

# Competitive Programming Algorithms and Topics

Ubiratan Neto

<b>1</b>	<b>Template</b>	<b>1</b>
1.1	Template Code . . . . .	1
<b>2</b>	<b>Data Structures</b>	<b>2</b>
2.1	Segment Tree using Pointers . . . . .	2
2.2	Range Update Segment Tree . . . . .	2
2.3	Range Update Binary Indexed Tree . . . . .	3
2.4	Trie . . . . .	3
2.5	STL Ordered Set . . . . .	3
<b>3</b>	<b>Uncategorized</b>	<b>4</b>
3.1	Longest Increasing Subsequence . . . . .	4
3.2	Inversion Count - Merge Sort . . . . .	4
3.3	Mo's Decomposition . . . . .	4
<b>4</b>	<b>Geometry</b>	<b>5</b>
4.1	Closest Pair of Points . . . . .	5
4.2	2D Structures . . . . .	5
4.3	2D Geometry Functions . . . . .	7
4.4	Convex Hull - Monotone Chain Algorithm . . . . .	9
<b>5</b>	<b>Graphs</b>	<b>10</b>
5.1	Dynammic Connectivity - connected(u,v) query . . . . .	10
5.2	Bellman Ford Shortest Path . . . . .	11
5.3	Eulerian Circuits/Paths . . . . .	11
5.4	Kosaraju SCC . . . . .	12
5.5	Centroid Decomposition . . . . .	12
5.6	Floyd Warshall Shortest Path . . . . .	13
5.7	Tarjan's Bridge/Articulations Algorithm . . . . .	13
5.8	Max Flow Dinic's Algorithm . . . . .	14
5.9	HLD . . . . .	15
5.10	LCA . . . . .	15

<b>6</b>	<b>Math and Number Theory</b>	<b>16</b>
6.1	Binomial Coefficient DP . . . . .	16
6.2	Erathostenes Sieve + Logn Prime Factorization . . . . .	16
6.3	Matrix Exponentiation . . . . .	16
6.4	Fast Fourier Transform - Recursive and Iterative . . . . .	16
<b>7</b>	<b>String Algorithms</b>	<b>18</b>
7.1	KMP Failure Function + String Matching . . . . .	18
7.2	Z-Function . . . . .	18
7.3	Rolling Hash . . . . .	18
7.4	Suffix Array + Linear Sort . . . . .	19

## 1. Template

### 1.1. Template Code

```

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

## 2. Data Structures

### 2.1. Segment Tree using Pointers

```

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

```

```

62         if(!root->right) root->right = new node();
63         node *left = query(root->left, l, m, a, m);
64         node *right = query(root->right, m+1, r, m+1, b);
65         node *ans = new node();
66         combine(ans, left, right);
67         return ans;
68     }
69 }

```

### 2.2. Range Update Segment Tree

```

1  /* Range update Segment Tree Implementation */
2  /* The first node (ROOT) is defined to 1 (1 - index impl) */
3  /* N is the maximum number of elements given by the statement */
4  /* Lazy can be inside node structure instead of being another structure */
5
6  #define ROOT 1
7  #define N MAX_INPUT
8
9  struct node{
10     //attributes of node
11 };
12
13 node tree[4*N];
14 node lazy[4*N];
15
16 node combine(node a, node b){
17     node res;
18     //combine operations
19     return res;
20 }
21
22 void propagate(int root, int l, int r){
23     //return if there is no update
24     //update tree using lazy node
25     if(l != r){
26         //propagate for left and right child
27     }
28     //reset lazy node
29 }
30
31 void range_update(int root, int l, int r, int a, int b, long long val){
32     if(l == a && r == b){
33         //lazy operation using val
34         return;
35     }
36
37     int m = (l+r)/2;
38
39     if(b <= m) range_update(2*root, l, m, a, b, val);
40     else if(m < a) range_update(2*root+1, m+1, r, a, b, val);
41     else {
42         range_update(2*root, l, m, a, m, val);
43         range_update(2*root+1, m+1, r, m+1, b, val);
44     }
45
46     propagate(root, l, r);
47     propagate(2*root, l, m);
48     propagate(2*root+1, m+1, r);
49     tree[root] = combine(tree[2*root], tree[2*root+1]);
50 }
51
52 node query(int root, int l, int r, int a, int b){
53     propagate(root, l, r);

```

```

54 if(l == a && r == b) return tree[root];
55
56 int m = (l+r)/2;
57 if(b <= m) return query(2*root, l, m, a, b);
58 else if(m < a) return query(2*root+1, m+1, r, a, b);
59 else {
60     node left = query(2*root, l, m, a, m);
61     node right = query(2*root+1, m+1, r, m+1, b);
62     node ans = combine(left, right);
63     return ans;
64 }
65 }

```

### 2.3. Range Update Binary Indexed Tree

```

1  /* Range Update Binary Indexed Tree Implementation */
2  /* Tree is 1 - index */
3  /* Point Update Binary Indexed Tree operations are used as auxiliar */
4  /* N is defined as the maximum number of elements (given by the statement) */
5
6  #define N MAX_INPUT
7
8  int bit[2][N+1];
9
10 void init(int n){
11     for(int i=1; i<=n; i++){
12         bit[0][i] = 0;
13         bit[1][i] = 0;
14     }
15 }
16
17 //auxiliar functions
18
19 void update(int *bit, int idx, int val, int n){
20     for(int i = idx; i <= n; i += i&-i){
21         bit[i]+=val;
22     }
23 }
24
25 int query(int *bit, int idx){
26     int ans = 0;
27     for(int i=idx; i>0; i -= i&-i){
28         ans += bit[i];
29     }
30     return ans;
31 }
32
33 //end of auxiliar functions
34
35 void range_update(int l, int r, int val, int n){
36     update(bit[0], l, val, n);
37     update(bit[0], r+1, -val, n);
38     update(bit[1], l, val*(l-1), n);
39     update(bit[1], r+1, -val*r, n);
40 }
41
42 int prefix_query(int idx){
43     return query(bit[0],idx)*idx - query(bit[1], idx);
44 }
45
46 int range_query(int l, int r){
47     return prefix_query(r) - prefix_query(l-1);
48 }

```

### 2.4. Trie

```

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

```

### 2.5. STL Ordered Set

```

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

```

### 3. Uncategorized

#### 3.1. Longest Increasing Subsequence

```

1 struct LIS{
2     vector<int> seq;
3
4     void calculate(vector<int> & v){
5         int n = v.size();
6         seq.resize(n+1);
7         for(int i=1; i<=n; i++) seq[i] = INT_MAX;
8         seq[0] = INT_MIN;
9         for(int i=0; i<n; i++){
10             int index = lower_bound(seq.begin(), seq.end(), v[i]) - seq.begin();
11             index--;
12             seq[index+1] = min(seq[index+1], v[i]);
13         }
14     }
15 } lis;

```

#### 3.2. Inversion Count - Merge Sort

```

1 long long 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     long long 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 }

```

#### 3.3. Mo's Decomposition

```

1 struct query{
2     int id, l, r, ans;
3     bool operator<(const query & b) const {
4         return l/(int)sqrt(n) < b.l/(int)sqrt(n) || l/(int)sqrt(n) ==
5             b.l/(int)sqrt(n) && r < b.r;
6     }
7 };
8 struct SqrtDecomposition {
9
10    vector<query> q;
11
12    void init(int n){
13        q.resize(n);
14    }
15
16    void add(int idx){
17
18    }
19
20    void remove(int idx){
21
22    }
23
24    int answer_query(){
25
26    }
27
28    void calculate(){
29        sort(q.begin(), q.end());
30        int l = 0, r = -1;
31        for(int i=0; i<q.size(); i++){
32            while(q[i].l < l) add(--l);
33            while(r < q[i].r) add(++r);
34            while(q[i].l > l) remove(l++);
35            while(r > q[i].r) remove(r--);
36            q[i].ans = answer_query();
37        }
38    }
39
40    void print_ans(){
41        sort(q.begin(), q.end(), [](const query & a, const query & b){
42            return a.id < b.id;
43        });
44
45        for(query x : q){
46            printf("%d\n", x.ans);
47        }
48    }
49 } mos;

```

## 4. Geometry

### 4.1. Closest Pair of Points

```

1  /* Closest Pair of Points Problem Implementation */
2  /* Divide and Conquer Approach */
3  /* Using the observation of only checking points inside min_dist x min_dist
   from mid */
4  /* Binary search boosts search for the right border start point */
5
6  struct vec2 {
7      long long x, y;
8  };
9
10 bool cmp(vec2 a, vec2 b) {
11     return a.x < b.x || (a.x == b.x && a.y < b.y);
12 }
13
14 pair<vec2, vec2> ans;
15
16 long long solve(vector<vec2> &a) {
17
18     long long mid = a[a.size()/2].x;
19     int n = a.size();
20
21     vector<vec2> l;
22     vector<vec2> r;
23     int i = 0;
24     for(; i < a.size()/2; i++) l.push_back(a[i]);
25     for(; i < a.size(); i++) r.push_back(a[i]);
26
27     long long d = LLONG_MAX;
28
29     if(l.size() > 1) {
30         d = min(d, solve(l));
31     } if(r.size() > 1) {
32         d = min(d, solve(r));
33     }
34
35     a.clear();
36
37     vector<vec2> ll;
38     vector<vec2> rr;
39
40
41     int j = 0;
42     i = 0;
43     for(int k=0; k<n; k++){
44         if(i < l.size() && j < r.size()){
45             if(r[j].y <= l[i].y){
46                 if((r[j].x - mid)*(r[j].x - mid) < d) {
47                     rr.push_back(r[j]);
48                 }
49                 a.push_back(r[j++]);
50             }
51             else {
52                 if((l[i].x - mid)*(l[i].x - mid) < d) {
53                     ll.push_back(l[i]);
54                 }
55                 a.push_back(l[i++]);
56             }
57         }
58         else if(i < l.size()){
59             if((l[i].x - mid)*(l[i].x - mid) < d) {
60                 ll.push_back(l[i]);

```

```

61     }
62     a.push_back(l[i++]);
63 }
64 else {
65     if((r[j].x - mid)*(r[j].x - mid) < d) {
66         rr.push_back(r[j]);
67     }
68     a.push_back(r[j++]);
69 }
70 }
71
72 for(int i = 0; i < ll.size(); i++) {
73
74     int ini = 0, end = rr.size()-1;
75     int j;
76     while(ini < end) {
77         j = (ini + end) / 2;
78         if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
79             ini = j+1;
80         else end = j;
81     }
82     j = ini;
83
84     for(; j < rr.size(); j++) {
85         if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
86         long long cur = (ll[i].x - rr[j].x)*(ll[i].x - rr[j].x)
87             + (ll[i].y - rr[j].y)*(ll[i].y - rr[j].y);
88         if(cur < d) {
89             d = cur;
90             long long cur2 = (ans.first.x - ans.second.x)*(ans.first.x -
91                 ans.second.x)
92                 + (ans.first.y - ans.second.y)*(ans.first.y - ans.second.y);
93             if(cur < cur2)
94                 ans = { ll[i], rr[j] };
95         }
96     }
97     return d;
98 }

```

### 4.2. 2D Structures

```

1  struct Point2D {
2      int x,y;
3
4      Point2D(){
5          x = 0;
6          y = 0;
7      }
8
9      Point2D(int x, int y) : x(x), y(y) {}
10
11     Point2D operator+(const Point2D b) const{
12         return Point2D(x + b.x, y + b.y);
13     }
14
15     Point2D operator-(const Point2D b) const{
16         return Point2D(x - b.x, y - b.y);
17     }
18
19     bool operator<(const Point2D b) const{
20         return x < b.x || (x == b.x && y < b.y);
21     }
22 }

```

```

23 void operator=(const Point2D b) const{
24     x = b.x;
25     y = b.y;
26 }
27
28 double distanceTo(Point2D b){
29     return sqrt((x - b.x)*(x - b.x) + (y - b.y)*(y - b.y));
30 }
31
32 int distanceTo2(Point2D b){
33     return (x - b.x)*(x - b.x) + (y - b.y)*(y - b.y);
34 }
35
36 };
37
38 struct Vector2D {
39     int x,y;
40
41     Vector2D(){
42         x = 0;
43         y = 0;
44     }
45
46     Vector2D(int x, int y) : x(x), y(y) {}
47
48     Vector2D(Point2D a, Point2D b){
49         x = b.x - a.x;
50         y = b.y - a.y;
51     }
52
53     Vector2D operator+(const Vector2D b) const{
54         return Vector2D(x + b.x, y + b.y);
55     }
56
57     Vector2D operator-(const Vector2D b) const{
58         return Vector2D(x - b.x, y - b.y);
59     }
60
61     void operator=(const Vector2D b) const{
62         x = b.x;
63         y = b.y;
64     }
65
66     int operator*(const Vector2D b) const{
67         return (x*b.x + y*b.y);
68     }
69
70     int operator^(const Vector2D b) const{
71         return x*b.y - y*b.x;
72     }
73
74     bool operator<(const Vector2D b) const{
75         return x < b.x || (x == b.x && y < b.y);
76     }
77
78     Vector2D scale(int n){
79         return Vector2D(x*n, y*n);
80     }
81
82     double size(){
83         return sqrt(x*x + y*y);
84     }
85
86     int size2(){
87         return x*x + y*y;

```

```

88     }
89
90     Vector2D normalize(){
91         return Vector2D((double)x/size(), (double)y/size());
92     }
93 };
94
95 struct Line2D {
96     Point2D p, q;
97     Vector2D v;
98     Vector2D normal;
99
100     int a,b,c;
101
102     Line2D() {
103         p = Point2D();
104         q = Point2D();
105         v = Vector2D();
106         normal = Vector2D();
107         a = 0;
108         b = 0;
109         c = 0;
110     }
111
112     void operator=(const Line2D l) const{
113         a = l.a;
114         b = l.b;
115         c = l.c;
116         p = l.p;
117         q = l.q;
118         v = l.v;
119         normal = l.normal;
120     }
121
122     Line2D(Point2D r, Point2D s){
123         p = r;
124         q = s;
125         v = Vector2D(r, s);
126         normal = Vector2D(-v.y, v.x);
127         a = -v.y;
128         b = v.x;
129         c = -(a*p.x + b*p.y);
130     }
131
132     Line2D(Point2D r, Vector2D s){
133         p = r;
134         q = Point2D(p.x + s.x, p.y + s.y);
135         v = s;
136         normal = Vector2D(-v.y, v.x);
137         a = -v.y;
138         b = v.x;
139         c = -(a*p.x + b*p.y);
140     }
141 }
142
143 };

```

## 4.3. 2D Geometry Functions

```

1 struct Geo2D {
2
3     double distancePointLine(Point2D p, Line2D l){
4         return double(1.*abs(l.a*p.x + l.b*p.y + l.c)/l.normal.size());
5     }
6
7     double distancePointSegment(Point2D p, Line2D l){
8         int dot1 = Vector2D(l.p, p)*Vector2D(l.p, l.q);
9         int dot2 = Vector2D(l.q, p)*Vector2D(l.q, l.p);
10
11         if(dot1 >= 0 && dot2 >= 0) return distancePointLine(p, l);
12         else return min(p.distanceTo(l.p), p.distanceTo(l.q));
13     }
14
15     double distancePointRay(Point2D p, Line2D l){
16         int dot = Vector2D(l.p, p)*l.v;
17         if(dot >= 0) return distancePointLine(p, l);
18         else return p.distanceTo(l.p);
19     }
20
21     Point2D closestPointInSegment(Point2D p, Line2D s){
22         //returns closest point from p in segment s
23         Vector2D u = s.v.normalize();
24         Vector2D w(s.p, p);
25         Vector2D res = u.scale(u*w);
26         if(u*w < 0 || u*w > s.p.distanceTo(s.q)){
27             if(p.distanceTo(s.p) < p.distanceTo(s.q)) return s.p;
28             else return s.q;
29         }
30         else return Point2D(s.p.x + res.x, s.p.y + res.y);
31     }
32
33     bool intersectionSegmentSegment(Line2D s1, Line2D s2){
34         //Assuming that endpoints are ordered by x
35         if(s1.p.x > s1.q.x) swap(s1.p, s1.q);
36         if(s2.p.x > s2.q.x) swap(s2.p, s2.q);
37         if(abs(s1.v^s2.v) <= EPS){
38             Vector2D v1(s2.p, s1.p);
39             if(s1.p.x == s1.q.x && s2.p.x == s2.q.x && s1.p.x == s2.p.x){
40                 Point2D ans1, ansr;
41                 if(s1.p.y > s1.q.y) swap(s1.p, s1.q);
42                 if(s2.p.y > s2.q.y) swap(s2.p, s2.q);
43                 if(s1.p.y <= s2.p.y) ans1 = s2.p;
44                 else ans1 = s1.p;
45                 if(s2.q.y <= s1.q.y) ansr = s2.q;
46                 else ansr = s1.q;
47                 if(ans1.x == ansr.x && ans1.y == ansr.y){
48                     //cout << ansr.x << " " << ansr.y << endl;
49                     return true;
50                 }
51                 else if(ansr.y < ans1.y){
52                     //cout << "Empty" << endl;
53                     return false;
54                 }
55                 else {
56                     if(ans1.x == ansr.x && ans1.y > ansr.y) swap(ans1, ansr);
57                     //cout << ans1.x << " " << ans1.y << endl << ansr.x << " " <<
58                     ansr.y << endl;
59                     return true;
60                 }
61             }
62             else if(abs(s1.v^v1) <= EPS){
63                 Point2D ans1, ansr;

```

```

63         if(s1.p.x <= s2.p.x) ans1 = s2.p;
64         else ans1 = s1.p;
65         if(s2.q.x <= s1.q.x) ansr = s2.q;
66         else ansr = s1.q;
67         if(ans1.x == ansr.x && ans1.y == ansr.y){
68             //cout << ansr.x << " " << ansr.y << endl;
69             return true;
70         }
71         else if(ansr.x < ans1.x){
72             //cout << "Empty" << endl;
73             return false;
74         }
75         else {
76             if(ans1.x == ansr.x && ans1.y > ansr.y) swap(ans1, ansr);
77             //cout << ans1.x << " " << ans1.y << endl << ansr.x << " " <<
78             ansr.y << endl;
79             return true;
80         }
81         else {
82             //cout << "Empty" << endl;
83             return false;
84         }
85     }
86
87     else {
88         int a1 = s1.q.y - s1.p.y;
89         int b1 = s1.p.x - s1.q.x;
90         int c1 = a1*s1.p.x + b1*s1.p.y;
91         int a2 = s2.q.y - s2.p.y;
92         int b2 = s2.p.x - s2.q.x;
93         int c2 = a2*s2.p.x + b2*s2.p.y;
94         int det = a1*b2 - a2*b1;
95         double x = (double)(b2*c1 - b1*c2)/(double)det*1.;
96         double y = (double)(a1*c2 - a2*c1)/(double)det*1.;
97         if(s1.p.x-EPS <= x && x <= s1.q.x+EPS && s2.p.x-EPS <= x && x <=
98         s2.q.x+EPS){
99             //cout << x << " " << y << endl;
100             return true;
101         }
102         else {
103             //cout << "Empty" << endl;
104             return false;
105         }
106     }
107 }
108
109 double distanceSegmentSegment(Line2D l1, Line2D l2){
110     if((l1.v^l2.v) != 0){
111         Line2D r1(l1.p, l1.q);
112         Line2D r2(l1.q, l1.p);
113         Line2D r3(l2.p, l2.q);
114         Line2D r4(l2.q, l2.p);
115
116         int cross1 = (Vector2D(r3.p, r1.p)^r3.v);
117         int cross2 = (Vector2D(r3.p, r1.q)^r3.v);
118         if(cross2 < cross1) swap(cross1, cross2);
119
120         bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
121         r3) > distancePointLine(r1.q, r3));
122
123         cross1 = (Vector2D(r1.p, r3.p)^r1.v);
124         cross2 = (Vector2D(r1.p, r3.q)^r1.v);
125         if(cross2 < cross1) swap(cross1, cross2);

```

```

125     bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p,
126     r1) > distancePointLine(r3.q, r1));
127
128     cross1 = (Vector2D(r3.p, r2.p)^r3.v);
129     cross2 = (Vector2D(r3.p, r2.q)^r3.v);
130     if(cross2 < cross1) swap(cross1, cross2);
131
132     bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
133     r3) > distancePointLine(r2.q, r3));
134
135     cross1 = (Vector2D(r2.p, r3.p)^r2.v);
136     cross2 = (Vector2D(r2.p, r3.q)^r2.v);
137     if(cross2 < cross1) swap(cross1, cross2);
138
139     bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p,
140     r2) > distancePointLine(r3.q, r2));
141
142     cross1 = (Vector2D(r4.p, r1.p)^r4.v);
143     cross2 = (Vector2D(r4.p, r1.q)^r4.v);
144     if(cross2 < cross1) swap(cross1, cross2);
145
146     bool ok5 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
147     r4) > distancePointLine(r1.q, r4));
148
149     cross1 = (Vector2D(r1.p, r4.p)^r1.v);
150     cross2 = (Vector2D(r1.p, r4.q)^r1.v);
151     if(cross2 < cross1) swap(cross1, cross2);
152
153     bool ok6 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p,
154     r1) > distancePointLine(r4.q, r1));
155
156     cross1 = (Vector2D(r4.p, r2.p)^r4.v);
157     cross2 = (Vector2D(r4.p, r2.q)^r4.v);
158     if(cross2 < cross1) swap(cross1, cross2);
159
160     bool ok7 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
161     r4) > distancePointLine(r2.q, r4));
162
163     cross1 = (Vector2D(r2.p, r4.p)^r2.v);
164     cross2 = (Vector2D(r2.p, r4.q)^r2.v);
165     if(cross2 < cross1) swap(cross1, cross2);
166
167     bool ok8 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p,
168     r2) > distancePointLine(r4.q, r2));
169
170     if(ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8) return 0;
171
172     double ans = distancePointSegment(l1.p, l2);
173     ans = min(ans, distancePointSegment(l1.q, l2));
174     ans = min(ans, distancePointSegment(l2.p, l1));
175     ans = min(ans, distancePointSegment(l2.q, l1));
176     return ans;
177 }
178
179 double distanceSegmentRay(Line2D s, Line2D r){
180     if((s.v^r.v) != 0){
181         Line2D r1(s.p, s.q);
182         Line2D r2(s.q, s.p);
183
184         int cross1 = (Vector2D(r.p, r1.p)^r.v);
185         int cross2 = (Vector2D(r.p, r1.q)^r.v);

```

```

182         if(cross2 < cross1) swap(cross1, cross2);
183
184         bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p, r
185         > distancePointLine(r1.q, r));
186
187         cross1 = (Vector2D(r1.p, r.p)^r1.v);
188         cross2 = (Vector2D(r1.p, r.q)^r1.v);
189         if(cross2 < cross1) swap(cross1, cross2);
190
191         bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r1
192         > distancePointLine(r.q, r1));
193
194         cross1 = (Vector2D(r.p, r2.p)^r.v);
195         cross2 = (Vector2D(r.p, r2.q)^r.v);
196         if(cross2 < cross1) swap(cross1, cross2);
197
198         bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r
199         > distancePointLine(r2.q, r));
200
201         cross1 = (Vector2D(r2.p, r.p)^r2.v);
202         cross2 = (Vector2D(r2.p, r.q)^r2.v);
203         if(cross2 < cross1) swap(cross1, cross2);
204
205         bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r2
206         > distancePointLine(r.q, r2));
207
208         if(ok1 && ok2 && ok3 && ok4) return 0;
209     }
210
211     double ans = INF;
212     int dot = Vector2D(s.p, r.p)*Vector2D(r.p, s.q);
213     if(dot >= 0) ans = min(ans, distancePointLine(r.p, s));
214     else ans = min(ans, min(r.p.distanceTo(s.p), r.p.distanceTo(s.q)));
215
216     dot = Vector2D(r.p, s.p)*r.v;
217     if(dot >= 0) ans = min(ans, distancePointLine(s.p, r));
218     else ans = min(ans, r.p.distanceTo(s.p));
219
220     dot = Vector2D(r.p, s.q)*r.v;
221     if(dot >= 0) ans = min(ans, distancePointLine(s.q, r));
222     else ans = min(ans, r.p.distanceTo(s.q));
223
224     return ans;
225 }
226
227 double distanceSegmentLine(Line2D s, Line2D l){
228     if((s.v^l.v) == 0){
229         return distancePointLine(s.p, l);
230     }
231
232     int cross1 = (Vector2D(l.p, s.p)^l.v);
233     int cross2 = (Vector2D(l.p, s.q)^l.v);
234     if(cross2 < cross1) swap(cross1, cross2);
235     if(cross1 <= 0 && cross2 >= 0) return 0;
236     else return min(distancePointLine(s.p, l), distancePointLine(s.q, l));
237 }
238
239 double distanceLineRay(Line2D l, Line2D r){
240     if((l.v^r.v) == 0){
241         return distancePointLine(r.p, l);
242     }

```



```

243 int cross1 = (Vector2D(l.p, r.p)^l.v);
244 int cross2 = (Vector2D(l.p, r.q)^l.v);
245 if(cross2 < cross1) swap(cross1, cross2);
246 if((cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, l) >
distancePointLine(r.q, l))) return 0;
return distancePointLine(r.p, l);
}

247
248
249
250 double distanceLineLine(Line2D l1, Line2D l2){
251     if((l1.v^l2.v) == 0){
252         return distancePointLine(l1.p, l2);
253     }
254     else return 0;
255 }
256
257 double distanceRayRay(Line2D r1, Line2D r2){
258     if((r1.v^r2.v) != 0){
259
260         int cross1 = (Vector2D(r1.p, r2.p)^r1.v);
261         int cross2 = (Vector2D(r1.p, r2.q)^r1.v);
262         if(cross2 < cross1) swap(cross1, cross2);
263         bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
r1) > distancePointLine(r2.q, r1));
264
265         cross1 = (Vector2D(r2.p, r1.p)^r2.v);
266         cross2 = (Vector2D(r2.p, r1.q)^r2.v);
267         if(cross2 < cross1) swap(cross1, cross2);
268
269         bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
r2) > distancePointLine(r1.q, r2));
270
271         if(ok1 && ok2) return 0;
272     }
273 }
274
275 double ans = INF;
276 int dot = Vector2D(r2.p, r1.p)*r2.v;
277 if(dot >= 0) ans = min(ans, distancePointLine(r1.p, r2));
278 else ans = min(ans, r2.p.distanceTo(r1.p));
279
280 dot = Vector2D(r1.p, r2.p)*r1.v;
281 if(dot >= 0) ans = min(ans, distancePointLine(r2.p, r1));
282 else ans = min(ans, r1.p.distanceTo(r2.p));
283
284 return ans;
285
286 }
287
288 } geo2d;

```

#### 4.4. Convex Hull - Monotone Chain Algorithm

```

1 struct ConvexHull {
2     vector< Point2D > points, lower, upper;
3
4     ConvexHull(){}
5
6     void calculate(vector<Point2D> v){
7         sort(v.begin(), v.end());
8         for(int i=0; i<v.size(); i++){
9             while(upper.size() >= 2 && (Vector2D(upper[upper.size()-2],
upper.back())^Vector2D(upper.back(), v[i])) >= 0LL) upper.pop_back();
10            upper.push_back(v[i]);
11        }
12        reverse(v.begin(), v.end());
13        for(int i=0; i<v.size(); i++){
14            while(lower.size() >= 2 && (Vector2D(lower[lower.size()-2],
lower.back())^Vector2D(lower.back(), v[i])) >= 0LL) lower.pop_back();
15            lower.push_back(v[i]);
16        }
17        for(int i=upper.size()-2; i>=0; i--) points.push_back(upper[i]);
18        for(int i=lower.size()-2; i>=0; i--) points.push_back(lower[i]);
19    }
20
21    double area(){
22        double area = points.back().x*points[0].y - points.back().y*points[0].x;
23        for(int i=0; i<points.size()-1; i++){
24            area += points[i].x*points[i+1].y - points[i].y*points[i+1].x;
25        }
26        return area/2.;
27    }
28
29    int area2(){
30        int area2 = points.back().x*points[0].y - points.back().y*points[0].x;
31        for(int i=0; i<points.size()-1; i++){
32            area2 += points[i].x*points[i+1].y - points[i].y*points[i+1].x;
33        }
34        return area2;
35    }
36
37    long double perimeter(){
38        long double val = Vector2D(points[0], points.back()).size();
39        for(int i=0; i<points.size()-1; i++){
40            val += Vector2D(points[i], points[i+1]).size();
41        }
42        return val;
43    }
44
45 } chull;

```

## 5. Graphs

### 5.1. 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(1, u, v, l, i));
114             events[l].r = i;
115         }
116         else if(op == "conn"){
117             events.push_back(event(2, u, v, -1, -1));
118         }
119     }
120     for(int i=0; i<q; i++){
121         if(events[i].op == 0){
122             if(events[i].r == -1){
123                 events[i].r = events.size();
124                 events.push_back(event(1, events[i].u, events[i].v, events[i].l,
125                     events[i].r));
126             }
127         }
128     }

```

```

124 }
125 }
126 }

```

## 5.2. Bellman Ford Shortest Path

```

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

```

## 5.3. Eulerian Circuits/Paths

```

1 //Graph - Euler path
2
3 //for undirected graph
4 //circuit - 2 vertex with odd grades
5 //simple path - all vertex with even grades
6 //this algorithm generates a circuit, if you need a path between u,v
7 //create a new edge u-v, compute circuit u..u, then delete the last u
8
9 //for directed graph
10 //circuit - all vertex needs enter grade = exit grade
11 //path - one vertex needs to have one more enter grade
12 //and the other needs to have one more exit grade
13 //this algorithm generates a circuit, if you need a path between u,v
14 //create a new edge u-v, considering that u have one more enter grade
15 //and v one more exit grade
16
17 struct EulerianCircuit {
18     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
39         walk.push_back(u);
40     }
41
42     void directed_euler(int u){
43         while(!adj[u].empty()){
44             int v = * (--adj[u].end());
45
46             adj[u].erase(v);
47
48             euler(v);
49         }
50
51         walk.push_back(u);
52     }
53
54 };

```

## 5.4. Kosaraju SCC

```

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

```

```

64         comp++;
65         dfst(vertex[i], comp);
66     }
67 }
68 for(int i=0; i<n; i++){
69     for(int j=0; j<adj[i].size(); j++){
70         int v = adj[i][j];
71         if(find(scc[i]) != find(scc[v])){
72             scc_adj[scc[i]].push_back(scc[v]);
73             merge(scc[i], scc[v]);
74         }
75     }
76 }
77 }

```

## 5.5. Centroid Decomposition

```

1 /* Centroid Decomposition Implementation */
2 /* c_p[] contains the centroid predecessor on centroid tree */
3 /* removed[] says if the node was already selected as a centroid (limit the
   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 #define N MAX_N
8
9 vector<int> adj[N];
10 bool removed[N];
11 int L[N], subsz[N];
12 int c_p[N];
13
14 void init(int n){
15     for(int i=0; i<=n; i++){
16         removed[i] = false;
17         adj[i].clear();
18         L[i] = 0;
19         subsz[i] = 1;
20         c_p[i] = -1;
21     }
22 }
23
24 void centroid_subsz(int u, int p){
25     subsz[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_subsz(v, u);
30         subsz[u] += subsz[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(subsz[v] > subsz[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_subsz(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 }

```

## 5.6. Floyd Warshall Shortest Path

```

1 int dist[N][N][N];
2
3 void relax(int i, int j, int k){
4     dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j]);
5 }
6
7 void floyd_warshall(){
8     for(int k=0; k<=n; k++){
9         for(int i=1; i<=n; i++){
10             for(int j=1; j<=n; j++){
11                 if(i==j) dist[k][i][j] = 0;
12                 else dist[k][i][j] = INF;
13             }
14         }
15     }
16     for(int k=1; k<=n; k++){
17         for(int i=1; i<=n; i++){
18             for(int j=1; j<=n; j++){
19                 relax(i,j,k);
20             }
21         }
22     }
23 }

```

## 5.7. Tarjan's Bridge/Articulations Algorithm

```

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

```

## 5.8. Max Flow Dinic's Algorithm

```

1 struct Dinic {
2
3     struct FlowEdge{
4         int v, rev, c;
5         FlowEdge() {}
6         FlowEdge(int v, int c, int rev) : v(v), c(c), rev(rev) {}
7     };
8
9     vector< vector<FlowEdge> > adj;
10    vector<int> level, used;
11    int src, snk;
12    int sz;
13    int max_flow;
14    Dinic(){}
15    Dinic(int n){
16        src = 0;
17        snk = n+1;
18        adj.resize(n+2, vector< FlowEdge >());
19        level.resize(n+2);
20        used.resize(n+2);
21        sz = n+2;
22        max_flow = 0;
23    }
24
25    void add_edge(int u, int v, int c){
26        int id1 = adj[u].size();
27        int id2 = adj[v].size();
28        adj[u].pb(FlowEdge(v, c, id2));
29        adj[v].pb(FlowEdge(u, 0, id1));
30    }
31
32    void add_to_src(int v, int c){
33        adj[src].pb(FlowEdge(v, c, -1));
34    }
35
36    void add_to_snk(int u, int c){
37        adj[u].pb(FlowEdge(snk, c, -1));
38    }
39
40    bool bfs(){
41        for(int i=0; i<sz; i++){
42            level[i] = -1;
43        }
44
45        level[src] = 0;
46        queue<int> q; q.push(src);
47
48        while(!q.empty()){
49            int cur = q.front();
50            q.pop();
51            for(FlowEdge e : adj[cur]){
52                if(level[e.v] == -1 && e.c > 0){
53                    level[e.v] = level[cur]+1;
54                    q.push(e.v);
55                }
56            }
57        }
58
59        return (level[snk] == -1 ? false : true);
60    }
61
62    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
95    };

```

## 5.9. HLD

```

1 //Uses Segment tree
2
3
4 int L[N], vis[N], vis2[N], P[N], ch[N], subsh[N], st[N], ed[N], heavy[N];
5 int t = 0;
6 vector<int> adj[N];
7 int n,q;
8
9 void init(int n){
10     t = 0;
11     for(int i=0; i<=n; i++){
12         vis[i] = false;
13         vis2[i] = false;
14         adj[i].clear();
15         ch[i] = i;
16         L[i] = 0;
17         P[i] = -1;
18         subsh[i] = 1;
19         heavy[i] = -1;
20     }
21 }
22
23 void pre_dfs(int u){
24     vis[u] = true;
25     for(int i=0; i<adj[u].size(); i++){
26         int v = adj[u][i];
27         if(vis[v]) continue;
28         P[v] = u;
29         L[v]=L[u]+1;
30         pre_dfs(v);
31         if(heavy[u] == -1 || subsh[heavy[u]] < subsh[v]) heavy[u] = v;
32         subsh[u]+=subsh[v];
33     }
34 }
35
36 void st_dfs(int u){
37     vis2[u] = true;
38     st[u]=t;
39     v[t++] = //segtree value
40     if(heavy[u] != -1){
41         ch[heavy[u]] = ch[u];
42         st_dfs(heavy[u]);
43     }
44     for(int i=0; i<adj[u].size(); i++){
45         int v = adj[u][i];
46         if(vis2[v] || v == heavy[u]) continue;
47         st_dfs(v);
48     }
49     ed[u] = t;
50     v[t++] = 0; //trick
51 }
52
53 void update() {
54 }
55
56 void query() {
57 }
58
59

```

## 5.10. LCA

```

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

```

## 6. Math and Number Theory

### 6.1. 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 }

```

### 6.2. Erathostenes Sieve + Logn Prime Factorization

```

1  /* Erathostenes Sieve Implementation */
2  /* Calculate primes from 2 to N */
3  /* lf[i] stores the lowest prime factor of i(logn factorization) */
4
5  bitset<N> prime;
6  int lf[N];
7
8  void run_sieve(int n){
9      for(int i=0; i<=n; i++) lf[i] = i;
10     prime.set();
11     prime[0] = false;
12     prime[1] = false;
13     for(int p = 2; p*p <= n; p++){
14         if(prime[p]){
15             for(int i=p*p; i<=n; i+=p){
16                 prime[i] = false;
17                 lf[i] = min(lf[i], p);
18             }
19         }
20     }
21 }

```

### 6.3. Matrix Exponentiation

```

1  /* Matrix Exponentiation Implementation */
2
3  typedef vector< vector<int> > Matrix;
4
5  Matrix operator *(const Matrix & a, const Matrix & b){
6      Matrix c(a.size(), vector<int>(b[0].size()));
7      for(int i = 0; i<a.size(); i++){
8          for(int j = 0; j<b[0].size(); j++){
9              for(int k = 0; k<b.size(); k++){
10                 c[i][j] += (a[i][k]*b[k][j]);
11             }
12         }
13     }
14     return c;
15 }
16
17 Matrix exp(Matrix & a, int k){
18     if(k == 1) return a;
19     Matrix c = exp(a, k/2);
20     c = c*c;
21     if(k%2) c = c*a;
22     return c;
23 }

```

### 6.4. Fast Fourier Transform - Recursive and Iterative

```

1  /* Fast Fourier Transform Implementation */
2  /* Complex numbers implemented by hand */
3  /* Poly needs to have degree of next power of 2 (result poly has size
   next_pot2(2*n) */
4  /* Uses Roots of Unity Idea ( $Z^n = 1$ , divide and conquer strategy)
5  /* Inverse FFT only changes to the conjugate of Primitive Root of Unity */
6  /* Remember to use round to get integer value of Coefficients of Poly C */
7  /* Iterative FFT is way faster (bit reversal idea + straightforward conquer
   for each block of each size) */
8  /* std::complex doubles the execution time */
9
10 struct FFT{
11
12     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             this->a = b.a;
31             this->b = b.b;
32         }
33
34         Complex operator+(const Complex & y) const {

```



```

35     return Complex(a + y.a, b + y.b);
36 }
37
38 Complex operator-(const Complex & y) const {
39     return Complex(a - y.a, b - y.b);
40 }
41
42 Complex operator*(const Complex & y) const {
43     return Complex(a*y.a - b*y.b, a*y.b + b*y.a);
44 }
45
46 Complex operator/(const double & x) const {
47     return Complex(a/x, b/x);
48 }
49
50 Complex operator/(const Complex & y) const {
51     return (*this)*(y.conjugate())/y.size2();
52 }
53
54 };
55
56 struct Poly{
57     vector<Complex> c;
58     Poly() {}
59
60     Poly(int n){
61         int sz = (31 - __builtin_clz(n)%32) + 1;
62         c.resize((1 << (sz-1) == n ? n : (1<<sz))<<1);
63     }
64
65     int size() const{
66         return (int)c.size();
67     }
68
69 };
70
71 inline Complex PrimitiveRootOfUnity(int n){
72     const double PI = acos(-1);
73     return Complex(cos(2*PI/(double)n), sin(2*PI/(double)n));
74 }
75
76 inline Complex InversePrimitiveRootOfUnity(int n){
77     const double PI = acos(-1);
78     return Complex(cos(-2*PI/(double)n), sin(-2*PI/(double)n));
79 }
80
81 void DFT(Poly & A, bool inverse){
82     int n = A.size();
83     int lg = 0;
84     while(n > 0) lg++, n>>=1;
85     n = A.size();
86     lg-=2;
87
88     for(int i=0; i<n; i++){
89         int j = 0;
90         for(int b=0; b <= lg; b++){
91             if(i & (1 << b)) j |= (1 << (lg - b));
92         }
93         if(i < j) swap(A.c[i], A.c[j]);
94     }
95
96     for(int len=2; len <= n; len <= 1){
97         Complex w;
98         if(inverse) w = InversePrimitiveRootOfUnity(len);
99         else w = PrimitiveRootOfUnity(len);

```

```

100
101     for(int i=0; i<n; i+=len){
102         Complex x(1,0);
103         for(int j=0; j<len/2; j++){
104             Complex u = A.c[i+j], v = x*A.c[i+j+len/2];
105             A.c[i+j] = u + v;
106             A.c[i+j+len/2] = u - v;
107             x = x*w;
108         }
109     }
110 }
111
112 if(inverse) for(int i=0; i<n; i++) A.c[i] = A.c[i]/n;
113 }
114
115 /* Skipable */
116 Poly RecursiveFFT(Poly A, int n, Complex w){
117     if(n == 1) return A;
118
119     Poly A_even(n/2), A_odd(n/2);
120
121     for(int i=0; i<n; i+=2){
122         A_even.c[i/2] = A.c[i];
123         A_odd.c[i/2] = A.c[i+1];
124     }
125
126     Poly F_even = RecursiveFFT(A_even, n/2, w*w);
127     Poly F_odd = RecursiveFFT(A_odd, n/2, w*w);
128     Poly F(n);
129     Complex x(1, 0);
130
131     for(int i=0; i<n/2; i++){
132         F.c[i] = F_even.c[i] + x*F_odd.c[i];
133         F.c[i + n/2] = F_even.c[i] - x*F_odd.c[i];
134         x = x*w;
135     }
136
137     return F;
138 }
139
140 /* Skipable */
141 Poly Convolution(Poly & F_A, Poly & F_B){
142     Poly F_C(F_A.size()>>1);
143     for(int i=0; i<F_A.size(); i++) F_C.c[i] = F_A.c[i]*F_B.c[i];
144     return F_C;
145 }
146
147 Poly multiply(Poly & A, Poly & B){
148     DFT(A, false);
149
150     DFT(B, false);
151
152     Poly C = Convolution(A, B);
153
154     DFT(C, true);
155
156     return C;
157 }
158
159 }fft;

```

## 7. String Algorithms

### 7.1. KMP Failure Function + String Matching

```

1  /* Knuth - Morris - Pratt Algorithm */
2
3  struct KMP{
4      vector<int> pi;
5
6      string t;
7
8      vector<int> matches;
9
10     KMP() {}
11
12     KMP(string s){
13         pi.resize(s.size());
14         t = s;
15     }
16
17     void calculate() {
18         int n = t.size();
19         pi[0] = 0;
20         for(int i = 1; i < n; i++) {
21             pi[i] = pi[i-1];
22             while(pi[i] > 0 && t[i] != t[pi[i]]) pi[i] = pi[pi[i]-1];
23             if(t[i] == t[pi[i]]) pi[i]++;
24         }
25     }
26
27     void matching(string s){
28         int j = 0;
29         int n = s.size();
30         for(int i=0; i<n; i++){
31             while(j > 0 && s[i] != t[j]) j = pi[j-1];
32             if(s[i] == t[j]) j++;
33             if(j == t.size()){
34                 matches.push_back(i-t.size()+1);
35                 j = pi[j-1];
36             }
37         }
38     }
39
40 };

```

### 7.2. Z-Function

```

1  /* Z-function */
2  /* Calculate the size K of the largest substring which is a prefix */
3
4  struct ZFunction{
5
6      vector<int> z;
7
8      string t;
9
10     ZFunction() {}
11
12     ZFunction(string s){
13         t = s;
14         z.resize(t.size());
15     }
16
17     void calculate() {

```

```

18     int n = t.size();
19     z[0] = 0;
20     int l = 0, r = 0;
21     for(int i=1; i<n; i++){
22         if(i > r){
23             l = i;
24             r = i;
25         }
26         z[i] = min(z[i-1], r-i+1);
27         while(i + z[i] < n && t[i + z[i]] == t[z[i]]) z[i]++;
28         if(i + z[i] > r){
29             l = i;
30             r = i + z[i]-1;
31         }
32     }
33 }
34
35 };

```

### 7.3. Rolling Hash

```

1  /* Rolling Hash Implementation */
2  /* Uses 1-indexed string */
3
4  struct RollingHash{
5
6      long long BASE = 137
7      long long PRIME = (int)1e9+9;
8
9      string a;
10     vector<long long> hash;
11     vector<long long> base;
12     vector<long long> invBase;
13
14     RollingHash() {}
15
16     RollingHash(string s){
17         a = s;
18         hash.resize(s.size());
19         base.resize(s.size());
20         invBase.resize(s.size());
21     }
22
23     long long expo(long long a, long long k){
24         if(k == 0) return 1LL;
25         else if(k == 1) return a;
26         long long aux = expo(a, k/2);
27         aux %= PRIME;
28         aux *= aux;
29         aux %= PRIME;
30         if(k%2) aux *= a;
31         aux %= PRIME;
32         return aux;
33     }
34
35     void calculate(string a){
36         base[0] = 1;
37         invBase[0] = 1;
38         hash[0] = 0;
39         for(int i=1; i<=a.size(); i++){
40             hash[i] += BASE*hash[i-1] + a[i-1];
41             hash[i] %= PRIME;
42             base[i] = base[i-1]*BASE;
43             base[i] %= PRIME;

```

```

44     invBase[i] = expo(base[i], PRIME-2);
45 }
46 }
47
48 long long query(int i, int j){
49     return ((h[j] - h[i-1])*invBase[j-i+1] + PRIME)%PRIME;
50 }
51
52 };

```

#### 7.4. 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;
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          tmp.resize(max(257LL, n+1));
22          sa.resize(n+1);
23          sa_aux.resize(n+1);
24          lcp.resize(n+1);
25          block = 0;
26      }
27
28      bool suffixcmp(int i, int j){
29          if(rnk[i] != rnk[j]) return rnk[i] < rnk[j];
30          i+=block, j+=block;
31          i%=n;
32          j%=n;
33          return rnk[i] < rnk[j];
34      }
35
36      void suffixSort(int MAX_VAL){
37          for(int i=0; i<=MAX_VAL; i++) tmp[i] = 0;
38          for(int i=0; i<n; i++) tmp[rnk[i]]++;
39          for(int i=1; i<=MAX_VAL; i++) tmp[i] += tmp[i-1];
40          for(int i = n-1; i>=0; i--){
41              int aux = sa[i]-block;
42              aux%=n;
43              if(aux < 0) aux+=n;
44              sa_aux[--tmp[rnk[aux]]] = aux;
45          }
46          for(int i=0; i<n; i++) sa[i] = sa_aux[i];
47          tmp[0] = 0;
48          for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
49          for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
50      }
51

```

```

52 void calculate(){
53     s+='\0';
54     n++;
55     for(int i=0; i<n; i++){
56         sa[i] = i;
57         rnk[i] = s[i];
58         tmp[i] = 0;
59     }
60     suffixSort(256);
61     block = 1;
62     while(tmp[n-1] != n-1){
63         suffixSort(tmp[n-1]);
64         block*=2;
65     }
66     for(int i=0; i<n-1; i++) sa[i] = sa[i+1];
67     n--;
68     tmp[0] = 0;
69     for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
70     for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
71     s.pop_back();
72 }
73
74 void calculate_lcp(){
75     int last = 0;
76     for(int i=0; i<n; i++){
77         if(rnk[i] == n-1) continue;
78         int x = rnk[i];
79         lcp[x] = max(0LL, last-1);
80         while(sa[x] + lcp[x] < n && sa[x+1] + lcp[x] < n && s[sa[x]+lcp[x]] ==
81             s[sa[x+1]+lcp[x]]){
82             lcp[x]++;
83         }
84         last = lcp[x];
85     }
86
87     int lcp(int x, int y){
88         if(x == y) return n - x;
89         if(rnk[x] > rnk[y]) swap(x,y);
90         return rmq(rnk[x], rnk[y]-1);
91     }
92
93 };

```