Competitive Programming Algorithms and Topics

Ubiratan Neto

1. Template

1.1. Template Code

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

#define int long long
#define double long double
#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
#define M 1000000007
const double PI = acos(-1);

using namespace std;

inline int mod(int n, int m){ int ret = n%m; if(ret < 0) ret += m; return ret; }

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

int32_t main(){
  DESYNC;

}
```

2. Data Structures

2.1. Dynamic Segment Tree

```
namespace DynamicSegmentTree{

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

  void combine(node *ans, node *left, node *right){
    //combine operation
  }

  void propagate(node * root, int l, int r){
    //check if exists lazy

    //apply lazy on node

    //propagate
    if(!root->left) root->left = new node();
    if(!root->right) root->right = new node();

    if(l != r){
      //propagate operation
    }

    //reset lazy
  }

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

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

```
namespace SegmentTree{

  struct node{
    //attributes of node
    int lazy = 0;
    node() {}
  };

  struct Tree{
    vector<node> st;
    Tree(){}

    Tree(int n){
      st.resize(4*n);
    }

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

    void propagate(int cur, 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 build(int cur, int l, int r){
      if(l == r){
        //leaf operation
        return;
      }

      int m = (l+r)>>1;
      build(2*cur, l, m);
      build(2*cur + 1, m+1, r);
      st[cur] = combine(st[2*cur], st[2*cur+1]);

    }

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

      int m = (l+r)/2;

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

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

    node query(int cur, int l, int r, int a, int b){
      propagate(cur, l, r);
      if(l == a && r == b) return st[cur];

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

}
```

2.3. Fenwick Tree

```
struct BIT {

  vector<int> bit;

  BIT() {}

  int n;

  BIT(int n) {
    this->n = n;
    bit.resize(n+1);
  }

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

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

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

  //int bit 0-1 it finds the index of k-th element active
  int kth(int k) {
    int cur = 0;
    int acc = 0;
    for(int i = 19; i >= 0; i--) {
      if(cur + (1<<i) > n) continue;
      if(acc + bit[cur + (1<<i)] < k) {
        cur += (1<<i);
        acc += bit[cur];
      }
    }
    return ++cur;
  }

};
```

2.4. Trie

```
namespace Trie{

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

  struct Tree{

    node *t;

    Tree(){
      t = new node();
    }

    void add(){
      node *cur = t;

    }

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

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

  };

}
```

2.5. STL Ordered Set

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

//NAMESPACE
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
```

2.6. Convex Hull Trick

```
struct ConvexHullTrick {
  //max cht, suppose lines are added in crescent order of a
  vector<line> cht;
  ConvexHullTrick() {}
  struct line{
    int id, a, b;
    line() {}
    line(int id, int a, int b) : id(id), a(a), b(b) {}
  };
  bool cmp(const line & a, const line & b){
    return (a.a < b.a || (a.a == b.a && a.b > b.b));
  }
  bool remove(line & a, line & b, line & c){
    if((a.a - c.a)*(c.b - b.b) <= (b.a - c.a)*(c.b - a.b)) return true;
    else return false;
  }

  void add(line & v){
    if(cht.empty()){
      cht.push_back(v);
    }
    else {
      if(cht.back().a == v.a) return;
      while(cht.size() > 1 && remove(cht[cht.size()-2], cht.back(), v)){
        cht.pop_back();
      }
      cht.push_back(v);
    }
  }

  void preprocess_cht(vector< line > & v){
    sort(v.begin(), v.end(), cmp);
    cht.clear();
    for(int i=0; i<v.size(); i++){
      add(v[i]);
    }
  }

  int f(int i, int x){
    return cht[i].a*x + cht[i].b;
  }

  //return line index
  int query(int x){
    if(cht.size() == 0) return -1;
    if(cht.size() == 1) return 0;
    int l = 0, r = cht.size()-2;
    int ans= cht.size()-1;
    while(l <= r){
      int m = (l+r)/2;
      int y1 = f(m, x);
      int y2 = f(m+1, x);
      if(y1 >= y2){
        ans = m;
        r = m-1;
      }
      else l = m+1;
    }
    return ans;
  }

};
```

2.7. Lichao Segment Tree - Convex Hull Trick

```
namespace Lichao{
  //min lichao tree

  struct line {
    int a, b;
    line() {
      a = 0;
      b = INF;
    }
    line(int a, int b) : a(a), b(b) {}
    int eval(int x){
      return a*x + b;
    }
  };

  struct node {
    node * left, * right;
    line ln;
    node(){
      left = NULL;
      right = NULL;
    }
  };

  struct Tree {

    node * root;

    Tree() {
      root = new node();
    }

    void add(node * root, int l, int r, line ln){
      if(!root->left) root->left = new node();
      if(!root->right) root->right = new node();
      int m = (l+r)>>1;
      bool left = ln.eval(l) < (root->ln).eval(l);
      bool mid = ln.eval(m) < (root->ln).eval(m);

      if(mid){
        swap(root->ln, ln);
      }

      if(l == r) return;
      else if(left != mid) add(root->left, l, m, ln);
      else add(root->right, m+1, r, ln);
    }

    int query(node * root, int l, int r, int x){
      if(!root->left) root->left = new node();
      if(!root->right) root->right = new node();
      int m = (l+r)>>1;
      if(l == r) return (root->ln).eval(x);
      else if(x < m) return min((root->ln).eval(x), query(root->left, l, m, x));
      else return min((root->ln).eval(x), query(root->right, m+1, r, x));
    }

  };

}
```

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

```
namespace Lichao{
  //min lichao tree working with doubles

  struct line {
    double a, b;
    line() {
      a = 0;
      b = INF;
    }
    line(double a, double b) : a(a), b(b) {}
    double eval(double x){
      return a*x + b;
    }
  };

  struct node {
    node * left, * right;
    line ln;
    node(){
      left = NULL;
      right = NULL;
    }
  };

  struct Tree {

    node * root;

    Tree() {
      root = new node();
    }

    void add(node * root, double l, double r, line ln){
      if(!root->left) root->left = new node();
      if(!root->right) root->right = new node();
      double m = (l+r)/2.;
      bool left = ln.eval(l) < (root->ln).eval(l);
      bool mid = ln.eval(m) < (root->ln).eval(m);

      if(mid){
        swap(root->ln, ln);
      }

      if(abs(r-l) <= 1e-9) return;
      else if(left != mid) add(root->left, l, m, ln);
      else add(root->right, m, r, ln);
    }

    double query(node * root, double l, double r, double x){
      if(!root->left) root->left = new node();
      if(!root->right) root->right = new node();
      double m = (l+r)/2.;
      if(abs(r-l) <= 1e-9) return (root->ln).eval(x);
      else if(x < m) return min((root->ln).eval(x), query(root->left, l, m, x));
      else return min((root->ln).eval(x), query(root->right, m, r, x));
    }

  };

}
```

3. Uncategorized

3.1. Coordinate Compression

```
struct Compresser {

  vector<int> value;

  Compresser() {}

  Compresser(int n){
    value.resize(n);
  }

  void compress(vector<int> & v){
    vector<int> tmp;
    set<int> s;
    for(int i=0; i<v.size(); i++) s.insert(v[i]);
    for(int x : s) tmp.pb(x);
    for(int i=0; i<v.size(); i++){
      int idx = lower_bound(tmp.begin(), tmp.end(), v[i]) - tmp.begin();
      value[idx] = v[i];
      v[i] = idx;
    }
  }

} compresser;
```

3.2. Longest Increasing Subsequence

```
/* Use upper_bound to swap to longest non decreasing subsequence */

struct LIS{

  vector<int> seq;

  LIS() {}

  LIS(int n){
    seq.resize(n+1);
  }

  void calculate(vector<int> & v){
    int n = v.size();
    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]);
    }
  }

};
```

3.3. Inversion Count - Merge Sort

```
int 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]);
  int 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;
}
```

3.4. Mo’s Decomposition

```
namespace Mos {

  int sqr;

  struct query{
    int id, l, r, ans;
    bool operator<(const query & b) const {
      if(l/sqr != b.l/sqr) return l/sqr < b.l/sqr;
      return (l/sqr) % 2 ? r > b.r : r < b.r;
    }
  };

  struct QueryDecomposition {

    vector<query> q;

    QueryDecomposition(int n, int nq){
      q.resize(nq);
      sqr = (int)sqrt(n);
    }

    void read(){

    }

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

      for(query x : q){
        cout << x.ans << endl;
      }
    }
  };

}
```

4. Geometry

4.1. 2D Structures

```
//////////////////////////////////  Geometry Structures ////////////////////////////////////

namespace Geo2D {

  struct Point {
    int x,y;

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

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

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

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

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

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

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

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

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

    bool operator==(const Point b){
      return x == b.x && y == b.y;
    }

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

    int squareDistanceTo(Point b){
      return (x - b.x)*(x - b.x) + (y - b.y)*(y - b.y);
    }

    bool operator<(const Point & p) const{
      return tie(x,y) < tie(p.x, p.y);
    }

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

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

    //Only with double type
    Point normalize(){
      return Point((double)x/size(), (double)y/size());
    }

    void rotate(double ang){
      double xx = x, yy = y;
      x = xx*cos(ang) + yy*-sin(ang);
      y = xx*sin(ang) + yy*cos(ang);
    }

  };

  struct Line {
    Point p, q;
    Point v;
    Point normal;

    int a,b,c;

    Line() {
      p = Point();
      q = Point();
      v = Point();
      normal = Point();
      a = 0;
      b = 0;
      c = 0;
    }

    Line(int aa, int bb, int cc){
      a = aa;
      b = bb;
      c = cc;
      normal = Point(a,b);
      v = Point(-normal.y, normal.x);
      p = Point();
      q = Point();
    }

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

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

    void flip_sign(){
      a = -a, b = -b, c = -c;
    }

    void normalize(){
      if(a < 0) flip_sign();
      else if(a == 0 && b < 0) flip_sign();
      else if(a == 0 && b == 0 && c < 0) flip_sign();
      int g = max(a, max(b,c));
      if(a != 0) g = gcd(g, a); if(b != 0) g = gcd(g,b); if(c != 0) g = gcd(g,c);
      if(g > 0) a/=g, b/=g, c/=g;
    }

    bool operator<(const Line & l) const{
      return tie(a,b,c) < tie(l.a, l.b, l.c);
    }

  };

  struct Circle{
    Point c;
    double r;
    Circle() {}
    Circle(Point center, double radius) : c(center), r(radius) {}

    bool operator=(Circle circ){
      c = circ.c;
      r = circ.r;
    }

    pair<Point, Point> getTangentPoints(Point p){
      //p needs to be outside the circle
      double d = p.distanceTo(c);
      double ang = asin(1.*r/d);
      Point v1(p, c);
      v1.rotate(ang);
      Point v2(p, c);
      v2.rotate(-ang);
      v1 = v1.scale(sqrt(d*d - r*r)/d);
      v2 = v2.scale(sqrt(d*d - r*r)/d);
      Point p1(v1.x + p.x, v1.y + p.y);
      Point p2(v2.x + p.x, v2.y + p.y);
      return make_pair(p1,p2);
    }

    double sectorArea(double ang){
      return (ang*r*r)/2.;
    }

    double arcLength(double ang){
      return ang*r;
    }

    double sectorArea(Point p1, Point p2){
      double h = p1.distanceTo(p2);
      double ang = acos(1. - h*h/r*r);
      return sectorArea(ang);
    }

    double arcLength(Point p1, Point p2){
      double h = p1.distanceTo(p2);
      double ang = acos(1. - (h*h)/(2*r*r));
      return arcLength(ang);
    }

    bool inside(const Point & p){
      if(Point(c,p).size() + EPS < r) return true;
      else if(r + EPS < Point(c,p).size()) return false;
      else return true;
    }

  };

}

////////////////////////////////// End of  Geometry Structures /////////////////////////////
```

4.2. 2D Geometry Functions

```
////////////////////////////////// Geometry  Algorithms ////////////////////////////////////

namespace Geo2D {

  double distancePointLine(Point p, Line l){
    if(l.normal.squareSize() == 0) return INF;
    return (double)(1.*abs(l.a*p.x + l.b*p.y + l.c))/l.normal.size();
  }

  double distancePointSegment(Point p, Line l){
    int dot1 = Point(l.p, p)*Point(l.p, l.q);
    int dot2 = Point(l.q, p)*Point(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(Point p, Line l){
    int dot = Point(l.p, p)*l.v;
    if(dot >= 0) return distancePointLine(p, l);
    else return p.distanceTo(l.p);
  }

  Point closestPointInSegment(Point p, Line s){
    //returns closest point from p in segment s
    Point u = s.v.normalize();
    Point w(s.p, p);
    Point 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 Point(s.p.x + res.x, s.p.y + res.y);
  }

  Point intersectionSegmentSegment(Line s1, Line s2){
    //Assumes that intersection exists
    //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){

      //parallel segments
      Point v1(s2.p, s1.p);
      if(s1.p.x == s1.q.x && s2.p.x == s2.q.x && s1.p.x == s2.p.x){
        Point 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 Point(ansr.x, ansr.y);
        }
        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 Point(INF, INF);
        }
      }
      else if(abs(s1.v^v1) <= EPS){
        Point 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 Point(ansr.x, ansr.y);
        }
        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 Point(INF, INF);
        }
      }

    }
    else {
      //general case
      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.;
      //cout << x << " " << y << endl;
      return Point(x,y);
    }
  }


  double distanceSegmentSegment(Line l1, Line l2){
    if(l1.p == l2.p && l1.q == l2.q) return 0;
    if(l1.q == l2.p && l1.p == l2.q) return 0;
    if((l1.v^l2.v) != 0){

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

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

  bool ok2 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p, r1) > distancePointLine(r3.q, r1));

  cross1 = (Point(r3.p, r2.p)^r3.v);
  cross2 = (Point(r3.p, r2.q)^r3.v);
  if(cross2 < cross1) swap(cross1, cross2);

  bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r3) > distancePointLine(r2.q, r3));

  cross1 = (Point(r2.p, r3.p)^r2.v);
  cross2 = (Point(r2.p, r3.q)^r2.v);
  if(cross2 < cross1) swap(cross1, cross2);

  bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r3.p, r2) > distancePointLine(r3.q, r2));

  cross1 = (Point(r4.p, r1.p)^r4.v);
  cross2 = (Point(r4.p, r1.q)^r4.v);
  if(cross2 < cross1) swap(cross1, cross2);

  bool ok5 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r1.p, r4) > distancePointLine(r1.q, r4));

  cross1 = (Point(r1.p, r4.p)^r1.v);
  cross2 = (Point(r1.p, r4.q)^r1.v);
  if(cross2 < cross1) swap(cross1, cross2);

  bool ok6 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p, r1) > distancePointLine(r4.q, r1));

  cross1 = (Point(r4.p, r2.p)^r4.v);
  cross2 = (Point(r4.p, r2.q)^r4.v);
  if(cross2 < cross1) swap(cross1, cross2);

  bool ok7 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r4) > distancePointLine(r2.q, r4));

  cross1 = (Point(r2.p, r4.p)^r2.v);
  cross2 = (Point(r2.p, r4.q)^r2.v);
  if(cross2 < cross1) swap(cross1, cross2);

  bool ok8 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r4.p, r2) > distancePointLine(r4.q, r2));

  if(ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8) return 0;

 }

 double ans = distancePointSegment(l1.p, l2);
 ans = min(ans, distancePointSegment(l1.q, l2));
 ans = min(ans, distancePointSegment(l2.p, l1));
 ans = min(ans, distancePointSegment(l2.q, l1));
 return ans;
}

double distanceSegmentRay(Line s, Line r){
 if((s.v^r.v) != 0){
  Line r1(s.p, s.q);
  Line r2(s.q, s.p);

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

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

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

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

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

  bool ok3 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r2.p, r) > distancePointLine(r2.q, r));

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

  bool ok4 = (cross1 <= 0 && cross2 >= 0) || (distancePointLine(r.p, r2) > distancePointLine(r.q, r2));

  if(ok1 && ok2 && ok3 && ok4) return 0;
 }


 double ans = INF;
 int dot = Point(s.p, r.p)*Point(r.p, s.q);
 if(dot >= 0) ans = min(ans, distancePointLine(r.p, s));
 else ans = min(ans, min(r.p.distanceTo(s.p), r.p.distanceTo(s.q)));

 dot = Point(r.p, s.p)*r.v;
 if(dot >= 0) ans = min(ans, distancePointLine(s.p, r));
 else ans = min(ans, r.p.distanceTo(s.p));

 dot = Point(r.p, s.q)*r.v;
 if(dot >= 0) ans = min(ans, distancePointLine(s.q, r));
 else ans = min(ans, r.p.distanceTo(s.q));

 return ans;

}

double distanceSegmentLine(Line s, Line l){
 if((s.v^l.v) == 0){
  return distancePointLine(s.p, l);
 }

 int cross1 = (Point(l.p, s.p)^l.v);
```

```
    int cross2 = (Point(l.p, s.q)^l.v);
    if(cross2 < cross1) swap(cross1, cross2);
    if(cross1 <= 0 && cross2 >= 0) return 0;
    else return min(distancePointLine(s.p, l), distancePointLine(s.q,l));

  }

  double distanceLineRay(Line l, Line r){
    if((l.v^r.v) == 0){
      return distancePointLine(r.p, l);
    }

    int cross1 = (Point(l.p, r.p)^l.v);
    int cross2 = (Point(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(Line l1, Line l2){
    if((l1.v^l2.v) == 0){
      return distancePointLine(l1.p, l2);
    }
    else return 0;
  }

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

      int cross1 = (Point(r1.p, r2.p)^r1.v);
      int cross2 = (Point(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 = (Point(r2.p, r1.p)^r2.v);
      cross2 = (Point(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 = Point(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 = Point(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;

  }

  double circleCircleIntersection(Circle c1, Circle c2){

    if((c1.r+c2.r)*(c1.r+c2.r) <= (c2.c.x-c1.c.x)*(c2.c.x-c1.c.x) + (c2.c.y-c1.c.y)*(c2.c.y-c1.c.y)){
      return 0;
    }
    if((c1.r-c2.r)*(c1.r-c2.r) >= (c2.c.x-c1.c.x)*(c2.c.x-c1.c.x) + (c2.c.y-c1.c.y)*(c2.c.y-c1.c.y)){
      return PI*min(c1.r, c2.r)*min(c1.r, c2.r);
    }
    double x1 = c1.c.x, x2 = c2.c.x, y1 = c1.c.y, y2 = c2.c.y, r1 = c1.r, r2 = c2.r;
    double d = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
    double r1sqr = c1.r*c1.r;
    double r2sqr = c2.r*c2.r;
    double dsqr = d*d;

    double alpha1 = acos(((c1.r + c2.r)*(c1.r - c2.r) + dsqr)/(2.*d*r1));
    double alpha2 = acos(((c2.r + c1.r)*(c2.r - c1.r) + dsqr)/(2.*d*r2));
    double area1 = r1sqr*(alpha1 - sin(alpha1)*cos(alpha1));
    double area2 = r2sqr*(alpha2 - sin(alpha2)*cos(alpha2));

    return area1 + area2;

  }

  vector<Point> intersectionLineCircle(Line l, Circle circ){
    //NOT TESTED!!!!!!!!
    //no intersection
    if((l.c*l.c)/(circ.r*circ.r) > l.a*l.a + l.b*l.b) return vector<Point>();

    double x0 = -l.a*l.c/(l.a*l.a+l.b*l.b), y0 = -l.b*l.c/(l.a*l.a+l.b*l.b);
    //one intersection
    if(abs((l.c*l.c)/(circ.r*circ.r) - (l.a*l.a + l.b*l.b)) <= EPS){
      vector<Point> ret;
      ret.pb(Point(x0,y0));
      return ret;
    }

    //general case
    double d = circ.r*circ.r - (l.c*l.c)/(l.a*l.a+l.b*l.b);
    double mult = sqrt(d/(l.a*l.a+l.b*l.b));

    Point p1(x0 + l.b*mult, y0 - l.a*mult);
    Point p2(x0 - l.b*mult, y0 + l.a*mult);

    vector<Point> ret;
    ret.pb(p1); ret.pb(p2);
    return ret;
  }

  vector<Point> intersectionCircleCircle(Circle c1, Circle c2){
    //NOT TESTED!!!!!!!!
    //translate first circle to origin
    Point translation = c1.c;
    c1.c = Point(0,0);
    c2.c = c2.c - translation;

    //check if centers are equal
    if(c1.c == c2.c){
      //if radius are equal = infinite intersections(return 3 points to indicate), else = no intersection(empty)
      if(c1.r == c2.r){
        vector<Point> ret;
        ret.pb(Point());
        ret.pb(Point());
        ret.pb(Point());
        return ret;
      }
      else return vector<Point>();
  }

  //general case
  Line l(-2*c2.c.x,-2*c2.c.y, c2.c.x*c2.c.x + c2.c.y*c2.c.y + c1.r*c1.r - c2.r*c2.r);

  vector<Point> ret = intersectionLineCircle(l, c1);

  for(Point & p : ret){
    p = p + translation;
  }

  return ret;
}

Point barycenter(Point & a, Point & b, Point & c, double pA, double pB, double pC){
  Point ret = (a.scale(pA) + b.scale(pB) + c.scale(pC));
  ret.x /= (pA + pB + pC);
  ret.y /= (pA + pB + pC);
  return ret;
}

Point circumcenter(Point & a, Point & b, Point & c){
  double pA = Point(b,c).squareSize(), pB = Point(a,c).squareSize(), pC = Point(a,b).squareSize();
  return barycenter(a,b,c, pA*(pB+pC-pA), pB*(pC+pA-pB), pC*(pA+pB-pC));
}

Point centroid(Point & a, Point & b, Point & c){
  return barycenter(a,b,c,1,1,1);
}

Point incenter(Point & a, Point & b, Point & c){
  return barycenter(a,b,c, Point(b,c).size(), Point(a,c).size(), Point(a,b).size());
}

Point excenter(Point & a, Point & b, Point & c){
  return barycenter(a,b,c, -Point(b,c).size(), Point(a,c).size(), Point(a,b).size());
}

Point orthocenter(Point & a, Point & b, Point & c){
  double pA = Point(b,c).squareSize(), pB = Point(a,c).squareSize(), pC = Point(a,b).squareSize();
  return barycenter(a, b, c, (pA+pB-pC)*(pC+pA-pB), (pB+pC-pA)*(pA+pB-pC), (pC+pA-pB)*(pB+pC-pA));
}

Circle minimumCircle(vector<Point> & v){
  Circle circ(Point(0,0), 1e-14);
  random_shuffle(v.begin(), v.end());
  for(int i=0; i<v.size(); i++){
    if(!circ.inside(v[i])){
      circ = Circle(v[i], 0);
      for(int j=0; j<i; j++){
        if(!circ.inside(v[j])){
          circ = Circle((v[i] + v[j]).scale(0.5), Point(v[i], v[j]).size()*0.5);
          for(int k = 0; k<j; k++){
            if(!circ.inside(v[k])){
              Point center = circumcenter(v[i], v[j], v[k]);
              circ = Circle(center, Point(center, v[k]).size());
            }
          }
        }
      }
    }
  }
  return circ;
}

long long ClosestPairOfPoints(vector<Point> &a) {
  //returns square of distance
  long long mid = a[a.size()/2].x;
  int n = a.size();

  vector<Point> l;
  vector<Point> 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, ClosestPairOfPoints(l));
  } if(r.size() > 1) {
    d = min(d, ClosestPairOfPoints(r));
  }

  a.clear();

  vector<Point> ll;
  vector<Point> 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;
        }
      }
    }
    return d;
  }

}

////////////////////////////////// End of Geometry  Algorithms //////////////////////////////
```

4.3. Convex Hull - Monotone Chain Algorithm

```
namespace Geo2D {

  struct ConvexHull {

    vector< Point > points, lower, upper;

    ConvexHull(){}

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

    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;
    }

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

    bool insideHull(Point p){

      auto it = lower_bound(lower.begin(), lower.end(),  p);
      if(it != lower.end() && *it == p) return true;
      it = lower_bound(upper.begin(), upper.end(), p);
      if(it != upper.end() && *it == p) return true;

      if(p.x == upper[0].x){
        if(p.y > upper[0].y){
          //upper
          if(upper[1].x != upper[0].x) return false;
          else if(p.y <= upper[1].y) return true;
        }
        else {
          //lower
          if(lower[1].x != lower[0].x) return false;
          else if(p.y >= lower[1].y) return true;
        }
        return false;
      }
      Point v1,v2;
      //upper or lower
      int ansu = -1, ansl = -1;
      int l = 0, r = upper.size()-2;
      while(l <= r){
        int m = (l+r)>>1LL;
        if(upper[m].x < p.x && p.x <= upper[m+1].x){
          ansu = m;
          break;
        }
        else if(upper[m+1].x < p.x) l = m+1;
        else r = m-1;
      }
      l = 0, r = lower.size()-2;
      while(l <= r){
        int m = (l+r)>>1LL;
        if(lower[m].x < p.x && p.x <= lower[m+1].x){
          ansl = m;
          break;
        }
        else if(lower[m+1].x < p.x) l = m+1;
        else r = m-1;
      }
      if(ansu == -1 || ansl == -1) return false;
      bool oku = false, okl = false;
      v1 = Point(upper[ansu], upper[ansu+1]);
      v2 = Point(upper[ansu], p);
      oku = ((v1^v2) <= 0);
      v1 = Point(lower[ansl], lower[ansl+1]);
      v2 = Point(lower[ansl], p);
      okl = ((v1^v2) >= 0);
      if(oku && okl) return true;
      else return false;
    }

  };

}
```

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

```
struct SCC {

  vector< vector<int> > adj_t;
  vector< vector<int> > scc_adj;
  vector<int> ed;
  int tempo,comp;
  vector<bool> vis;
  vector<int> scc;

  SCC() {}

  SCC(int n){
    tempo = 0;
    adj_t.resize(n+1, vector<int>());
    scc_adj.resize(n+1, vector<int>());
    ed.resize(n+1);
    comp = 0;
    vis.resize(n+1);
    scc.resize(n+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);
    }
  }

  void calculate(int n){
    for(int i=0; i<=n; i++){
      vis[i] = false;
    }
    for(int i=1; i<=n; i++){
      if(!vis[i]){
        dfs(i);
      }
    }

    vector< ii > vertex(n);

    for(int i=0; i<n; i++){
      vis[i] = false;
      vertex[i] = ii(i+1, ed[i+1]);
    }

    sort(vertex.begin(), vertex.end(), [](const ii & a, const ii & b) {
    return a.ss > b.ss; });

    for(int i=0; i<vertex.size(); i++){
      if(!vis[vertex[i].ff]){
        comp++;
        dfst(vertex[i].ff,comp);
      }
    }
    for(int i=1; i<=n; i++){
      for(int j=0; j<adj[i].size(); j++){
        int v = adj[i][j];
        scc_adj[scc[i]].push_back(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) */

struct CentroidDecomposition {
  vector<bool> removed;
  vector<int> L, subsz;
  vector<int> c_p;

  CentroidDecomposition() {}

  CentroidDecomposition(int n){
    removed.resize(n+1);
    L.resize(n+1);
    c_p.resize(n+1);
    subsz.resize(n+1);
    for(int i=0; i<=n;i++){
      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

```
struct FloydWarshall {

  vector< vector< vector<int> > > dist;

  FloydWarshall() {}

  FloydWarshall(int n){
    dist.resize(n+1, vector< vector< int > >(n+1, vector<int>(n+1)));
  }

  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 calculate(){
    for(int k=0; k<dist.size(); k++){
      for(int i=1; i<dist.size(); i++){
        for(int j=1; j<dist.size(); j++){
          if(i==j) dist[k][i][j] = 0;
          else dist[k][i][j] = INF;
        }
      }
    }
    for(int k=1; k<dist.size(); k++){
      for(int i=1; i<dist.size(); i++){
        for(int j=1; j<dist.size(); 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 calculate(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]]){
        calculate(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, cap;
    FlowEdge() {}
    FlowEdge(int v, int c, int cap, int rev) : v(v), c(c), cap(cap), 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, c, id2));
    adj[v].pb(FlowEdge(u, 0, 0, id1));
  }

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

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

  }

  vector< ii > mincut(){
    bool vis[sz];
    for(int i=0; i<sz; i++) vis[i] = false;
    queue<int> q;
    q.push(src);
    vis[src] = true;
    while(!q.empty()){
      int cur = q.front();
      q.pop();
      for(FlowEdge e : adj[cur]){
        if(e.c > 0 && !vis[e.v]){
          q.push(e.v);
          vis[e.v] = true;
        }
      }
    }
    vector< ii > cut;
    for(int i=1; i<=sz-2; i++){
      if(!vis[i]) continue;
      for(FlowEdge e : adj[i]){
        if(1 <= e.v && e.v <= sz-2 && !vis[e.v] && e.cap > 0 && e.c == 0) cut.pb(ii(i, e.v));
      }
    }
    return cut;
  }

  vector< ii > min_edge_cover(){
    bool covered[sz];
    for(int i=0; i<sz; i++) covered[i] = false;
    vector< ii > edge_cover;
    for(int i=1; i<sz-1; i++){
      for(FlowEdge e : adj[i]){
        if(e.cap == 0 || e.v > sz-2) continue;
        if(e.c == 0){
          edge_cover.pb(ii(i, e.v));
          covered[i] = true;
          covered[e.v] = true;
          break;
        }
      }
    }
    for(int i=1; i<sz-1; i++){
      for(FlowEdge e : adj[i]){
        if(e.cap == 0 || e.v > sz-2) continue;
        if(e.c == 0) continue;
        if(!covered[i] || !covered[e.v]){
          edge_cover.pb(ii(i, e.v));
          covered[i] = true;
          covered[e.v] = true;
        }
      }
    }
    return edge_cover;
  }

};
```

5.9. HLD

```
struct HLD {

  vector<int> L, vis, vis2, P, ch, subsz, st, ed, heavy;
  int t = 0;

  HLD () {}

  HLD(int n){
    L.resize(n+1);
    vis.resize(n+1);
    vis2.resize(n+1);
    P.resize(n+1);
    ch.resize(n+1);
    subsz.resize(n+1);
    st.resize(n+1);
    ed.resize(n+1);
    heavy.resize(n+1);
    t = 0;
    for(int i=0; i<=n; i++){
      ch[i] = i;
      P[i] = -1;
      heavy[i] = -1;
    }
  }

  void precalculate(int u){
    vis[u] = true;
    subsz[u] = 1;
    for(int i=0; i<adj[u].size(); i++){
      int v = adj[u][i];
      if(vis[v]) continue;
      P[v] = u;
      L[v]=L[u]+1;
      precalculate(v);
      if(heavy[u] == -1 || subsz[heavy[u]] < subsz[v]) heavy[u] = v;
      subsz[u]+=subsz[v];
    }
  }

  void build(int u){
    vis2[u] = true;
    st[u]=t;
    v[t++] = //segtree value
    if(heavy[u] != -1){
      ch[heavy[u]] = ch[u];
      build(heavy[u]);
    }
    for(int i=0; i<adj[u].size(); i++){
      int v = adj[u][i];
      if(vis2[v] || v == heavy[u]) continue;
      build(v);
    }
    ed[u] = t;
    v[t++] = 0; //trick
  }

  void update(){
    //update path / subtree / edge
  }

  int query(){
    long long ans;
    if(a == b) return 0;
    while(ch[a] != ch[b]){
      if(L[ch[b]] > L[ch[a]]) swap(a,b);
      //query from st[ch[a]] to st[a]
      a = P[ch[a]];
    }
    if(L[b] < L[a]) swap(b,a);
    if(st[a]+1 <= st[b]) //query from st[a]+1 to st[b]
    return ans;
  }

};
```

5.10. LCA

```
struct LCA {

  int tempo;
  vector<int> st, ed, dad, anc[20];
  vector<bool> vis;

  void init(int n){
    tempo = 0;
    st.resize(n+1);
    ed.resize(n+1);
    dad.resize(n+1);
    for(int i=0; i<20; i++) anc[i].resize(n+1);
    vis.resize(n+1);
    for(int i=0; i<=n; i++) vis[i] = false;
  }

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

  bool is_ancestor(int u, int v){
    return st[u] <= st[v] && st[v] <= ed[u];
  }

  int query(int u, int v){
    if(is_ancestor(u,v)) return u;
    for(int i=19; i>=0; i--){
      if(anc[i][u] == -1) continue;
      if(!is_ancestor(anc[i][u],v)) u = anc[i][u];
    }
    return dad[u];
  }

  void precalculate(){
    dad[1] = -1;
    dfs(1);
    for(int i=1; i<st.size(); i++){
      anc[0][i] = dad[i];
    }
    for(int i=1; i<20; i++){
      for(int j=1; j<st.size(); j++){
          if(anc[i-1][j] != -1){
            anc[i][j] = anc[i-1][anc[i-1][j]];
          }
          else {
            anc[i][j] = -1;
          }
      }
    }
  }

} lca;
```

6. Math and Number Theory

6.1. Diophantine Equations + CRT

```
namespace NT{

  int GCD(int a, int b){
    if(a == 0) return b;
    else return GCD(b%a, a);
  }

  tuple<int,int> ExtendedEuclidean(int a, int b){
    //solves ax+by = gcd(a,b)
    //careful when a or b equal to 0
    if(a == 0) return make_tuple(0,1);
    int x,y;
    tie(x,y) = ExtendedEuclidean(b%a, a);
    return make_tuple(y - (b/a)*x, x);
  }

  bool FailDiophantine = false;

  tuple<int,int> Diophantine(int a, int b, int c){
    FailDiophantine = false;
    //finds a solution for ax+by = c
    //given a solution (x,y), all solutions have the form (x + m*(b/gcd(a,b)), y - m*(a/(gcd(a,b))), multiplied by (c/g)

    int g = GCD(a,b);

    if(g == 0 || c%g != 0) {
      FailDiophantine = true;
      return make_tuple(0,0);
    }

    int x,y;

    tie(x,y) = ExtendedEuclidean(a, b);

    return make_tuple(x*(c/g), y*(c/g));
  }

  bool FailCRT = false;

  tuple<int,int> CRT(vector<int> & a, vector<int> & n){
    FailCRT = false;
    for(int i=0; i<a.size(); i++) a[i] = mod(a[i], n[i]);
    int ans = a[0];
    int modulo = n[0];

    for(int i=1; i<a.size(); i++){
      int x,y;
      tie(x,y) = ExtendedEuclidean(modulo, n[i]);
      int g = GCD(modulo, n[i]);

      if(g == 0 || (a[i] - ans)%g != 0){
        FailCRT = true;
        return make_tuple(0,0);
      }

      ans = mod(ans + (x*(a[i] - ans)/g)%(n[i]/g) * modulo, modulo*n[i]/g);
      modulo = modulo*n[i]/g;
    }

    return make_tuple(ans, modulo);
  }

}
```

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

```
/* Baby-Step Giant-Step Shank's Algorithm */

namespace NT {

  int discrete_log(int a, int b, int p){
    a %= p, b %= p;

    if(b == 1) return 0;

    int cnt = 0, t = 1;
    for(int g = gcd(a, p); g != 1; g = gcd(a, p)){
      if(b % g) return -1;

      p /= g, b /= g, t = t * a / g % p;
      cnt++;

      if(b == t) return cnt;
    }

    map<int, int> hash;
    int m = (sqrt(p) + 1);
    int base = b;

    for(int i = 0; i != m; ++i){
      hash[base] = i;
      base = base * a % p;
    }

    base = 1;
    for(int i=0; i<m; i++){
      base = (base*a)%p;
    }

    int cur = t;
    for(int i = 1; i <= m + 1; ++i){
      cur = cur * base % p;
      if(hash.count(cur)) return i * m - hash[cur] + cnt;
    }
    return -1;
  }

}
```

6.3. 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];
}
```

6.4. Erathostenes Sieve + Logn Prime Factorization

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

namespace NT {

  const int MAX_N = 1123456;

  bitset<MAX_N> prime;
  vector<int> primes;
  int lf[MAX_N];
  int totient[MAX_N];

  void 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);
        }
      }
    }
    for(int i=2; i<=n; i++) if(prime[i]) primes.pb(i);
  }

  void EulerTotient(int n){
    for(int i=0; i<=n; i++) totient[i] = i;
    for(int p = 2; p <= n; p++){
      if(totient[p] == p){
        totient[p] = p-1;
        for(int i=p+p; i<=n; i+=p){
          totient[i] = (totient[i]/p) * (p-1);
        }
      }
    }
  }

};
```

6.5. Segmented Sieve

```
/* Segmented Erathostenes Sieve */
/* Needs primes up to sqrt(N) - Use normal sieve to get them */

namespace NT {

  bitset<MAX_N> prime;
  vector<int> primes;
  vector<int> seg_primes;

  void Sieve(int n){
    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;
        }
      }
    }
    for(int i=2; i<=n; i++) if(prime[i]) primes.pb(i);
  }

  void SegmentedSieve(int l, int r){
    prime.set();
    seg_primes.clear();
    for(int p : primes){
      int start = l - l%p - p;
      while(start < l) start += p;
      if(p == start) start += p;
      for(int i = start; i<=r; i+=p){
        prime[i-l] = false;
      }
    }
    for(int i=0; i<r-l+1; i++){
      if(prime[i]){
        seg_primes.pb(l+i);
      }
    }
  }

}
```

6.6. Matrix Exponentiation

```
/* Matrix Exponentiation Implementation */

struct Matrix{
  vector< vector<int> > m;
  Matrix() {}
  Matrix(int l, int c){
    m.resize(l, vector<int>(c));
  }

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


  Matrix exp(int k){
    if(k == 1) return *this;
    Matrix c = (*this).exp(k/2);
    c = c*c;
    if(k%2) c = c*(*this);
    return c;
  }
};
```

6.7. 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 (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 */

namespace 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;
  }

};
```

7. String Algorithms

7.1. KMP Failure Function + String Matching

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

struct KMP{
  vector<int> pi;

  vector<int> matches;

  KMP() {}

  void calculate(string t) {
    int n = t.size();
    pi.resize(n);
    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;

  ZFunction() {}

  void calculate(string t){
    int n = t.size();
    z.resize(n);
    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. 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, pot, sp[22];

  int block, n;

  string s;

  SuffixArray() {}

  SuffixArray(string t){
    s = t;
    n = t.size();
    rnk.resize(n+1);
    for(int i=0; i<22; i++) sp[i].resize(n+1);
    pot.resize(n+1);
    tmp.resize(max(257LL, n+1));
    sa.resize(n+1);
    sa_aux.resize(n+1);
    lcp.resize(n+1);
    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();
    sa.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(0LL,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];
    }
  }

  void build_lcp() {
    int k = 0;
    for(int j = 0; (1<<j) <= 2*n; j++) {
      for(; k <= n && k < (1<<j); k++) {
          pot[k] = j-1;
      }
    }
    for(int i=0; i<n; i++){
      sp[0][i] = lcp[i];
    }
    for(int i = 1; (1<<i) <= n; i++) {
      for(int j = 0; j+(1<<i) <= n; j++) {
        sp[i][j] = min(sp[i-1][j], sp[i-1][j+(1<<(i-1))]);
      }
    }
  }

  int query_lcp(int x, int y){
    if(x == y) return n - x;
    if(rnk[x] > rnk[y]) swap(x,y);
    int l = rnk[x], r = rnk[y]-1;
    return min(sp[pot[r-l+1]][l], sp[pot[r-l+1]][r-(1LL<<pot[r-l+1])+1]);
  }

};
```

7.4. Rolling Hash

```
namespace Hash{

  int B1, B2, M1, M2;

  void init(){
    B1 = rand()%65536;
    B2 = rand()%65536;
    M1 = 1000000007;
    M2 = 1000000009;
  }

  struct RollingHash{

    vector< ii > hash;
    vector< ii > base;

    RollingHash() {}

    void calculate(string s){
      int n = s.size();
      hash.resize(n+1); base.resize(n+1);
      base[0] = ii(1, 1);
      hash[0] = ii(0, 0);
      for(int i=1; i<=n; i++){
        int val = (int)(s[i-1]);
        base[i] = ii(mod(base[i-1].ff*B1, M1), mod(base[i-1].ss*B2, M2));
        hash[i] = ii(mod(hash[i-1].ff*B1 + val, M1), mod(hash[i-1].ss*B2 + val, M2));
      }
    }

    ii query(int l, int r){
      ii ret;
      ret.ff = mod(hash[r].ff - hash[l-1].ff*base[r-l+1].ff, M1);
      ret.ss = mod(hash[r].ss - hash[l-1].ss*base[r-l+1].ss, M2);
      return ret;
    }

  };

}
```