Competitive Programming Algorithms and Topics

Ubiratan Neto

1. Template

1.1. Template Code

```
#include <bits/stdc++.h>

#define int long long
#define ff first
#define ss second
#define endl '\n'
#define ii pair<int, int>
#define DESYNC ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0)
#define pb push_back
#define vi vector<int>
#define vii vector< ii >
#define EPS 1e-9
#define INF 1e18
#define ROOT 1

using namespace std;

inline int mod(int n){ return (n%1000000007); }

int gcd(int a, int b){
  if(a == 0 || b == 0) return 0;
  if(b == 1) return b;
  else return gcd(b, a%b);
}

int32_t main(){
  DESYNC;

}
```

2. Data Structures

2.1. Segment Tree using Pointers

```
/* Segment Tree implementation using pointers */
/* Can be adapted to Persistent Segment Tree */

struct node {
  node *left, *right;
  //attributes of node
  node() {
    //initialize attributes
    left = NULL;
    right = NULL;
  }
};

struct DynamicSegmentTree{
  void combine(node *ans, node *left, node *right){
    //combine operation
  }
  void build(node *root, int l, int r){
    if(l == r){
      root->sum = v[l];
      return;
    }
    int m = (l+r) >> 1;
    if(!root->left) root->left = new node();
    if(!root->right) root->right = new node();
    build(root->left, l, m);
    build(root->right, m+1, r);
    combine(root, root->left, root->right);
  }

  void update(node *root, int l, int r, int idx, int val){
    if(l == r && l == idx){
      //do leaf operation
      return;
    }
    int m = (l+r) >> 1;
    if(idx <= m){
      if(!root->left) root->left = new node();
      update(root->left, l, m, idx, val);
    }
    else {
      if(!root->right) root->right = new node();
      update(root->right, m+1, r, idx, val);
    }
    combine(root, root->left, root->right);
  }

  node* query(node *root, int l, int r, int a, int b){
    if(l == a && r == b){
      return root;
    }
    int m = (l+r) >> 1;
    if(b <= m){
      if(!root->left) root->left = new node();
      return query(root->left, l, m, a, b);
    }
    else if(m < a){
      if(!root->right) root->right = new node();
      return query(root->right, m+1, r, a, b);
    }
    if(!root->left) root->left = new node();
    if(!root->right) root->right = new node();
    node *left = query(root->left, l,m,a,m);
    node *right = query(root->right, m+1, r, m+1, b);
    node *ans = new node();
    combine(ans, left, right);
    return ans;
  }
}
```

2.2. Range Update Segment Tree

```
/* Range update Segment Tree Implementation */
/* The first node (ROOT) is defined to 1 (1 - index impl) */
/* N is the maximum number of elements given by the statement */
/* Lazy can be inside node structure instead of being another structure */

#define ROOT 1
#define N MAX_INPUT

struct node{
  //attributes of node
};

node tree[4*N];
node lazy[4*N];

node combine(node a, node b){
  node res;
  //combine operations
  return res;
}

void propagate(int root, int l , int r){
  //return if there is no update
  //update tree using lazy node
  if(l != r){
    //propagate for left and right child
  }
  //reset lazy node
}

void range_update(int root, int l, int r, int a, int b, long long val){
  if(l == a && r == b){
    //lazy operation using val
    return;
  }

  int m = (l+r)/2;

  if(b <= m) range_update(2*root, l, m, a, b, val);
  else if(m < a) range_update(2*root+1, m+1, r, a, b, val);
  else {
    range_update(2*root, l, m, a, m, val);
    range_update(2*root+1, m+1, r, m+1, b, val);
  }

  propagate(root, l , r);
  propagate(2*root, l, m);
  propagate(2*root+1, m+1, r);
  tree[root] = combine(tree[2*root], tree[2*root+1]);
}

node query(int root, int l, int r, int a, int b){
  propagate(root, l, r);
```

```
  if(l == a && r == b) return tree[root];

  int m = (l+r)/2;
  if(b <= m) return query(2*root, l, m, a, b);
  else if(m < a) return query(2*root+1, m+1, r, a, b);
  else {
    node left = query(2*root, l, m, a, m);
    node right = query(2*root+1, m+1, r, m+1, b);
    node ans = combine(left, right);
    return ans;
  }
}
```

2.3. Range Update Binary Indexed Tree

```
/* Range Update Binary Indexed Tree Implementation */
/* Tree is 1 - index */
/* Point Update Binary Indexed Tree operations are used as auxiliar */
/* N is defined as the maximum number of elements (given by the statement) */

#define N MAX_INPUT

int bit[2][N+1];

void init(int n){
  for(int i=1; i<=n; i++){
    bit[0][i] = 0;
    bit[1][i] = 0;
  }
}

//auxiliar functions

void update(int *bit, int idx, int val, int n){
  for(int i = idx; i <= n; i += i&-i){
    bit[i]+=val;
  }
}

int query(int *bit, int idx){
  int ans = 0;
  for(int i=idx; i>0; i -= i&-i){
    ans += bit[i];
  }
  return ans;
}

//end of auxiliar functions

void range_update(int l, int r, int val, int n){
  update(bit[0], l, val, n);
  update(bit[0], r+1, -val, n);
  update(bit[1], l, val*(l-1), n);
  update(bit[1], r+1, -val*r, n);
}

int prefix_query(int idx){
  return query(bit[0],idx)*idx - query(bit[1], idx);
}

int range_query(int l, int r){
  return prefix_query(r) - prefix_query(l-1);
}
```

2.4. Trie

```
struct tnode {
  tnode *adj[SIZE_NODE];
  tnode(){
    for(int i=0; i<SIZE_NODE; i++) adj[i] =  NULL;
  }
};

struct Trie{

  tnode *t;

  void init(){
    t = new tnode();
  }

  void add(){
    tnode *cur = t;

  }

  int query(){
    tnode *cur = t;
  }

  void remove(){
    tnode *cur = t;
  }

} trie;
```

2.5. STL Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

typedef tree<
int, //change for pair<int,int> to use like multiset
null_type,
less<int>, //change for pair<int,int> to use like multiset
rb_tree_tag,
tree_order_statistics_node_update>
ordered_set;

//int differ = 0; for multiset

//ordered_set myset; //declares a stl ordered set
//myset.insert(1); //inserts
//myset.insert(make_pair(1, differ++)); //insertion for multiset
//myset.find_by_order(k)//returns an iterator to the k-th element (or
    returns the end)
//myset.order_of_key(x)//returns the number of elements strictly less than x
//myset.order_of_key(myset.lower_bound(make_pair(x, 0))) //for multisets
```

3. Uncategorized

3.1. Longest Increasing Subsequence

```
struct LIS{
  vector<int> seq;

  void calculate(vector<int> & v){
    int n = v.size();
    seq.resize(n+1);
    for(int i=1; i<=n; i++) seq[i] = INT_MAX;
    seq[0] = INT_MIN;
    for(int i=0; i<n; i++){
      int index = lower_bound(seq.begin(), seq.end(), v[i]) - seq.begin();
      index--;
      seq[index+1] = min(seq[index+1], v[i]);
    }
  }
} lis;
```

3.2. Inversion Count - Merge Sort

```
long long mergesort_count(vector<int> & v){
  vector<int> a,b;
  if(v.size() == 1) return 0;
  for(int i=0; i<v.size()/2; i++) a.push_back(v[i]);
  for(int i=v.size()/2; i<v.size(); i++) b.push_back(v[i]);
  long long ans = 0;
  ans += mergesort_count(a);
  ans += mergesort_count(b);
  a.push_back(LLONG_MAX);
  b.push_back(LLONG_MAX);
  int x = 0, y = 0;
  for(int i=0; i<v.size(); i++){
    if(a[x] <= b[y]){
      v[i] = a[x++];
    }
    else {
      v[i] = b[y++];
      ans += a.size() - x -1;
    }
  }
  return ans;
}
```

3.3. Mo's Decomposition

```
struct query{
  int id, l, r, ans;
  bool operator<(const query & b) const {
    return l/(int)sqrt(n) < b.l/(int)sqrt(n) || l/(int)sqrt(n) ==
    b.l/(int)sqrt(n) && r < b.r;
  }
};

struct SqrtDecomposition {

  vector<query> q;

  void init(int n){
    q.resize(n);
  }

  void add(int idx){

  }

  void remove(int idx){

  }

  int answer_query(){

  }

  void calculate(){
    sort(q.begin(), q.end());
    int l = 0, r = -1;
    for(int i=0; i<q.size(); i++){
      while(q[i].l < l) add(--l);
      while(r < q[i].r) add(++r);
      while(q[i].l > l) remove(l++);
      while(r > q[i].r) remove(r--);
      q[i].ans = answer_query();
    }
  }

  void print_ans(){
    sort(q.begin(), q.end(), [](const query & a, const query & b){
      return a.id < b.id;
    });

    for(query x : q){
      printf("%d\n", x.ans);
    }
  }
} mos;
```

4. Geometry

4.1. Closest Pair of Points

```
/* Closest Pair of Points Problem Implementation */
/* Divide and Conquer Approach */
/* Using the observation of only checking points inside min_dist x min_dist
    from mid */
/* Binary search boosts search for the right border start point */

struct vec2 {
  long long x, y;
};

bool cmp(vec2 a, vec2 b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}

pair<vec2, vec2> ans;

long long solve(vector<vec2> &a) {

  long long mid = a[a.size()/2].x;
  int n = a.size();

  vector<vec2> l;
  vector<vec2> r;
  int i = 0;
  for(; i < a.size()/2; i++) l.push_back(a[i]);
  for(; i < a.size(); i++) r.push_back(a[i]);

  long long d = LLONG_MAX;

  if(l.size() > 1) {
    d = min(d, solve(l));
  } if(r.size() > 1) {
    d = min(d, solve(r));
  }

  a.clear();

  vector<vec2> ll;
  vector<vec2> rr;


  int j = 0;
  i = 0;
  for(int k=0; k<n; k++){
    if(i < l.size() && j < r.size()){
      if(r[j].y <= l[i].y){
        if((r[j].x - mid)*(r[j].x - mid) < d) {
          rr.push_back(r[j]);
        }
        a.push_back(r[j++]);
      }
      else {
        if((l[i].x - mid)*(l[i].x - mid) < d) {
          ll.push_back(l[i]);
        }
        a.push_back(l[i++]);
      }
    }
    else if(i < l.size()){
      if((l[i].x - mid)*(l[i].x - mid) < d) {
        ll.push_back(l[i]);
      }
      a.push_back(l[i++]);
    }
    else {
      if((r[j].x - mid)*(r[j].x - mid) < d) {
          rr.push_back(r[j]);
        }
        a.push_back(r[j++]);
    }
  }

  for(int i = 0; i < ll.size(); i++) {

    int ini = 0, end = rr.size()-1;
    int j;
    while(ini < end) {
      j = (ini + end) / 2;
      if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
        ini = j+1;
      else end = j;
    }
    j = ini;

    for(; j < rr.size(); j++) {
      if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
      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);
      if(cur < d) {
        d = cur;
        long long cur2 =  (ans.first.x - ans.second.x)*(ans.first.x -
    ans.second.x)
              +(ans.first.y - ans.second.y)*(ans.first.y - ans.second.y);
        if(cur < cur2)
          ans = { ll[i], rr[j] };
      }
    }
  }
  return d;
}
```

4.2. 2D Structures

```
struct Point2D {
  int x,y;

  Point2D(){
    x = 0;
    y = 0;
  }

  Point2D(int x, int y) : x(x), y(y) {}

  Point2D operator+(const Point2D b) const{
    return Point2D(x + b.x, y + b.y);
  }

  Point2D operator-(const Point2D b) const{
    return Point2D(x - b.x, y - b.y);
  }

  bool operator<(const Point2D b) const{
    return x < b.x || (x == b.x && y < b.y);
  }

```

```
  void operator=(const Point2D b) const{
    x = b.x;
    y = b.y;
  }

  double distanceTo(Point2D b){
    return sqrt((x - b.x)*(x - b.x) + (y - b.y)*(y - b.y));
  }

  int distanceTo2(Point2D b){
    return (x - b.x)*(x - b.x) + (y - b.y)*(y - b.y);
  }

};

struct Vector2D {
  int x,y;

  Vector2D(){
    x = 0;
    y = 0;
  }

  Vector2D(int x, int y) : x(x), y(y) {}

  Vector2D(Point2D a, Point2D b){
    x = b.x - a.x;
    y = b.y - a.y;
  }

  Vector2D operator+(const Vector2D b) const{
    return Vector2D(x + b.x, y + b.y);
  }

  Vector2D operator-(const Vector2D b) const{
    return Vector2D(x - b.x, y - b.y);
  }

  void operator=(const Vector2D b) const{
    x = b.x;
    y = b.y;
  }

  int operator*(const Vector2D b) const{
    return (x*b.x + y*b.y);
  }

  int operator^(const Vector2D b) const{
    return x*b.y - y*b.x;
  }

  bool operator<(const Vector2D b) const{
    return x < b.x || (x == b.x && y < b.y);
  }

  Vector2D scale(int n){
    return Vector2D(x*n, y*n);
  }

  double size(){
    return sqrt(x*x + y*y);
  }

  int size2(){
    return x*x + y*y;
  }

  Vector2D normalize(){
    return Vector2D((double)x/size(), (double)y/size());
  }

};

struct Line2D {
  Point2D p, q;
  Vector2D v;
  Vector2D normal;

  int a,b,c;

  Line2D() {
    p = Point2D();
    q = Point2D();
    v = Vector2D();
    normal = Vector2D();
    a = 0;
    b = 0;
    c = 0;
  }

  void operator=(const Line2D l) const{
    a = l.a;
    b = l.b;
    c = l.c;
    p = l.p;
    q = l.q;
    v = l.v;
    normal = l.normal;
  }

  Line2D(Point2D r, Point2D s){
    p = r;
    q = s;
    v = Vector2D(r, s);
    normal = Vector2D(-v.y, v.x);
    a = -v.y;
    b = v.x;
    c = -(a*p.x + b*p.y);
  }

  Line2D(Point2D r, Vector2D s){
    p = r;
    q = Point2D(p.x + s.x, p.y + s.y);
    v = s;
    normal = Vector2D(-v.y, v.x);
    a = -v.y;
    b = v.x;
    c = -(a*p.x + b*p.y);
  }

};
```

4.3. 2D Geometry Functions

```
struct Geo2D {

  double distancePointLine(Point2D p, Line2D l){
    return double(1.*abs(l.a*p.x + l.b*p.y + l.c))/l.normal.size();
  }

  double distancePointSegment(Point2D p, Line2D l){
    int dot1 = Vector2D(l.p, p)*Vector2D(l.p, l.q);
    int dot2 = Vector2D(l.q, p)*Vector2D(l.q, l.p);

    if(dot1 >= 0 && dot2 >= 0) return distancePointLine(p, l);
    else return min(p.distanceTo(l.p), p.distanceTo(l.q));
  }

  double distancePointRay(Point2D p, Line2D l){
    int dot = Vector2D(l.p, p)*l.v;
    if(dot >= 0) return distancePointLine(p, l);
    else return p.distanceTo(l.p);
  }

  Point2D closestPointInSegment(Point2D p, Line2D s){
    //returns closest point from p in segment s
    Vector2D u = s.v.normalize();
    Vector2D w(s.p, p);
    Vector2D res = u.scale(u*w);
    if(u*w < 0 || u*w > s.p.distanceTo(s.q)){
      if(p.distanceTo(s.p) < p.distanceTo(s.q)) return s.p;
      else return s.q;
    }
    else return Point2D(s.p.x + res.x, s.p.y + res.y);
  }

  bool intersectionSegmentSegment(Line2D s1, Line2D s2){
    //Assuming that endpoints are ordered by x
    if(s1.p.x > s1.q.x) swap(s1.p, s1.q);
    if(s2.p.x > s2.q.x) swap(s2.p, s2.q);
    if(abs(s1.v^s2.v) <= EPS){
      Vector2D v1(s2.p, s1.p);
      if(s1.p.x == s1.q.x && s2.p.x == s2.q.x && s1.p.x == s2.p.x){
        Point2D ansl, ansr;
        if(s1.p.y > s1.q.y) swap(s1.p, s1.q);
        if(s2.p.y > s2.q.y) swap(s2.p, s2.q);
        if(s1.p.y <= s2.p.y) ansl = s2.p;
        else ansl = s1.p;
        if(s2.q.y <= s1.q.y) ansr = s2.q;
        else ansr = s1.q;
        if(ansl.x == ansr.x && ansl.y == ansr.y){
          //cout << ansr.x << " " << ansr.y << endl;
          return true;
        }
        else if(ansr.y < ansl.y){
          //cout << "Empty" << endl;
          return false;
        }
        else {
          if(ansl.x == ansr.x && ansl.y > ansr.y) swap(ansl, ansr);
          //cout << ansl.x << " " << ansl.y << endl << ansr.x << " " << ansr.y << endl;
          return true;
        }
      }
      else if(abs(s1.v^v1) <= EPS){
        Point2D ansl, ansr;
        if(s1.p.x <= s2.p.x) ansl = s2.p;
        else ansl = s1.p;
        if(s2.q.x <= s1.q.x) ansr = s2.q;
        else ansr = s1.q;
        if(ansl.x == ansr.x && ansl.y == ansr.y){
          //cout << ansr.x << " " << ansr.y << endl;
          return true;
        }
        else if(ansr.x < ansl.x){
          //cout << "Empty" << endl;
          return false;
        }
        else {
          if(ansl.x == ansr.x && ansl.y > ansr.y) swap(ansl, ansr);
          //cout << ansl.x << " " << ansl.y << endl << ansr.x << " " << ansr.y << endl;
          return true;
        }
      }
      else {
        //cout << "Empty" << endl;
        return false;
      }

    }
    else {
      int a1 = s1.q.y - s1.p.y;
      int b1 = s1.p.x - s1.q.x;
      int c1 = a1*s1.p.x + b1*s1.p.y;
      int a2 = s2.q.y - s2.p.y;
      int b2 = s2.p.x - s2.q.x;
      int c2 = a2*s2.p.x + b2*s2.p.y;
      int det = a1*b2 - a2*b1;
      double x = (double)(b2*c1 - b1*c2)/(double)det*1.;
      double y = (double)(a1*c2 - a2*c1)/(double)det*1.;
      if(s1.p.x-EPS <= x && x <= s1.q.x+EPS && s2.p.x-EPS <= x && x <= s2.q.x+EPS){
        //cout << x << " " << y << endl;
        return true;
      }
      else {
        //cout << "Empty" << endl;
        return false;
      }
    }
  }

  double distanceSegmentSegment(Line2D l1, Line2D l2){
    if((l1.v^l2.v) != 0){

      Line2D r1(l1.p, l1.q);
      Line2D r2(l1.q, l1.p);
      Line2D r3(l2.p, l2.q);
      Line2D r4(l2.q, l2.p);

      int cross1 = (Vector2D(r3.p, r1.p)^r3.v);
      int cross2 = (Vector2D(r3.p, r1.q)^r3.v);
      if(cross2 < cross1) swap(cross1, cross2);

      bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p, r3) > distancePointLine(r1.q, r3));

      cross1 = (Vector2D(r1.p, r3.p)^r1.v);
      cross2 = (Vector2D(r1.p, r3.q)^r1.v);
      if(cross2 < cross1) swap(cross1, cross2);
```

```
125
126       bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p,
          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,
          r3) > distancePointLine(r2.q, r3));
133
134       cross1 = (Vector2D(r2.p, r3.p)^r2.v);
135       cross2 = (Vector2D(r2.p, r3.q)^r2.v);
136       if(cross2 < cross1) swap(cross1, cross2);
137
138       bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p,
          r2) > distancePointLine(r3.q, r2));
139
140       cross1 = (Vector2D(r4.p, r1.p)^r4.v);
141       cross2 = (Vector2D(r4.p, r1.q)^r4.v);
142       if(cross2 < cross1) swap(cross1, cross2);
143
144       bool ok5 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p,
          r4) > distancePointLine(r1.q, r4));
145
146       cross1 = (Vector2D(r1.p, r4.p)^r1.v);
147       cross2 = (Vector2D(r1.p, r4.q)^r1.v);
148       if(cross2 < cross1) swap(cross1, cross2);
149
150       bool ok6 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p,
          r1) > distancePointLine(r4.q, r1));
151
152       cross1 = (Vector2D(r4.p, r2.p)^r4.v);
153       cross2 = (Vector2D(r4.p, r2.q)^r4.v);
154       if(cross2 < cross1) swap(cross1, cross2);
155
156       bool ok7 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p,
          r4) > distancePointLine(r2.q, r4));
157
158       cross1 = (Vector2D(r2.p, r4.p)^r2.v);
159       cross2 = (Vector2D(r2.p, r4.q)^r2.v);
160       if(cross2 < cross1) swap(cross1, cross2);
161
162       bool ok8 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p,
          r2) > distancePointLine(r4.q, r2));
163
164       if(ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8) return 0;
165
166     }
167
168     double ans = distancePointSegment(l1.p, l2);
169     ans = min(ans, distancePointSegment(l1.q, l2));
170     ans = min(ans, distancePointSegment(l2.p, l1));
171     ans = min(ans, distancePointSegment(l2.q, l1));
172     return ans;
173   }
174
175   double distanceSegmentRay(Line2D s, Line2D r){
176     if((s.v^r.v) != 0){
177       Line2D r1(s.p, s.q);
178       Line2D r2(s.q, s.p);
179
180       int cross1 = (Vector2D(r.p, r1.p)^r.v);
181       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)
          > distancePointLine(r1.q, r));
185
186       cross1 = (Vector2D(r1.p, r.p)^r1.v);
187       cross2 = (Vector2D(r1.p, r.q)^r1.v);
188       if(cross2 < cross1) swap(cross1, cross2);
189
190       bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r1)
          > distancePointLine(r.q, r1));
191
192       cross1 = (Vector2D(r.p, r2.p)^r.v);
193       cross2 = (Vector2D(r.p, r2.q)^r.v);
194       if(cross2 < cross1) swap(cross1, cross2);
195
196       bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r)
          > distancePointLine(r2.q, r));
197
198       cross1 = (Vector2D(r2.p, r.p)^r2.v);
199       cross2 = (Vector2D(r2.p, r.q)^r2.v);
200       if(cross2 < cross1) swap(cross1, cross2);
201
202       bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r2)
          > distancePointLine(r.q, r2));
203
204       if(ok1 && ok2 && ok3 && ok4) return 0;
205     }
206
207
208     double ans = INF;
209     int dot = Vector2D(s.p, r.p)*Vector2D(r.p, s.q);
210     if(dot >= 0) ans = min(ans, distancePointLine(r.p, s));
211     else ans = min(ans, min(r.p.distanceTo(s.p), r.p.distanceTo(s.q)));
212
213     dot = Vector2D(r.p, s.p)*r.v;
214     if(dot >= 0) ans = min(ans, distancePointLine(s.p, r));
215     else ans = min(ans, r.p.distanceTo(s.p));
216
217     dot = Vector2D(r.p, s.q)*r.v;
218     if(dot >= 0) ans = min(ans, distancePointLine(s.q, r));
219     else ans = min(ans, r.p.distanceTo(s.q));
220
221     return ans;
222
223   }
224
225   double distanceSegmentLine(Line2D s, Line2D l){
226     if((s.v^l.v) == 0){
227       return distancePointLine(s.p, l);
228     }
229
230     int cross1 = (Vector2D(l.p, s.p)^l.v);
231     int cross2 = (Vector2D(l.p, s.q)^l.v);
232     if(cross2 < cross1) swap(cross1, cross2);
233     if(cross1 <= 0 && cross2 >= 0) return 0;
234     else return min(distancePointLine(s.p, l), distancePointLine(s.q,l));
235
236   }
237
238   double distanceLineRay(Line2D l, Line2D r){
239     if((l.v^r.v) == 0){
240       return distancePointLine(r.p, l);
241     }
242
```

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

  double distanceLineLine(Line2D l1, Line2D l2){
    if((l1.v^l2.v) == 0){
      return distancePointLine(l1.p, l2);
    }
    else return 0;
  }

  double distanceRayRay(Line2D r1, Line2D r2){
    if((r1.v^r2.v) != 0){

      int cross1 = (Vector2D(r1.p, r2.p)^r1.v);
      int cross2 = (Vector2D(r1.p, r2.q)^r1.v);
      if(cross2 < cross1) swap(cross1, cross2);
      bool ok1 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r1) > distancePointLine(r2.q, r1));

      cross1 = (Vector2D(r2.p, r1.p)^r2.v);
      cross2 = (Vector2D(r2.p, r1.q)^r2.v);
      if(cross2 < cross1) swap(cross1, cross2);

      bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p, r2) > distancePointLine(r1.q, r2));

      if(ok1 && ok2) return 0;

    }

    double ans = INF;
    int dot = Vector2D(r2.p, r1.p)*r2.v;
    if(dot >= 0) ans = min(ans, distancePointLine(r1.p, r2));
    else ans = min(ans, r2.p.distanceTo(r1.p));

    dot = Vector2D(r1.p, r2.p)*r1.v;
    if(dot >= 0) ans = min(ans, distancePointLine(r2.p, r1));
    else ans = min(ans, r1.p.distanceTo(r2.p));

    return ans;

  }

} geo2d;
```

4.4. Convex Hull - Monotone Chain Algorithm

```
struct ConvexHull {
  vector< Point2D > points, lower, upper;

  ConvexHull(){}

  void calculate(vector<Point2D> v){
    sort(v.begin(), v.end());
    for(int i=0; i<v.size(); i++){
      while(upper.size() >= 2 && (Vector2D(upper[upper.size()-2], upper.back())^Vector2D(upper.back(), v[i])) >= 0LL) upper.pop_back();
      upper.push_back(v[i]);
    }
    reverse(v.begin(), v.end());
    for(int i=0; i<v.size(); i++){
      while(lower.size() >= 2 && (Vector2D(lower[lower.size()-2], lower.back())^Vector2D(lower.back(), v[i])) >= 0LL) lower.pop_back();
      lower.push_back(v[i]);
    }
    for(int i=upper.size()-2; i>=0; i--) points.push_back(upper[i]);
    for(int i=lower.size()-2; i>=0; i--) points.push_back(lower[i]);
  }

  double area(){
    double area = points.back().x*points[0].y - points.back().y*points[0].x;
    for(int i=0; i<points.size()-1; i++){
      area += points[i].x*points[i+1].y - points[i].y*points[i+1].x;
    }
    return area/2.;
  }

  int area2(){
    int area2 = points.back().x*points[0].y - points.back().y*points[0].x;
    for(int i=0; i<points.size()-1; i++){
      area2 += points[i].x*points[i+1].y - points[i].y*points[i+1].x;
    }
    return area2;
  }

  long double perimeter(){
    long double val = Vector2D(points[0], points.back()).size();
    for(int i=0; i<points.size()-1; i++){
      val += Vector2D(points[i], points[i+1]).size();
    }
    return val;
  }

} chull;
```

5. Graphs

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

```
/* Dynammic Connectivity Implementation */
/* Uses Divide and Conquer Offline approach */
/* Able to answer if two vertex <u,v> are connected */
/* No multi-edges allowed */
/* DSU + Rollback is used to backtrack merges */
/* N is defined as the maximum graph size given by input */

#define N MAX_INPUT

int uf[N];
int sz[N];

struct event{
  int op, u, v, l, r;
  event() {}
  event(int o, int a, int b, int x, int y) : op(o), u(a), v(b), l(x), r(y) {}
};

map< pair<int, int>, int > edge_to_l;
stack< pair<int*,int> > hist;
vector<event> events;

int init(int n){
  for(int i=0; i<=n; i++){
    uf[i] = i;
    sz[i] = 1;
  }
}

int find(int u){
  if(uf[u] == u) return u;
  else return find(uf[u]);
}

void merge(int u, int v){
  int a = find(u);
  int b = find(v);
  if(a == b) return;
  if(sz[a] < sz[b]){
    hist.push(make_pair(&uf[a], uf[a]));
    uf[a] = b;
    hist.push(make_pair(&sz[b], sz[b]));
    sz[b]+= sz[a];
  }
  else {
    hist.push(make_pair(&uf[b], uf[b]));
    hist.push(make_pair(&sz[a], sz[a]));
    uf[b] = a;
    sz[a]+=sz[b];
  }
}

int snap(){
  return hist.size();
}

void rollback(int t){
  while(hist.size() > t){
    pair<int*, int> aux = hist.top();
    hist.pop();
    *aux.first = aux.second;
  }
}

void solve(int l, int r){
  if(l == r){
    if(events[l].op == 2){
      if(find(events[l].u) == find(events[l].v)) cout << "YES" << endl;
      else cout << "NO" << endl;
    }
    return;
  }

  int m = (l+r)/2;
  //doing for [L,m]
  int t = snap();
  for(int i=l; i<=r; i++){
    if(events[i].op == 0 || events[i].op == 1){
      if(events[i].l <= l && m <= events[i].r) merge(events[i].u,
    events[i].v);
    }
  }
  solve(l, m);
  rollback(t);

  //doing for [m+1, R]
  t = snap();
  for(int i=l; i<=r; i++){
    if(events[i].op == 0 || events[i].op == 1){
      if(events[i].l <= m+1 && r <= events[i].r) merge(events[i].u,
    events[i].v);
    }
  }
  solve(m+1, r);
  rollback(t);
}

void offline_process(){
  int n, q;
  cin >> n >> q; //number of vertex and queries
  init(n);
  for(int i=0; i<q; i++){
    string op;
    int u,v;
    cin >> op >> u >> v; //add, remove or query for u,v
    if(u > v) swap(u,v);
    if(op == "add"){
      events.push_back(event(0, u, v, i, -1));
      edge_to_l[make_pair(u,v)] = i;
    }
    else if(op == "rem"){
      int l = edge_to_l[make_pair(u,v)];
      events.push_back(event(1, u, v, l, i));
      events[l].r = i;
    }
    else if(op == "conn"){
      events.push_back(event(2, u, v, -1, -1));
    }
  }
  for(int i=0; i<q; i++){
    if(events[i].op == 0){
      if(events[i].r == -1){
        events[i].r = events.size();
        events.push_back(event(1, events[i].u, events[i].v, events[i].l,
    events[i].r));
      }
```

```
    }
  }
}
```

5.2. Bellman Ford Shortest Path

```
struct BellmanFord{

  struct edges {
    int u, v, weight;
    edges(int u , int v, int weight) :
    u(u),
    v(v),
    weight(weight) {}
  };

  vector<int> dist;

  vector<edges> e;

  bool cycle = false;

  BellmanFord(){}

  BellmanFord(int n, int m){
    dist.resize(n+1);
    e.resize(m+1);
  }

  void calculate(int source){
    for(int i=0; i<=dist.size(); i++){
      dist[i] = INT_MAX;
    }
    dist[source] = 0;
    for(int k=0; k<dist.size()-1; k++){
      for(int i=0; i<e.size(); i++){
        if(dist[e[i].v] > dist[e[i].u] + e[i].weight){
          dist[e[i].v] = dist[e[i].u] + e[i].weight;
        }
      }
    }
    for(int i=0; i<e.size(); i++){
      if(dist[e[i].v] > dist[e[i].u] + e[i].weight){
        cycle = true;
      }
    }
  }
};
```

5.3. Eulerian Circuits/Paths

```
//Graph - Euler path

//for undirected graph
//circuit - 2 vertex with odd grades
//simple path - all vertex with even grades
//this algorithm generates a circuit, if you need a path between u,v
//create a new edge u-v, compute circuit u..u, then delete the last u

//for directed graph
//circuit - all vertex needs enter grade = exit grade
//path - one vertex needs to have one more enter grade
//and the other needs to have one more exit grade
//this algorithm generates a circuit, if you need a path between u,v
//create a new edge u-v, considering that u have one more enter grade
//and v one more exit grade

struct EulerianCircuit {
  vector< set<int> > adj;
  vector<int> walk;
  vector<int> deg;
  int s, t;

  EulerianCiruit();
  EulerianCircuit(int n){
    deg.resize(n+1);
    adj.resize(n+1);
  }

  void undirected_euler(int u){
    while(!adj[u].empty()){
      int v = *(--adj[u].end());

      adj[u].erase(v);
      adj[v].erase(adj[v].find(u));

      euler(v);
    }

    walk.push_back(u);
  }

  void directed_euler(int u){
    while(!adj[u].empty()){
      int v = *(--adj[u].end());

      adj[u].erase(v);

      euler(v);
    }

    walk.push_back(u);
  }

};
```

5.4. Kosaraju SCC

```
//Implementation uses dsu (cut parallel edges)

vector<int> adj[N];
vector<int> adj_t[N];
vector<int> scc_adj[N];
int ed[N];
int tempo,comp;
bool vis[N];
int scc[N];

int init(int n){
  tempo = 0;
  for(int i=0; i<n; i++){
    adj[i].clear();
    adj_t[i].clear();
    scc_adj[i].clear();
    ed[i] = -1;
          vis[i] = false;
          uf[i]  = i;
          sz[i] = 1;
  }
}

void dfs(int u){
  vis[u] = true;
  for(int i=0; i<adj[u].size(); i++){
    int v = adj[u][i];
    if(!vis[v]) dfs(v);
  }
  ed[u] = ++tempo;
}

void dfst(int u, int comp){
  scc[u] = comp;
  vis[u] = true;
  for(int i=0; i<adj_t[u].size(); i++){
    int v = adj_t[u][i];
    if(!vis[v]) dfst(v,comp);
  }
}

bool cmp_end(const int & a, const int & b){
  return ed[a] > ed[b];
}

void calculate_scc(int n){
  for(int i=0; i<n; i++){
    vis[i] = false;
  }
  for(int i=0; i<n; i++){
    if(!vis[i]){
      dfs(i);
    }
  }
  vector<int> vertex(n+1);
  for(int i=0; i<n; i++){
    vis[i] = false;
    vertex[i] = i;
  }
  sort(vertex.begin(), vertex.end(), cmp_end);
  comp=-1;
  for(int i=0; i<vertex.size(); i++){
    if(!vis[vertex[i]]){
      comp++;
      dfst(vertex[i],comp);
    }
  }
  for(int i=0; i<n; i++){
    for(int j=0; j<adj[i].size(); j++){
      int v = adj[i][j];
      if(find(scc[i]) != find(scc[v])){
        scc_adj[scc[i]].push_back(scc[v]);
        merge(scc[i], scc[v]);
      }
    }
  }
}
```

5.5. Centroid Decomposition

```
/* Centroid Decomposition Implementation */
/* c_p[] contains the centroid predecessor on centroid tree */
/* removed[] says if the node was already selected as a centroid (limit the
    subtree search) */
/* L[] contains the height of the vertex (from root) on centroid tree (Max
    is logN) */
/* N is equal to the maximum size of tree (given by statement) */

#define N MAX_N

vector<int> adj[N];
bool removed[N];
int L[N], subsz[N];
int c_p[N];

void init(int n){
  for(int i=0; i<=n;i++){
    removed[i] = false;
    adj[i].clear();
    L[i] = 0;
    subsz[i] = 1;
    c_p[i] = -1;
  }
}

void centroid_subsz(int u, int p){
  subsz[u]= 1;
  for(int i=0; i<adj[u].size(); i++){
    int v = adj[u][i];
    if(v == p || removed[v]) continue;
    centroid_subsz(v,u);
    subsz[u] += subsz[v];
  }
}

int find_centroid(int u, int p, int sub){
  for(int i=0; i<adj[u].size(); i++){
    int v = adj[u][i];
    if(v == p || removed[v]) continue;
    if(subsz[v] > subsz[sub]/2){
      return find_centroid(v, u, sub);
    }
  }
  return u;
}

void centroid_decomp(int u, int p, int r){
```

```
  centroid_subsz(u,-1);
  int centroid = find_centroid(u, -1, u);
  L[centroid] = r;
  c_p[centroid] = p;
  removed[centroid] = true;

  //problem pre-processing

  for(int i=0; i<adj[centroid].size(); i++){
    int v = adj[centroid][i];
    if(removed[v]) continue;
    centroid_decomp(v, centroid, r+1);
  }
}
```

5.6. Floyd Warshall Shortest Path

```
int dist[N][N][N];

void relax(int i, int j, int k){
  dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j]);
}

void floyd_warshall(){
  for(int k=0; k<=n; k++){
    for(int i=1; i<=n; i++){
      for(int j=1; j<=n; j++){
        if(i==j) dist[k][i][j] = 0;
        else dist[k][i][j] = INF;
      }
    }
  }
  for(int k=1; k<=n; k++){
    for(int i=1; i<=n; i++){
      for(int j=1; j<=n; j++){
        relax(i,j,k);
      }
    }
  }
}
```

5.7. Tarjan's Bridge/Articulations Algorithm

```
//Graph - Tarjan Bridges Algorithm

//calculate bridges, articulations and all connected components

struct Tarjan{
  int cont = 0;
  vector<int> st;
  vector<int> low;
  vector< ii > bridges;
  vector<bool> isArticulation;

  Tarjan() {}
  Tarjan(int n){
    st.resize(n+1);
    low.resize(n+1);
    isArticulation(n+1);
    cont = 0;
    bridges.clear();
  }

  void tarjan(int u, int p){
    st[u] = low[u] = ++cont;
    int son = 0;
    for(int i=0; i<adj[u].size(); i++){
      if(adj[u][i]==p){
        p = 0;
        continue;
      }
      if(!st[adj[u][i]]){
        tarjan(adj[u][i], u);
        low[u] = min(low[u], low[adj[u][i]]);
        if(low[adj[u][i]] >= st[u]) isArticulation[u] = true; //check
    articulation

        if(low[adj[u][i]] > st[u]){ //check if its a bridge
          bridges.push_back(ii(u, adj[u][i]));
        }

        son++;
      }
      else low[u] = min(low[u], st[adj[u][i]]);
    }

    if(p == -1){
      if(son > 1) isArticulation[u] = true;
      else isArticulation[u] = false;
    }
  }
};
```

5.8. Max Flow Dinic's Algorithm

```
struct Dinic {

  struct FlowEdge{
    int v, rev, c;
    FlowEdge() {}
    FlowEdge(int v, int c, int rev) : v(v), c(c), rev(rev) {}
  };

  vector< vector<FlowEdge> >  adj;
  vector<int> level, used;
  int src, snk;
  int sz;
  int max_flow;
  Dinic(){}
  Dinic(int n){
    src = 0;
    snk = n+1;
    adj.resize(n+2, vector< FlowEdge >());
    level.resize(n+2);
    used.resize(n+2);
    sz = n+2;
    max_flow = 0;
  }

  void add_edge(int u, int v, int c){
    int id1 = adj[u].size();
    int id2 = adj[v].size();
    adj[u].pb(FlowEdge(v, c, id2));
    adj[v].pb(FlowEdge(u, 0, id1));
  }

  void add_to_src(int v, int c){
    adj[src].pb(FlowEdge(v, c, -1));
  }

  void add_to_snk(int u, int c){
    adj[u].pb(FlowEdge(snk, c, -1));
  }

  bool bfs(){
    for(int i=0; i<sz; i++){
      level[i] = -1;
    }

    level[src] = 0;
    queue<int> q; q.push(src);

    while(!q.empty()){
      int cur = q.front();
      q.pop();
      for(FlowEdge e : adj[cur]){
        if(level[e.v] == -1 && e.c > 0){
          level[e.v] = level[cur]+1;
          q.push(e.v);
        }
      }
    }

    return (level[snk] == -1 ? false : true);
  }

  int send_flow(int u, int flow){
    if(u == snk) return flow;

    for(int &i = used[u]; i<adj[u].size(); i++){
      FlowEdge &e = adj[u][i];

      if(level[u]+1 != level[e.v] || e.c <= 0) continue;

      int new_flow = min(flow, e.c);
      int adjusted_flow = send_flow(e.v, new_flow);

      if(adjusted_flow > 0){
        e.c -= adjusted_flow;
        if(e.rev != -1) adj[e.v][e.rev].c += adjusted_flow;
        return adjusted_flow;
      }
    }

    return 0;
  }

  void calculate(){
    if(src == snk){max_flow = -1; return;} //not sure if needed

    max_flow = 0;

    while(bfs()){
      for(int i=0; i<sz; i++) used[i] = 0;
      while(int inc = send_flow(src, INF)) max_flow += inc;
    }

  }

};
```

5.9. HLD

```
1
2  //Uses Segment tree
3
4  int L[N], vis[N], vis2[N], P[N], ch[N], subsz[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     subsz[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 || subsz[heavy[u]] < subsz[v]) heavy[u] = v;
32     subsz[u]+=subsz[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
57 void query(){
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
61 } lca;
```

6. Math and Number Theory

6.1. Binomial Coefficient DP

```
/* Dynammic Programming for Binomial Coefficient Calculation */
/* Using Stiefel Rule C(n, k) = C(n-1, k) + C(n-1, k-1) */

int binomial(int n ,int k){
  int c[n+10][k + 10];
  memset(c, 0 , sizeof c);
  c[0][0] = 1;
  for(int i = 1;i<=n;i++){
    for(int j = min(i, k);j>0;j--){
      c[i][j] = c[i-1][j] + c[i-1][j-1];
    }
  }
  return c[n][k];
}
```

6.2. Erathostenes Sieve + Logn Prime Factorization

```
/* Erasthostenes Sieve Implementation */
/* Calculate primes from 2 to N */
/* lf[i] stores the lowest prime factor of i(logn factorization) */

bitset<N> prime;
int lf[N];

void run_sieve(int n){
  for(int i=0; i<=n; i++) lf[i] = i;
  prime.set();
  prime[0] = false;
  prime[1] = false;
  for(int p = 2; p*p <= n; p++){
    if(prime[p]){
      for(int i=p*p; i<=n; i+=p){
        prime[i] = false;
        lf[i] = min(lf[i], p);
      }
    }
  }
}
```

6.3. Matrix Exponentiation

```
/* Matrix Exponentiation Implementation */

typedef vector< vector<int> > Matrix;

Matrix operator *(const Matrix & a, const Matrix & b){
  Matrix c(a.size(), vector<int>(b[0].size()));
  for(int i = 0; i<a.size(); i++){
    for(int j = 0; j<b[0].size(); j++){
      for(int k = 0; k<b.size(); k++){
        c[i][j] += (a[i][k]*b[k][j]);
      }
    }
  }
  return c;
}

Matrix exp(Matrix & a, int k){
  if(k == 1) return a;
  Matrix c = exp(a, k/2);
  c = c*c;
  if(k%2) c = c*a;
  return c;
}
```

6.4. Fast Fourier Transform - Recursive and Iterative

```
/* Fast Fourier Transform Implementation */
/* Complex numbers implemented by hand */
/* Poly needs to have degree of next power of 2 (result poly has size
    next_pot2(2*n) */
/* Uses Roots of Unity Idea (Z^n = 1, divide and conquer strategy)
/* Inverse FFT only changes to the conjugate of Primitive Root of Unity */
/* Remember to use round to get integer value of Coefficients of Poly C */
/* Iterative FFT is way faster (bit reversal idea + straightforward conquer
    for each block of each size) */
/* std::complex doubles the execution time */

struct FFT{

  FFT() {}

  struct Complex{
    double a, b;

    Complex(double a, double b) : a(a), b(b) {}

    Complex() : a(0), b(0) {}

    Complex conjugate() const {
      return Complex(a, -b);
    }

    double size2() const {
      return a*a + b*b;
    }

    void operator=(const Complex & b){
      this->a = b.a;
      this->b = b.b;
    }

    Complex operator+(const Complex & y) const {
```

```
      return Complex(a + y.a, b + y.b);
    }

    Complex operator-(const Complex & y) const {
      return Complex(a - y.a, b - y.b);
    }

    Complex operator*(const Complex & y) const {
      return Complex(a*y.a - b*y.b, a*y.b + b*y.a);
    }

    Complex operator/(const double & x) const {
      return Complex(a/x, b/x);
    }

    Complex operator/(const Complex & y) const {
      return (*this)*(y.conjugate()/y.size2());
    }

  };

  struct Poly{
    vector<Complex> c;
    Poly() {}

    Poly(int n){
      int sz = (31 - __builtin_clz(n)%32) + 1;
      c.resize((1 << (sz-1) == n ? n : (1<<sz))<<1);
    }

    int size() const{
      return (int)c.size();
    }

  };

  inline Complex PrimitiveRootOfUnity(int n){
    const double PI = acos(-1);
    return Complex(cos(2*PI/(double)n), sin(2*PI/(double)n));
  }

  inline Complex InversePrimitiveRootOfUnity(int n){
    const double PI = acos(-1);
    return Complex(cos(-2*PI/(double)n), sin(-2*PI/(double)n));
  }

  void DFT(Poly & A, bool inverse){
    int n = A.size();
    int lg = 0;
    while(n > 0) lg++, n>>=1;
    n = A.size();
    lg-=2;

    for(int i=0; i<n; i++){
      int j = 0;
      for(int b=0; b <= lg; b++){
        if(i & (1 << b)) j |= (1 << (lg - b));
      }
      if(i < j) swap(A.c[i], A.c[j]);
    }

    for(int len=2; len <= n; len <<= 1){
      Complex w;
      if(inverse) w = InversePrimitiveRootOfUnity(len);
      else w = PrimitiveRootOfUnity(len);

      for(int i=0; i<n; i+=len){
        Complex x(1,0);
        for(int j=0; j<len/2; j++){
          Complex u = A.c[i+j], v = x*A.c[i+j+len/2];
          A.c[i+j] = u + v;
          A.c[i+j+len/2] = u - v;
          x = x*w;
        }
      }
    }

    if(inverse) for(int i=0; i<n; i++) A.c[i] = A.c[i]/n;
  }

  /* Skipable */
  Poly RecursiveFFT(Poly A, int n, Complex w){
    if(n == 1) return A;

    Poly A_even(n/2), A_odd(n/2);

    for(int i=0; i<n; i+=2){
      A_even.c[i/2] = A.c[i];
      A_odd.c[i/2] = A.c[i+1];
    }

    Poly F_even = RecursiveFFT(A_even, n/2, w*w);
    Poly F_odd = RecursiveFFT(A_odd, n/2, w*w);
    Poly F(n);
    Complex x(1, 0);

    for(int i=0; i<n/2; i++){
      F.c[i] = F_even.c[i] + x*F_odd.c[i];
      F.c[i + n/2] = F_even.c[i] -  x*F_odd.c[i];
      x = x*w;
    }

    return F;
  }
  /* Skipable */

  Poly Convolution(Poly & F_A, Poly & F_B){
    Poly F_C(F_A.size()>>1);
    for(int i=0; i<F_A.size(); i++) F_C.c[i] = F_A.c[i]*F_B.c[i];
    return F_C;
  }

  Poly multiply(Poly & A, Poly & B){
    DFT(A, false);

    DFT(B, false);

    Poly C = Convolution(A, B);

    DFT(C, true);

    return C;
  }

}fft;
```

7. String Algorithms

7.1. KMP Failure Function + String Matching

```
/* Knuth - Morris - Pratt Algorithm */

struct KMP{
  vector<int> pi;

  string t;

  vector<int> matches;

  KMP() {}

  KMP(string s){
    pi.resize(s.size());
    t = s;
  }

  void calculate() {
    int n = t.size();
    pi[0] = 0;
    for(int i = 1; i < n; i++) {
      pi[i] = pi[i-1];
      while(pi[i] > 0 && t[i] != t[pi[i]]) pi[i] = pi[pi[i]-1];
      if(t[i] == t[pi[i]]) pi[i]++;
    }
  }

  void matching(string s){
    int j = 0;
    int n = s.size();
    for(int i=0; i<n; i++){
      while(j > 0 && s[i] != t[j]) j = pi[j-1];
      if(s[i] == t[j]) j++;
      if(j == t.size()){
        matches.push_back(i-t.size()+1);
        j = pi[j-1];
      }
    }
  }

};
```

7.2. Z-Function

```
/* Z-function */
/* Calculate the size K of the largest substring which is a prefix */

struct ZFunction{

  vector<int> z;

  string t;

  ZFunction() {}

  ZFunction(string s){
    t = s;
    z.resize(t.size());
  }

  void calculate(){
    int n = t.size();
    z[0] = 0;
    int l = 0, r = 0;
    for(int i=1; i<n; i++){
      if(i > r){
        l = i;
        r = i;
      }
      z[i] = min(z[i-l], r-i+1);
      while(i + z[i] < n && t[i + z[i]] == t[z[i]]) z[i]++;
      if(i + z[i] > r){
        l = i;
        r = i + z[i]-1;
      }
    }
  }

};
```

7.3. Rolling Hash

```
/* Rolling Hash Implementation */
/* Uses 1-indexed string */

struct RollingHash{

  long long BASE = 137
  long long PRIME = (int)1e9+9;

  string a;
  vector<long long> hash;
  vector<long long> base;
  vector<long long> invBase;

  RollingHash() {}

  RollingHash(string s){
    a = s;
    hash.resize(s.size());
    base.resize(s.size());
    invBase.resize(s.size());
  }

  long long expo(long long a, long long k){
    if(k == 0) return 1LL;
    else if(k == 1) return a;
    long long aux = expo(a, k/2);
    aux %= PRIME;
    aux *= aux;
    aux %= PRIME;
    if(k%2) aux *= a;
    aux %= PRIME;
    return aux;
  }

  void calculate(string a){
    base[0] = 1;
    invBase[0] = 1;
    hash[0] = 0;
    for(int i=1; i<=a.size(); i++){
      hash[i]+= BASE*hash[i-1] + a[i-1];
      hash[i] % = PRIME;
      base[i] = base[i-1]*BASE;
      base[i] %= PRIME;
```

```
      invBase[i] = expo(base[i], PRIME-2);
    }
  }

  long long query(int i, int j){
    return ((h[j] - h[i-1])*invBase[j-i+1] + PRIME)%PRIME;
  }

};
```

7.4. Suffix Array + Linear Sort

```
/* Suffix Array using Counting Sort Implementation */
/* rnk is inverse of sa array */
/* aux arrays are needed for sorting step */
/* inverse sorting (using rotating arrays and blocks of power of 2) */
/* rmq data structure needed for calculating lcp of two non adjacent
    suffixes sorted */

struct SuffixArray{

  vector<int> rnk,tmp,sa, sa_aux, lcp;

  int block=0, n;

  string s;

  SuffixArray() {}

  SuffixArray(string t){
    s = t;
    n = t.size();
    rnk.resize(n);
    tmp.resize(n);
    sa.resize(n);
    sa_aux.resize(n);
    lcp.resize(n);
    block = 0;
  }

  bool suffixcmp(int i, int j){
    if(rnk[i] != rnk[j]) return rnk[i] < rnk[j];
    i+=block, j+=block;
    i%=n;
    j%=n;
    return rnk[i] < rnk[j];
  }

  void suffixSort(int MAX_VAL){
    for(int i=0; i<=MAX_VAL; i++) tmp[i] = 0;
    for(int i=0; i<n; i++) tmp[rnk[i]]++;
    for(int i=1; i<=MAX_VAL; i++) tmp[i] += tmp[i-1];
    for(int i = n-1; i>=0; i--){
        int aux = sa[i]-block;
        aux%=n;
        if(aux < 0) aux+=n;
        sa_aux[--tmp[rnk[aux]]] = aux;
    }
    for(int i=0; i<n; i++) sa[i] = sa_aux[i];
    tmp[0] = 0;
    for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
    for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
  }

  void calculate(){
    s+='\0';
    n++;
    for(int i=0; i<n; i++){
      sa[i] = i;
      rnk[i] = s[i];
      tmp[i] = 0;
    }
    suffixSort(256);
    block = 1;
    while(tmp[n-1] != n-1){
      suffixSort(tmp[n-1]);
      block*=2;
    }
    for(int i=0; i<n-1; i++) sa[i] = sa[i+1];
    n--;
    tmp[0] = 0;
    for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
    for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
    s.pop_back();
  }

  void calculate_lcp(){
    int last = 0;
    for(int i=0; i<n; i++){
      if(rnk[i] == n-1) continue;
      int x = rnk[i];
      lcp[x] = max(0,last-1);
      while(sa[x] + lcp[x] < n && sa[x+1] + lcp[x] < n && s[sa[x]+lcp[x]] ==
    s[sa[x+1]+lcp[x]]){
        lcp[x]++;
      }
      last = lcp[x];
    }
  }

  int lcp(int x, int y){
    if(x == y) return n - x;
    if(rnk[x] > rnk[y]) swap(x,y);
    return rmq(rnk[x], rnk[y]-1);
  }

};
```