////////// End of Geometry Structures
    
```

4.2. 2D Geometry Functions

```

1  ////////////////////////////////////////////////// Geometry Algorithms
2  //////////////////////////////////////////////////
3  namespace Geo2D {
4
5      double distancePointLine(Point p, Line l){
6          if(l.normal.squareSize() == 0) return INF;
7          return (double)(l.a*p.x + l.b*p.y + l.c)/l.normal.size();
8      }
9
10     double distancePointSegment(Point p, Line l){
11         int dot1 = Point(l.p, p)*Point(l.p, l.q);
12         int dot2 = Point(l.q, p)*Point(l.q, l.p);
13
14         if(dot1 >= 0 && dot2 >= 0) return distancePointLine(p, l);
15         else return min(p.distanceTo(l.p), p.distanceTo(l.q));
16     }
17
18     double distancePointRay(Point p, Line l){
19         int dot = Point(l.p, p)*l.v;
20         if(dot >= 0) return distancePointLine(p, l);
21         else return p.distanceTo(l.p);
22     }
23
24     Point closestPointInSegment(Point p, Line s){
25         //returns closest point from p in segment s
26         Point u = s.v.normalize();
27         Point w(s.p, p);
28         Point res = u.scale(u*w);
29         if(u*w < 0 || u*w > s.p.distanceTo(s.q)){
30             if(p.distanceTo(s.p) < p.distanceTo(s.q)) return s.p;
31             else return s.q;
32         }
33         else return Point(s.p.x + res.x, s.p.y + res.y);
34     }
35
36     Point intersectionSegmentSegment(Line s1, Line s2){
37         //Assumes that intersection exists
38         //Assuming that endpoints are ordered by x
39         if(s1.p.x > s1.q.x) swap(s1.p, s1.q);
40         if(s2.p.x > s2.q.x) swap(s2.p, s2.q);
41
42         if(abs(s1.v^s2.v) <= EPS){
43
44             //parallel segments
    
```

```

45     Point v1(s2.p, s1.p);
46     if(s1.p.x == s1.q.x && s2.p.x == s2.q.x && s1.p.x == s2.p.x){
47         Point ans1, ansr;
48         if(s1.p.y > s1.q.y) swap(s1.p, s1.q);
49         if(s2.p.y > s2.q.y) swap(s2.p, s2.q);
50         if(s1.p.y <= s2.p.y) ans1 = s2.p;
51         else ans1 = s1.p;
52         if(s2.q.y <= s1.q.y) ansr = s2.q;
53         else ansr = s1.q;
54         if(ans1.x == ansr.x && ans1.y == ansr.y){
55             //cout << ansr.x << " " << ansr.y << endl;
56             return Point(ansr.x, ansr.y);
57         }
58         else {
59             if(ans1.x == ansr.x && ans1.y > ansr.y) swap(ans1, ansr);
60             //cout << ans1.x << " " << ans1.y << endl << ansr.x << " " <<
61             ansr.y << endl;
62             return Point(INF, INF);
63         }
64     }
65     else if(abs(s1.v^v1) <= EPS){
66         Point ans1, ansr;
67         if(s1.p.x <= s2.p.x) ans1 = s2.p;
68         else ans1 = s1.p;
69         if(s2.q.x <= s1.q.x) ansr = s2.q;
70         else ansr = s1.q;
71         if(ans1.x == ansr.x && ans1.y == ansr.y){
72             //cout << ansr.x << " " << ansr.y << endl;
73             return Point(ansr.x, ansr.y);
74         }
75         else {
76             if(ans1.x == ansr.x && ans1.y > ansr.y) swap(ans1, ansr);
77             //cout << ans1.x << " " << ans1.y << endl << ansr.x << " " <<
78             ansr.y << endl;
79             return Point(INF, INF);
80         }
81     }
82     else {
83         //general case
84         int a1 = s1.q.y - s1.p.y;
85         int b1 = s1.p.x - s1.q.x;
86         int c1 = a1*s1.p.x + b1*s1.p.y;
87         int a2 = s2.q.y - s2.p.y;
88         int b2 = s2.p.x - s2.q.x;
89         int c2 = a2*s2.p.x + b2*s2.p.y;
90         int det = a1*b2 - a2*b1;
91
92         double x = (double)(b2*c1 - b1*c2)/(double)det*1.;
93         double y = (double)(a1*c2 - a2*c1)/(double)det*1.;
94         //cout << x << " " << y << endl;
95         return Point(x,y);
96     }
97 }
98
99
100 double distanceSegmentSegment(Line l1, Line l2){
101     if(l1.p == l2.p && l1.q == l2.q) return 0;
102     if(l1.q == l2.p && l1.p == l2.q) return 0;
103     if((l1.v^l2.v) != 0){
104
105         Line r1(l1.p, l1.q);
106         Line r2(l1.q, l1.p);
107         Line r3(l2.p, l2.q);
    
```

```

108     Line r4(l2.q, l2.p);
109
110     int cross1 = (Point(r3.p, r1.p)^r3.v);
111     int cross2 = (Point(r3.p, r1.q)^r3.v);
112     if(cross2 < cross1) swap(cross1, cross2);
113
114     bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
115 r3) > distancePointLine(r1.q, r3));
116
117     cross1 = (Point(r1.p, r3.p)^r1.v);
118     cross2 = (Point(r1.p, r3.q)^r1.v);
119     if(cross2 < cross1) swap(cross1, cross2);
120
121     bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p,
122 r1) > distancePointLine(r3.q, r1));
123
124     cross1 = (Point(r3.p, r2.p)^r3.v);
125     cross2 = (Point(r3.p, r2.q)^r3.v);
126     if(cross2 < cross1) swap(cross1, cross2);
127
128     bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
129 r3) > distancePointLine(r2.q, r3));
130
131     cross1 = (Point(r2.p, r3.p)^r2.v);
132     cross2 = (Point(r2.p, r3.q)^r2.v);
133     if(cross2 < cross1) swap(cross1, cross2);
134
135     bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p,
136 r2) > distancePointLine(r3.q, r2));
137
138     cross1 = (Point(r4.p, r1.p)^r4.v);
139     cross2 = (Point(r4.p, r1.q)^r4.v);
140     if(cross2 < cross1) swap(cross1, cross2);
141
142     bool ok5 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
143 r4) > distancePointLine(r1.q, r4));
144
145     cross1 = (Point(r1.p, r4.p)^r1.v);
146     cross2 = (Point(r1.p, r4.q)^r1.v);
147     if(cross2 < cross1) swap(cross1, cross2);
148
149     bool ok6 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p,
150 r1) > distancePointLine(r4.q, r1));
151
152     cross1 = (Point(r4.p, r2.p)^r4.v);
153     cross2 = (Point(r4.p, r2.q)^r4.v);
154     if(cross2 < cross1) swap(cross1, cross2);
155
156     bool ok7 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
157 r4) > distancePointLine(r2.q, r4));
158
159     cross1 = (Point(r2.p, r4.p)^r2.v);
160     cross2 = (Point(r2.p, r4.q)^r2.v);
161     if(cross2 < cross1) swap(cross1, cross2);
162
163     bool ok8 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p,
164 r2) > distancePointLine(r4.q, r2));
165
166     if(ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8) return 0;
167
168 }
169
170 double ans = distancePointSegment(l1.p, l2);
171 ans = min(ans, distancePointSegment(l1.q, l2));

```

```

164     ans = min(ans, distancePointSegment(l2.p, l1));
165     ans = min(ans, distancePointSegment(l2.q, l1));
166     return ans;
167 }
168
169 double distanceSegmentRay(Line s, Line r){
170     if((s.v^r.v) != 0){
171         Line r1(s.p, s.q);
172         Line r2(s.q, s.p);
173
174         int cross1 = (Point(r.p, r1.p)^r.v);
175         int cross2 = (Point(r.p, r1.q)^r.v);
176         if(cross2 < cross1) swap(cross1, cross2);
177
178         bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p, r)
179 > distancePointLine(r1.q, r));
180
181         cross1 = (Point(r1.p, r.p)^r1.v);
182         cross2 = (Point(r1.p, r.q)^r1.v);
183         if(cross2 < cross1) swap(cross1, cross2);
184
185         bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r1)
186 > distancePointLine(r.q, r1));
187
188         cross1 = (Point(r.p, r2.p)^r.v);
189         cross2 = (Point(r.p, r2.q)^r.v);
190         if(cross2 < cross1) swap(cross1, cross2);
191
192         bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r)
193 > distancePointLine(r2.q, r));
194
195         cross1 = (Point(r2.p, r.p)^r2.v);
196         cross2 = (Point(r2.p, r.q)^r2.v);
197         if(cross2 < cross1) swap(cross1, cross2);
198
199         bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r2)
200 > distancePointLine(r.q, r2));
201
202         if(ok1 && ok2 && ok3 && ok4) return 0;
203     }
204
205     double ans = INF;
206     int dot = Point(s.p, r.p)*Point(r.p, s.q);
207     if(dot >= 0) ans = min(ans, distancePointLine(r.p, s));
208     else ans = min(ans, min(r.p.distanceTo(s.p), r.p.distanceTo(s.q)));
209
210     dot = Point(r.p, s.p)*r.v;
211     if(dot >= 0) ans = min(ans, distancePointLine(s.p, r));
212     else ans = min(ans, r.p.distanceTo(s.p));
213
214     dot = Point(r.p, s.q)*r.v;
215     if(dot >= 0) ans = min(ans, distancePointLine(s.q, r));
216     else ans = min(ans, r.p.distanceTo(s.q));
217
218     return ans;
219 }
220
221 double distanceSegmentLine(Line s, Line l){
222     if((s.v^l.v) == 0){
223         return distancePointLine(s.p, l);
224     }
225
226     int cross1 = (Point(l.p, s.p)^l.v);

```

```

225     int cross2 = (Point(l.p, s.q)^l.v);
226     if(cross2 < cross1) swap(cross1, cross2);
227     if(cross1 <= 0 && cross2 >= 0) return 0;
228     else return min(distancePointLine(s.p, l), distancePointLine(s.q,l));
229 }
230
231 double distanceLineRay(Line l, Line r){
232     if((l.v^r.v) == 0){
233         return distancePointLine(r.p, l);
234     }
235
236     int cross1 = (Point(l.p, r.p)^l.v);
237     int cross2 = (Point(l.p, r.q)^l.v);
238     if(cross2 < cross1) swap(cross1, cross2);
239     if((cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, l) >
240     distancePointLine(r.q, l))) return 0;
241     return distancePointLine(r.p, l);
242 }
243
244 double distanceLineLine(Line l1, Line l2){
245     if((l1.v^l2.v) == 0){
246         return distancePointLine(l1.p, l2);
247     }
248     else return 0;
249 }
250
251 double distanceRayRay(Line r1, Line r2){
252     if((r1.v^r2.v) != 0){
253
254         int cross1 = (Point(r1.p, r2.p)^r1.v);
255         int cross2 = (Point(r1.p, r2.q)^r1.v);
256         if(cross2 < cross1) swap(cross1, cross2);
257         bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
258         r1) > distancePointLine(r2.q, r1));
259
260         cross1 = (Point(r2.p, r1.p)^r2.v);
261         cross2 = (Point(r2.p, r1.q)^r2.v);
262         if(cross2 < cross1) swap(cross1, cross2);
263
264         bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
265         r2) > distancePointLine(r1.q, r2));
266
267         if(ok1 && ok2) return 0;
268     }
269
270     double ans = INF;
271     int dot = Point(r2.p, r1.p)*r2.v;
272     if(dot >= 0) ans = min(ans, distancePointLine(r1.p, r2));
273     else ans = min(ans, r2.p.distanceTo(r1.p));
274
275     dot = Point(r1.p, r2.p)*r1.v;
276     if(dot >= 0) ans = min(ans, distancePointLine(r2.p, r1));
277     else ans = min(ans, r1.p.distanceTo(r2.p));
278     return ans;
279 }
280
281 double circleCircleIntersection(Circle c1, Circle c2){
282
283     if((c1.r+c2.r)*(c1.r+c2.r) <= (c2.c.x-c1.c.x)*(c2.c.x-c1.c.x) +
284     (c2.c.y-c1.c.y)*(c2.c.y-c1.c.y)){
285         return 0;

```

```

286     }
287     if((c1.r-c2.r)*(c1.r-c2.r) >= (c2.c.x-c1.c.x)*(c2.c.x-c1.c.x) +
288     (c2.c.y-c1.c.y)*(c2.c.y-c1.c.y)){
289         return PI*min(c1.r, c2.r)*min(c1.r, c2.r);
290     }
291     double x1 = c1.c.x, x2 = c2.c.x, y1 = c1.c.y, y2 = c2.c.y, r1 = c1.r, r2
292     = c2.r;
293     double d = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
294     double r1sqr = c1.r*c1.r;
295     double r2sqr = c2.r*c2.r;
296     double dsqr = d*d;
297
298     double alpha1 = acos(((c1.r + c2.r)*(c1.r - c2.r) + dsqr)/(2.*d*r1));
299     double alpha2 = acos(((c2.r + c1.r)*(c2.r - c1.r) + dsqr)/(2.*d*r2));
300     double areal = r1sqr*(alpha1 - sin(alpha1)*cos(alpha1));
301     double area2 = r2sqr*(alpha2 - sin(alpha2)*cos(alpha2));
302
303     return areal + area2;
304 }
305
306 vector<Point> intersectionLineCircle(Line l, Circle circ){
307     //NOT TESTED!!!!!!!
308     //no intersection
309     if((l.c*l.c)/(circ.r*circ.r) > l.a*l.a + l.b*l.b) return vector<Point>();
310
311     double x0 = -l.a*l.c/(l.a*l.a+l.b*l.b), y0 = -l.b*l.c/(l.a*l.a+l.b*l.b);
312     //one intersection
313     if(abs((l.c*l.c)/(circ.r*circ.r) - (l.a*l.a + l.b*l.b)) <= EPS){
314         vector<Point> ret;
315         ret.pb(Point(x0,y0));
316         return ret;
317     }
318     //general case
319     double d = circ.r*circ.r - (l.c*l.c)/(l.a*l.a+l.b*l.b);
320     double mult = sqrt(d/(l.a*l.a+l.b*l.b));
321
322     Point p1(x0 + l.b*mult, y0 - l.a*mult);
323     Point p2(x0 - l.b*mult, y0 + l.a*mult);
324
325     vector<Point> ret;
326     ret.pb(p1); ret.pb(p2);
327     return ret;
328 }
329
330 vector<Point> intersectionCircleCircle(Circle c1, Circle c2){
331     //NOT TESTED!!!!!!!
332     //translate first circle to origin
333     Point translation = c1.c;
334     c1.c = Point(0,0);
335     c2.c = c2.c - translation;
336
337     //check if centers are equal
338     if(c1.c == c2.c){
339         //if radius are equal = infinite intersections(return 3 points to
340         indicate), else = no intersection(empty)
341         if(c1.r == c2.r){
342             vector<Point> ret;
343             ret.pb(Point());
344             ret.pb(Point());
345             ret.pb(Point());
346             return ret;
347         }
348         else return vector<Point>();

```

```

348     }
349
350     //general case
351     Line l(-2*c2.c.x,-2*c2.c.y, c2.c.x*c2.c.x + c2.c.y*c2.c.y + c1.r*c1.r -
        c2.r*c2.r);
352
353     vector<Point> ret = intersectionLineCircle(l, c1);
354
355     for(Point & p : ret){
356         p = p + translation;
357     }
358
359     return ret;
360 }
361
362 Point barycenter(Point & a, Point & b, Point & c, double pA, double pB,
    double pC){
363     Point ret = (a.scale(pA) + b.scale(pB) + c.scale(pC));
364     ret.x /= (pA + pB + pC);
365     ret.y /= (pA + pB + pC);
366     return ret;
367 }
368
369 Point circumcenter(Point & a, Point & b, Point & c){
370     double pA = Point(b,c).squareSize(), pB = Point(a,c).squareSize(), pC =
        Point(a,b).squareSize();
371     return barycenter(a,b,c, pA*(pB+pC-pA), pB*(pC+pA-pB), pC*(pA+pB-pC));
372 }
373
374 Point centroid(Point & a, Point & b, Point & c){
375     return barycenter(a,b,c,1,1,1);
376 }
377
378 Point incenter(Point & a, Point & b, Point & c){
379     return barycenter(a,b,c, Point(b,c).size(), Point(a,c).size(),
        Point(a,b).size());
380 }
381
382 Point excenter(Point & a, Point & b, Point & c){
383     return barycenter(a,b,c, -Point(b,c).size(), Point(a,c).size(),
        Point(a,b).size());
384 }
385
386 Point orthocenter(Point & a, Point & b, Point & c){
387     double pA = Point(b,c).squareSize(), pB = Point(a,c).squareSize(), pC =
        Point(a,b).squareSize();
388     return barycenter(a, b, c, (pA+pB-pC)*(pC+pA-pB), (pB+pC-pA)*(pA+pB-pC),
        (pC+pA-pB)*(pB+pC-pA));
389 }
390
391 Circle minimumCircle(vector<Point> & v){
392     Circle circ(Point(0,0), 1e-14);
393     random_shuffle(v.begin(), v.end());
394     for(int i=0; i<v.size(); i++){
395         if(!circ.inside(v[i])){
396             circ = Circle(v[i], 0);
397             for(int j=0; j<i; j++){
398                 if(!circ.inside(v[j])){
399                     circ = Circle((v[i] + v[j]).scale(0.5), Point(v[i],
        v[j]).size()*0.5);
400                     for(int k = 0; k<j; k++){
401                         if(!circ.inside(v[k])){
402                             Point center = circumcenter(v[i], v[j], v[k]);
403                             circ = Circle(center, Point(center, v[k]).size());
404                         }

```

```

405                     }
406                 }
407             }
408         }
409     }
410     return circ;
411 }
412
413 long long ClosestPairOfPoints(vector<Point> &a) {
414     //returns square of distance
415     long long mid = a[a.size()/2].x;
416     int n = a.size();
417
418     vector<Point> l;
419     vector<Point> r;
420     int i = 0;
421     for(; i < a.size()/2; i++) l.push_back(a[i]);
422     for(; i < a.size(); i++) r.push_back(a[i]);
423
424     long long d = LLONG_MAX;
425
426     if(l.size() > 1) {
427         d = min(d, ClosestPairOfPoints(l));
428     } if(r.size() > 1) {
429         d = min(d, ClosestPairOfPoints(r));
430     }
431
432     a.clear();
433
434     vector<Point> ll;
435     vector<Point> rr;
436
437
438     int j = 0;
439     i = 0;
440     for(int k=0; k<n; k++){
441         if(i < l.size() && j < r.size()){
442             if(r[j].y <= l[i].y){
443                 if((r[j].x - mid)*(r[j].x - mid) < d) {
444                     rr.push_back(r[j]);
445                 }
446                 a.push_back(r[j++]);
447             }
448             else {
449                 if((l[i].x - mid)*(l[i].x - mid) < d) {
450                     ll.push_back(l[i]);
451                 }
452                 a.push_back(l[i++]);
453             }
454         }
455         else if(i < l.size()){
456             if((l[i].x - mid)*(l[i].x - mid) < d) {
457                 ll.push_back(l[i]);
458             }
459             a.push_back(l[i++]);
460         }
461         else {
462             if((r[j].x - mid)*(r[j].x - mid) < d) {
463                 rr.push_back(r[j]);
464             }
465             a.push_back(r[j++]);
466         }
467     }
468
469     for(int i = 0; i < ll.size(); i++) {

```

```

470     int ini = 0, end = rr.size()-1;
471     int j;
472
473     while(ini < end) {
474         j = (ini + end) / 2;
475         if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
476             ini = j+1;
477         else end = j;
478     }
479
480     j = ini;
481
482     for(; j < rr.size(); j++) {
483         if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
484         long long cur = (ll[i].x - rr[j].x)*(ll[i].x - rr[j].x) + (ll[i].y
485 - rr[j].y)*(ll[i].y - rr[j].y);
486         if(cur < d) {
487             d = cur;
488         }
489     }
490     return d;
491 }
492 }
493
494 }
495
496 ////////////////////////////////////////////////// End of Geometry Algorithms
497 //////////////////////////////////////

```

4.3. Convex Hull - Monotone Chain Algorithm

```

1 namespace Geo2D {
2
3 struct ConvexHull {
4
5     vector< Point > points, lower, upper;
6
7     ConvexHull(){}
8
9     void calculate(vector<Point> v){
10         sort(v.begin(), v.end());
11         for(int i=0; i<v.size(); i++){
12             while(upper.size() >= 2 && (Point(upper[upper.size()-2],
13 upper.back())^Point(upper.back(), v[i])) >= 0LL) upper.pop_back();
14             upper.push_back(v[i]);
15         }
16         reverse(v.begin(), v.end());
17         for(int i=0; i<v.size(); i++){
18             while(lower.size() >= 2 && (Point(lower[lower.size()-2],
19 lower.back())^Point(lower.back(), v[i])) >= 0LL) lower.pop_back();
20             lower.push_back(v[i]);
21         }
22         for(int i=upper.size()-2; i>=0; i--) points.push_back(upper[i]);
23         for(int i=lower.size()-2; i>=0; i--) points.push_back(lower[i]);
24         reverse(lower.begin(), lower.end());
25     }
26
27     double area(){
28         double area = points.back().x*points[0].y -
29 points.back().y*points[0].x;
30         for(int i=0; i<points.size()-1; i++){
31             area += points[i].x*points[i+1].y - points[i].y*points[i+1].x;
32         }
33         return area/2.;
34     }
35
36     int area2(){
37         int area2 = points.back().x*points[0].y - points.back().y*points[0].x;
38         for(int i=0; i<points.size()-1; i++){
39             area2 += points[i].x*points[i+1].y - points[i].y*points[i+1].x;
40         }
41         return area2;
42     }
43
44     double perimeter(){
45         double val = Point(points[0], points.back()).size();
46         for(int i=0; i<points.size()-1; i++){
47             val += Point(points[i], points[i+1]).size();
48         }
49         return val;
50     }
51
52     bool insideHull(Point p){
53
54         auto it = lower_bound(lower.begin(), lower.end(), p);
55         if(it != lower.end() && *it == p) return true;
56         it = lower_bound(upper.begin(), upper.end(), p);
57         if(it != upper.end() && *it == p) return true;
58
59         if(p.x == upper[0].x){
60             if(p.y > upper[0].y){
61                 //upper
62                 if(upper[1].x != upper[0].x) return false;
63                 else if(p.y <= upper[1].y) return true;

```

```

61     }
62     else {
63         //lower
64         if(lower[l].x != lower[0].x) return false;
65         else if(p.y >= lower[l].y) return true;
66     }
67     return false;
68 }
69 Point v1,v2;
70 //upper or lower
71 int ansu = -1, ans1 = -1;
72 int l = 0, r = upper.size()-2;
73 while(l <= r){
74     int m = (l+r)>>1LL;
75     if(upper[m].x < p.x && p.x <= upper[m+1].x){
76         ansu = m;
77         break;
78     }
79     else if(upper[m+1].x < p.x) l = m+1;
80     else r = m-1;
81 }
82 l = 0, r = lower.size()-2;
83 while(l <= r){
84     int m = (l+r)>>1LL;
85     if(lower[m].x < p.x && p.x <= lower[m+1].x){
86         ans1 = m;
87         break;
88     }
89     else if(lower[m+1].x < p.x) l = m+1;
90     else r = m-1;
91 }
92 if(ansu == -1 || ans1 == -1) return false;
93 bool oku = false, okl = false;
94 v1 = Point(upper[ansu], upper[ansu+1]);
95 v2 = Point(upper[ansu], p);
96 oku = ((v1^v2) <= 0);
97 v1 = Point(lower[ans1], lower[ans1+1]);
98 v2 = Point(lower[ans1], p);
99 okl = ((v1^v2) >= 0);
100 if(oku && okl) return true;
101 else return false;
102 }
103
104 };
105
106 }

```

5. Graphs

5.1. Dynammmic Connectivity - connected(u,v) query

```

1  /* Dynamicm Connectivity Implementation */
2  /* Uses Divide and Conquer Offline approach */
3  /* Able to answer if two vertex <u,v> are connected */
4  /* No multi-edges allowed */
5  /* DSU + Rollback is used to backtrack merges */
6  /* N is defined as the maximum graph size given by input */
7
8  #define N MAX_INPUT
9
10 int uf[N];
11 int sz[N];
12
13 struct event{
14     int op, u, v, l, r;
15     event() {}
16     event(int o, int a, int b, int x, int y) : op(o), u(a), v(b), l(x), r(y) {}
17 };
18
19 map< pair<int, int>, int > edge_to_l;
20 stack< pair<int*,int> > hist;
21 vector<event> events;
22
23 int init(int n){
24     for(int i=0; i<=n; i++){
25         uf[i] = i;
26         sz[i] = 1;
27     }
28 }
29
30 int find(int u){
31     if(uf[u] == u) return u;
32     else return find(uf[u]);
33 }
34
35 void merge(int u, int v){
36     int a = find(u);
37     int b = find(v);
38     if(a == b) return;
39     if(sz[a] < sz[b]){
40         hist.push(make_pair(&uf[a], uf[a]));
41         uf[a] = b;
42         hist.push(make_pair(&sz[b], sz[b]));
43         sz[b] += sz[a];
44     }
45     else {
46         hist.push(make_pair(&uf[b], uf[b]));
47         hist.push(make_pair(&sz[a], sz[a]));
48         uf[b] = a;
49         sz[a] += sz[b];
50     }
51 }
52
53 int snap(){
54     return hist.size();
55 }
56
57 void rollback(int t){
58     while(hist.size() > t){
59         pair<int*,int> aux = hist.top();
60         hist.pop();
61         *aux.first = aux.second;

```

```

62 }
63 }
64
65 void solve(int l, int r){
66     if(l == r){
67         if(events[l].op == 2){
68             if(find(events[l].u) == find(events[l].v)) cout << "YES" << endl;
69             else cout << "NO" << endl;
70         }
71         return;
72     }
73
74     int m = (l+r)/2;
75     //doing for [L,m]
76     int t = snap();
77     for(int i=l; i<=r; i++){
78         if(events[i].op == 0 || events[i].op == 1){
79             if(events[i].l <= l && m <= events[i].r) merge(events[i].u,
80                 events[i].v);
81         }
82     }
83     solve(l, m);
84     rollback(t);
85
86     //doing for [m+1, R]
87     t = snap();
88     for(int i=l; i<=r; i++){
89         if(events[i].op == 0 || events[i].op == 1){
90             if(events[i].l <= m+1 && r <= events[i].r) merge(events[i].u,
91                 events[i].v);
92         }
93     }
94     solve(m+1, r);
95     rollback(t);
96 }
97
98 void offline_process(){
99     int n, q;
100     cin >> n >> q; //number of vertex and queries
101     init(n);
102     for(int i=0; i<q; i++){
103         string op;
104         int u,v;
105         cin >> op >> u >> v; //add, remove or query for u,v
106         if(u > v) swap(u,v);
107         if(op == "add"){
108             events.push_back(event(0, u, v, i, -1));
109             edge_to_l[make_pair(u,v)] = i;
110         }
111         else if(op == "rem"){
112             int l = edge_to_l[make_pair(u,v)];
113             events.push_back(event(1, u, v, l, i));
114             events[l].r = i;
115         }
116         else if(op == "conn"){
117             events.push_back(event(2, u, v, -1, -1));
118         }
119     }
120     for(int i=0; i<q; i++){
121         if(events[i].op == 0){
122             if(events[i].r == -1){
123                 events[i].r = events.size();
124                 events.push_back(event(1, events[i].u, events[i].v, events[i].l,
125                     events[i].r));
126             }
127         }
128     }

```

```

124 }
125 }
126 }

```

5.2. Bellman Ford Shortest Path

```

1 struct BellmanFord{
2
3     struct edges {
4         int u, v, weight;
5         edges(int u, int v, int weight) :
6             u(u),
7             v(v),
8             weight(weight) {}
9     };
10
11     vector<int> dist;
12
13     vector<edges> e;
14
15     bool cycle = false;
16
17     BellmanFord() {}
18
19     BellmanFord(int n, int m){
20         dist.resize(n+1);
21         e.resize(m+1);
22     }
23
24     void calculate(int source){
25         for(int i=0; i<=dist.size(); i++){
26             dist[i] = INT_MAX;
27         }
28         dist[source] = 0;
29         for(int k=0; k<dist.size()-1; k++){
30             for(int i=0; i<e.size(); i++){
31                 if(dist[e[i].v] > dist[e[i].u] + e[i].weight){
32                     dist[e[i].v] = dist[e[i].u] + e[i].weight;
33                 }
34             }
35         }
36         for(int i=0; i<e.size(); i++){
37             if(dist[e[i].v] > dist[e[i].u] + e[i].weight){
38                 cycle = true;
39             }
40         }
41     }
42
43 };

```

5.3. Eulerian Circuits/Paths

```

1 //Graph - Euler path
2
3 //for undirected graph
4 //circuit - 2 vertex with odd grades
5 //simple path - all vertex with even grades
6 //this algorithm generates a circuit, if you need a path between u,v
7 //create a new edge u-v, compute circuit u..u, then delete the last u
8
9 //for directed graph
10 //circuit - all vertex needs enter grade = exit grade
11 //path - one vertex needs to have one more enter grade
12 //and the other needs to have one more exit grade
13 //this algorithm generates a circuit, if you need a path between u,v
14 //create a new edge u-v, considering that u have one more enter grade
15 //and v one more exit grade
16
17 struct EulerianCircuit {
18
19     vector< set<int> > adj;
20     vector<int> walk;
21     vector<int> deg;
22     int s, t;
23
24     EulerianCircuit();
25     EulerianCircuit(int n){
26         deg.resize(n+1);
27         adj.resize(n+1);
28     }
29
30     void undirected_euler(int u){
31         while(!adj[u].empty()){
32             int v = * (--adj[u].end());
33
34             adj[u].erase(v);
35             adj[v].erase(adj[v].find(u));
36
37             euler(v);
38         }
39
40         walk.push_back(u);
41     }
42
43     void directed_euler(int u){
44         while(!adj[u].empty()){
45             int v = * (--adj[u].end());
46
47             adj[u].erase(v);
48
49             euler(v);
50         }
51
52         walk.push_back(u);
53     }
54
55 };

```

5.4. Kosaraju SCC

```

1 struct SCC {
2
3     vector< vector<int> > adj_t;
4     vector< vector<int> > scc_adj;
5     vector<int> ed;
6     int tempo, comp;
7     vector<bool> vis;
8     vector<int> scc;
9
10    SCC() {}
11
12    SCC(int n){
13        tempo = 0;
14        adj_t.resize(n+1, vector<int>());
15        scc_adj.resize(n+1, vector<int>());
16        ed.resize(n+1);
17        comp = 0;
18        vis.resize(n+1);
19        scc.resize(n+1);
20    }
21
22    void dfs(int u){
23        vis[u] = true;
24        for(int i=0; i<adj_t[u].size(); i++){
25            int v = adj_t[u][i];
26            if(!vis[v]) dfs(v);
27        }
28        ed[u] = ++tempo;
29    }
30
31    void dfst(int u, int comp){
32        scc[u] = comp;
33        vis[u] = true;
34        for(int i=0; i<adj_t[u].size(); i++){
35            int v = adj_t[u][i];
36            if(!vis[v]) dfst(v, comp);
37        }
38    }
39
40    void calculate(int n){
41        for(int i=0; i<=n; i++){
42            vis[i] = false;
43        }
44        for(int i=1; i<=n; i++){
45            if(!vis[i]){
46                dfs(i);
47            }
48        }
49
50        vector< ii > vertex(n);
51
52        for(int i=0; i<n; i++){
53            vis[i] = false;
54            vertex[i] = ii(i+1, ed[i+1]);
55        }
56
57        sort(vertex.begin(), vertex.end(), [](const ii & a, const ii & b) {
58            return a.ss > b.ss; });
59
60        for(int i=0; i<vertex.size(); i++){
61            if(!vis[vertex[i].ff]){
62                comp++;
63                dfst(vertex[i].ff, comp);

```



```

63     }
64     }
65     for(int i=1; i<=n; i++){
66         for(int j=0; j<adj[i].size(); j++){
67             int v = adj[i][j];
68             scc_adj[scc[i]].push_back(scc[v]);
69         }
70     }
71 }
72
73 };

```

5.5. Centroid Decomposition

```

1  /* Centroid Decomposition Implementation */
2  /* c_p[] contains the centroid predecessor on centroid tree */
3  /* removed[] says if the node was already selected as a centroid (limit the
4     subtree search) */
5  /* L[] contains the height of the vertex (from root) on centroid tree (Max
6     is logN) */
7  /* N is equal to the maximum size of tree (given by statement) */
8
9  struct CentroidDecomposition {
10     vector<bool> removed;
11     vector<int> L, subz;
12     vector<int> c_p;
13
14     CentroidDecomposition() {}
15
16     CentroidDecomposition(int n){
17         removed.resize(n+1);
18         L.resize(n+1);
19         c_p.resize(n+1);
20         subz.resize(n+1);
21         for(int i=0; i<=n; i++){
22             c_p[i] = -1;
23         }
24     }
25
26     void centroid_subz(int u, int p){
27         subz[u] = 1;
28         for(int i=0; i<adj[u].size(); i++){
29             int v = adj[u][i];
30             if(v == p || removed[v]) continue;
31             centroid_subz(v, u);
32             subz[u] += subz[v];
33         }
34     }
35
36     int find_centroid(int u, int p, int sub){
37         for(int i=0; i<adj[u].size(); i++){
38             int v = adj[u][i];
39             if(v == p || removed[v]) continue;
40             if(subz[v] > subz[sub]/2){
41                 return find_centroid(v, u, sub);
42             }
43         }
44         return u;
45     }
46
47     void centroid_decomp(int u, int p, int r){
48         centroid_subz(u, -1);
49         int centroid = find_centroid(u, -1, u);
50         L[centroid] = r;

```

```

49     c_p[centroid] = p;
50     removed[centroid] = true;
51
52     //problem pre-processing
53
54     for(int i=0; i<adj[centroid].size(); i++){
55         int v = adj[centroid][i];
56         if(removed[v]) continue;
57         centroid_decomp(v, centroid, r+1);
58     }
59 }
60 };

```

5.6. Floyd Warshall Shortest Path

```

1  struct FloydWarshall {
2
3     vector< vector< vector<int> > > dist;
4
5     FloydWarshall() {}
6
7     FloydWarshall(int n){
8         dist.resize(n+1, vector< vector< int > >(n+1, vector<int>(n+1)));
9     }
10
11     void relax(int i, int j, int k){
12         dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j]);
13     }
14
15     void calculate(){
16         for(int k=0; k<dist.size(); k++){
17             for(int i=1; i<dist.size(); i++){
18                 for(int j=1; j<dist.size(); j++){
19                     if(i==j) dist[k][i][j] = 0;
20                     else dist[k][i][j] = INF;
21                 }
22             }
23         }
24         for(int k=1; k<dist.size(); k++){
25             for(int i=1; i<dist.size(); i++){
26                 for(int j=1; j<dist.size(); j++){
27                     relax(i, j, k);
28                 }
29             }
30         }
31     }
32
33 };

```

5.7. Tarjan's Bridge/Articulations Algorithm

```

1 //Graph - Tarjan Bridges Algorithm
2
3 //calculate bridges, articulations and all connected components
4
5 struct Tarjan{
6     int cont = 0;
7     vector<int> st;
8     vector<int> low;
9     vector<int> bridges;
10    vector<bool> isArticulation;
11
12    Tarjan() {}
13
14    Tarjan(int n){
15        st.resize(n+1);
16        low.resize(n+1);
17        isArticulation.resize(n+1);
18        cont = 0;
19        bridges.clear();
20    }
21
22    void calculate(int u, int p){
23        st[u] = low[u] = ++cont;
24        int son = 0;
25        for(int i=0; i<adj[u].size(); i++){
26            if(adj[u][i]==p){
27                p = 0;
28                continue;
29            }
30            if(!st[adj[u][i]]){
31                calculate(adj[u][i], u);
32                low[u] = min(low[u], low[adj[u][i]]);
33                if(low[adj[u][i]] >= st[u]) isArticulation[u] = true; //check
34                articulation
35
36                if(low[adj[u][i]] > st[u]){ //check if its a bridge
37                    bridges.push_back(ii(u, adj[u][i]));
38                }
39                son++;
40            }
41            else low[u] = min(low[u], st[adj[u][i]]);
42        }
43
44        if(p == -1){
45            if(son > 1) isArticulation[u] = true;
46            else isArticulation[u] = false;
47        }
48    }
49 };

```

5.8. Max Flow Dinic's Algorithm

```

1 struct Dinic {
2
3     struct FlowEdge{
4         int v, rev, c, cap;
5         FlowEdge() {}
6         FlowEdge(int v, int c, int cap, int rev) : v(v), c(c), cap(cap),
7             rev(rev) {}
8     };
9
10    vector< vector<FlowEdge> > adj;
11    vector<int> level, used;
12    int src, snk;
13    int sz;
14    int max_flow;
15    Dinic(){}
16    Dinic(int n){
17        src = 0;
18        snk = n+1;
19        adj.resize(n+2, vector< FlowEdge >());
20        level.resize(n+2);
21        used.resize(n+2);
22        sz = n+2;
23        max_flow = 0;
24    }
25
26    void add_edge(int u, int v, int c){
27        int id1 = adj[u].size();
28        int id2 = adj[v].size();
29        adj[u].pb(FlowEdge(v, c, c, id2));
30        adj[v].pb(FlowEdge(u, 0, 0, id1));
31    }
32
33    void add_to_src(int v, int c){
34        adj[src].pb(FlowEdge(v, c, c, -1));
35    }
36
37    void add_to_snk(int u, int c){
38        adj[u].pb(FlowEdge(snk, c, c, -1));
39    }
40
41    bool bfs(){
42        for(int i=0; i<sz; i++){
43            level[i] = -1;
44        }
45
46        level[src] = 0;
47        queue<int> q; q.push(src);
48
49        while(!q.empty()){
50            int cur = q.front();
51            q.pop();
52            for(FlowEdge e : adj[cur]){
53                if(level[e.v] == -1 && e.c > 0){
54                    level[e.v] = level[cur]+1;
55                    q.push(e.v);
56                }
57            }
58        }
59
60        return (level[snk] == -1 ? false : true);
61    }
62
63    int send_flow(int u, int flow){

```

```

63     if(u == snk) return flow;
64
65     for(int &i = used[u]; i<adj[u].size(); i++){
66         FlowEdge &e = adj[u][i];
67
68         if(level[u]+1 != level[e.v] || e.c <= 0) continue;
69
70         int new_flow = min(flow, e.c);
71         int adjusted_flow = send_flow(e.v, new_flow);
72
73         if(adjusted_flow > 0){
74             e.c -= adjusted_flow;
75             if(e.rev != -1) adj[e.v][e.rev].c += adjusted_flow;
76             return adjusted_flow;
77         }
78     }
79
80     return 0;
81 }
82
83 void calculate(){
84     if(src == snk){max_flow = -1; return;} //not sure if needed
85
86     max_flow = 0;
87
88     while(bfs()){
89         for(int i=0; i<sz; i++) used[i] = 0;
90         while(int inc = send_flow(src, INF)) max_flow += inc;
91     }
92 }
93
94 vector< ii > mincut(){
95     bool vis[sz];
96     for(int i=0; i<sz; i++) vis[i] = false;
97     queue<int> q;
98     q.push(src);
99     vis[src] = true;
100     while(!q.empty()){
101         int cur = q.front();
102         q.pop();
103         for(FlowEdge e : adj[cur]){
104             if(e.c > 0 && !vis[e.v]){
105                 q.push(e.v);
106                 vis[e.v] = true;
107             }
108         }
109     }
110     vector< ii > cut;
111     for(int i=1; i<=sz-2; i++){
112         if(!vis[i]) continue;
113         for(FlowEdge e : adj[i]){
114             if(1 <= e.v && e.v <= sz-2 && !vis[e.v] && e.cap > 0 && e.c == 0)
115                 cut.pb(ii(i, e.v));
116         }
117     }
118     return cut;
119 }
120
121 vector< ii > min_edge_cover(){
122     bool covered[sz];
123     for(int i=0; i<sz; i++) covered[i] = false;
124     vector< ii > edge_cover;
125     for(int i=1; i<sz-1; i++){
126         for(FlowEdge e : adj[i]){

```

```

127         if(e.cap == 0 || e.v > sz-2) continue;
128         if(e.c == 0){
129             edge_cover.pb(ii(i, e.v));
130             covered[i] = true;
131             covered[e.v] = true;
132             break;
133         }
134     }
135 }
136 for(int i=1; i<sz-1; i++){
137     for(FlowEdge e : adj[i]){
138         if(e.cap == 0 || e.v > sz-2) continue;
139         if(e.c == 0) continue;
140         if(!covered[i] || !covered[e.v]){
141             edge_cover.pb(ii(i, e.v));
142             covered[i] = true;
143             covered[e.v] = true;
144         }
145     }
146 }
147 return edge_cover;
148 }
149
150 };

```

5.9. HLD

```

1 struct HLD {
2
3     vector<int> L, vis, vis2, P, ch, subsize, st, ed, heavy;
4     int t = 0;
5
6     HLD () {}
7
8     HLD(int n){
9         L.resize(n+1);
10        vis.resize(n+1);
11        vis2.resize(n+1);
12        P.resize(n+1);
13        ch.resize(n+1);
14        subsize.resize(n+1);
15        st.resize(n+1);
16        ed.resize(n+1);
17        heavy.resize(n+1);
18        t = 0;
19        for(int i=0; i<=n; i++){
20            ch[i] = i;
21            P[i] = -1;
22            heavy[i] = -1;
23        }
24    }
25
26    void precalculate(int u){
27        vis[u] = true;
28        subsize[u] = 1;
29        for(int i=0; i<adj[u].size(); i++){
30            int v = adj[u][i];
31            if(vis[v]) continue;
32            P[v] = u;
33            L[v]=L[u]+1;
34            precalculate(v);
35            if(heavy[u] == -1 || subsize[heavy[u]] < subsize[v]) heavy[u] = v;
36            subsize[u]+=subsize[v];
37        }

```

```

38 }
39
40 void build(int u){
41     vis2[u] = true;
42     st[u]=t;
43     v[t++] = //segtree value
44     if(heavy[u] != -1){
45         ch[heavy[u]] = ch[u];
46         build(heavy[u]);
47     }
48     for(int i=0; i<adj[u].size(); i++){
49         int v = adj[u][i];
50         if(vis2[v] || v == heavy[u]) continue;
51         build(v);
52     }
53     ed[u] = t;
54     v[t++] = 0; //trick
55 }
56
57 void update(){
58     //update path / subtree / edge
59 }
60
61 int query(){
62     long long ans;
63     if(a == b) return 0;
64     while(ch[a] != ch[b]){
65         if(L[ch[b]] > L[ch[a]]) swap(a,b);
66         //query from st[ch[a]] to st[a]
67         a = P[ch[a]];
68     }
69     if(L[b] < L[a]) swap(b,a);
70     if(st[a]+1 <= st[b]) //query from st[a]+1 to st[b]
71     return ans;
72 }
73
74 };

```

5.10. LCA

```

1 struct LCA {
2
3     int tempo;
4     vector<int> st, ed, dad, anc[20];
5     vector<bool> vis;
6
7     void init(int n){
8         tempo = 0;
9         st.resize(n+1);
10        ed.resize(n+1);
11        dad.resize(n+1);
12        for(int i=0; i<20; i++) anc[i].resize(n+1);
13        vis.resize(n+1);
14        for(int i=0; i<=n; i++) vis[i] = false;
15    }
16
17    void dfs(int u){
18        vis[u] = true;
19        st[u] = tempo++;
20        for(int i=0; i<adj[u].size(); i++){
21            int v = adj[u][i];
22            if(!vis[v]){
23                dad[v] = u;
24                dfs(v);
25            }
26        }
27        ed[u] = tempo++;
28    }
29
30    bool is_ancestor(int u, int v){
31        return st[u] <= st[v] && st[v] <= ed[u];
32    }
33
34    int query(int u, int v){
35        if(is_ancestor(u,v)) return u;
36        for(int i=19; i>=0; i--){
37            if(anc[i][u] == -1) continue;
38            if(!is_ancestor(anc[i][u],v)) u = anc[i][u];
39        }
40        return dad[u];
41    }
42
43    void precalculate(){
44        dad[1] = -1;
45        dfs(1);
46        for(int i=1; i<st.size(); i++){
47            anc[0][i] = dad[i];
48        }
49        for(int i=1; i<20; i++){
50            for(int j=1; j<st.size(); j++){
51                if(anc[i-1][j] != -1){
52                    anc[i][j] = anc[i-1][anc[i-1][j]];
53                }
54                else {
55                    anc[i][j] = -1;
56                }
57            }
58        }
59    }
60
61 } lca;

```

6. Math and Number Theory

6.1. Diophantine Equations + CRT

```

1 namespace NT{
2
3     int GCD(int a, int b){
4         if(a == 0) return b;
5         else return GCD(b%a, a);
6     }
7
8     tuple<int,int> ExtendedEuclidean(int a, int b){
9         //solves ax+by = gcd(a,b)
10        //careful when a or b equal to 0
11        if(a == 0) return make_tuple(0,1);
12        int x,y;
13        tie(x,y) = ExtendedEuclidean(b%a, a);
14        return make_tuple(y - (b/a)*x, x);
15    }
16
17    bool FailDiophantine = false;
18
19    tuple<int,int> Diophantine(int a, int b, int c){
20        FailDiophantine = false;
21        //finds a solution for ax+by = c
22        //given a solution (x,y), all solutions have the form (x +
23        m*(b/gcd(a,b)), y - m*(a/(gcd(a,b))), multiplied by (c/g)
24
25        int g = GCD(a,b);
26
27        if(g == 0 || c%g != 0) {
28            FailDiophantine = true;
29            return make_tuple(0,0);
30        }
31
32        int x,y;
33
34        tie(x,y) = ExtendedEuclidean(a, b);
35
36        return make_tuple(x*(c/g), y*(c/g));
37    }
38
39    bool FailCRT = false;
40
41    tuple<int,int> CRT(vector<int> & a, vector<int> & n){
42        FailCRT = false;
43        for(int i=0; i<a.size(); i++) a[i] = mod(a[i], n[i]);
44        int ans = a[0];
45        int modulo = n[0];
46
47        for(int i=1; i<a.size(); i++){
48            int x,y;
49            tie(x,y) = ExtendedEuclidean(modulo, n[i]);
50            int g = GCD(modulo, n[i]);
51
52            if(g == 0 || (a[i] - ans)%g != 0){
53                FailCRT = true;
54                return make_tuple(0,0);
55            }
56
57            ans = mod(ans + (x*(a[i] - ans)/g)*(n[i]/g) * modulo, modulo*n[i]/g);
58            modulo = modulo*n[i]/g;
59        }
60        return make_tuple(ans, modulo);

```

```

61     }
62
63 }
```

6.2. Discrete Logarithm - Shanks Baby-Step Giant-Step

```

1  /* Baby-Step Giant-Step Shank's Algorithm */
2
3  namespace NT {
4
5      int discrete_log(int a, int b, int p){
6          a %= p, b %= p;
7
8          if(b == 1) return 0;
9
10         int cnt = 0, t = 1;
11         for(int g = gcd(a, p); g != 1; g = gcd(a, p)){
12             if(b % g) return -1;
13
14             p /= g, b /= g, t = t * a / g % p;
15             cnt++;
16
17             if(b == t) return cnt;
18         }
19
20         map<int, int> hash;
21         int m = (sqrt(p) + 1);
22         int base = b;
23
24         for(int i = 0; i != m; ++i){
25             hash[base] = i;
26             base = base * a % p;
27         }
28
29         base = 1;
30         for(int i=0; i<m; i++){
31             base = (base*a)%p;
32         }
33
34         int cur = t;
35         for(int i = 1; i <= m + 1; ++i){
36             cur = cur * base % p;
37             if(hash.count(cur)) return i * m - hash[cur] + cnt;
38         }
39         return -1;
40     }
41
42 }
```

6.3. Binomial Coefficient DP

```

1  /* Dynamic Programming for Binomial Coefficient Calculation */
2  /* Using Stiefel Rule  $C(n, k) = C(n-1, k) + C(n-1, k-1)$  */
3
4  int binomial(int n, int k){
5      int c[n+10][k + 10];
6      memset(c, 0, sizeof c);
7      c[0][0] = 1;
8      for(int i = 1; i<=n; i++){
9          for(int j = min(i, k); j>0; j--){
10             c[i][j] = c[i-1][j] + c[i-1][j-1];
11         }
12     }
13     return c[n][k];
14 }

```

6.4. Erathosthenes Sieve + Logn Prime Factorization

```

1  /* Erathosthenes Sieve Implementation + Euler's Totient */
2  /* Calculate primes from 2 to N */
3  /* lf[i] stores the lowest prime factor of i(logn factorization) */
4
5  namespace NT {
6
7      const int MAX_N = 1123456;
8
9      bitset<MAX_N> prime;
10     vector<int> primes;
11     int lf[MAX_N];
12     int totient[MAX_N];
13
14     void Sieve(int n){
15         for(int i=0; i<=n; i++) lf[i] = i;
16         prime.set();
17         prime[0] = false;
18         prime[1] = false;
19         for(int p = 2; p*p <= n; p++){
20             if(prime[p]){
21                 for(int i=p*p; i<=n; i+=p){
22                     prime[i] = false;
23                     lf[i] = min(lf[i], p);
24                 }
25             }
26         }
27         for(int i=2; i<=n; i++) if(prime[i]) primes.pb(i);
28     }
29
30     void EulerTotient(int n){
31         for(int i=0; i<=n; i++) totient[i] = i;
32         for(int p = 2; p <= n; p++){
33             if(totient[p] == p){
34                 totient[p] = p-1;
35                 for(int i=p*p; i<=n; i+=p){
36                     totient[i] = (totient[i]/p) * (p-1);
37                 }
38             }
39         }
40     }
41 }
42 };

```

6.5. Segmented Sieve

```

1  /* Segmented Erathosthenes Sieve */
2  /* Needs primes up to sqrt(N) - Use normal sieve to get them */
3
4  namespace NT {
5
6      bitset<MAX_N> prime;
7      vector<int> primes;
8      vector<int> seg_primes;
9
10     void Sieve(int n){
11         prime.set();
12         prime[0] = false;
13         prime[1] = false;
14         for(int p = 2; p*p <= n; p++){
15             if(prime[p]){
16                 for(int i=p*p; i<=n; i+=p){
17                     prime[i] = false;
18                 }
19             }
20         }
21         for(int i=2; i<=n; i++) if(prime[i]) primes.pb(i);
22     }
23
24     void SegmentedSieve(int l, int r){
25         prime.set();
26         seg_primes.clear();
27         for(int p : primes){
28             int start = l - l%p - p;
29             while(start < l) start += p;
30             if(p == start) start += p;
31             for(int i = start; i<=r; i+=p){
32                 prime[i-1] = false;
33             }
34         }
35         for(int i=0; i<r-l+1; i++){
36             if(prime[i]){
37                 seg_primes.pb(l+i);
38             }
39         }
40     }
41 }
42 }

```

6.6. Matrix Exponentiation

```

1  /* Matrix Exponentiation Implementation */
2
3  struct Matrix{
4      vector< vector<int> > m;
5      Matrix() {}
6      Matrix(int l, int c){
7          m.resize(l, vector<int>(c));
8      }
9
10     Matrix operator *(Matrix b) const{
11         Matrix c(m.size(), b.m[0].size());
12         for(int i = 0; i<m.size(); i++){
13             for(int j = 0; j<b.m[0].size(); j++){
14                 for(int k = 0; k<b.m.size(); k++){
15                     c.m[i][j] += (m[i][k]*b.m[k][j]);
16                 }
17             }
18         }
19         return c;
20     }
21
22     Matrix exp(int k){
23         if(k == 1) return *this;
24         Matrix c = (*this).exp(k/2);
25         c = c*c;
26         if(k%2) c = c*(*this);
27         return c;
28     }
29 };
30

```

6.7. Fast Fourier Transform - Recursive and Iterative

```

1  /* Fast Fourier Transform Implementation */
2  /* Complex numbers implemented by hand */
3  /* Poly needs to have degree of next power of 2 (result poly has size
   next_pot2(2*n) */
4  /* Uses Roots of Unity ( $Z^n = 1$ , divide and conquer strategy)
5  /* Inverse FFT only changes to the conjugate of Primitive Root of Unity */
6  /* Remember to use round to get integer value of Coefficients of Poly C */
7  /* Iterative FFT is way faster (bit reversal idea + straightforward conquer
   for each block of each size) */
8  /* std::complex doubles the execution time */
9
10 namespace FFT{
11
12     struct Complex{
13         double a, b;
14
15         Complex(double a, double b) : a(a), b(b) {}
16
17         Complex() : a(0), b(0) {}
18
19         Complex conjugate() const {
20             return Complex(a, -b);
21         }
22
23         double size2() const {
24             return a*a + b*b;
25         }
26
27         void operator=(const Complex & b){

```

```

28         this->a = b.a;
29         this->b = b.b;
30     }
31     Complex operator+(const Complex & y) const {
32         return Complex(a + y.a, b + y.b);
33     }
34     Complex operator-(const Complex & y) const {
35         return Complex(a - y.a, b - y.b);
36     }
37     Complex operator*(const Complex & y) const {
38         return Complex(a*y.a - b*y.b, a*y.b + b*y.a);
39     }
40     Complex operator/(const double & x) const {
41         return Complex(a/x, b/x);
42     }
43     Complex operator/(const Complex & y) const {
44         return (*this)*(y.conjugate()/y.size2());
45     }
46
47 };
48
49 struct Poly{
50     vector<Complex> c;
51     Poly() {}
52
53     Poly(int n){
54         int sz = (31 - __builtin_clz(n)%32) + 1;
55         c.resize((1 << (sz-1)) == n ? n : (1<<sz)<<1);
56     }
57
58     int size() const{
59         return (int)c.size();
60     }
61
62 };
63
64 inline Complex PrimitiveRootOfUnity(int n){
65     const double PI = acos(-1);
66     return Complex(cos(2*PI/(double)n), sin(2*PI/(double)n));
67 }
68
69 inline Complex InversePrimitiveRootOfUnity(int n){
70     const double PI = acos(-1);
71     return Complex(cos(-2*PI/(double)n), sin(-2*PI/(double)n));
72 }
73
74 void DFT(Poly & A, bool inverse){
75     int n = A.size();
76     int lg = 0;
77     while(n > 0) lg++, n>>=1;
78     n = A.size();
79     lg-=2;
80
81     for(int i=0; i<n; i++){
82         int j = 0;
83         for(int b=0; b <= lg; b++){
84             if(i & (1 << b)) j |= (1 << (lg - b));
85         }
86         if(i < j) swap(A.c[i], A.c[j]);
87     }
88
89     for(int len=2; len <= n; len <= 1){
90         Complex w;
91         if(inverse) w = InversePrimitiveRootOfUnity(len);
92         else w = PrimitiveRootOfUnity(len);

```

```

93     for(int i=0; i<n; i+=len){
94         Complex x(1,0);
95         for(int j=0; j<len/2; j++){
96             Complex u = A.c[i+j], v = x*A.c[i+j+len/2];
97             A.c[i+j] = u + v;
98             A.c[i+j+len/2] = u - v;
99             x = x*w;
100         }
101     }
102 }
103 }
104 }
105 if(inverse) for(int i=0; i<n; i++) A.c[i] = A.c[i]/n;
106 }
107
108 /* Skipable */
109 Poly RecursiveFFT(Poly A, int n, Complex w){
110     if(n == 1) return A;
111
112     Poly A_even(n/2), A_odd(n/2);
113
114     for(int i=0; i<n; i+=2){
115         A_even.c[i/2] = A.c[i];
116         A_odd.c[i/2] = A.c[i+1];
117     }
118
119     Poly F_even = RecursiveFFT(A_even, n/2, w*w);
120     Poly F_odd = RecursiveFFT(A_odd, n/2, w*w);
121     Poly F(n);
122     Complex x(1, 0);
123
124     for(int i=0; i<n/2; i++){
125         F.c[i] = F_even.c[i] + x*F_odd.c[i];
126         F.c[i + n/2] = F_even.c[i] - x*F_odd.c[i];
127         x = x*w;
128     }
129
130     return F;
131 }
132
133 /* Skipable */
134 Poly Convolution(Poly & F_A, Poly & F_B){
135     Poly F_C(F_A.size()>>1);
136     for(int i=0; i<F_A.size(); i++) F_C.c[i] = F_A.c[i]*F_B.c[i];
137     return F_C;
138 }
139
140 Poly multiply(Poly & A, Poly & B){
141     DFT(A, false);
142
143     DFT(B, false);
144
145     Poly C = Convolution(A, B);
146
147     DFT(C, true);
148
149     return C;
150 }
151 }
152 };

```

7. String Algorithms

7.1. KMP Failure Function + String Matching

```

1  /* Knuth - Morris - Pratt Algorithm */
2
3  struct KMP{
4      vector<int> pi;
5
6      vector<int> matches;
7
8      KMP() {}
9
10     void calculate(string t) {
11         int n = t.size();
12         pi.resize(n);
13         pi[0] = 0;
14         for(int i = 1; i < n; i++) {
15             pi[i] = pi[i-1];
16             while(pi[i] > 0 && t[i] != t[pi[i]]) pi[i] = pi[pi[i]-1];
17             if(t[i] == t[pi[i]]) pi[i]++;
18         }
19     }
20
21     void matching(string s){
22         int j = 0;
23         int n = s.size();
24         for(int i=0; i<n; i++){
25             while(j > 0 && s[i] != t[j]) j = pi[j-1];
26             if(s[i] == t[j]) j++;
27             if(j == t.size()){
28                 matches.push_back(i-t.size()+1);
29                 j = pi[j-1];
30             }
31         }
32     }
33 }
34 };

```


7.2. Z-Function

```

1  /* Z-function */
2  /* Calculate the size K of the largest substring which is a prefix */
3
4  struct ZFunction{
5
6      vector<int> z;
7
8      ZFunction() {}
9
10     void calculate(string t){
11         int n = t.size();
12         z.resize(n);
13         z[0] = 0;
14         int l = 0, r = 0;
15         for(int i=1; i<n; i++){
16             if(i > r){
17                 l = i;
18                 r = i;
19             }
20             z[i] = min(z[i-1], r-i+1);
21             while(i + z[i] < n && t[i + z[i]] == t[z[i]]) z[i]++;
22             if(i + z[i] > r){
23                 l = i;
24                 r = i + z[i]-1;
25             }
26         }
27     }
28 };
29

```

7.3. Suffix Array + Linear Sort

```

1  /* Suffix Array using Counting Sort Implementation */
2  /* rnk is inverse of sa array */
3  /* aux arrays are needed for sorting step */
4  /* inverse sorting (using rotating arrays and blocks of power of 2) */
5  /* rmq data structure needed for calculating lcp of two non adjacent
6     suffixes sorted */
7
8  struct SuffixArray{
9
10     vector<int> rnk,tmp,sa, sa_aux, lcp, pot, sp[22];
11
12     int block, n;
13
14     string s;
15
16     SuffixArray() {}
17
18     SuffixArray(string t){
19         s = t;
20         n = t.size();
21         rnk.resize(n+1);
22         for(int i=0; i<22; i++) sp[i].resize(n+1);
23         pot.resize(n+1);
24         tmp.resize(max(257LL, n+1));
25         sa.resize(n+1);
26         sa_aux.resize(n+1);
27         lcp.resize(n+1);
28         block = 0;
29     }
30

```

```

30 bool suffixcmp(int i, int j){
31     if(rnk[i] != rnk[j]) return rnk[i] < rnk[j];
32     i+=block, j+=block;
33     i%=n;
34     j%=n;
35     return rnk[i] < rnk[j];
36 }
37
38 void suffixSort(int MAX_VAL){
39     for(int i=0; i<=MAX_VAL; i++) tmp[i] = 0;
40     for(int i=0; i<n; i++) tmp[rnk[i]]++;
41     for(int i=1; i<=MAX_VAL; i++) tmp[i] += tmp[i-1];
42     for(int i = n-1; i>=0; i--){
43         int aux = sa[i]-block;
44         aux%=n;
45         if(aux < 0) aux+=n;
46         sa_aux[--tmp[rnk[aux]]] = aux;
47     }
48     for(int i=0; i<n; i++) sa[i] = sa_aux[i];
49     tmp[0] = 0;
50     for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
51     for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
52 }
53
54 void calculate(){
55     s+='\0';
56     n++;
57     for(int i=0; i<n; i++){
58         sa[i] = i;
59         rnk[i] = s[i];
60         tmp[i] = 0;
61     }
62     suffixSort(256);
63     block = 1;
64     while(tmp[n-1] != n-1){
65         suffixSort(tmp[n-1]);
66         block*=2;
67     }
68     for(int i=0; i<n-1; i++) sa[i] = sa[i+1];
69     n--;
70     tmp[0] = 0;
71     for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
72     for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
73     s.pop_back();
74     sa.pop_back();
75 }
76
77 void calculate_lcp(){
78     int last = 0;
79     for(int i=0; i<n; i++){
80         if(rnk[i] == n-1) continue;
81         int x = rnk[i];
82         lcp[x] = max(0LL, last-1);
83         while(sa[x] + lcp[x] < n && sa[x+1] + lcp[x] < n && s[sa[x]+lcp[x]] ==
84             s[sa[x+1]+lcp[x]]){
85             lcp[x]++;
86         }
87         last = lcp[x];
88     }
89 }
90
91 void build_lcp() {
92     int k = 0;
93     for(int j = 0; (1<<j) <= 2*n; j++) {
94         for(; k <= n && k < (1<<j); k++) {

```

```

94     pot[k] = j-1;
95 }
96 }
97 for(int i=0; i<n; i++){
98     sp[0][i] = lcp[i];
99 }
100 for(int i = 1; (1<<i) <= n; i++) {
101     for(int j = 0; j+(1<<i) <= n; j++) {
102         sp[i][j] = min(sp[i-1][j], sp[i-1][j+(1<<(i-1))]);
103     }
104 }
105 }
106
107 int query_lcp(int x, int y){
108     if(x == y) return n - x;
109     if(rnk[x] > rnk[y]) swap(x,y);
110     int l = rnk[x], r = rnk[y]-1;
111     return min(sp[pot[r-l+1]][l], sp[pot[r-l+1]][r-(1LL<<pot[r-l+1])+1]);
112 }
113
114 };

```

7.4. Rolling Hash

```

1 namespace Hash{
2
3     int B1, B2, M1, M2;
4
5     void init(){
6         B1 = rand()%65536;
7         B2 = rand()%65536;
8         M1 = 1000000007;
9         M2 = 1000000009;
10    }
11
12    struct RollingHash{
13
14        vector< ii > hash;
15        vector< ii > base;
16
17        RollingHash() {}
18
19        void calculate(string s){
20            int n = s.size();
21            hash.resize(n+1); base.resize(n+1);
22            base[0] = ii(1, 1);
23            hash[0] = ii(0, 0);
24            for(int i=1; i<=n; i++){
25                int val = (int)(s[i-1]);
26                base[i] = ii(mod(base[i-1].ff*B1, M1), mod(base[i-1].ss*B2, M2));
27                hash[i] = ii(mod(hash[i-1].ff*B1 + val, M1), mod(hash[i-1].ss*B2 +
28                    val, M2));
29            }
30
31            ii query(int l, int r){
32                ii ret;
33                ret.ff = mod(hash[r].ff - hash[l-1].ff*base[r-l+1].ff, M1);
34                ret.ss = mod(hash[r].ss - hash[l-1].ss*base[r-l+1].ss, M2);
35                return ret;
36            }
37
38        };
39
40    }

```