

Competitive Programming Algorithms and Topics

Ubiratan Neto

1	Template	2	5.3	Suffix Array + Linear Sort	15
1.1	Template Code	2	5.4	Rolling Hash	16
2	Data Structures	2	5.5	Aho-Corasick	16
2.1	Dynamic Segment Tree	2	5.6	Suffix Automata - Tested ??	17
2.2	Segment Tree	3	6	Geometry	19
2.3	Fenwick Tree	3	6.1	2D Structures	19
2.4	Trie	4	6.2	2D Geometry Functions	20
2.5	STL Ordered Set	4	6.3	Convex Hull - Monotone Chain Algorithm	25
2.6	Convex Hull Trick	4	6.4	Rotation in 3D	26
2.7	Lichao Segment Tree - Convex Hull Trick	5	7	Graphs	26
2.8	Lichao Segment Tree - Convex Hull Trick - Double Type	5	7.1	Dynammic Connectivity - connected(u,v) query	26
2.9	Sparse Table	6	7.2	Bellman Ford Shortest Path	27
3	Uncategorized	6	7.3	Eulerian Circuits/Paths	28
3.1	Coordinate Compression	6	7.4	Kosaraju SCC	28
3.2	Longest Increasing Subsequence	6	7.5	Centroid Decomposition	29
3.3	LIS 2D	7	7.6	Floyd Warshall Shortest Path	29
3.4	Inversion Count - Merge Sort	8	7.7	Tarjan's Bridge/Articulations Algorithm	30
3.5	Mo's Decomposition	8	7.8	Max Flow Dinic's Algorithm	30
4	Math and Number Theory	8	7.9	HLD	31
4.1	Binomial Coefficient DP	8	7.10	LCA	33
4.2	Discrete Logarithm - Shanks Baby-Step Giant-Step	9	7.11	Stoer-Wagner Minimum Cut in Undirected Graphs	34
4.3	Erathostenes Sieve + Logn Prime Factorization	9	7.12	Block-Cut Tree	34
4.4	Diophantine Equations + CRT	10	7.13	Dominator Tree	35
4.5	Segmented Sieve	11	7.14	Minimum Path Cover Problem On DAG's	35
4.6	Matrix Exponentiation	11			
4.7	Fast Fourier Transform - Recursive and Iterative	11			
4.8	Count Divisors in $\text{cbrt}(n)$	13			
4.9	Count Prime Factors in $\text{cbrt}(n)$	13			
4.10	Miller Rabin Primality Test	13			
4.11	Sum of Divisors in a Range	14			
5	String Algorithms	14			
5.1	KMP Failure Function + String Matching	14			
5.2	Z-Function	14			

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 mp make_pair
10 #define mt make_tuple
11 #define DESYNC ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0)
12 #define pb push_back
13 #define vi vector<int>
14 #define vii vector< ii >
15 #define EPS 1e-9
16 #define INF 1e18
17 #define ROOT 1
18 #define M 1000000007
19 const double PI = acos(-1);
20
21 using namespace std;
22
23 inline int mod(int n, int m){ int ret = n%m; if(ret < 0) ret += m; return
    ret; }
24
25 int gcd(int a, int b){
26     if(a == 0 || b == 0) return 0;
27     else return abs(__gcd(a,b));
28 }
29
30 int32_t main(){
31     DESYNC;
32
33 }
```

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         Tree(int n){
13             st.resize(4*n);
14         }
15         node combine(node a, node b){
16             node res;
17             //combine operations
18             return res;
19         }
20         void propagate(int cur, int l, int r){
21             //return if there is no update
22             //update tree using lazy node
23             if(l != r){
24                 //propagate for left and right child
25             }
26             //reset lazy node
27         }
28         void build(int cur, int l, int r){
29             if(l == r){
30                 //leaf operation
31                 return;
32             }
33             int m = (l+r)>>1;
34             build(2*cur, l, m);
35             build(2*cur + 1, m+1, r);
36             st[cur] = combine(st[2*cur], st[2*cur+1]);
37         }
38
39         void range_update(int cur, int l, int r, int a, int b, long long val){
40             propagate(cur, l, r);
41             if(l == a && r == b){
42                 //lazy operation using val
43                 return;
44             }
45
46             int m = (l+r)/2;
47
48             if(b <= m) range_update(2*cur, l, m, a, b, val);
49             else if(m < a) range_update(2*cur+1, m+1, r, a, b, val);
50             else {
51                 range_update(2*cur, l, m, a, m, val);
52                 range_update(2*cur+1, m+1, r, m+1, b, val);
53             }
54
55             propagate(cur, l, r);
56             propagate(2*cur, l, m);
57             propagate(2*cur+1, m+1, r);
58             st[cur] = combine(st[2*cur], st[2*cur+1]);
59         }
60
61         node query(int cur, int l, int r, int a, int b){
62             propagate(cur, l, r);
63             if(l == a && r == b) return st[cur];

```

```

64
65         int m = (l+r)/2;
66         if(b <= m) return query(2*cur, l, m, a, b);
67         else if(m < a) return query(2*cur+1, m+1, r, a, b);
68         else {
69             node left = query(2*cur, l, m, a, m);
70             node right = query(2*cur+1, m+1, r, m+1, b);
71             node ans = combine(left, right);
72             return ans;
73         }
74     }
75 };
76
77 }

```

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        int query(){
23            node *cur = t;
24        }
25
26        void remove(){
27            node *cur = t;
28        }
29
30    };
31 }

```

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

```

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 }

```

2.9. Sparse Table

```

1  int spt[MAXN][LOGN];
2  int e[MAXN];
3
4  void spt_build(int *a, int n) {
5      for(int i = 0; i < n; i++) {
6          spt[i][0] = a[i];
7      }
8
9      for(int i = 1; (1<<i) <= n; i++) {
10         for(int j = 0; j+(1<<i) <= n; j++) {
11             spt[j][i] = min(spt[j][i-1], spt[j+(1<<(i-1))][i-1]);
12         }
13     }
14
15     int k = 0;
16     for(int j = 0; (1<<j) <= 2*n; j++) {
17         for(; k <= n && k < (1<<j); k++) {
18             e[k] = j-1;
19         }
20     }
21 }
22
23 int spt_rmq(int l, int r) {
24     return min(spt[l][e[sz]], spt[r-(1<<e[sz])+1][e[sz]]);
25 }

```

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      vector<int> ii > pointer;
7      int sz;
8      LIS() {}
9
10     LIS(int n){
11         seq.resize(n+1);
12         pointer.resize(n);
13     }
14
15     void calculate(vector<int> & v){
16         int n = v.size();
17         vector<int> aux(n+1);
18         for(int i=1; i<=n; i++){
19             seq[i] = INT_MAX;
20             aux[i] = -1;
21         }
22         seq[0] = INT_MIN;
23         aux[0] = -1;
24         for(int i=0; i<n; i++){
25             int index = lower_bound(seq.begin(), seq.end(), v[i]) - seq.begin();
26             index--;
27             if(seq[index+1] > v[i]){
28                 seq[index+1] = min(seq[index+1], v[i]);
29                 aux[index+1] = i;
30             }
31             pointer[i] = ii(index+1, aux[index]);
32         }
33         for(int i=n; i>=0; i--){
34             if(seq[i] != INT_MAX){
35                 sz = i;
36                 break;
37             }
38         }
39     }
40 };

```

3.3. LIS 2D

```

1 struct LIS2D{
2
3     struct node {
4         node *left, *right;
5         int mx = (int)1e18+1;
6         node() {
7             mx = (int)1e18+1;
8             left = NULL;
9             right = NULL;
10        }
11    };
12
13    LIS2D() {}
14
15    vector<node*> lis;
16    int L,R,size;
17
18    void combine(node *ans, node *left, node *right){
19        if(left && right) ans->mx = min(left->mx, right->mx);
20        else if(left) ans->mx = left->mx;
21        else if(right) ans->mx = right->mx;
22        else ans->mx = (int)1e18+1;
23    }
24
25    void update(node *root, int l, int r, int idx, int val){
26        if(l == r){
27            root->mx = min(root->mx, val);
28            return;
29        }
30        int m = (l+r) >> 1;
31        if(idx <= m){
32            if(!root->left) root->left = new node();
33            update(root->left, l, m, idx, val);
34        }
35        else{
36            if(!root->right) root->right = new node();
37            update(root->right, m+1, r, idx, val);
38        }
39        combine(root, root->left, root->right);
40    }
41
42    int query(node *root, int l, int r, int a, int b){
43        if(l == a && r == b){
44            return root->mx;
45        }
46        int m = (l+r) >> 1;
47        if(b <= m){
48            if(!root->left) return (int)1e18+1;
49            else return query(root->left, l, m, a, b);
50        }
51        else if(m < a){
52            if(!root->right) return (int)1e18+1;
53            else return query(root->right, m+1, r, a, b);
54        }
55        int left = (int)1e18+1;
56        int right = (int)1e18+1;
57        if(root->left) left = query(root->left, l,m,a,m);
58        if(root->right) right = query(root->right, m+1, r, m+1, b);
59        return min(left, right);
60    }
61
62    bool check(int id, int x, int y){
63        int val = query(lis[id], L, R, L, x-1);

```

```

64        return val < y;
65    }
66
67    void calculate(vector< ii > & v){
68        int n = v.size();
69        lis.resize(n+1);
70        set<int> sx;
71        vector<int> aux;
72        for(int i=0; i<n; i++){
73            sx.insert(v[i].ff);
74        }
75        for(int x : sx) aux.pb(x);
76        L = -1, R = sx.size();
77        for(int i=0; i<n; i++){
78            v[i].ff = lower_bound(aux.begin(), aux.end(), v[i].ff) - aux.begin();
79        }
80        for(int i=0; i<=n; i++){
81            lis[i] = new node();
82        }
83        update(lis[0], L, R, L, -(int)1e18-1);
84        size = 0;
85        for(ii par : v){
86            int x = par.ff, y = par.ss;
87            int l = 0, r = n-1;
88            int ans = 0;
89            while(l <= r){
90                int m = (l+r)>>1;
91                if(check(m, x, y)){
92                    ans = m;
93                    l = m+1;
94                }
95                else r = m-1;
96            }
97            size = max(size, ans+1);
98            update(lis[ans+1], L, R, x, y);
99        }
100    }
101
102    };
103
104    int32_t main(){
105        int n;
106        scanf("%d", &n);
107        vector< ii > v(n);
108        set<int> sx;
109        vector<int> aux;
110        for(int i=0; i<n; i++){
111            scanf("%d%d", &v[i].ff, &v[i].ss);
112        }
113        LIS2D lis2d;
114        lis2d.calculate(v);
115        printf("%d\n", lis2d.size);
116    }

```

3.4. 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 }
```

3.5. 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. Math and Number Theory

4.1. 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 }
```


4.2. 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 }

```

4.3. 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 };

```

4.4. 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     int s1 = x*(c/g), s2 = y*(c/g);
36
37     //shifts solution
38     int l = 0, r = 1e9;
39     int ans = -1;
40     while(l <= r){
41         int m = (l+r)>>1;
42         if(s2 + m*(a/g) >= 0){
43             ans = m;
44             r = m-1;
45         }
46         else l = m+1;
47     }
48     if(ans != -1){
49         s1 = s1 - ans*(b/g);
50         s2 = s2 + ans*(a/g);
51     }
52
53     l = 0, r = 1e9;
54     ans = -1;
55     while(l <= r){
56         int m = (l+r)>>1;
57         if(s1 + m*(a/g) >= 0){
58             ans = m;
59             r = m-1;
60         }
61         else l = m+1;
62     }
63     if(ans != -1){

```

```

63     s1 = s1 + ans*(b/g);
64     s2 = s2 - ans*(a/g);
65 }
66
67 l = 0, r = 1e9;
68 ans = -1;
69 while(l <= r){
70     int m = (l+r)>>1;
71     if(s1 - m*(a/g) <= s2 + m*(b/g)){
72         ans = m;
73         r = m-1;
74     }
75     else l = m+1;
76 }
77 if(ans != -1){
78     s1 = s1 - ans*(b/g);
79     s2 = s2 + ans*(a/g);
80 }
81
82 return make_tuple(s1, s2);
83 }
84
85 bool FailCRT = false;
86
87 tuple<int,int> CRT(vector<int> & a, vector<int> & n){
88     FailCRT = false;
89     for(int i=0; i<a.size(); i++) a[i] = mod(a[i], n[i]);
90     int ans = a[0];
91     int modulo = n[0];
92
93     for(int i=1; i<a.size(); i++){
94         int x,y;
95         tie(x,y) = ExtendedEuclidean(modulo, n[i]);
96         int g = GCD(modulo, n[i]);
97
98         if(g == 0 || (a[i] - ans)%g != 0){
99             FailCRT = true;
100             return make_tuple(0,0);
101         }
102
103         ans = mod(ans + (x*(a[i] - ans)/g)%(n[i]/g) * modulo, modulo*n[i]/g);
104         modulo = modulo*n[i]/g;
105     }
106
107     return make_tuple(ans, modulo);
108 }
109
110 }

```

4.5. Segmented Sieve

```

1  /* Segmented Erathostenes Sieve */
2  /* Needs primes up to sqrt(N) - Use normal sieve to get them */
3
4  namespace NT {
5
6      const int MAX_N = 1123456;
7      bitset<MAX_N> prime;
8      vector<int> primes;
9      vector<int> seg_primes;
10
11     void Sieve(int n){
12         prime.set();
13         prime[0] = false;
14         prime[1] = false;
15         for(int p = 2; p*p <= n; p++){
16             if(prime[p]){
17                 for(int i=p*p; i<=n; i+=p){
18                     prime[i] = false;
19                 }
20             }
21         }
22         for(int i=2; i<=n; i++) if(prime[i]) primes.pb(i);
23     }
24
25     void SegmentedSieve(int l, int r){
26         prime.set();
27         seg_primes.clear();
28         for(int p : primes){
29             int start = l - l%p - p;
30             while(start < l) start += p;
31             if(p == start) start += p;
32             for(int i = start; i<=r; i+=p){
33                 prime[i-l] = false;
34             }
35         }
36         for(int i=0; i<r-l+1; i++){
37             if(prime[i] && l+i > 1){
38                 seg_primes.pb(l+i);
39             }
40         }
41     }
42 }
43

```

4.6. 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

```

4.7. 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
4      next_pot2(2*n) */
5  /* Uses Roots of Unity (Z^n = 1, divide and conquer strategy)
6  /* Inverse FFT only changes to the conjugate of Primitive Root of Unity */
7  /* Remember to use round to get integer value of Coefficients of Poly C */
8  /* Iterative FFT is way faster (bit reversal idea + straightforward conquer
9      for each block of each size) */
10 /* std::complex doubles the execution time */
11
12 namespace FFT{
13
14     struct Complex{
15         double a, b;
16
17         Complex(double a, double b) : a(a), b(b) {}
18
19         Complex() : a(0), b(0) {}
20
21         Complex conjugate() const {
22             return Complex(a, -b);
23         }
24
25         double size2() const {
26             return a*a + b*b;
27         }
28
29         void operator=(const Complex & b){
30

```

```

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 struct Poly{
49     vector<Complex> c;
50     Poly() {}
51
52     Poly(int n){
53         int sz = (31 - __builtin_clz(n)%32) + 1;
54         c.resize((1 << (sz-1)) == n ? n : (1<<sz)<<1);
55     }
56
57     int size() const{
58         return (int)c.size();
59     }
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
94     for(int i=0; i<n; i+=len){
95         Complex x(1,0);
96         for(int j=0; j<len/2; j++){
97             Complex u = A.c[i+j], v = x*A.c[i+j+len/2];
98             A.c[i+j] = u + v;
99             A.c[i+j+len/2] = u - v;
100             x = x*w;
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 /* Skipable */
133
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 };

```

4.8. Count Divisors in $\text{cbt}(n)$

```

1 namespace NT{
2
3   int CountDivisors(int x){
4
5       int ans = 1;
6       for(int i=2; i*i*i <= x; i++){
7           int cnt = 1;
8           while(x%i == 0){
9               cnt++;
10              x/=i;
11          }
12          ans*=cnt;
13      }
14
15      if(PrimalityTest(x,15)) ans*=2;
16      else if((int)sqrt(x)*(int)sqrt(x) == x && PrimalityTest((int)sqrt(x),
17          15)) ans*=3;
18      else if(x != 1) ans*=4;
19
20      return ans;
21  }
22 }

```

4.9. Count Prime Factors in $\text{cbt}(n)$

```

1 namespace NT{
2
3   int CountPrimeFactors(int x){
4
5       int ans = 0;
6       for(int i=2; i*i*i <= x; i++){
7           while(x%i == 0){
8               ans++;
9               x/=i;
10          }
11      }
12
13      if(PrimalityTest(x, 10)) ans++;
14      else if((int)sqrt(x)*(int)sqrt(x) == x && PrimalityTest((int)sqrt(x),
15          10)) ans+=2;
16      else if(x != 1) ans+=2;
17
18      return ans;
19  }
20 }
21

```

4.10. Miller Rabin Primality Test

```

1 namespace NT{
2
3   int mulmod(int a, int b, int c){
4       int x = 0,y=a%c;
5
6       while(b > 0){
7
8           if(b%2 == 1){
9               x = (x+y)%c;
10          }
11
12          y = (y*2)%c;
13          b /= 2;
14      }
15
16      return x%c;
17  }
18
19  int expmod(int a, int k, int p){
20      if(k == 0) return 1;
21      if(k == 1) return a;
22
23      int aux = expmod(a, k/2, p);
24      aux = mulmod(aux, aux, p);
25
26      if(k%2) aux = mulmod(aux, a, p);
27      return aux;
28  }
29
30  bool PrimalityTest(int p, int iterations){
31      //Miller Rabin Primality Test
32      mt19937 mt_rand(time(0));
33
34      if(p < 2) return false;
35      if(p == 2) return true;
36      if(p%2 == 0) return false;
37
38      int fixed_s = p-1;
39      while(fixed_s%2 == 0) fixed_s /= 2;
40
41      for(int iter = 0; iter < iterations; iter++){
42
43          int s = fixed_s;
44
45          int a = mt_rand()%(p-1) + 1;
46          int b = expmod(a, s, p);
47
48          while(s != p-1 && b != 1 && b != p-1){
49              b = mulmod(b,b,p);
50              s *= 2;
51          }
52
53          if(b != p-1 && s%2 == 0) return false;
54      }
55
56      return true;
57  }
58
59 }
60
61 }

```

4.11. Sum of Divisors in a Range

```

1 namespace NT{
2
3     int SumOfDivisors(int a, int b){
4         int m = sqrt(b);
5         int s = 0;
6         for (int f = 1; f <= m; f++){
7             int x = (b/f)*(b/f) - max(m, (a-1)/f)*max(m, (a-1)/f) + (b/f) - max(m,
8                 (a-1)/f);
9             s += f *(b/f - (a-1)/f);
10            s += x/2;
11        }
12        return s;
13    }
14 }

```

5. String Algorithms

5.1. 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 };

```

5.2. 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

```

5.3. 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
   suffixes sorted */
6
7  struct SuffixArray{
8
9      vector<int> rnk,tmp,sa, sa_aux, lcp, pot, sp[22];
10
11      int block, n;
12
13      string s;
14
15      SuffixArray() {}
16
17      SuffixArray(string t){
18          s = t;
19          n = t.size();
20          rnk.resize(n+1);
21          for(int i=0; i<22; i++) sp[i].resize(n+1);
22          pot.resize(n+1);
23          tmp.resize(max(257LL, n+1));
24          sa.resize(n+1);
25          sa_aux.resize(n+1);
26          lcp.resize(n+1);
27          block = 0;
28      }
29
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]] ==
s[sa[x+1]+lcp[x]]){
84              lcp[x]++;
85          }
86          last = lcp[x];
87      }
88  }
89
90  void build_lcp_table() {
91      int k = 0;
92      for(int j = 0; (1<<j) <= 2*n; j++) {
93          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  int number_of_substrings(){
115      int ans = n - sa[0];
116      for(int i=0; i<n-1; i++){
117          int length = n - sa[i+1];
118          ans += length - lcp[i];
119      }
120      return ans;
121  }
122
123  };

```

5.4. 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 }

```

5.5. Aho-Corasick

```

1 map<char, int> *nxt;
2 int *slinks;
3 vector<int> *dlinks;
4
5 void aho_build(const vector<string>& words) {
6     int len_words = 1;
7     for(const string& w : words) {
8         len_words += w.size();
9     }
10    nxt = new map<char, int>[len_words];
11    dlinks = new vector<int>[len_words];
12    int root = 0, fre = 1;
13    for(int i = 0; i < words.size(); i++) {
14        const string& w = words[i];
15        int cur = root;
16        for(const char& c : w) {
17            if(nxt[cur].count(c)==0) {
18                nxt[cur][c] = fre++;

```

```

19            }
20            cur = nxt[cur][c];
21        }
22        dlinks[cur].push_back(i);
23    }
24
25    slinks = new int[len_words];
26    slinks[0] = -1;
27    queue<int> q;
28    for(const pair<char, int>& ch : nxt[root]) {
29        slinks[ch.second] = root;
30        q.push(ch.second);
31    }
32    while(!q.empty()) {
33        const int cur = q.front();
34        q.pop();
35        for(const pair<char, int>& ch : nxt[cur]) {
36            int sl = slinks[cur];
37            while(sl != root && nxt[sl].count(ch.first) == 0)
38                sl = slinks[sl];
39            if(nxt[sl].count(ch.first) != 0)
40                sl = nxt[sl][ch.first];
41            slinks[ch.second] = sl;
42            copy(dlinks[sl].begin(), dlinks[sl].end(),
43                back_inserter(dlinks[ch.second]));
44            q.push(ch.second);
45        }
46    }
47    vector< vector<int> > aho_matches(const vector<string>& words, const string&
48        text) {
49        int root = 0;
50        int cur = root;
51        vector< vector<int> > matches(text.size());
52        // vector< vector<int> > matches(words.size());
53        for(int i = 0; i < text.size(); i++) {
54            while(cur != root && nxt[cur].count(text[i]) == 0)
55                cur = slinks[cur];
56            if(nxt[cur].count(text[i]) != 0)
57                cur = nxt[cur][text[i]];
58
59            // returns matching words per position in text
60            for(int w_id : dlinks[cur]) {
61                matches[i-words[w_id].size()+1].push_back(w_id);
62            }
63
64            // // returns matching positions per word
65            // for(int w_id : dlinks[cur]) {
66            //     matches[w_id].push_back(i-words[w_id].size()+1);
67            // }
68        }
69        return matches;
70    }
71
72    int32_t main() {
73        vector<string> words;
74        words.push_back("he");
75        words.push_back("hers");
76        words.push_back("his");
77        words.push_back("she");
78        string text = "heishers sheishis hihershe!";
79        aho_build(words);
80        vector< vector<int> > matches = aho_matches(words, text);
81        for(int i = 0; i < matches.size(); i++) {

```



```

82     cout << i;
83     for(int id : matches[i]) {
84         cout << " " << words[id];
85     }
86     cout << endl;
87 }
88 // for(int i = 0; i < matches.size(); i++) {
89 //     cout << words[i];
90 //     for(int p : matches[i]) {
91 //         cout << " " << p;
92 //     }
93 //     cout << endl;
94 // }
95 }

```

5.6. Suffix Automata - Tested ??

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 struct SuffixAutomaton {
6     vector< map<char, int> > nxt;
7     vector<int> slink;
8     vector<int> len;
9     int lstr;
10    int root;
11    vector<bool> is_terminal;
12    int slen;
13    vector<vector<int>>> slink_tree;
14    //vector<int> terminals;
15
16    SuffixAutomaton(const string& s) {
17        slen = s.size();
18        // add root
19        nxt.push_back(map<char, int>());
20        len.push_back(0);
21        slink.push_back(-1);
22        is_terminal.push_back(false);
23        lstr = root = 0;
24
25        for(int i = 0; i < s.size(); i++) {
26            // add r
27            nxt.push_back(map<char, int>());
28            len.push_back(i+1);
29            slink.push_back(0);
30            is_terminal.push_back(false);
31            int r = nxt.size()-1;
32
33            // Find p (longest suffix of last r with edge with new character)
34            int p = lstr;
35            while(p >= 0 && nxt[p].count(s[i]) == 0) {
36                // Add edge with new character
37                nxt[p][s[i]] = r;
38                p = slink[p];
39            }
40            if(p != -1) {
41                // There is an suffix of last r that has edge to new character
42                int q = nxt[p][s[i]];
43                if(len[p] + 1 == len[q]) {
44                    // the longest suffix of new r is the logest of class q
45                    // There is no need to split
46                    slink[r] = q;
47                } else {

```

```

48                // Need to split
49                // Add q'. New class that longest sufux of r and q.
50                nxt.push_back(nxt[q]); // Copy from q
51                len.push_back(len[p]+1);
52                slink.push_back(slink[q]); // Copy from q
53                is_terminal.push_back(false);
54                int ql = nxt.size()-1;
55
56                slink[q] = ql;
57                slink[r] = ql;
58
59                // q' will have every suffix of p that was previously conected to q
60                while(p >= 0 && nxt[p][s[i]] == q) {
61                    nxt[p][s[i]] = ql;
62                    p = slink[p];
63                }
64            }
65        }
66        lstr = r;
67        if(i == s.size()-1) {
68            p = r;
69            while(p >= 0) {
70                //terminals.push_back(p);
71                is_terminal[p] = true;
72                p = slink[p];
73            }
74        }
75    }
76
77    bool is_substr(const string& s) {
78        int cur = root;
79        for(int i = 0; i < s.size(); i++) {
80            if(nxt[cur].count(s[i]) == 0) return false;
81            cur = nxt[cur][s[i]];
82        }
83        return true;
84    }
85
86    bool is_suffix(const string& s) {
87        int cur = root;
88        for(int i = 0; i < s.size(); i++) {
89            if(nxt[cur].count(s[i]) == 0) return false;
90            cur = nxt[cur][s[i]];
91        }
92        if(is_terminal[cur]) return true;
93        return false;
94    }
95
96    void dfs_num_substr(int v, int *dp) {
97        dp[v] = 1;
98        for(pair<char, int> ad : nxt[v]) {
99            if(dp[ad.second]==-1)
100                dfs_num_substr(ad.second, dp);
101            dp[v] += dp[ad.second];
102        }
103    }
104
105    int num_substr() {
106        int dp[nxt.size()];
107        memset(dp, -1, sizeof dp);
108        dfs_num_substr(root, dp);
109        return dp[root]-1; // Remove empty substring
110    }
111
112    void dfs_num_matches(int v, int *dp) {
113        dp[v] = 0;

```

```

113     if(is_terminal[v]) dp[v] = 1;
114     for(pair<char, int> ad : nxt[v]) {
115         if(dp[ad.second] == -1)
116             dfs_num_matches(ad.second, dp);
117         dp[v] += dp[ad.second];
118     }
119 }
120 int num_matches(const string& s) {
121     int cur = root;
122     for(int i = 0; i < s.size(); i++) {
123         if(nxt[cur].count(s[i]) == 0) return 0;
124         cur = nxt[cur][s[i]];
125     }
126     int dp[nxt.size()];
127     memset(dp, -1, sizeof dp);
128     dfs_num_matches(cur, dp);
129     return dp[cur];
130 }
131
132 void dfs_first_match(int v, int *dp) {
133     dp[v] = 0;
134     if(is_terminal[v]) dp[v] = 1;
135     for(pair<char, int> ad : nxt[v]) {
136         if(dp[ad.second] == -1) {
137             dfs_first_match(ad.second, dp);
138             dp[v] = max(dp[v], dp[ad.second]+1);
139         }
140     }
141 }
142 int first_match(const string& s) {
143     int cur = root;
144     for(int i = 0; i < s.size(); i++) {
145         if(nxt[cur].count(s[i]) == 0) return -1;
146         cur = nxt[cur][s[i]];
147     }
148     int dp[nxt.size()];
149     memset(dp, -1, sizeof dp);
150     dfs_first_match(cur, dp);
151     return slen-(dp[cur]-1)-s.size();
152 }
153
154 void dfs_all_matches(int v, vector<int>& ans) {
155     //cout << v << endl;
156     if(slink_tree[v].size()==0)
157         ans.push_back(len[v]);
158     for(int ad : slink_tree[v]) {
159         dfs_all_matches(ad, ans);
160     }
161 }
162 vector<int> all_matches(const string& s) {
163     slink_tree = vector<vector<int>>> (slink.size());
164     for(int i=0;i<slink.size();i++) {
165         if(slink[i] >= 0) slink_tree[slink[i]].push_back(i);
166     }
167     int cur = root;
168     for(int i = 0; i < s.size(); i++) {
169         if(nxt[cur].count(s[i]) == 0) return vector<int>();
170         cur = nxt[cur][s[i]];
171     }
172     vector<int> ans;
173     dfs_all_matches(cur, ans);
174     for(int i = 0; i < ans.size(); i++) {
175         ans[i] -= s.size();
176     }
177     // Last one is not valid

```

```

178     return ans;
179 }
180 };
181
182 int main() {
183     string s;
184     cin >> s;
185     SuffixAutomaton sa(s);
186     cout << sa.num_substr() << endl;
187     // cout << sa.terminals.size() << endl;
188     // for(int ter : sa.terminals) cout << ter << " ";
189     // cout << endl;
190     int T;
191     cin >> T;
192     string w;
193     while(T--) {
194         cin >> w;
195         cout << sa.is_substr(w) << endl;
196         cout << sa.is_suffix(w) << endl;
197         cout << sa.num_matches(w) << endl;
198         cout << sa.first_match(w) << endl;
199         SuffixAutomaton sb(s+"$"+w);
200         vector<int> matches = sb.all_matches(w);
201         for(int i : matches) cout << i << " ";
202         cout << endl;
203     }
204 }

```

6. Geometry

6.1. 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 struct Circle{
148     Point c;
149     double r;
150     Circle() {}
151     Circle(Point center, double radius) : c(center), r(radius) {}
152
153     bool operator=(Circle circ){
154         c = circ.c;
155         r = circ.r;
156     }
157
158     pair<Point, Point> getTangentPoints(Point p){
159         //p needs to be outside the circle
160         double d = p.distanceTo(c);
161         double ang = asin(l.*r/d);
162         Point v1(p, c);
163         v1.rotate(ang);
164         Point v2(p, c);
165         v2.rotate(-ang);
166         v1 = v1.scale(sqrt(d*d - r*r)/d);
167         v2 = v2.scale(sqrt(d*d - r*r)/d);
168         Point p1(v1.x + p.x, v1.y + p.y);
169         Point p2(v2.x + p.x, v2.y + p.y);
170         return make_pair(p1,p2);
171     }
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
////////////////////////////////////

```

6.2. 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.*abs(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 << " " <<
        ansr.y << endl;
61         return Point(INF, INF);
62     }
63 }
64 else if(abs(s1.v^v1) <= EPS){
65     Point ans1, ansr;
66     if(s1.p.x <= s2.p.x) ans1 = s2.p;
67     else ans1 = s1.p;
68     if(s2.q.x <= s1.q.x) ansr = s2.q;
69     else ansr = s1.q;
70     if(ans1.x == ansr.x && ans1.y == ansr.y){
71         //cout << ansr.x << " " << ansr.y << endl;
72         return Point(ansr.x, ansr.y);
73     }
74     else {
75         if(ans1.x == ansr.x && ans1.y > ansr.y) swap(ans1, ansr);
76         //cout << ans1.x << " " << ans1.y << endl << ansr.x << " " <<
        ansr.y << endl;
77         return Point(INF, INF);
78     }
79 }
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 double ans = distancePointSegment(l1.p, l2);
170 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
279     return ans;
280 }
281
282 double circleCircleIntersection(Circle c1, Circle c2){
283
284     if((c1.r+c2.r)*(c1.r+c2.r) <= (c2.c.x-c1.c.x)*(c2.c.x-c1.c.x) +
285 (c2.c.y-c1.c.y)*(c2.c.y-c1.c.y)){
286         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) +
(c2.c.y-c1.c.y)*(c2.c.y-c1.c.y)){
288         return PI*min(c1.r, c2.r)*min(c1.r, c2.r);
289     }
290     double x1 = c1.c.x, x2 = c2.c.x, y1 = c1.c.y, y2 = c2.c.y, r1 = c1.r, r2
= c2.r;
291     double d = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
292     double r1sqr = c1.r*c1.r;
293     double r2sqr = c2.r*c2.r;
294     double dsqr = d*d;
295
296     double alpha1 = acos(((c1.r + c2.r)*(c1.r - c2.r) + dsqr)/(2.*d*r1));
297     double alpha2 = acos(((c2.r + c1.r)*(c2.r - c1.r) + dsqr)/(2.*d*r2));
298     double areal = r1sqr*(alpha1 - sin(alpha1)*cos(alpha1));
299     double area2 = r2sqr*(alpha2 - sin(alpha2)*cos(alpha2));
300
301     return areal + area2;
302 }
303
304 vector<Point> intersectionLineCircle(Line l, Circle circ){
305     //NOT TESTED!!!!!!
306     //no intersection
307     if((l.c*x1.c)/(circ.r*circ.r) > l.a*l.a + l.b*l.b) return vector<Point>();
308
309     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);
310     //one intersection
311     if(abs((l.c*x1.c)/(circ.r*circ.r) - (l.a*l.a + l.b*l.b)) <= EPS){
312         vector<Point> ret;
313         ret.pb(Point(x0,y0));
314         return ret;
315     }
316 }
317
318 //general case
319 double d = circ.r*circ.r - (l.c*x1.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
indicate), else = no intersection(empty)
340         if(c1.r == c2.r){
341             vector<Point> ret;
342             ret.pb(Point());
343             ret.pb(Point());
344             ret.pb(Point());
345             return ret;
346         }
347         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     int j = 0;
438     i = 0;
439     for(int k=0; k<n; k++){
440         if(i < l.size() && j < r.size()){
441             if(r[j].y <= l[i].y){
442                 if((r[j].x - mid)*(r[j].x - mid) < d) {
443                     rr.push_back(r[j]);
444                 }
445                 a.push_back(r[j++]);
446             }
447             else {
448                 if((l[i].x - mid)*(l[i].x - mid) < d) {
449                     ll.push_back(l[i]);
450                 }
451                 a.push_back(l[i++]);
452             }
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
471     int ini = 0, end = rr.size()-1;
472     int j;
473
474     while(ini < end) {
475         j = (ini + end) / 2;
476         if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
477             ini = j+1;
478         else end = j;
479     }
480
481     j = ini;
482
483     for(; j < rr.size(); j++) {
484         if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
485         long long cur = (ll[i].x - rr[j].x)*(ll[i].x - rr[j].x) + (ll[i].y
- rr[j].y)*(ll[i].y - rr[j].y);
486         if(cur < d) {
487             d = cur;
488         }
489     }
490 }
491 return d;
492 }
493
494 }
495
496 ////////////////////////////////////// End of Geometry Algorithms
////////////////////////////////////

```


6.3. 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;
64                 }
65             }
66         }
67     }
68 }

```

```

61     }
62     else {
63         //lower
64         if(lower[1].x != lower[0].x) return false;
65         else if(p.y >= lower[1].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 }

```

6.4. Rotation in 3D

```

1 struct Vec3d{
2     double x,y,z;
3     Vec3d operator+(const Vec3d & rhs) const{
4         return {x+rhs.x, y+rhs.y, z+rhs.z};
5     }
6     Vec3d operator*(const double k) const {
7         return {k*x, k*y, k*z};
8     }
9     Vec3d operator-(const Vec3d & rhs) const{
10        return *this + rhs*-1;
11    }
12    Vec3d operator/(const double k) const {
13        return {x/k, y/k, z/k};
14    }
15    double operator*(const Vec3d & rhs) const{
16        return x*rhs.x+y*rhs.y+z*rhs.z;
17    }
18    double norm_sq() { return (*this)*(*this); }
19    double norm() { return sqrt(norm_sq()); }
20 };
21
22 Vec3d rotate(Vec3d p, Vec3d u /*unit vector*/, double ang){
23     double dot = p*u;
24     double co = cos(ang);
25     double si = sin(ang);
26     double x = u.x*dot*(1-co) + p.x*co + (u.y*p.z-u.z*p.y)*si;
27     double y = u.y*dot*(1-co) + p.y*co + (u.z*p.x-u.x*p.z)*si;
28     double z = u.z*dot*(1-co) + p.z*co + (u.x*p.y-u.y*p.x)*si;
29     return {x, y, z};
30 }

```

7. Graphs

7.1. Dynamic Connectivity - connected(u,v) query

```

1  /* Dynamic 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(l, 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(l, events[i].u, events[i].v, events[i].l,
125                     events[i].r));
126             }
127         }
128     }

```

```

124 }
125 }
126 }

```

7.2. 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     bool cycle = false;
14
15     BellmanFord(){}
16
17     BellmanFord(int n){
18         dist.resize(n+1);
19     }
20
21     void calculate(int source){
22         for(int i=0; i<dist.size(); i++){
23             dist[i] = INF;
24         }
25         dist[source] = 0;
26         for(int k=0; k<dist.size()-1; k++){
27             for(int i=0; i<e.size(); i++){
28                 if(dist[e[i].v] > dist[e[i].u] + e[i].weight){
29                     dist[e[i].v] = dist[e[i].u] + e[i].weight;
30                 }
31             }
32         }
33         for(int i=0; i<e.size(); i++){
34             if(dist[e[i].v] > dist[e[i].u] + e[i].weight){
35                 cycle = true;
36             }
37         }
38     }
39
40 };

```

7.3. 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     vector< set<int> > adj;
19     vector<int> walk;
20     vector<int> deg;
21     int s, t;
22
23     EulerianCircuit();
24     EulerianCircuit(int n){
25         deg.resize(n+1);
26         adj.resize(n+1);
27     }
28
29     void undirected_euler(int u){
30         while(!adj[u].empty()){
31             int v = * (--adj[u].end());
32
33             adj[u].erase(v);
34             adj[v].erase(adj[v].find(u));
35
36             euler(v);
37         }
38         walk.push_back(u);
39     }
40
41     void directed_euler(int u){
42         while(!adj[u].empty()){
43             int v = * (--adj[u].end());
44
45             adj[u].erase(v);
46
47             euler(v);
48         }
49         walk.push_back(u);
50     }
51 };
52
53
54
55

```

7.4. Kosaraju SCC

```

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

```

```

64     dfst(vertex.top(), comp);
65 }
66 vertex.pop();
67 }
68
69 //set< ii > edge_check; //eliminates duplicate edges (additional O(logn))
70
71 for(int i=1; i<=n; i++){
72     for(int j=0; j<adj[i].size(); j++){
73         int v = adj[i][j];
74         if(scc[i] == scc[v]) continue;
75         //if(edge_check.count(ii(scc[i], scc[v]))) continue; //eliminates
duplicate edges (additional O(logn))
76         scc_adj[scc[i]].push_back(scc[v]);
77         //edge_check.insert(ii(scc[i], scc[v])); //eliminates duplicate
edges (additional O(logn))
78     }
79 }
80 }
81
82 };

```

7.5. 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
subtree search) */
4  /* L[] contains the height of the vertex (from root) on centroid tree (Max
is logN) */
5  /* N is equal to the maximum size of tree (given by statement) */
6
7  struct CentroidDecomposition {
8      vector<bool> removed;
9      vector<int> L, subz;
10     vector<int> c_p;
11
12     CentroidDecomposition() {}
13
14     CentroidDecomposition(int n){
15         removed.resize(n+1);
16         L.resize(n+1);
17         c_p.resize(n+1);
18         subz.resize(n+1);
19         for(int i=0; i<=n; i++){
20             c_p[i] = -1;
21         }
22     }
23
24     void centroid_subz(int u, int p){
25         subz[u] = 1;
26         for(int i=0; i<adj[u].size(); i++){
27             int v = adj[u][i];
28             if(v == p || removed[v]) continue;
29             centroid_subz(v, u);
30             subz[u] += subz[v];
31         }
32     }
33
34     int find_centroid(int u, int p, int sub){
35         for(int i=0; i<adj[u].size(); i++){
36             int v = adj[u][i];
37             if(v == p || removed[v]) continue;
38             if(subz[v] > subz[sub]/2){

```

```

39         return find_centroid(v, u, sub);
40     }
41 }
42 return u;
43 }
44
45 void centroid_decomp(int u, int p, int r){
46     centroid_subz(u, -1);
47     int centroid = find_centroid(u, -1, u);
48     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 };

```

7.6. 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 };

```

7.7. 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<ii> 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 = -1){
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 };

```

7.8. 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 }
111 vector< ii > cut;
112 for(int i=1; i<=sz-2; i++){
113     if(!vis[i]) continue;
114     for(FlowEdge e : adj[i]){
115         if(1 <= e.v && e.v <= sz-2 && !vis[e.v] && e.cap > 0 && e.c == 0)
116             cut.pb(ii(i, e.v));
117     }
118 }
119 return cut;
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 };

```

7.9. HLD

```

1 struct HLD {
2
3     struct node{
4         //node values
5         int val = 0; //sets neutral value for the needed operation
6         int lazy = 0;
7         node() {}
8         node(int val) : val(val){
9             lazy = 0;
10        }
11
12        node merge(node b){
13            node ret;
14            //merge nodes
15            return ret;
16        }
17    };
18
19    struct SegmentTree{
20
21        vector<node> st;
22
23        SegmentTree() {}
24
25        void construct(int n){
26            st.resize(4*n);
27        }
28
29        void propagate(int cur, int l, int r){
30            //check if exists operation
31
32            //apply lazy
33
34            if(l != r){
35                //propagate lazy
36            }
37        }

```

```

38 //reset lazy
39 }
40
41 void build(int cur, int l, int r){
42     if(l == r){
43         //apply on leaf
44         return;
45     }
46
47     int m = (l+r)>>1;
48
49     build(2*cur, l, m);
50     build(2*cur+1, m+1, r);
51     st[cur] = st[2*cur].merge(st[2*cur+1]);
52 }
53
54
55 void update(int cur, int l, int r, int a, int b, int val){
56     propagate(cur, l, r);
57     if(b < l || r < a) return;
58     if(a <= l && r <= b){
59         //apply on lazy
60         propagate(cur, l, r);
61         return;
62     }
63     int m = (l+r)>>1;
64     update(2*cur, l, m, a, b, val);
65     update(2*cur+1, m+1, r, a, b, val);
66     st[cur] = st[2*cur].merge(st[2*cur+1]);
67 }
68
69 node query(int cur, int l, int r, int a, int b){
70     propagate(cur, l, r);
71     if(b < l || r < a) return node();
72     if(a <= l && r <= b) return st[cur];
73     int m = (l+r)>>1;
74     node lef = query(2*cur, l, m, a, b);
75     node rig = query(2*cur+1, m+1, r, a, b);
76     return lef.merge(rig);
77 }
78
79 } st;
80
81 vector<int> L, P, ch, subsz, in, out;
82 int t;
83
84 HLD () {}
85
86 HLD(int n){
87     L.resize(n+1);
88     P.resize(n+1);
89     ch.resize(n+1);
90     subsz.resize(n+1);
91     in.resize(n+1);
92     out.resize(n+1);
93     st.construct(n+1);
94     t = 0;
95     for(int i=0; i<=n; i++){
96         ch[i] = i;
97         P[i] = -1;
98         L[i] = 0;
99     }
100 }
101
102

```

```

103
104 void precalculate(int u, int p = -1){
105     subsz[u] = 1;
106     for(int &v : adj[u]){
107         if(v == p) continue;
108         P[v] = u;
109         L[v] = L[u]+1;
110         precalculate(v, u);
111         if(subsz[adj[u][0]] < subsz[v]) swap(adj[u][0], v);
112         subsz[u] += subsz[v];
113     }
114 }
115
116 void build(int u, int p = -1){
117     in[u] = ++t;
118     for(int v : adj[u]){
119         if(v == p) continue;
120         if(adj[u][0] == v){
121             ch[v] = ch[u];
122         }
123         build(v, u);
124     }
125     out[u] = t;
126 }
127
128 void calculate(int root = 1){
129     precalculate(root);
130     build(root);
131 }
132
133 void build_ds(){
134     st.build(1, 1, t);
135 }
136
137 void path_update(int a, int b, int val, bool edge_update = false){
138     while(ch[a] != ch[b]){
139         if(L[ch[b]] > L[ch[a]]) swap(a,b);
140         st.update(1, 1, t, in[ch[a]], in[a], val);
141         a = P[ch[a]];
142     }
143     if(L[b] < L[a]) swap(a,b);
144     if(in[a]+edge_update <= in[b]) st.update(1, 1, t, in[a]+edge_update, in[b], val);
145 }
146
147 void node_update(int u, int val){
148     st.update(1, 1, t, in[u], in[u], val);
149 }
150
151 void edge_update(int u, int v, int val){
152     if(L[u] > L[v]) swap(u, v);
153     st.update(1, 1, t, in[v], in[v], val);
154 }
155
156 void subtree_update(int u, int val, bool edge_update = false){
157     if(in[u] + edge_update <= out[u]) st.update(1, 1, t, in[u] + edge_update, out[u], val);
158 }
159
160 node path_query(int a, int b, bool edge_query = false){
161     node ans;
162     while(ch[a] != ch[b]){
163         if(L[ch[b]] > L[ch[a]]) swap(a,b);
164         ans = ans.merge(st.query(1, 1, t, in[ch[a]], in[a]));
165         a = P[ch[a]];

```



```

166     }
167     if(L[b] < L[a]) swap(a,b);
168     if(in[a]+edge_query <= in[b]) ans = ans.merge(st.query(1, 1, t,
169     in[a]+edge_query, in[b]));
170     return ans;
171 }
172 node node_query(int u){
173     return st.query(1, 1, t, in[u], in[u]);
174 }
175
176 node edge_query(int u, int v){
177     if(L[u] > L[v]) swap(u,v);
178     return st.query(1, 1, t, in[v], in[v]);
179 }
180
181 node subtree_query(int u, bool edge_query = false){
182     if(in[u] + edge_query <= out[u]) return st.query(1, 1, t, in[u] +
183     edge_query, out[u]);
184     else return node();
185 }
186 };

```

7.10. LCA

```

1 struct LCA {
2
3     int tempo;
4     vector<int> st, ed, dad, anc[20], L;
5     vector<bool> vis;
6
7     LCA() {}
8
9     LCA(int n){
10         tempo = 0;
11         st.resize(n+1);
12         ed.resize(n+1);
13         dad.resize(n+1);
14         L.resize(n+1);
15         for(int i=0; i<20; i++) anc[i].resize(n+1);
16         vis.resize(n+1);
17         for(int i=0; i<=n; i++) vis[i] = false;
18     }
19
20     void dfs(int u){
21         vis[u] = true;
22         st[u] = tempo++;
23         for(int i=0; i<adj[u].size(); i++){
24             int v = adj[u][i];
25             if(!vis[v]){
26                 dad[v] = u;
27                 L[v] = L[u]+1;
28                 dfs(v);
29             }
30         }
31         ed[u] = tempo++;
32     }
33
34     bool is_ancestor(int u, int v){
35         return st[u] <= st[v] && st[v] <= ed[u];
36     }
37
38     int query(int u, int v){
39         if(is_ancestor(u,v)) return u;
40         for(int i=19; i>=0; i--){
41             if(anc[i][u] == -1) continue;
42             if(!is_ancestor(anc[i][u],v)) u = anc[i][u];
43         }
44         return dad[u];
45     }
46
47     int distance(int u, int v){
48         return L[u] + L[v] - 2*L[query(u,v)];
49     }
50
51     void precalculate(){
52         dad[1] = -1;
53         L[1] = 0;
54         dfs(1);
55         for(int i=1; i<st.size(); i++){
56             anc[0][i] = dad[i];
57         }
58         for(int i=1; i<20; i++){
59             for(int j=1; j<st.size(); j++){
60                 if(anc[i-1][j] != -1){
61                     anc[i][j] = anc[i-1][anc[i-1][j]];
62                 }
63                 else {

```

```

64         anc[i][j] = -1;
65     }
66 }
67 }
68 }
69
70 };

```

7.11. Block-Cut Tree

```

1  vector<int> adj[N];
2  int vis[N];
3  int ini[N];
4  int bef[N];
5  bool art[N];
6  int num_bridges = 0;
7  int T = 0;
8
9  void dfs_tarjan(int v, int p) {
10     vis[v] = 1;
11     bef[v] = ini[v] = ++T;
12     art[v] = false;
13     int num_sub=0;
14     for(int i = 0; i < adj[v].size(); i++) {
15         int ad = adj[v][i];
16         if(ad == p) continue;
17         if(!vis[ad]) {
18             dfs_tarjan(ad, v);
19             if(bef[ad] > ini[v]) {
20                 // Bridge
21                 num_bridges++;
22             }
23             if(bef[ad] >= ini[v] && p != -1) {
24                 // v is an articulation
25                 art[v] = true;
26             }
27             num_sub++;
28         }
29         bef[v] = min(bef[v], ini[ad]);
30     }
31     if(p == -1 && num_sub > 1) {
32         // Root is an articulation
33         art[v] = true;
34     }
35 }
36
37 int curId;
38 vector<int> adjbct[2*112345];
39
40 void dfs_block_cut(int v, int id) {
41     vis[v] = 1;
42     if(id != -1) {
43         adjbct[v].pb(id);
44         adjbct[id].pb(v);
45     }
46     for(int i = 0; i < adj[v].size(); i++) {
47         int ad = adj[v][i];
48         if(!vis[ad]) {
49             if(bef[ad] >= ini[v]) {
50                 curId++;
51                 adjbct[v].pb(curId);
52                 adjbct[curId].pb(v);
53                 dfs_block_cut(ad, curId);
54             } else {

```

```

55         dfs_block_cut(ad, id);
56     }
57 }
58 }
59 }
60
61 int32_t main() {
62     num_bridges = 0;
63     T = 0;
64     memset(vis, 0, sizeof vis);
65     for(int i = 1; i <= n; i++)
66         dfs_tarjan(i, -1);
67
68     curId = n;
69     memset(vis, 0, sizeof vis);
70     for(int i = 1; i <= n; i++)
71         dfs_block_cut(i, -1);
72 }

```

7.12. Dominator Tree

```

1  template<typename T = int>
2  struct LinkDsu{
3      vector<int> r;
4      vector<T> best;
5      LinkDsu(int n = 0){
6          r = vector<int>(n); iota(r.begin(), r.end(), 0);
7          best = vector<T>(n);
8      }
9
10     int find(int u) {
11         if (r[u] == u)
12             return u;
13         else {
14             int v = find(r[u]);
15             if (best[r[u]] < best[u]) best[u] = best[r[u]];
16             return r[u] = v;
17         }
18     }
19
20     T eval(int u){ find(u); return best[u]; }
21     void link(int p, int u) { r[u] = p; }
22     void set(int u, T x) { best[u] = x; }
23 };
24
25 struct DominatorTree{
26     typedef vector<vector<int>> Graph;
27     vector<int> semi, dom, parent, st, from;
28     Graph succ, pred, bucket;
29     int r, n, tempo;
30
31     void dfs(int u, int p){
32         semi[u] = u;
33         from[st[u] = tempo++] = u;
34         parent[u] = p;
35         for (int v : succ[u]) {
36             pred[v].push_back(u);
37             if (semi[v] == -1) { dfs(v, u); }
38         }
39     }
40
41     void build(){
42         n = succ.size();
43         dom.assign(n, -1);

```

```

44 semi.assign(n, -1);
45 parent.assign(n, -1);
46 st.assign(n, 0);
47 from.assign(n, -1);
48 pred = Graph(n, vector<int>());
49 bucket = Graph(n, vector<int>());
50 LinkDsu<pair<int,int>> dsu(n);
51 tempo = 0;
52
53 dfs(r, r);
54 for(int i = 0; i < n; i++) dsu.set(i, make_pair(st[i], i));
55
56 for (int i = tempo - 1; i; i--) {
57     int u = from[i];
58     for (int v : pred[u]) {
59         int w = dsu.eval(v).second;
60         if (st[semi[w]] < st[semi[u]]) { semi[u] = semi[w]; }
61     }
62     dsu.set(u, make_pair(st[semi[u]], u));
63     bucket[semi[u]].push_back(u);
64     dsu.link(parent[u], u);
65     for(int v : bucket[parent[u]]) {
66         int w = dsu.eval(v).second;
67         dom[v] = semi[w] == parent[u] ? parent[u] : w;
68     }
69     bucket[parent[u]].clear();
70 }
71 for (int i = 1; i < tempo; i++) {
72     int u = from[i];
73     if (dom[u] != semi[u]) dom[u] = dom[dom[u]];
74 }
75 }
76
77 DominatorTree(const Graph & g, int s) : succ(g), r(s) {
78     build();
79 }
80 };

```

7.13. Minimum Path Cover Problem On DAG's

```

1 int32_t main(){
2     DESYNC;
3     int n,m;
4     cin >> n >> m;
5     Dinic dinic(n+n);
6     for(int i=1; i<=n; i++){
7         dinic.add_to_src(i, 1);
8         dinic.add_to_snk(i+n, 1);
9     }
10
11     for(int i=0; i<m; i++){
12         int u,v;
13         cin >> u >> v;
14         dinic.add_edge(u,v+n,1);
15     }
16
17     dinic.calculate();
18     for(int i=1; i<=n; i++){
19         for(Dinic::FlowEdge e : dinic.adj[i]){
20             if(e.cap == 1 && e.c == 0 && 1 <= e.v && e.v-n <= n){
21                 adj[i].pb(e.v-n);
22                 dg[e.v-n]++;
23             }
24         }
25     }

```

```

25 }
26
27 for(int i=1; i<=n; i++){
28     if(dg[i] == 0){
29         paths.pb(vector<int>());
30         go(i, paths.size()-1);
31     }
32 }
33
34 cout << paths.size() << endl;
35 for(int i=0; i<paths.size(); i++){
36     for(int v : paths[i]) cout << v << " ";
37     cout << endl;
38 }
39
40 }

```

7.14. Stoer-Wagner Minimum Cut in Undirected Graphs

```

1 /* Initialization */
2 int cost[n+1][n+1];
3 memset(cost, 0, sizeof cost);
4 while (m--) {
5     int u, v, c;
6     u = input.next();
7     v = input.next();
8     c = input.next();
9     cost[u][v] = c;
10    cost[v][u] = c;
11 }
12 /* Stoer-Wagner: global minimum cut in undirected graphs */
13 int min_cut = 1000000000;
14 bool added[n+1];
15 int vertex_cost[n+1];
16 for (int vertices_count = n; vertices_count > 1; --vertices_count) {
17     memset(added, 0, sizeof(added[0]) * (vertices_count + 1));
18     memset(vertex_cost, 0, sizeof(vertex_cost[0]) * (vertices_count + 1));
19     int s = -1, t = -1;
20     for (int i = 1; i <= vertices_count; ++i) {
21         int vert = 1;
22         while (added[vert]) ++vert;
23         for (int j = 1; j <= vertices_count; ++j)
24             if (!added[j] && vertex_cost[j] > vertex_cost[vert]) vert = j;
25         if (i == vertices_count - 1)
26             s = vert;
27         else if (i == vertices_count) {
28             t = vert;
29             min_cut = min(min_cut, vertex_cost[vert]);
30         }
31         added[vert] = 1;
32         for (int j = 1; j <= vertices_count; ++j) vertex_cost[j] +=
33             cost[vert][j];
34     }
35     for (int i = 1; i <= vertices_count; ++i) {
36         cost[s][i] += cost[t][i];
37         cost[i][s] += cost[i][t];
38     }
39     for (int i = 1; i <= vertices_count; ++i) {
40         cost[t][i] = cost[vertices_count][i];
41         cost[i][t] = cost[i][vertices_count];
42     }
43 }
44 printf("%d\n", min_cut);

```