[Data structures 5](#_Toc89359269)

[Fenwick tree 5](#_Toc89359270)

[Fenwick tree 2d 6](#_Toc89359271)

[Fenwick tree update range 7](#_Toc89359272)

[Segment tree 8](#_Toc89359273)

[Max sum range node 9](#_Toc89359274)

[Extended Segment tree 10](#_Toc89359275)

[Persistent segment tree 11](#_Toc89359276)

[Ordered set 12](#_Toc89359277)

[Quad Tree 13](#_Toc89359278)

[Sparse table 14](#_Toc89359279)

[SQRT Decomposition 15](#_Toc89359280)

[Treap 16](#_Toc89359281)

[Implicit Treap 16](#_Toc89359282)

[Ordered multiset 18](#_Toc89359283)

[Big Integer 21](#_Toc89359284)

[Mod int 24](#_Toc89359285)

[Graph 25](#_Toc89359286)

[Graph Data structures 25](#_Toc89359287)

[DSU 25](#_Toc89359288)

[LCA 28](#_Toc89359289)

[Heavy light decomposition 29](#_Toc89359290)

[Centroid decomposition 31](#_Toc89359291)

[Shortest path algorithms 32](#_Toc89359292)

[Dijkstra 32](#_Toc89359293)

[Bellmanford 33](#_Toc89359294)

[Floyed 34](#_Toc89359295)

[SPFA 35](#_Toc89359296)

[MST 35](#_Toc89359297)

[Kruskal 35](#_Toc89359298)

[Prim 37](#_Toc89359299)

[SMST O(n \* log(n)) 37](#_Toc89359300)

[Tarjan 40](#_Toc89359301)

[SCC 40](#_Toc89359302)

[Articulation points and bridges 41](#_Toc89359303)

[Edge classification 42](#_Toc89359304)

[2-SAT 42](#_Toc89359305)

[Flows 43](#_Toc89359306)

[Maximum bipartite matching 43](#_Toc89359307)

[Hopcroft Karp for bipartite matching 43](#_Toc89359308)

[Edmonds Karp 44](#_Toc89359309)

[Dinic 45](#_Toc89359310)

[Min cost Max flow 46](#_Toc89359311)

[Hungarian 48](#_Toc89359312)

[String 50](#_Toc89359313)

[Hashing 50](#_Toc89359314)

[KMP 50](#_Toc89359315)

[Trie 52](#_Toc89359316)

[Aho Corasick 53](#_Toc89359317)

[Suffix Automaton 55](#_Toc89359318)

[Suffix array 57](#_Toc89359319)

[Suffix array Faster 60](#_Toc89359320)

[Z algorithm 61](#_Toc89359321)

[Math 62](#_Toc89359322)

[Primes 62](#_Toc89359323)

[Sieve 62](#_Toc89359324)

[Linear Sieve 62](#_Toc89359325)

[Miller Rabin Primality Test 62](#_Toc89359326)

[Prime Factors 64](#_Toc89359327)

[Phi function 64](#_Toc89359328)

[Moebius function 64](#_Toc89359329)

[Extended Euclidean 65](#_Toc89359330)

[Linear Diophantine Equation 65](#_Toc89359331)

[Extended Euclidean for n variables 67](#_Toc89359332)

[Sum Sequence 68](#_Toc89359333)

[Sum Range Divisors 68](#_Toc89359334)

[Combinatorics 69](#_Toc89359335)

[nCr pre calculation 69](#_Toc89359336)

[Matrices 70](#_Toc89359337)

[Matrix class 71](#_Toc89359338)

[Mod 73](#_Toc89359339)

[Fast power 73](#_Toc89359340)

[Sum of powers 73](#_Toc89359341)

[Mod Inverse 73](#_Toc89359342)

[(a^n)%p=result , return n 74](#_Toc89359343)

[CRT 75](#_Toc89359344)

[FFT 75](#_Toc89359345)

[NTT 76](#_Toc89359346)

[Misc 78](#_Toc89359347)

[Bitmask 78](#_Toc89359348)

[Coordinate Compress 79](#_Toc89359349)

[Random numbers 79](#_Toc89359350)

[Custom hash 79](#_Toc89359351)

[Max histogram area 80](#_Toc89359352)

[Sorting 81](#_Toc89359353)

[LIS binary Search 82](#_Toc89359354)

[Mo algorithm 83](#_Toc89359355)

[floyd cycle detection algorithm 83](#_Toc89359356)

[Convex Hull Trick (Line Container) 84](#_Toc89359357)

[Java Scanner 84](#_Toc89359358)

[Geometry 87](#_Toc89359359)

[point 87](#_Toc89359360)

[rotate 87](#_Toc89359361)

[reflect 87](#_Toc89359362)

[Triangles 88](#_Toc89359363)

[Get Angles/Sides 88](#_Toc89359364)

[Point In Triangle 88](#_Toc89359365)

[Get largest Circle Inside the triangle 89](#_Toc89359366)

[Lines 89](#_Toc89359367)

[IsCollinear 89](#_Toc89359368)

[isPointOnRay 89](#_Toc89359369)

[isPointOnSegment 89](#_Toc89359370)

[distToLine 89](#_Toc89359371)

[distToSegment 90](#_Toc89359372)

[pointToSegment 90](#_Toc89359373)

[intersectSegments 90](#_Toc89359374)

[CCW 90](#_Toc89359375)

[Circles 91](#_Toc89359376)

[Find circle passes with 3 points 91](#_Toc89359377)

[intersectLineCircle 91](#_Toc89359378)

[Circle Circle Intersection 91](#_Toc89359379)

[Circle Circle Intersection Area 92](#_Toc89359380)

[Check if polygon is convex 92](#_Toc89359381)

[Convex hull 92](#_Toc89359382)

[Point in polygon O(log(n)) 93](#_Toc89359383)

[line sweep for lines intersections 94](#_Toc89359384)

[Pyramid Volume 96](#_Toc89359385)

# Data structures

## Fenwick tree

template<typename T> struct fenwick\_tree {

/\* can convert it to map, build what you need only

\* will be: memory O(q\*logn) ,time O(logn\*logn) \*/

vector<T> BIT;

int n;

fenwick\_tree(int n) : n(n), BIT(n + 1) {}

T getAccum(int idx) {

T sum = 0;

while (idx) {

sum += BIT[idx];

idx -= (idx & -idx);

}

return sum;

}

void add(int idx, T val) {

assert(idx != 0);

while (idx <= n) {

BIT[idx] += val;

idx += (idx & -idx);

}

}

T getValue(int idx) {

return getAccum(idx) - getAccum(idx - 1);

}

// ordered statistics tree // get index that has value >= accum

// values must by positive

int getIdx(T accum) {

int start = 1, end = n, rt = -1;

while (start <= end) {

int mid = start + end >> 1;

T val = getAccum(mid);

if (val >= accum)

rt = mid, end = mid - 1;

else start = mid + 1;

}

return rt;

}

//not tested (from topcoder)

//first index less than or equal accum O(logn) (same as getIdx)

int find(T accum) {

int i = 1, idx = 0;

while ((1 << i) <= n) i <<= 1;

for (; i > 0; i >>= 1) {

int tidx = idx + i;

if (tidx > n) continue;

if (accum >= BIT[tidx]) {

idx = tidx; accum -= BIT[tidx];

}

}

return idx;

}

};

### Fenwick tree 2d

template<typename T>

struct fenwick\_tree\_2d {

#define Lbit(x) (x&-x)

int n, m;

vector<vector<T>> BIT;

fenwick\_tree\_2d(int n, int m) :

n(n), m(m), BIT(n + 1, vector<T>(m + 1)) {

}

T getAccum(int i, int j) {

T sum = 0;

for (; i; i -= Lbit(i))

for (int idx = j; idx > 0; idx -= Lbit(idx))

sum += BIT[i][idx];

return sum;

}

void add(int i, int j, int val) {

assert(i != 0 && j != 0);

for (; i <= n; i += Lbit(i))

for (int idx = j; idx <= m; idx += Lbit(idx))

BIT[i][idx] += val;

}

T getRectangeSum(int x1, int y1, int x2, int y2) {

if (y1 > y2)

swap(y1, y2);

if (x1 > x2)

swap(x1, x2);

return getAccum(x2, y2) - getAccum(x1 - 1, y2) - getAccum(x2, y1 - 1)

+ getAccum(x1 - 1, y1 - 1);

}

};

### Fenwick tree update range

/\*

x[i] = a[i] - a[i-1] //a is original array

y[i] = x[i]\*(i-1)

sum(1,3) = a[1] + a[2] + a[3] = (x[1]) + (x[2] + x[1]) + (x[3] + x[2] + x[1])

= 3\*(x[1] + x[2] + x[3]) - 0\*x[1] - 1\*x[2] - 2\*x[3] //same equation but more complex

= sumX(1,3) \* 3 - sumY(1,3)

so sum(1,n) = sumX(1,n)\*n - sumY(1,n)

update:

x[l] += val,x[r+1] -= val

y[l] += val \*(l-1),y[r+1] -= r\*val

\*/

template<typename T>

class fenwick\_tree {

int n;

vector<T> x, y;

T getAccum(vector<T>& BIT, int idx) {

T sum = 0;

while (idx) {

sum += BIT[idx];

idx -= (idx & -idx);

}

return sum;

}

void add(vector<T>& BIT, int idx, T val) {

assert(idx != 0);

while (idx <= n) {

BIT[idx] += val;

idx += (idx & -idx);

}

}

T prefix\_sum(int idx) {

return getAccum(x, idx) \* idx - getAccum(y, idx);

}

public:

fenwick\_tree(int n) :

n(n), x(n + 1), y(n + 1) {

}

void update\_range(int l, int r, T val) {

add(x, l, val);

add(x, r + 1, -val);

add(y, l, val \* (l - 1));

add(y, r + 1, -val \* r);

}

T range\_sum(int l, int r) {

return prefix\_sum(r) - prefix\_sum(l - 1);

}

};

## Segment tree

/\*

for efficient memory (2\*n)

#define LEFT (idx+1)

#define MID ((start+end)>>1)

#define RIGHT (idx+((MID-start+1)<<1))

\*/

template<typename T>

class segment\_tree {//1-based

#define LEFT (idx<<1)

#define RIGHT (idx<<1|1)

#define MID ((start+end)>>1)

int n;

vector<T> tree, lazy;

T merge(const T& left, const T& right) {}

inline void pushdown(int idx, int start, int end) {

if (lazy[idx] == 0) return;

//update tree[idx] with lazy[idx]

tree[idx] += lazy[idx];

if (start != end) {

lazy[LEFT] += lazy[idx];

lazy[RIGHT] += lazy[idx];

}

lazy[idx] = 0; //clear lazy

}

inline void pushup(int idx) {

tree[idx] = merge(tree[LEFT], tree[RIGHT]);

}

void build(int idx, int start, int end) {

if (start == end) return;

build(LEFT, start, MID);

build(RIGHT, MID + 1, end);

pushup(idx);

}

void build(int idx, int start, int end, const vector<T>& arr) {

if (start == end) {

tree[idx] = arr[start];

return;

}

build(LEFT, start, MID, arr);

build(RIGHT, MID + 1, end, arr);

pushup(idx);

}

T query(int idx, int start, int end, int from, int to) {

pushdown(idx, start, end);

if (from <= start && end <= to)

return tree[idx];

if (to <= MID)

return query(LEFT, start, MID, from, to);

if (MID < from)

return query(RIGHT, MID + 1, end, from, to);

return merge(query(LEFT, start, MID, from, to),

query(RIGHT, MID + 1, end, from, to));

}

void update(int idx, int start, int end, int lq, int rq, const T& val) {

pushdown(idx, start, end);

if (rq < start || end < lq)

return;

if (lq <= start && end <= rq) {

lazy[idx] += val;//update lazy

pushdown(idx, start, end);

return;

}

update(LEFT, start, MID, lq, rq, val);

update(RIGHT, MID + 1, end, lq, rq, val);

pushup(idx);

}

public:

segment\_tree(int n) :n(n), tree(n << 2), lazy(n << 2) {}

segment\_tree(const vector<T>& v) {

n = v.size() - 1;

tree = vector<T>(n << 2);

lazy = vector<T>(n << 2);

build(1, 1, n, v);

}

T query(int l, int r) {

return query(1, 1, n, l, r);

}

void update(int l, int r, const T& val) {

update(1, 1, n, l, r, val);

}

#undef LEFT

#undef RIGHT

#undef MID

};

### Max sum range node

struct MSR\_Node {

ll left, right, mid, sum;

MSR\_Node(const ll& val) {//be careful from the empty subarray

left = right = mid = sum = val;

}

MSR\_Node(const MSR\_Node& a, const MSR\_Node& b) {

left = max(a.left, a.sum + b.left);

right = max(b.right, b.sum + a.right);

mid = max({ a.mid, b.mid, a.right + b.left });

sum = a.sum + b.sum;

}

ll getMax() {

return max({ left, right, mid, sum });

}

};

### Extended Segment tree

struct segtree {

segtree \*left = nullptr, \*right = nullptr;

int mx = 0;

segtree(int val = 0) :

mx(val) {

}

void extend() {

if (left == nullptr) {

left = new segtree();

right = new segtree();

}

}

void pushup() {

mx = max(left->mx, right->mx);

}

~segment\_tree() {

if (left == nullptr)return;

delete left;

delete right;

}

};

class extened\_segment\_tree {

#define MID ((start+end)>>1)

void update(segtree \*root, int start, int end, int pos, int val) {

if (pos < start || end < pos)

return;

if (start == end) {

root->mx = max(root->mx, val);

return;

}

root->extend();

update(root->left, start, MID, pos, val);

update(root->right, MID + 1, end, pos, val);

root->pushup();

}

int query(segtree \*root, int start, int end, int l, int r) {

if (root == nullptr || r < start || end < l)

return 0;

if (l <= start && end <= r)

return root->mx;

return max(query(root->left, start, MID, l, r),

query(root->right, MID + 1, end, l, r));

}

public:

int start, end;

segtree \*root;

extened\_segment\_tree() {

}

~extened\_segment\_tree() {

delete root;

}

extened\_segment\_tree(int start, int end) : start(start), end(end) {

root = new segtree();

}

void update(int pos, int val) {

update(root, start, end, pos, val);

}

int query(int l, int r) {

return query(root, start, end, l, r);

}

#undef MID

};

### Persistent segment tree

struct segtree {

    static segtree \*sentinel;

    segtree \*left, \*right;

    bool dirty = false;

    ll sum = 0, lazy = 0;

    segtree(ll val = 0) : sum(val) {

        left = right = this;

    }

    segtree(segtree \*left, segtree \*right) : left(left), right(right) {

        sum = left->sum + right->sum;

    }

};

segtree \*segtree::sentinel = new segtree();

class persistent\_segment\_tree {

#define MID ((start+end)>>1)

    segtree\* apply(segtree \*root, int start, int end, ll val) {

        segtree \*rt = new segtree(\*root);

        rt->dirty = true;

        rt->sum += (end - start + 1) \* val;

        rt->lazy += val;

        return rt;

    }

    void pushdown(segtree \*root, int start, int end) {

        if (root->dirty == false || start == end)

            return;

        root->left = apply(root->left, start, MID, root->lazy);

        root->right = apply(root->right, MID + 1, end, root->lazy);

        root->lazy = 0;

        root->dirty = 0;

    }

    segtree\* build(int start, int end, const vector<int> &v) {

        if (start == end)

            return new segtree(v[start]);

        return new segtree(build(start, MID, v), build(MID + 1, end, v));

    }

    segtree\* Set(segtree \*root, int start, int end, int pos, ll new\_val) {

        pushdown(root, start, end);

        if (pos < start || end < pos)

            return root;

        if (pos <= start && end <= pos)

            return new segtree(new\_val);

        return new segtree(Set(root->left, start, MID, pos, new\_val),

                Set(root->right, MID + 1, end, pos, new\_val));

    }

    segtree\* update(segtree \*root, int start, int end, int l, int r, ll val) {

        pushdown(root, start, end);

        if (r < start || end < l)

            return root;

        if (l <= start && end <= r)

            return apply(root, start, end, val);

        return new segtree(update(root->left, start, MID, l, r, val),

                update(root->right, MID + 1, end, l, r, val));

    }

    ll query(segtree \*root, int start, int end, int l, int r) {

        pushdown(root, start, end);

        if (r < start || end < l)

            return 0;

        if (l <= start && end <= r)

            return root->sum;

        return query(root->left, start, MID, l, r)

                + query(root->right, MID + 1, end, l, r);

    }

public:

    int start, end;

    vector<segtree\*> versions;

    persistent\_segment\_tree(int start, int end) :

            start(start), end(end) {

        versions.push\_back(segtree::sentinel);

    }

    persistent\_segment\_tree(const vector<int> &v) :

            start(0), end(v.size() - 1) {

        versions.push\_back(build(start, end, v));

    }

    void update(int l, int r, ll val) {

        versions.push\_back(update(versions.back(), start, end, l, r, val));

    }

    ll query(int time, int l, int r) {

        return query(versions[time], start, end, l, r);

    }

#undef MID

};

## Ordered set

#include<bits/stdc++.h>

#include<ext/pb\_ds/assoc\_container.hpp>

#include<ext/pb\_ds/tree\_policy.hpp>

using namespace std;

using namespace \_\_gnu\_pbds;

template<typename key>

using ordered\_set = tree<key, null\_type, less<key>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

/\*

find\_by\_order(k) :

It returns to an iterator to the k-th element (counting from zero) in the set in O(logn) time.

To find the first element k must be zero.

order\_of\_key(k) :

It returns to the number of items that are strictly smaller than our item k in O(logn) time. \*/

## Quad Tree

//set NIL with defult value

template<typename T>

struct node {

    node<T> \*child[4];

    T val;

    node(T val = NIL) : val(val) {

        memset(child, 0, sizeof child);

    }

    void push\_up() {//merge the 4 childs

        val = NIL;

        for (int i = 0; i < 4; i++)

            if (child[i] != nullptr) {}

    }

    ~node() {

        for (int i = 0; i < 4; i++)

            if (child[i] != nullptr)

                delete child[i];

    }

};

template<typename T>

void update(node<T> \*&root, int r1, int r2, int c1, int c2, int x, int y, T val) {

    if (x < r1 || x > r2 || y < c1 || y > c2)

        return;

    if (root == nullptr)

        root = new node<T>();

    if (r1 == r2 && c1 == c2) {

        //update

        return;

    }

    int rmid = (r1 + r2) / 2, cmid = (c1 + c2) / 2;

    update(root->child[0], r1, rmid, c1, cmid, x, y, val);

    update(root->child[1], r1, rmid, cmid + 1, c2, x, y, val);

    update(root->child[2], rmid + 1, r2, c1, cmid, x, y, val);

    update(root->child[3], rmid + 1, r2, cmid + 1, c2, x, y, val);

    root->push\_up();

}

template<typename T>

T query(node<T> \*root, int r1, int r2, int c1, int c2, int x, int y) {

    if (root == nullptr || x < r1 || x > r2 || y < c1 || y > c2)

        return NIL;

    if (r1 == r2 && c1 == c2)

        return root->val;

    int rmid = (r1 + r2) / 2, cmid = (c1 + c2) / 2;

    query(root->child[0], r1, rmid, c1, cmid, x, y);

    query(root->child[1], r1, rmid, cmid + 1, c2, x, y);

    query(root->child[2], rmid + 1, r2, c1, cmid, x, y);

    query(root->child[3], rmid + 1, r2, cmid + 1, c2, x, y);

}

node<T> \*seg = new node<T>();

## Sparse table

template<typename T>

struct sparse\_table {

vector<vector<T>> sparseTable;

using F = function<T(T, T)>;

F merge;

static int LOG2(int x) { //floor(log2(x))

return 31 - \_\_builtin\_clz(x);

}

sparse\_table(vector<T>& v, F merge) :

merge(merge) {

int n = v.size();

int logN = LOG2(n);

sparseTable = vector<vector<T>>(logN + 1);

sparseTable[0] = v;

for (int k = 1, len = 1; k <= logN; k++, len <<= 1) {

sparseTable[k].resize(n);

for (int i = 0; i + len < n; i++)

sparseTable[k][i] = merge(sparseTable[k - 1][i],

sparseTable[k - 1][i + len]);

}

}

T query(int l, int r) {

int k = LOG2(r - l + 1); // max k ==> 2^k <= length of range

//check first 2^k from left and last 2^k from right //overlap

return merge(sparseTable[k][l], sparseTable[k][r - (1 << k) + 1]);

}

T query\_shifting(int l, int r) {

T res;

bool first = true;

for (int i = (int)sparseTable.size() - 1; i >= 0; i--)

if (l + (1 << i) - 1 <= r) {

if (first)

res = sparseTable[i][l];

else

res = merge(res, sparseTable[i][l]);

first = false;

l += (1 << i);

}

return res;

}

};

## SQRT Decomposition

//zero based SQRT\_Decomposition with lazy propagation

template<typename update\_type, typename query\_type>

class SQRT\_Decomposition {

struct Bucket {

int l, r;

update\_type lazy;

Bucket(int l, int r) : l(l), r(r) {

//set default value to lazy

//build bucket for the first time

}

void build() {

//update all bucket with lazy if have

//rebuild the bucket

//clear lazy

}

//update all bucket

//just update lazy

void update(const update\_type& val) {}

//update range in bucket

void update(int start, int end, const update\_type& val) {

if (start == l && end == r) {

update(val);

return;

}

//update bucket

//rebuild the bucket if need

}

//query about all bucket

//calc with lazy

query\_type query() {}

//query about range in bucket

query\_type query(int start, int end) {

if (start == l && end == r)

return query();

//push lazy if have

//calc

}

};

int n, sqrtN;

vector<Bucket> bucket;

int begin(int idx) {

return idx \* sqrtN;

}

int end(int idx) {

return min(sqrtN \* (idx + 1), n) - 1;

}

int which\_block(int idx) {

return idx / sqrtN;

}

public:

SQRT\_Decomposition(int n) {

this->n = n;

sqrtN = sqrt(n);

for (int i = 0; i < n; i += sqrtN)

bucket.push\_back(Bucket(i, min(i + sqrtN, n) - 1));

}

void update(int left, int right, update\_type val) {

int st = which\_block(left), ed = which\_block(right);

bucket[st].update(left, min(bucket[st].r, right), val);

for (int i = st + 1; i < ed; i++)

bucket[i].update(val);

if (st != ed) bucket[ed].update(bucket[ed].l, right, val);

}

query\_type query(int left, int right) {

int st = which\_block(left), ed = which\_block(right);

query\_type rt = bucket[st].query(left, min(bucket[st].r, right));

for (int i = st + 1; i < ed; i++)

rt += bucket[i].query();

if (st != ed) rt += bucket[ed].query(bucket[ed].l, right);

return rt;

}

};

## **Treap**

### **Implicit** Treap

enum DIR { L, R };

template<typename T> struct cartesian\_tree {

static cartesian\_tree<T>\* sentinel;

T key = T();

int priority = 0, size = 0, reverse = false;

cartesian\_tree\* child[2];

cartesian\_tree() {

child[L] = child[R] = this;

}

cartesian\_tree(const T& x, int y) : key(x), priority(y) {

size = 1;

child[L] = child[R] = sentinel;

}

void push\_down() {

if (!reverse) return;

reverse = 0;

child[L]->doRevese();

child[R]->doRevese();

}

void doReverse() {

reverse ^= 1;

swap(child[L], child[R]);

}

void push\_up() {

size = child[L]->size + child[R]->size + 1;

}

};

template<typename T>

cartesian\_tree<T>\* cartesian\_tree<T>::sentinel = new cartesian\_tree<T>();

template<typename T, template<typename > class cartesian\_tree>

class implicit\_treap { //1 based

typedef cartesian\_tree<T> node;

typedef cartesian\_tree<T>\* nodeptr;

#define emptyNode cartesian\_tree<T>::sentinel

nodeptr root;

void split(nodeptr root, nodeptr& l, nodeptr& r, int firstXElment) {

if (root == emptyNode) {

l = r = emptyNode;

return;

}

root->push\_down();

if (firstXElment <= root->child[L]->size) {

split(root->child[L], l, root->child[L], firstXElment);

r = root;

}

else {

split(root->child[R], root->child[R], r,

firstXElment - root->child[L]->size - 1);

l = root;

}

root->push\_up();

}

nodeptr merge(nodeptr l, nodeptr r) {

l->push\_down();

r->push\_down();

if (l == emptyNode || r == emptyNode)

return (l == emptyNode ? r : l);

if (l->priority > r->priority) {

l->child[R] = merge(l->child[R], r);

l->push\_up();

return l;

}

r->child[L] = merge(l, r->child[L]);

r->push\_up();

return r;

}

vector<nodeptr> split\_range(int s, int e) { // [x<s,s<=x<=e,e<x]

nodeptr l, m, r, tmp;

split(root, l, tmp, s - 1);

split(tmp, m, r, e - s + 1);

return { l,m,r };

}

public:

implicit\_treap() : root(emptyNode) {}

int size() {

return root->size;

}

void insert(int pos, const T& key) {

nodeptr tmp = new node(key, rand());

nodeptr l, r;

split(root, l, r, pos - 1);

root = merge(merge(l, tmp), r);

}

void push\_back(const T& value) {

root = merge(root, new node(value, rand()));

}

T getByIndex(int pos) {

vector<nodeptr> tmp = split\_range(pos, pos);

nodeptr l = tmp[0], m = tmp[1], r = tmp[2];

T rt = m->key;

root = merge(merge(l, m), r);

return rt;

}

void erase(int pos) {

vector<nodeptr> tmp = split\_range(pos, pos);

nodeptr l = tmp[0], m = tmp[1], r = tmp[2];

delete m;

root = merge(l, r);

}

void cyclic\_shift(int s, int e) { //to the right

vector<nodeptr> tmp = split\_range(s, e);

nodeptr l = tmp[0], m = tmp[1], r = tmp[2];

nodeptr first, second;

split(m, first, second, e - s);

root = merge(merge(merge(l, second), first), r);

}

void reverse\_range(int s, int e) {

vector<nodeptr> tmp = split\_range(s, e);

nodeptr l = tmp[0], m = tmp[1], r = tmp[2];

m->reverse ^= 1;

root = merge(merge(l, m), r);

}

node range\_query(int s, int e) {

vector<nodeptr> tmp = split\_range(s, e);

nodeptr l = tmp[0], m = tmp[1], r = tmp[2];

node rt = \*m;

root = merge(merge(l, m), r);

return rt;

}

};

### Ordered multiset

enum DIR { L, R };

template<typename T>

struct cartesian\_tree {

static cartesian\_tree<T>\* sentinel;

T key = T();

int priority = 0, frequency = 0, size = 0;

cartesian\_tree\* child[2];

cartesian\_tree() {

child[L] = child[R] = this;

}

cartesian\_tree(const T& x, int y) : key(x), priority(y) {

size = frequency = 1;

child[L] = child[R] = sentinel;

}

void push\_down() {

}

void push\_up() {

size = child[L]->size + child[R]->size + frequency;

}

};

template<typename T> //be

cartesian\_tree<T>\* cartesian\_tree<T>::sentinel = new cartesian\_tree<T>();

template<typename T>

void split(cartesian\_tree<T>\* root, T key, cartesian\_tree<T>\*& l,

cartesian\_tree<T>\*& r) {

if (root == cartesian\_tree<T>::sentinel) {

l = r = cartesian\_tree<T>::sentinel;

return;

}

root->push\_down();

if (root->key <= key) {

split(root->child[R], key, root->child[R], r);

l = root;

}

else {

split(root->child[L], key, l, root->child[L]);

r = root;

}

root->push\_up();

}

template<typename T>

cartesian\_tree<T>\* merge(cartesian\_tree<T>\* l, cartesian\_tree<T>\* r) {

l->push\_down();

r->push\_down();

if (l == cartesian\_tree<T>::sentinel || r == cartesian\_tree<T>::sentinel)

return (l == cartesian\_tree<T>::sentinel ? r : l);

if (l->priority > r->priority) {

l->child[R] = merge(l->child[R], r);

l->push\_up();

return l;

}

r->child[L] = merge(l, r->child[L]);

r->push\_up();

return r;

}

template<typename T, template<typename > class cartesian\_tree>

class treap {

typedef cartesian\_tree<T> node;

typedef node\* nodeptr;

#define emptyNode node::sentinel

nodeptr root;

void insert(nodeptr& root, nodeptr it) {

if (root == emptyNode) {

root = it;

}

else if (it->priority > root->priority) {

split(root, it->key, it->child[L], it->child[R]);

root = it;

}

else

insert(root->child[root->key < it->key], it);

root->push\_up();

}

bool increment(nodeptr root, const T& key) {

if (root == emptyNode)

return 0;

if (root->key == key) {

root->frequency++;

root->push\_up();

return root;

}

bool rt = increment(root->child[root->key < key], key);

root->push\_up();

return rt;

}

nodeptr find(nodeptr root, const T& key) {

if (root == emptyNode || root->key == key)

return root;

return find(root->child[root->key < key], key);

}

void erase(nodeptr& root, const T& key) {

if (root == emptyNode)

return;

if (root->key == key) {

if (--(root->frequency) == 0)

root = merge(root->child[L], root->child[R]);

}

else

erase(root->child[root->key < key], key);

root->push\_up();

}

T kth(nodeptr root, int k) {

if (root->child[L]->size >= k)

return kth(root->child[L], k);

k -= root->child[L]->size;

if (k <= root->frequency)

return root->key;

return kth(root->child[R], k - root->frequency);

}

int order\_of\_key(nodeptr root, const T& key) {

if (root == emptyNode)

return 0;

if (key < root->key)

return order\_of\_key(root->child[L], key);

if (key == root->key)

return root->child[L]->size;

return root->child[L]->size + root->frequency

+ order\_of\_key(root->child[R], key);

}

public:

treap() : root(emptyNode) {}

void insert(const T& x) {

if (increment(root, x)) //change it to find(x) to make it as a set

return;

insert(root, new node(x, rand()));

}

void erase(const T& x) {

erase(root, x);

}

bool find(const T& x) {

return (find(root, x) != emptyNode);

}

int get\_kth\_number(int k) {

assert(1 <= k && k <= size());

return kth(root, k);

}

int order\_of\_key(const T& x) {

return order\_of\_key(root, x);

}

int size() {

return root->size;

}

};

## Big Integer

class BigInt {

private:

#define CUR (\*this)

    const int BASE = 1000000000;//1e9

    vector<int> v;

public:

    BigInt(const long long &val = 0) {

        CUR = val;

    }

    BigInt(const string &val) {

        CUR = val;

    }

    int size() const {

        return v.size();

    }

    bool zero() const {

        return v.empty();

    }

    BigInt& operator =(long long val) {

        v.clear();

        while (val) {

            v.push\_back(val % BASE);

            val /= BASE;

        }

        return CUR;

    }

    BigInt& operator =(const BigInt &a) {

        v = a.v;

        return CUR;

    }

    BigInt& operator=(const string &s) {

        CUR = 0;

        for (const char &ch : s)

            CUR = CUR \* 10 + (ch - '0');

        return CUR;

    }

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*compare\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    bool operator <(const BigInt &a) const {

        if (a.size() != size())

            return size() < a.size();

        for (int i = size() - 1; i >= 0; i--) {

            if (v[i] != a.v[i])

                return v[i] < a.v[i];

        }

        return false;

    }

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*add\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    BigInt operator +(const BigInt &a) const {

        BigInt res = CUR;

        int idx = 0, carry = 0;

        while (idx < a.size() || carry) {

            if (idx < a.size())

                carry += a.v[idx];

            if (idx == res.size())

                res.v.push\_back(0);

            res.v[idx] += carry;

            carry = res.v[idx] / BASE;

            res.v[idx] %= BASE;

            idx++;

        }

        return res;

    }

    BigInt& operator +=(const BigInt &a) {

        CUR = CUR + a;

        return CUR;

    }

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*multiply\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    BigInt operator \*(const BigInt &a) const {

        BigInt res;

        if (CUR.zero() || a.zero())

            return res;

        res.v.resize(size() + a.size());

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

            if (v[i] == 0)

                continue;

            long long carry = 0;

            for (int j = 0; carry || j < a.size(); j++) {

                carry += 1LL \* v[i] \* (j < a.size() ? a.v[j] : 0);

                while (i + j >= res.size())

                    res.v.push\_back(0);

                carry += res.v[i + j];

                res.v[i + j] = carry % BASE;

                carry /= BASE;

            }

        }

        while (!res.v.empty() && res.v.back() == 0)

            res.v.pop\_back();

        return res;

    }

    BigInt& operator \*=(const BigInt &a) {

        CUR = CUR \* a;

        return CUR;

    }

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Division\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    BigInt& operator /=(const int &a) {

        ll carry = 0;

        for (int i = (int) v.size() - 1; i >= 0; i--) {

            ll cur = v[i] + carry \* BASE;

            v[i] = cur / a;

            carry = cur % a;

        }

        while (!v.empty() && v.back() == 0)

            v.pop\_back();

        return CUR;

    }

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*print\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    friend ostream& operator<<(ostream &out, const BigInt &a) {

        out << (a.zero() ? 0 : a.v.back());

        for (int i = (int) a.v.size() - 2; i >= 0; i--)

            out << setfill('0') << setw(9) << a.v[i];

        return out;

    }

#undef CUR

};

## Mod int

struct modint {

#define CUR (\*this)

static const int MOD = 1e9 + 7;

int val;

modint(const long long& a = 0) {

val = a % MOD;

if (val < 0) val += MOD;

}

modint& operator+=(const modint& a) {

if ((val += a.val) >= MOD) val -= MOD;

return CUR;

}

modint operator+(const modint& a) const {

modint c = CUR;

c += a;

return c;

}

modint& operator-=(const modint& a) {

if ((val -= a.val) < 0) val += MOD;

return CUR;

}

modint operator-(const modint& a) const {

modint c = CUR;

c -= a;

return c;

}

modint operator\*(const modint& a) const {

return modint((1LL \* this->val \* a.val) % MOD);

}

modint& operator\*=(const modint& a) {

CUR = CUR \* a;

return CUR;

}

modint operator/(const modint& a) {

return CUR \* power(a, MOD - 2);

}

modint& operator/=(const modint& a) {

CUR = CUR / a;

return CUR;

}

static modint power(modint x, long long y) {

modint res = 1;

while (y > 0) {

if (y & 1) res \*= x;

x \*= x; y >>= 1;

}

return res;

}

friend ostream& operator<<(ostream& out, const modint& a) {

out << a.val;

return out;

}

#undef CUR

};

# Graph

## Graph Data structures

### DSU

struct DSU {

vector<int> rank, parent, size;

vector<vector<int>> component;

int forsets;

DSU(int n) {

size = rank = parent = vector<int>(n + 1, 1);

component = vector<vector<int>>(n + 1);

forsets = n;

for (int i = 0; i <= n; i++) {

parent[i] = i;

component[i].push\_back(i);

}

}

int find\_set(int v) {

if (v == parent[v])

return v;

return parent[v] = find\_set(parent[v]);

}

void link(int par, int node) {

parent[node] = par;

size[par] += size[node];

for (const int& it : component[node])

component[par].push\_back(it);

component[node].clear();

if (rank[par] == rank[node])

rank[par]++;

forsets--;

}

bool union\_sets(int v, int u) {

v = find\_set(v), u = find\_set(u);

if (v != u) {

if (rank[v] < rank[u])

swap(v, u);

link(v, u);

}

return v != u;

}

bool same\_set(int v, int u) {

return find\_set(v) == find\_set(u);

}

int size\_set(int v) {

return size[find\_set(v)];

}

};

#### DSU bipartiteness

struct DSU\_bipartiteness {

vector<int> bipartite, rank;

vector<pair<int, int>> parent;

DSU\_bipartiteness(int n) {

bipartite = rank = vector<int>(n + 1, 1);

parent = vector<pair<int, int>>(n + 1);

for (int i = 0; i <= n; i++)

parent[i] = { i, 0 };

}

pair<int, int> find\_set(int x) {

if (x == parent[x].first)

return parent[x];

int parity = parent[x].second;

parent[x] = find\_set(parent[x].first);

parent[x].second ^= parity;

return parent[x];

}

void union\_sets(int x, int y) {

pair<int, int> p = find\_set(x);

x = p.first;

int paX = p.second;

p = find\_set(y);

y = p.first;

int paY = p.second;

if (x == y) {

if (paX == paY)

bipartite[x] = false;

}

else {

if (rank[x] < rank[y])

swap(x, y);

parent[y] = { x, paX ^ paY ^ 1 };

bipartite[x] &= bipartite[y];

if (rank[x] == rank[y])

rank[x]++;

}

}

bool is\_bipartite(int x) {

return bipartite[find\_set(x).first];

}

};

#### DSU rollback

struct dsu\_save {

int v, rnkv, u, rnku;

dsu\_save() {}

dsu\_save(int \_v, int \_rnkv, int \_u, int \_rnku)

: v(\_v), rnkv(\_rnkv), u(\_u), rnku(\_rnku) {}

};

struct dsu\_with\_rollbacks {

vector<int> p, rnk;

int comps;

stack<dsu\_save> op;

dsu\_with\_rollbacks() {}

dsu\_with\_rollbacks(int n) {

p.resize(n + 1);

rnk.resize(n + 1);

for (int i = 1; i <= n; i++) {

p[i] = i;

rnk[i] = 0;

}

comps = n;

}

int find\_set(int v) {

return (v == p[v]) ? v : find\_set(p[v]);

}

bool same\_group(int v, int u) {

v = find\_set(v);

u = find\_set(u);

if (v == u)

return false;

comps--;

if (rnk[v] > rnk[u])

swap(v, u);

op.push(dsu\_save(v, rnk[v], u, rnk[u]));

p[v] = u;

if (rnk[u] == rnk[v])

rnk[u]++;

return true;

}

void snapshot() {

// this function save the current trees (merged) and don't rollback them any more

while (!op.empty())

op.pop();

}

void rollback() {

// you can erase the while loop if you want to rollback just the last merge

while (!op.empty()) {

dsu\_save x = op.top();

op.pop();

comps++;

p[x.v] = x.v;

rnk[x.v] = x.rnkv;

p[x.u] = x.u;

rnk[x.u] = x.rnku;

}

}

};

### LCA

class LCA {

    int n, logN, root = 1;

    vector<int> depth;

    vector<vector<int>> adj, lca;

    void dfs(int node, int parent) {

        lca[node][0] = parent;

        depth[node] = (~parent ? depth[parent] + 1 : 0);

        for (int k = 1; k <= logN; k++) {

            int up\_parent = lca[node][k - 1];

            if (~up\_parent)

lca[node][k] = lca[up\_parent][k - 1];

        }

        for (int child : adj[node])

            if (child != parent)

dfs(child, node);

    }

public:

    LCA(const vector<vector<int>> &\_adj, int root = 1) : root(root), adj(\_adj) {

        adj = \_adj;

        n = adj.size() - 1;

        logN = log2(n);

        lca = vector<vector<int>>(n + 1, vector<int>(logN + 1, -1));

        depth = vector<int>(n + 1);

        dfs(root, -1);

    }

    int get\_LCA(int x, int y) {

        if (depth[x] < depth[y])

            swap(x, y);

        for (int k = logN; k >= 0; k--)

            if (depth[x] - (1 << k) >= depth[y])

                x = lca[x][k];

        if (x == y)

            return x;

        for (int k = logN; k >= 0; k--) {

            if (lca[x][k] != lca[y][k]) {

                x = lca[x][k], y = lca[y][k];

            }

        }

        return lca[x][0];

    }

    int get\_distance(int u, int v) {

        return depth[u] + depth[v] - 2 \* depth[get\_LCA(u, v)];

    }

    int shifting(int node, int dist) {

        for (int i = logN; i >= 0 && ~node; i--)

            if (dist & (1 << i))

                node = lca[node][i];

        return node;

    }

};

### Heavy light decomposition

//1-based,if value in node,just update it after build chains

//don't forget to call build\_chains after add edges.

class heavy\_light\_decomposition {

    int n, is\_value\_in\_edge;

    vector<int> parent, depth, heavy, root, pos\_in\_array, pos\_to\_node, size;

    const static int merge(int a, int b); //implement it

    struct array\_ds { //implement it

        int n;

        array\_ds(int n) : n(n) {}

    } seg;

    struct TREE {

        int cnt\_edges = 1;

        vector<vector<int>> adj;

        //need for value in edges

        vector<vector<int>> edge\_idx;

        //edge\_to need for undirected tree //end of edge in directed tree

        vector<int> edge\_to, edge\_cost;

        TREE(int n) : adj(n + 1), edge\_idx(n + 1), edge\_to(n + 1), edge\_cost(n + 1) {

        }

        void add\_edge(int u, int v, int c) {

            adj[u].push\_back(v);

            adj[v].push\_back(u);

            edge\_idx[u].push\_back(cnt\_edges);

            edge\_idx[v].push\_back(cnt\_edges);

            edge\_cost[cnt\_edges] = c;

            cnt\_edges++;

        }

    } tree;

    int dfs\_hld(int node) {

        int size = 1, max\_sub\_tree = 0;

        for (int i = 0; i < (int) tree.adj[node].size(); i++) {

            int ch = tree.adj[node][i], edge\_idx = tree.edge\_idx[node][i];

            if (ch != parent[node]) {

                tree.edge\_to[edge\_idx] = ch;

                parent[ch] = node;

                depth[ch] = depth[node] + 1;

                int child\_size = dfs\_hld(ch);

                if (child\_size > max\_sub\_tree)

                    heavy[node] = ch, max\_sub\_tree = child\_size;

                size += child\_size;

            }

        }

        return size;

    }

    vector<tuple<int, int, bool>> get\_path(int u, int v) { //l,r,must\_reverse?

        vector<pair<int, int>> tmp[2];

        bool idx = 1;

        while (root[u] != root[v]) {

            if (depth[root[u]] > depth[root[v]]) {

                swap(u, v);

                idx = !idx;

            }

            //if value in edges ,you need value of root[v] also (connecter edge)

            tmp[idx].push\_back( { pos\_in\_array[root[v]], pos\_in\_array[v] });

            v = parent[root[v]];

        }

        if (depth[u] > depth[v]) {

            swap(u, v);

            idx = !idx;

        }

        if (!is\_value\_in\_edge || u != v)

            tmp[idx].push\_back( { pos\_in\_array[u] + is\_value\_in\_edge, pos\_in\_array[v] });

        reverse(all(tmp[1]));

        vector<tuple<int, int, bool>> rt;

        for (int i = 0; i < 2; i++)

            for (auto &it : tmp[i])

                rt.emplace\_back(it.first, it.second, i == 0);

        return rt; //u is LCA

    }

public:

    heavy\_light\_decomposition(int n, bool is\_value\_in\_edge) :

            n(n), is\_value\_in\_edge(is\_value\_in\_edge), seg(n + 1), tree(n + 1) {

        heavy = vector<int>(n + 1, -1);

        parent = depth = root = pos\_in\_array = pos\_to\_node = size = vector<int>(n + 1);

    }

    void add\_edge(int u, int v, int c = 0) {

        tree.add\_edge(u, v, c);

    }

    void build\_chains(int src = 1) {

        parent[src] = -1;

        dfs\_hld(src);

        for (int chain\_root = 1, pos = 1; chain\_root <= n; chain\_root++) {

            if (parent[chain\_root] == -1 || heavy[parent[chain\_root]] != chain\_root)

                for (int j = chain\_root; j != -1; j = heavy[j]) {

                    root[j] = chain\_root;

                    pos\_in\_array[j] = pos++;

                    pos\_to\_node[pos\_in\_array[j]] = j;

                }

        }

        if (is\_value\_in\_edge)

            for (int i = 1; i < n; i++)

                update\_edge(i, tree.edge\_cost[i]);

    }

    void update\_node(int node, int value) {

        seg.update(pos\_in\_array[node], value);

    }

    void update\_edge(int edge\_idx, int value) {

        update\_node(tree.edge\_to[edge\_idx], value);

    }

    void update\_path(int u, int v, ll c) {

        vector<tuple<int, int, bool>> intervals = get\_path(u, v);

        for (auto &it : intervals)

            seg.update(get<0>(it), get<1>(it), c);

    }

    node query\_in\_path(int u, int v) {

        vector<tuple<int, int, bool>> intervals = get\_path(u, v);

        //initial value,check if handling u == v

        node query\_res = 0;

        for (auto &it : intervals) {

            int l, r;

            bool rev;

            tie(l, r, rev) = it;

            node cur = seg.query(l, r);

            if (rev) cur.reverse();

            query\_res = node(query\_res, cur);

        }

        return query\_res;

    }

};

### Centroid decomposition

class centroid\_decomposition {

vector<bool> centroidMarked;

vector<int> size;

void dfsSize(int node, int par) {

size[node] = 1;

for (int ch : adj[node])

if (ch != par && !centroidMarked[ch]) {

dfsSize(ch, node);

size[node] += size[ch];

}

}

int getCenter(int node, int par, int size\_of\_tree) {

for (int ch : adj[node]) {

if (ch == par || centroidMarked[ch]) continue;

if (size[ch] \* 2 > size\_of\_tree)

return getCenter(ch, node, size\_of\_tree);

}

return node;

}

int getCentroid(int src) {

dfsSize(src, -1);

int centroid = getCenter(src, -1, size[src]);

centroidMarked[centroid] = true;

return centroid;

}

int decomposeTree(int root) {

root = getCentroid(root);

solve(root);

for (int ch : adj[root]) {

if (centroidMarked[ch])

continue;

int centroid\_of\_subtree = decomposeTree(ch);

//note: root and centroid\_of\_subtree probably not have a direct edge in adj

centroidTree[root].push\_back(centroid\_of\_subtree);

centroidParent[centroid\_of\_subtree] = root;

}

return root;

}

void calc(int node, int par) {

//TO-DO

for (int ch : adj[node]) if (ch != par && !centroidMarked[ch])

calc(ch, node);

}

void add(int node, int par) {

//TO-DO

for (int ch : adj[node]) if (ch != par && !centroidMarked[ch])

add(ch, node);

}

void remove(int node, int par) {

//TO-DO

for (int ch : adj[node]) if (ch != par && !centroidMarked[ch])

remove(ch, node);

}

void solve(int root) {

//add root

for (int ch : adj[root])

if (!centroidMarked[ch]) {

calc(ch, root);

add(ch, root);

}

//TO-DO //remove root

for (int ch : adj[root])

if (!centroidMarked[ch])

remove(ch, root);

}

public:

int n, root;

vector<vector<int>> adj, centroidTree;

vector<int> centroidParent;

centroid\_decomposition(vector<vector<int>> &adj) : adj(adj) {

n = (int) adj.size() - 1;

size = vector<int>(n + 1);

centroidTree = vector<vector<int>>(n + 1);

centroidParent = vector<int>(n + 1, -1);

centroidMarked = vector<bool>(n + 1);

root = decomposeTree(1);

}

};

## Shortest path algorithms

### Dijkstra

struct edge {

int from, to, weight;

edge() { from = to = weight = 0;}

edge(int from, int to, int weight) :

from(from), to(to), weight(weight) {

}

bool operator <(const edge& other) const {

return weight > other.weight;

}

};

vector<vector<edge>> adj;

//O(E\*log(v))

void dijkstra(int src, int dest = -1) {

priority\_queue<edge> q;

vector<int> dis(adj.size(), INT\_MAX), prev(adj.size(), -1);

q.push(edge(-1, src, 0));

dis[src] = 0;

while (!q.empty()) {

edge e = q.top();

q.pop();

if (e.weight > dis[e.to])

continue;

prev[e.to] = e.from;

if (e.to == dest)

break;

for (edge ne : adj[e.to])

if (dis[ne.to] > dis[e.to] + ne.weight) {

ne.weight = dis[ne.to] = dis[e.to] + ne.weight;

q.push(ne);

}

}

vector<int> path;

while (dest != -1) {

path.push\_back(dest);

dest = prev[dest];

}

reverse(path.begin(), path.end());

}

### Bellmanford

vector<edge> edgeList;

//O(V\*E)

void bellmanford(int n, int src, int dest = -1) {

vector<int> dis(n + 1, oo), prev(n + 1, -1);

dis[src] = 0;

bool negativeCycle = false;

int last = -1, tmp = n;

while (tmp--) {

last = -1;

for (edge e : edgeList)

if (dis[e.to] > dis[e.from] + e.weight) {

dis[e.to] = dis[e.from] + e.weight;

prev[e.to] = e.from;

last = e.to;

}

if (last == -1)

break;

if (tmp == 0)

negativeCycle = true;

}

if (last != -1) {

for (int i = 0; i < n; i++)

last = prev[last];

vector<int> cycle;

for (int cur = last; cur != last || cycle.size() > 1; cur = prev[cur])

cycle.push\_back(cur);

reverse(cycle.begin(), cycle.end());

}

vector<int> path;

while (dest != -1) {

path.push\_back(dest);

dest = prev[dest];

}

reverse(path.begin(), path.end());

}

#### Difference constraints

void difference\_constraints() {

int m;

cin >> m;

int cnt = 1;

while (m--) {

string x1, x2;

int w; // x1 - x2 <= w

cin >> x1 >> x2 >> w;

map<string, int> id;

if (id.find(x1) == id.end())

id[x1] = cnt++;

if (id.find(x2) == id.end())

id[x2] = cnt++;

edgeList.emplace\_back(id[x2], id[x1], w);

}

for (int i = 1; i < cnt; i++)

edgeList.emplace\_back(cnt, i, 0);

bellmanford(cnt, cnt);

}

### Floyed

vector<vector<int>> adj, par;

// adj[i][j] = oo , adj[i][i] = 0 , par[i][j] = i

void init(int n) {

par = adj = vector<vector<int>>(n + 1, vector<int>(n + 1, oo));

for (int i = 1; i <= n; i++)

adj[i][i] = 0;

for (int i = 1; i <= n; i++)

for (int j = 1; j <= n; j++)

par[i][j] = i;

}

void floyd() {

for (int k = 1; k < adj.size(); k++)

for (int i = 1; i < adj.size(); i++)

for (int j = 1; j < adj.size(); j++)

if (adj[i][j] > adj[i][k] + adj[k][j]) {

adj[i][j] = adj[i][k] + adj[k][j];

par[i][j] = par[k][j];

}

}

void buildPath(int src, int dest) {

vector<int> path;

while (src != dest) {

path.push\_back(dest);

dest = par[src][dest];

}

path.push\_back(src);

reverse(path.begin(), path.end());

}

### SPFA

vector<vector<edge>> adj;

enum visit {finished, in\_queue, not\_visited };

void spfa(int src) {

int n = adj.size();

vector<int> dis(n, INF), prev(n, -1), state(n, not\_visited);

dis[src] = 0;

deque<int> q;

q.push\_back(src);

while (!q.empty()) {

int u = q.front();

q.pop\_front();

state[u] = finished;

for (auto &e : adj[u]) {

if (dis[e.to] > dis[e.from] + e.cost) {

dis[e.to] = dis[e.from] + e.cost;

prev[e.to] = e.from;

if (state[e.to] == not\_visited) {

q.push\_back(e.to);

} else if (state[e.to] == finished) {

q.push\_front(e.to);

}

state[e.to] = in\_queue;

}

}

}

}

## MST

### Kruskal

struct edge {

int from, to;

ll weight;

edge() {

from = to = weight = 0;

}

edge(int from, int to, ll weight) :

from(from), to(to), weight(weight) {

}

bool operator <(const edge& other) const {

return weight < other.weight;

}

};

vector<edge> edgeList;

pair<int, vector<edge>> MST\_Kruskal(int n) {//O(edges\*log(edges))

DSU uf(n);

vector < edge > edges;

int mstCost = 0;

sort(edgeList.begin(), edgeList.end());

for (auto e : edgeList)

if (uf.union\_sets(e.from, e.to)) {

mstCost += e.weight;

edges.push\_back(e);

}

if (edges.size() != n - 1)

return { INT\_MAX,vector<edge>() };

return { mstCost,edges };

}

int miniMax(int src, int dest, int n) {

int max = INT\_MIN;

DSU uf(n);

sort(edgeList.begin(), edgeList.end());

for (auto e : edgeList) {

if (uf.same\_set(src, dest))

return max;

uf.union\_sets(e.from, e.to);

max = e.weight;

}

return max;

}

//O(edges\*log(edges) + nodes\*nodes)

pair<int, vector<edge>> SMST\_Kruskal(int n) {

DSU uf(n);

sort(edgeList.begin(), edgeList.end());

vector<edge> take, leave;

int mstCost = 0;

for (auto e : edgeList)

if (uf.union\_sets(e.from, e.to)) {

mstCost += e.weight;

take.push\_back(e);

}

else leave.push\_back(e);

pair<int, vector<edge>> ret = { INT\_MAX, vector<edge>() };

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

uf = DSU(n);

vector <edge> edges;

mstCost = 0;

for (int j = 0; j < take.size(); j++) {

if (i == j)

continue;

uf.union\_sets(take[j].from, take[j].to);

mstCost += take[j].weight;

edges.push\_back(take[j]);

}

for (edge e : leave) {

if (uf.union\_sets(e.from, e.to)) {

mstCost += e.weight;

edges.push\_back(e);

break;

}

}

if (edges.size() == n - 1 && ret.first < mstCost)

ret = { mstCost, edges };

}

return ret;

}

### Prim

struct edge {

int from, to, weight;

edge() {

from = to = weight = 0;

}

edge(int from, int to, int weight) : from(from), to(to), weight(weight) {}

bool operator <(const edge& other) const {

return weight > other.weight;

}

};

vector<vector<edge>> adj;

vector<edge> prim(int node) {

vector<bool> vis(adj.size());

priority\_queue<edge> q;

vector<edge> edges;

q.push(edge(-1, node, 0));

while (!q.empty()) {

edge e = q.top();

q.pop();

if (vis[e.to])

continue;

vis[e.to] = true;

if (e.from != -1)

edges.push\_back(e);

for (edge ch : adj[e.to])

if (!vis[ch.to])

q.push(ch);

}

return edges;//check it connected or not

}

### SMST O(n \* log(n))

struct edge {

int from, to;

ll weight;

edge() {

from = to = weight = 0;

}

edge(int from, int to, ll weight) :

from(from), to(to), weight(weight) {

}

bool operator <(const edge& other) const {

return weight < other.weight;

}

};

int MST\_Kruskal(int n, vector<edge> edgeList, vector<edge>& take,

vector<edge>& leave) {

DSU uf(n);

vector<edge> edges;

sort(edgeList.begin(), edgeList.end());

int mst\_cost = 0;

for (auto e : edgeList)

if (uf.union\_sets(e.from, e.to)) {

take.push\_back(e);

mst\_cost += e.weight;

}

else leave.push\_back(e);

return mst\_cost;

}

struct LCA {

#define INIT { -1, -2 }

struct data {

int lca = -1;

pair<int, int> max\_edges = INIT; //first max,second max (distinct)

};

pair<int, int> merge(pair<int, int> a, pair<int, int> b) {

if (a.first < b.first)

swap(a, b);

if (b.first == a.first)

a.second = max(a.second, b.second);

else if (b.first > a.second)

a.second = b.first;

return a;

}

int logN;

vector<vector<data>> lca;

vector<vector<edge>> adj;

vector<int> depth;

void dfs(int node, int par) {

for (edge e : adj[node])

if (e.to != par) {

depth[e.to] = depth[node] + 1;

lca[e.to][0].max\_edges.first = e.weight;

lca[e.to][0].lca = node;

dfs(e.to, node);

}

}

LCA(int n, vector<edge>& edges) :

adj(n + 1) {

for (auto& e : edges) {

adj[e.from].push\_back(e);

adj[e.to].push\_back(edge(e.to, e.from, e.weight));

}

logN = log2(n);

depth = vector<int>(n + 1);

lca = vector<vector<data>>(n + 1, vector<data>(logN + 1));

dfs(1, -1);

for (int k = 1; k <= logN; k++)

for (int node = 1; node <= n; node++) {

int par = lca[node][k - 1].lca;

if (~par) {

lca[node][k].lca = lca[par][k - 1].lca;

lca[node][k].max\_edges = merge(lca[node][k - 1].max\_edges,

lca[par][k - 1].max\_edges);

}

}

}

pair<int, int> max\_two\_edges(int u, int v) {

pair<int, int> ans = INIT;

if (depth[u] < depth[v]) swap(u, v);

for (int i = logN; i >= 0; i--)

if (depth[u] - (1 << i) >= depth[v]) {

ans = merge(ans, lca[u][i].max\_edges);

u = lca[u][i].lca;

}

if (u == v) return ans;

for (int i = logN; i >= 0; i--)

if (lca[u][i].lca != lca[v][i].lca) {

ans = merge(ans, lca[u][i].max\_edges);

ans = merge(ans, lca[v][i].max\_edges);

u = lca[u][i].lca;

v = lca[v][i].lca;

}

ans = merge(ans, lca[u][0].max\_edges);

ans = merge(ans, lca[v][0].max\_edges);

return ans;

}

};

int main() {

run();

int t;

cin >> t;

for (int I = 1; I <= t; I++) {

int n, e;

cin >> n >> e;

vector<edge> edgeList(e);

for (auto& it : edgeList)

cin >> it.from >> it.to >> it.weight;

vector<edge> take, leave;

int mst\_cost = MST\_Kruskal(n, edgeList, take, leave);

if (take.size() != n - 1) {

cout << "No way\n";

continue;

}

LCA tree(n, take);

ll rt = INF;

for (edge e : leave) {

pair<int, int> p = tree.max\_two\_edges(e.from, e.to);

rt = min(rt, mst\_cost - p.first + e.weight);

}

if (rt == INF)

cout << "No second way\n";

else

cout << rt << endl;

}

}

## Tarjan

### SCC

vector<vector<int>> adj, scc;

vector<set<int>> dag;

vector<int> dfs\_num, dfs\_low, compId;

vector<bool> inStack;

stack<int> stk;

int timer;

void dfs(int node) {

dfs\_num[node] = dfs\_low[node] = ++timer;

stk.push(node);

inStack[node] = 1;

for (int child : adj[node])

if (!dfs\_num[child]) {

dfs(child);

dfs\_low[node] = min(dfs\_low[node], dfs\_low[child]);

}

else if (inStack[child])

dfs\_low[node] = min(dfs\_low[node], dfs\_num[child]);

//can be dfs\_low[node] = min(dfs\_low[node], dfs\_low[child]);

if (dfs\_low[node] == dfs\_num[node]) {

scc.push\_back(vector<int>());

int v = -1;

while (v != node) {

v = stk.top();

stk.pop();

inStack[v] = 0;

scc.back().push\_back(v);

compId[v] = scc.size() - 1;

}

}

}

void SCC() {

timer = 0;

dfs\_num = dfs\_low = compId = vector<int>(adj.size());

inStack = vector<bool>(adj.size());

scc = vector<vector<int>>();

for (int i = 1; i < adj.size(); i++)

if (!dfs\_num[i]) dfs(i);

}

void DAG() {

dag = vector<set<int>>(scc.size());

for (int i = 1; i < adj.size(); i++)

for (int j : adj[i])

if (compId[i] != compId[j])

dag[compId[i]].insert(compId[j]);

}

### Articulation points and bridges

vector<vector<int>> adj;

vector<int> dfs\_num, dfs\_low;

vector<bool> articulation\_point;

vector<pair<int, int>> bridge;

stack<pair<int, int>> edges;

vector<vector<pair<int, int>>> BCC; //biconnected components

int timer, cntChild;

void dfs(int node, int par) {

dfs\_num[node] = dfs\_low[node] = ++timer;

for (int child : adj[node]) {

if (par != child && dfs\_num[child] < dfs\_num[node])

edges.push({ node, child });

if (!dfs\_num[child]) {

if (par == -1)

cntChild++;

dfs(child, node);

if (dfs\_low[child] >= dfs\_num[node]) {

articulation\_point[node] = 1;

//get biconnected component

BCC.push\_back(vector<pair<int, int>>());

pair<int, int> edge;

do {

edge = edges.top();

BCC.back().push\_back(edge);

edges.pop();

} while (edge.first != node || edge.second != child);

}

//can be (dfs\_low[child] == dfs\_num[child])

if (dfs\_low[child] > dfs\_num[node])

bridge.push\_back({ node, child });

dfs\_low[node] = min(dfs\_low[node], dfs\_low[child]);

}

else if (child != par)

dfs\_low[node] = min(dfs\_low[node], dfs\_num[child]);

}

}

void articulation\_points\_and\_bridges() {

timer = 0;

dfs\_num = dfs\_low = vector<int>(adj.size());

articulation\_point = vector<bool>(adj.size());

bridge = vector<pair<int, int>>();

for (int i = 1; i < adj.size(); i++)

if (!dfs\_num[i]) {

cntChild = 0;

dfs(i, -1);

articulation\_point[i] = cntChild > 1;

}

}

### Edge classification

vector<vector<int>> adj;

vector<int> start, finish;

int timer;

void dfsEdgeClassification(int node) {

start[node] = timer++;

for (int child : adj[node]) {

if (start[child] == -1)

dfsEdgeClassification(child);

else {

if (finish[child] == -1)

; // Back Edge

else if (start[node] < start[child])

; // Forward Edge

else; // Cross Edge

}

}

finish[node] = timer++;

}

## 2-SAT

int n;

int Not(int x) {

return (x > n ? x - n : x + n);

}

void addEdge(int a, int b) {

adj[Not(a)].push\_back(b);

adj[Not(b)].push\_back(a);

}

void add\_xor\_edge(int a, int b) {

addEdge(Not(a), Not(b));

addEdge(a, b);

}

bool \_2SAT(vector<int>& value) {

SCC();

for (int i = 1; i <= n; i++)

if (compId[i] == compId[Not(i)])

return false;

vector<int> assign(scc.size(), -1);

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

if (assign[i] == -1) {

assign[i] = true;

assign[compId[Not(scc[i].back())]] = false;

}

for (int i = 1; i <= n; i++)

value[i] = assign[compId[i]];

return true;

}

## Flows

### Maximum bipartite matching

vector<vector<int>> adj;

vector<int> rowAssign, colAssign, vis;//make vis array instance of vector

int test\_id;

bool canMatch(int i) {

if (vis[i] == test\_id) return false;

vis[i] = test\_id;

for (int j : adj[i])

if (colAssign[j] == -1) {

colAssign[j] = i;

rowAssign[i] = j;

return true;

}

for (int j : adj[i])

if (canMatch(colAssign[j])) {

colAssign[j] = i;

rowAssign[i] = j;

return true;

}

return false;

}

// O(rows \* edges) //number of operation could by strictly less than order (1e5\*1e5->AC)

int maximum\_bipartite\_matching(int rows, int cols) {

int maxFlow = 0;

rowAssign = vector<int>(rows, -1);

colAssign = vector<int>(cols, -1);

vis = vector<int>(rows);

for (int i = 0; i < rows; i++) {

test\_id++;

if (canMatch(i)) maxFlow++;

}

vector<pair<int, int>> matches;

for (int j = 0; j < cols; j++)

if (~colAssign[j]) matches.push\_back( { colAssign[j], j });

return maxFlow;

}

### Hopcroft Karp for bipartite matching

//O(sqrt(V) \* E)

struct Hopcroft\_Karp {//1-based

#define NIL 0

#define INF INT\_MAX

int n, m;

vector<vector<int>> adj;

vector<int> rowAssign, colAssign, dist;

bool bfs() {

queue<int> q;

dist = vector<int>(adj.size(), INF);

for (int i = 1; i <= n; i++)

if (rowAssign[i] == NIL) {

dist[i] = 0;

q.push(i);

}

while (!q.empty()) {

int cur = q.front();

q.pop();

if (dist[cur] >= dist[NIL])break;

for (auto& nxt : adj[cur]) {

if (dist[colAssign[nxt]] == INF) {

dist[colAssign[nxt]] = dist[cur] + 1;

q.push(colAssign[nxt]);

}

}

}

return dist[NIL] != INF;

}

bool dfs(int i) {

if (i == NIL)

return true;

for (int j : adj[i]) {

if (dist[colAssign[j]] == dist[i] + 1 && dfs(colAssign[j])) {

colAssign[j] = i;

rowAssign[i] = j;

return true;

}

}

dist[i] = INF;

return false;

}

Hopcroft\_Karp(int n, int m)

:n(n), m(m), adj(n + 1), rowAssign(n + 1), colAssign(m + 1) {

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

}

int maximum\_bipartite\_matching() {

int rt = 0;

while (bfs()) {

for (int i = 1; i <= n; i++)

if (rowAssign[i] == NIL && dfs(i))

rt++;

}

return rt;

}

};

### Edmonds Karp

//O( V \* E \* E)

#define INF 0x3f3f3f3f3f3f3f3fLL

int n;

int capacity[101][101];

int getPath(int src, int dest, vector<int> &parent) {

    parent = vector<int>(n + 1, -1);

    queue<pair<int, int>> q;

    q.push( { src, INF });

    while (q.size()) {

        int cur = q.front().first, flow = q.front().second;

        q.pop();

        if (cur == dest) return flow;

        for (int i = 1; i <= n; i++)

            if (parent[i] == -1 && capacity[cur][i]) {

                parent[i] = cur;

                q.push( { i, min(flow, capacity[cur][i]) });

                if (i == dest)  return q.back().second;

            }

    }

    return 0;

}

int Edmonds\_Karp(int source, int sink) {

    int max\_flow = 0;

    int new\_flow = 0;

    vector<int> parent(n + 1, -1);

    while (new\_flow = getPath(source, sink, parent)) {

        max\_flow += new\_flow;

        int cur = sink;

        while (cur != source) {

            int prev = parent[cur];

            capacity[prev][cur] -= new\_flow;

            capacity[cur][prev] += new\_flow;

            cur = prev;

        };

    }

    return max\_flow;

}

### Dinic

//O(V\*V\*E) more faster

struct Dinic { //0-based

    struct flowEdge {

        int from, to;

        ll cap, flow = 0;

        flowEdge(int from, int to, ll cap) :

                from(from), to(to), cap(cap) {

        }

    };

    vector<flowEdge> edges;

    int n, m = 0, source, sink;

    vector<vector<int>> adj;

    vector<int> level, ptr;

    Dinic(int n, int source, int sink) :

            n(n), source(source), sink(sink), adj(n), level(n), ptr(n) {

    }

    void addEdge(int u, int v, ll cap) {

        edges.emplace\_back(u, v, cap);

        edges.emplace\_back(v, u, 0);

        adj[u].push\_back(m);

        adj[v].push\_back(m + 1);

        m += 2;

    }

    bool bfs() {

        queue<int> q;

        level = vector<int>(n, -1);

        level[source] = 0;

        q.push(source);

        while (!q.empty()) {

            int cur = q.front();

            q.pop();

            for (auto &id : adj[cur]) {

                if (edges[id].cap - edges[id].flow <= 0)

                    continue;

                int nxt = edges[id].to;

                if (level[nxt] != -1)

                    continue;

                level[nxt] = level[cur] + 1;

                q.push(nxt);

            }

        }

        return level[sink] != -1;

    }

    ll dfs(int node, ll cur\_flow) {

        if (cur\_flow == 0 || node == sink)

            return cur\_flow;

        for (int &cid = ptr[node]; cid < adj[node].size(); cid++) {

            int id = adj[node][cid];

            int nxt = edges[id].to;

            if (level[node] + 1 != level[nxt] || edges[id].cap - edges[id].flow <= 0)

                continue;

            ll tmp = dfs(nxt, min(cur\_flow, edges[id].cap - edges[id].flow));

            if (tmp == 0)

continue;

            edges[id].flow += tmp;

            edges[id ^ 1].flow -= tmp;

            return tmp;

        }

        return 0;

    }

    ll flow() {

        ll max\_flow = 0;

        while (bfs()) {

            fill(ptr.begin(), ptr.end(), 0);

            while (ll pushed = dfs(source, INF))

                max\_flow += pushed;

        }

        return max\_flow;

    }

};

### Min cost Max flow

struct MCMF { //0-based

    struct edge {

        int from, to, cost, cap, flow, backEdge;

        edge() {

            from = to = cost = cap = flow = backEdge = 0;

        }

        edge(int from, int to, int cost, int cap, int flow, int backEdge) :

                from(from), to(to), cost(cost), cap(cap), flow(flow), backEdge(

                        backEdge) {

        }

        bool operator <(const edge &other) const {

            return cost < other.cost;

        }

    };

    int n, src, dest;

    vector<vector<edge>> adj;

    const int OO = 1e9;

    MCMF(int n, int src, int dest) : n(n), src(src), dest(dest), adj(n) {}

    void addEdge(int u, int v, int cost, int cap) {

        edge e1 = edge(u, v, cost, cap, 0, adj[v].size());

        edge e2 = edge(v, u, -cost, 0, 0, adj[u].size());

        adj[u].push\_back(e1);

        adj[v].push\_back(e2);

    }

    pair<int, int> minCostMaxFlow() {

        int maxFlow = 0, cost = 0;

        while (true) {

            vector<pair<int, int>> path = spfa();

            if (path.empty())

                break;

            int new\_flow = OO;

            for (auto &it : path) {

                edge &e = adj[it.first][it.second];

                new\_flow = min(new\_flow, e.cap - e.flow);

            }

            for (auto &it : path) {

                edge &e = adj[it.first][it.second];

                e.flow += new\_flow;

                cost += new\_flow \* e.cost;

                adj[e.to][e.backEdge].flow -= new\_flow;

            }

            maxFlow += new\_flow;

        }

        return {maxFlow,cost};

    }

    enum visit { finished, in\_queue, not\_visited };

    vector<pair<int, int>> spfa() {

        vector<int> dis(n, OO), prev(n, -1), from\_edge(n), state(n,

                not\_visited);

        deque<int> q;

        dis[src] = 0;

        q.push\_back(src);

        while (!q.empty()) {

            int u = q.front();

            q.pop\_front();

            state[u] = finished;

            for (int i = 0; i < adj[u].size(); i++) {

                edge e = adj[u][i];

                if (e.flow >= e.cap || dis[e.to] <= dis[u] + e.cost)

                    continue;

                dis[e.to] = dis[u] + e.cost;

                prev[e.to] = u;

                from\_edge[e.to] = i;

                if (state[e.to] == in\_queue) continue;

                if (state[e.to] == finished

                        || (!q.empty() && dis[q.front()] > dis[e.to]))

                    q.push\_front(e.to);

                else

                    q.push\_back(e.to);

                state[e.to] = in\_queue;

            }

        }

        if (dis[dest] == OO)

            return {};

        vector<pair<int, int>> path;

        int cur = dest;

        while (cur != src) {

            path.push\_back( { prev[cur], from\_edge[cur] });

            cur = prev[cur];

        }

        reverse(path.begin(), path.end());

        return path;

    }

};

### Hungarian

// nodes are 0-based

/\* There are n workers and n tasks.

You know exactly how much you need to pay each worker to perform one or another task.

You also know that every worker can only perform one task.

Your goal is to assign each worker some a task,

while minimizing your expenses.

\*/

// fill vector a with costs

// if you want maximizie final cost then you will multiply edges cost with -1

// this algorithm works only on bipartite graph

// the maximum matching must equal to n

template<typename T>

class hungarian {

public:

int n;

int m;

vector< vector<T> > a;

vector<T> u;

vector<T> v;

vector<int> pa;

vector<int> pb;

vector<int> way;

vector<T> minv;

vector<bool> used;

T inf;

hungarian(int \_n, int \_m) : n(\_n), m(\_m) {

assert(n <= m);

a = vector< vector<T> >(n, vector<T>(m));

u = vector<T>(n + 1);

v = vector<T>(m + 1);

pa = vector<int>(n + 1, -1);

pb = vector<int>(m + 1, -1);

way = vector<int>(m, -1);

minv = vector<T>(m);

used = vector<bool>(m + 1);

inf = numeric\_limits<T>::max();

}

inline void add\_row(int i) {

fill(minv.begin(), minv.end(), inf);

fill(used.begin(), used.end(), false);

pb[m] = i;

pa[i] = m;

int j0 = m;

do {

used[j0] = true;

int i0 = pb[j0];

T delta = inf;

int j1 = -1;

for (int j = 0; j < m; j++) {

if (!used[j]) {

T cur = a[i0][j] - u[i0] - v[j];

if (cur < minv[j]) {

minv[j] = cur;

way[j] = j0;

}

if (minv[j] < delta) {

delta = minv[j];

j1 = j;

}

}

}

for (int j = 0; j <= m; j++) {

if (used[j]) {

u[pb[j]] += delta;

v[j] -= delta;

}

else {

minv[j] -= delta;

}

}

j0 = j1;

} while (pb[j0] != -1);

do {

int j1 = way[j0];

pb[j0] = pb[j1];

pa[pb[j0]] = j0;

j0 = j1;

} while (j0 != m);

}

inline T current\_score() {

return -v[m];

}

inline T solve() {

for (int i = 0; i < n; i++) {

add\_row(i);

}

return current\_score();

}

};

# String

## Hashing

struct hashing {

int MOD, BASE;

vector<int> Hash, modInv;

hashing(string s, int MOD, int BASE, char first\_char = 'a') :

MOD(MOD), BASE(BASE), Hash(sz(s) + 1), modInv(sz(s) + 1) {

modInv[0] = 1;

ll base = 1;

for (int i = 1; i <= sz(s); i++) {

Hash[i] = (Hash[i - 1] + (s[i - 1] - first\_char + 1) \* base) % MOD;

modInv[i] = power(base, MOD - 2, MOD);

base = (base \* BASE) % MOD;

}

}

int getHash(int l, int r) { //1-based

return (1LL \* (Hash[r] - Hash[l - 1] + MOD) % MOD \* modInv[l]) % MOD;

}

};

//MOD = 1e9 + 9 ,BASE = 31

//MOD = 2000000011 ,BASE = 53 ->careful of overflow

//\*\*\*\*\*\*\*\*\*\*\*\*\*

//MOD = 998634293,BASE = 953

//MOD = 986464091,BASE = 1013

## KMP

struct KMP {

string pattern;

vector<int> longestPrefix;

KMP(string& str) :pattern(str) {

failure\_function();

}

int fail(int k, char nxt) {

while (k > 0 && pattern[k] != nxt)

k = longestPrefix[k - 1];

if (nxt == pattern[k]) k++;

return k;

}

void failure\_function() {

int n = pattern.size();

longestPrefix = vector<int>(n);

for (int i = 1, k = 0; i < n; i++)

longestPrefix[i] = k = fail(k, pattern[i]);

}

void match(const string& str) {

int n = str.size();

int m = pattern.size();

for (int i = 0, k = 0; i < n; i++) {

k = fail(k, str[i]);

if (k == m) {

cout << i - m + 1 << endl; //0-based

k = longestPrefix[k - 1]; // if you want next match

}

}

}

};

vector<bool> suffix\_pal(string s) { //[i..n-1] pal?

string r = s;

reverse(all(r));

vector<bool> v(s.size());

v[0] = (s == r);

string pattern = r + "#" + s;

int n = pattern.size();

vector<int> longestPrefix(n);

int k = 0;

for (int i = 1; i < n; i++) {

while (k > 0 && pattern[k] != pattern[i])

k = longestPrefix[k - 1];

if (pattern[i] == pattern[k]) k++;

longestPrefix[i] = k;

}

while (k > 0) {

v[s.size() - k] = true;

k = longestPrefix[k - 1];

}

return v;

}

vector<bool> prefix\_pal(string s) { // [0..i] pal?

string r = s;

reverse(all(r));

vector<bool> v(s.size());

v.back() = (s == r);

string pattern = s + "#" + r;

int n = pattern.size();

vector<int> longestPrefix(n);

int k = 0;

for (int i = 1; i < n; i++) {

while (k > 0 && pattern[k] != pattern[i])

k = longestPrefix[k - 1];

if (pattern[i] == pattern[k])

k++;

longestPrefix[i] = k;

}

while (k > 0) {

v[k - 1] = true;

k = longestPrefix[k - 1];

}

return v;

}

//frq[i] = number of occur s[0..i] in s

vector<int> build\_fre\_prefix(const string& s) {

KMP kmp(s);

kmp.failure\_function();

vector<int> f = kmp.longestPrefix;

int n = sz(s);

vector<int> frq(n);

for (int i = n - 1; i >= 0; i--)

if (f[i]) frq[f[i] - 1] += frq[i] + 1;

for (auto& it : frq)it++;

return frq;

}

## Trie

class trie {

    struct trie\_node {

        bool is\_leaf = false;

        map<char, int> next;

        bool have\_next(char ch) {

            return next.find(ch) != next.end();

        }

        int& operator[](char ch) {

            return next[ch];

        }

    };

    vector<trie\_node> t;

public:

    trie() {

        t.push\_back(trie\_node());

    }

    void insert(const string &s) {

        int root = 0;

        for (const char &ch : s) {

            if (!t[root].have\_next(ch)) {

                t.push\_back(trie\_node());

                t[root][ch] = t.size() - 1;

            }

            root = t[root][ch];

        }

        t[root].is\_leaf = true;

    }

    bool find(const string &s) {

        int root = 0;

        for (const char &ch : s) {

            if (!t[root].have\_next(ch))

                return false;

            root = t[root][ch];

        }

        return t[root].is\_leaf;

    }

};

## Aho Corasick

struct aho\_corasick {

    struct trie\_node {

        vector<int> pIdxs; //probably take memory limit

        map<char, int> next;

        int fail;

        trie\_node() : fail(0) {}

        bool have\_next(char ch) {

            return next.find(ch) != next.end();

        }

        int& operator[](char ch) {

            return next[ch];

        }

    };

    vector<trie\_node> t;

    vector<string> patterns;

    vector<int> end\_of\_pattern;

    vector<vector<int>> adj;

    int insert(const string &s, int patternIdx) {

        int root = 0;

        for (const char &ch : s) {

            if (!t[root].have\_next(ch)) {

                t.push\_back(trie\_node());

                t[root][ch] = t.size() - 1;

            }

            root = t[root][ch];

        }

        t[root].pIdxs.push\_back(patternIdx);

        return root;

    }

    int next\_state(int cur, char ch) {

        while (cur > 0 && !t[cur].have\_next(ch))

            cur = t[cur].fail;

        if (t[cur].have\_next(ch))

            return t[cur][ch];

        return 0;

    }

    void buildAhoTree() {

        queue<int> q;

        for (auto &child : t[0].next)

            q.push(child.second);

        while (!q.empty()) {

            int cur = q.front();

            q.pop();

            for (auto &child : t[cur].next) {

                int k = next\_state(t[cur].fail, child.first);

                t[child.second].fail = k;

                vector<int> &idxs = t[child.second].pIdxs;

                //dp[child.second] = max(dp[child.second],dp[k]);

                idxs.insert(idxs.end(), all(t[k].pIdxs));

                q.push(child.second);

            }

        }

    }

    void buildFailureTree() {

        adj = vector<vector<int>>(t.size());

        for (int i = 1; i < t.size(); i++)

            adj[t[i].fail].push\_back(i);

    }

    aho\_corasick(const vector<string> &\_patterns) {

        t.push\_back(trie\_node());

        patterns = \_patterns;

        end\_of\_pattern = vector<int>(patterns.size());

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

            end\_of\_pattern[i] = insert(patterns[i], i);

        buildAhoTree();

        //buildFailureTree();

    }

    vector<vector<int>> match(const string &str) {

        int k = 0;

        vector<vector<int>> rt(patterns.size());

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

            k = next\_state(k, str[i]);

            for (auto &it : t[k].pIdxs)

                rt[it].push\_back(i);

        }

        return rt;

    }

};

## Suffix Automaton

struct suffix\_automaton {

    struct state {

        int len, link = 0, cnt = 0;

        bool terminal = false, is\_clone = false;

        map<char, int> next;

        state(int len = 0) : len(len) {}

        bool have\_next(char ch) {

            return next.find(ch) != next.end();

        }

        void clone(const state &other, int nlen) {

            len = nlen;

            next = other.next;

            link = other.link;

            is\_clone = true;

        }

    };

    vector<state> st;

    int last = 0;

    suffix\_automaton() {

        st.push\_back(state());

        st[0].link = -1;

    }

    suffix\_automaton(const string &s) : suffix\_automaton() {

        for (char ch : s) extend(ch);

        for (int cur = last; cur > 0; cur = st[cur].link)

            st[cur].terminal = true;

    }

    void extend(char c) {

        int cur = st.size();

        st.push\_back(state(st[last].len + 1));

        st[cur].cnt = 1;

        int p = last;

        last = cur;

        while (p != -1 && !st[p].have\_next(c)) {

            st[p].next[c] = cur;

            p = st[p].link;

        }

        if (p == -1) return;

        int q = st[p].next[c];

        if (st[p].len + 1 == st[q].len) {

            st[cur].link = q;

            return;

        }

        int clone = st.size();

        st.push\_back(state());

        st[clone].clone(st[q], st[p].len + 1);

        while (p != -1 && st[p].next[c] == q) {

            st[p].next[c] = clone;

            p = st[p].link;

        }

        st[q].link = st[cur].link = clone;

    }

    void calc\_number\_of\_occurrences() {

        vector<vector<int>> lvl(st[last].len + 1);

        for (int i = 1; i < st.size(); i++)

            lvl[st[i].len].push\_back(i);

        for (int i = st[last].len; i >= 0; i--)

            for (auto cur : lvl[i])

                st[st[cur].link].cnt += st[cur].cnt;

    }

    vector<ll> dp;

    ll Count(int cur) { //count number of paths

        ll &rt = dp[cur];

        if (rt) return rt;

        rt = 1;

        for (auto ch : st[cur].next)

            rt += Count(ch.second);

        return rt;

    }

    string kth\_substring(ll k) { //1-based,different substring,0 = ""

        assert(k <= Count(0));

        string rt;

        int cur = 0;

        while (k > 0) {

            for (auto ch : st[cur].next) {

                if (Count(ch.second) < k)

                    k -= Count(ch.second);

                else {

                    rt += ch.first;

                    cur = ch.second;

                    k--;

                    break;

                }

            }

        }

        return rt;

    }

    string longest\_common\_substring(const string &t) {

        int cur = 0, l = 0, mx = 0, idx = 0;

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

            while (cur > 0 && !st[cur].have\_next(t[i])) {

                cur = st[cur].link;

                l = st[cur].len;

            }

            if (st[cur].have\_next(t[i])) {

                cur = st[cur].next[t[i]];

                l++;

            }

            if (l > mx) {

                mx = l;

                idx = i;

            }

        }

        return t.substr(idx - mx + 1, mx);

    }

};

## Suffix array

class suffix\_array {

    int getOrder(int a) const {

        return (a < (int) order.size() ? order[a] : 0);

    }

    void radix\_sort(int k) {

        vector<int> frq(n), tmp(n);

        for (auto &it : suf)

            frq[getOrder(it + k)]++;

        for (int i = 1; i < n; i++)

            frq[i] += frq[i - 1];

        for (int i = n - 1; i >= 0; i--)

            tmp[--frq[getOrder(suf[i] + k)]] = suf[i];

        suf = tmp;

    }

public:

    int n;

    string s;

    vector<int> suf, lcp, order; // order store position of suffix i in suf array

    suffix\_array(const string &s) :

            n(s.size() + 1), s(s) {

        suf = order = vector<int>(n);

        vector<int> newOrder(n);

        for (int i = 0; i < n; i++)

            suf[i] = i;

        { //sort according to first character

            vector<int> tmp(n);

            for (int i = 0; i < n; i++)

                tmp[i] = s[i];

            sort(all(tmp));

            for (int i = 0; i < n; i++)

                order[i] = (lower\_bound(all(tmp), s[i]) - tmp.begin());

        }

        for (int len = 1; newOrder.back() != n - 1; len <<= 1) {

            auto cmp = [&](const int &a, const int &b) {

                if (order[a] != order[b])

                    return order[a] < order[b];

                return getOrder(a + len) < getOrder(b + len);

            };

            //sort(all(suf), cmp); //run in 576ms  (n<=4e5)

            radix\_sort(len); //sort second part

            radix\_sort(0); //sort first part

            newOrder[0] = 0;

            for (int i = 1; i < n; i++)

                newOrder[i] = newOrder[i - 1] + cmp(suf[i - 1], suf[i]);

            for (int i = 0; i < n; i++)

                order[suf[i]] = newOrder[i];

        }

        buildLCP();

    }

    /\*

     \* longest common prefix

     \* O(n)

     \* lcp[i] = lcp(suf[i],suf[i-1])

     \*/

    void buildLCP() {

        lcp = vector<int>(n);

        int k = 0;

        for (int i = 0; i < n - 1; i++) {

            int pos = order[i];

            int j = suf[pos - 1];

            while (s[i + k] == s[j + k])

                k++;

            lcp[pos] = k;

            if (k)

                k--;

        }

    }

    int LCP\_by\_order(int a, int b) {

        if (a > b) swap(a, b);

        int mn = n - suf[a] - 1;

        for (int k = a + 1; k <= b; k++)

            mn = min(mn, lcp[k]);

        return mn; }

    //LCP(i,j) : longest common prefix between suffix i and suffix j

    int LCP(int i, int j) {

        //return LCP\_by\_order(order[i],order[j]);

        if (order[j] < order[i])

            swap(i, j);

        int mn = n - i - 1;

        for (int k = order[i] + 1; k <= order[j]; k++)

            mn = min(mn, lcp[k]);

        return mn;

    }

    //compare s[a.first..a.second] with s[b.first..b.second]

    //-1:a<b ,0:a==b,1:a>b

    int compare\_substrings(pair<int, int> a, pair<int, int> b) {

        int lcp = min({ LCP(a.first, b.first), a.second - a.first + 1,

b.second - b.first + 1 });

        a.first += lcp;

b.first += lcp;

        if (a.first <= a.second) {

            if (b.first <= b.second) {

                if (s[a.first] == s[b.first])

                    return 0;

                return (s[a.first] < s[b.first] ? -1 : 1);

            }

            return 1;

        }

        return (b.first <= b.second ? -1 : 0);

    }

    pair<int, int> find\_string(const string &x) {

        int st = 0, ed = n;

        for (int i = 0; i < sz(x) && st < ed; i++) {

            auto cmp = [&](int a, int b) {

                if (a == -1)

                    return x[i] < s[b + i];

                return s[a + i] < x[i];

            };

            st = lower\_bound(suf.begin() + st, suf.begin() + ed, -1, cmp)

                    - suf.begin();

            ed = upper\_bound(suf.begin() + st, suf.begin() + ed, -1, cmp)

                    - suf.begin();

        }

        return {st,ed-1};

    }

};

ll number\_of\_different\_substrings(string s) {

    int n = s.size();

    suffix\_array sa(s);

    ll cnt = 0;

    for (int i = 0; i <= n; i++)

        cnt += n - sa.suf[i] - sa.lcp[i];

    return cnt;

}

string longest\_common\_substring(const string &s1, const string &s2) {

    suffix\_array sa(s1 + "#" + s2);

    vector<int> suf = sa.suf, lcp = sa.lcp;

    auto type = [&](int idx) {

        return idx <= s1.size();

    };

    int mx = 0, idx = 0;

    int len = s1.size() + 1 + s2.size();

    for (int i = 1; i <= len; i++)

        if (type(suf[i - 1]) != type(suf[i]) && lcp[i] > mx) {

            mx = lcp[i];

            idx = min(suf[i - 1], suf[i]);

        }

    return s1.substr(idx, mx);

}

int longest\_common\_substring(const vector<string> &v) {

    int n = v.size();

    int len = n - 1;

    for (auto &it : v)

        len += it.size();

    string s(len, '.');

    vector<int> type(len + 1, n), frq(n + 1);

    for (int i = 0, j = 0; i < v.size(); i++) {

        if (i) s[j] = 'z' + i; //review this

        for (char ch : v[i]) {

            s[j] = ch;

            type[j] = i;

            j++;

        }

    }

    suffix\_array sa(s);

    vector<int> suf = sa.suf, lcp = sa.lcp;

    monoqueue q;

    int st = 0, ed = 0, cnt = 0, mx = 0;

    while (st <= s.size()) {

        while (ed <= s.size() && cnt < v.size()) {

            q.push(lcp[ed], ed);

            if (++frq[type[suf[ed]]] == 1)

                cnt++;

            ed++;

        }

        q.pop(st);

        if (cnt == v.size()) mx = max(mx, q.getMin()); //st+1,ed

        if (--frq[type[suf[st]]] == 0) cnt--;

        st++;

    }

    return mx;

}

string kth\_substring(string s, int k) { //1-based,repated

    int n = s.size();

    suffix\_array sa(s);

    vector<int> suf = sa.suf, lcp = sa.lcp;

    for (int i = 1; i <= n; i++) {

        int len = n - suf[i];

        int cnt = 0;

        for (int l = 1; l <= len; l++) {

            cnt++;

            int st = i + 1, ed = n, ans = i;

            while (st <= ed) {

                int md = st + ed >> 1;

                if (sa.LCP\_by\_order(i, md) >= l)

                    st = md + 1, ans = md;

                else   ed = md - 1;

            }

            cnt += ans - i;

            if (cnt >= k) return s.substr(suf[i], l);

        }

        k -= len;

    }

    assert(0);

}

### Suffix array Faster

class suffix\_array {

    const static int alpha = 128;

    int getOrder(int a) const {

        return (a < (int) order.size() ? order[a] : 0);

    }

public:

    int n;

    string s;

    vector<int> suf, order, lcp; // order store position of suffix i in suf array

    suffix\_array(const string &s) : n(s.size() + 1), s(s) {

        suf = order = lcp = vector<int>(n);

        vector<int> bucket\_idx(n), newOrder(n), newsuff(n);

        vector<int> prev(n), head(alpha, -1);

        for (int i = 0; i < n; i++) {

            prev[i] = head[s[i]];

            head[s[i]] = i;

        }

        int buc = -1, idx = 0;

        for (int i = 0; i < alpha; i++) {

            if (head[i] == -1) continue;

            bucket\_idx[++buc] = idx;

            for (int j = head[i]; ~j; j = prev[j])

                suf[idx++] = j, order[j] = buc;

        }

        int len = 1;

        do {

            auto cmp = [&](int a, int b) {

                if (order[a] != order[b])

                    return order[a] < order[b];

                return getOrder(a + len) < getOrder(b + len);

            };

            for (int i = 0; i < n; i++) {

                int j = suf[i] - len;

                if (j < 0)

                    continue;

                newsuff[bucket\_idx[order[j]]++] = j;

            }

            for (int i = 1; i < n; i++) {

                suf[i] = newsuff[i];

                bool cmpres = cmp(suf[i - 1], suf[i]);

                newOrder[suf[i]] = newOrder[suf[i - 1]] + cmpres;

                if (cmpres)

                    bucket\_idx[newOrder[suf[i]]] = i;

            }

            order = newOrder;

            len <<= 1;

        } while (order[suf[n - 1]] != n - 1);

    }

};

## Z algorithm

/\* z[i] equal the length of the longest substring starting from s[i]

which is also a prefix of s \*/

vector<int> z\_algo(string s) {

    int n = s.size();

    vector<int> z(n);

    z[0] = n;

    for (int i = 1, L = 1, R = 1; i < n; i++) {

        int k = i - L;

        if (z[k] + i >= R) {

            L = i;

            R = max(R, i);

            while (R < n && s[R - L] == s[R]) R++;

            z[i] = R - L;

        } else z[i] = z[k];

    }

    return z;

}

# Math

## Primes

### Sieve

const int N = 1e8;

bool isPrime[N + 1];

vector<int> prime;

void sieve() {

memset(isPrime, true, sizeof(isPrime));

isPrime[0] = isPrime[1] = false;

for (int i = 4; i <= N; i += 2)

isPrime[i] = false;

for (int i = 3; i \* i <= N; i += 2)

if (isPrime[i])

for (int j = i \* i; j <= N; j += i + i)

isPrime[j] = false;

for (int i = 1; i <= N; i++)

if (isPrime[i])

prime.push\_back(i);

}

### Linear Sieve

const int N = 1e7;

int lpf[N + 1];

vector<int> prime;

void sieve() {

for (int i = 2; i <= N; i++) {

if (lpf[i] == 0) {

lpf[i] = i;

prime.push\_back(i);

}

for (int j : prime) {

if (j > lpf[i] || 1LL \* i \* j > N)break;

lpf[i \* j] = j;

}

}

}

### Miller Rabin Primality Test

const int ITER = 4;

mt19937 rng(chrono::steady\_clock::now().time\_since\_epoch().count());

bool millerTest(ll n, ll d) {

ll a = uniform\_int\_distribution<ll>(2, n - 2)(rng);

a = fpow(a, d, n);

if (a == 1 || a == n - 1)

return true;

d <<= 1;

while (d != n - 1) {

a = a \* a % n;

if (a == 1) return false;

if (a == n - 1) return true;

d <<= 1;

}

return false;

}

bool isPrime(ll n) {

if (n <= 1) return false;

if (n <= 3) return true;

if (!(n & 1)) return false;

ll d = n - 1;

while (!(d & 1))

d >>= 1;

for (int i = 0; i < ITER; i++)

if (!millerTest(n, d))

return false;

return true;

}

bool isPrimeSquare(ll n) {

ll sq = round(sqrt(n));

if (sq \* sq < n) {

sq++;

if (sq \* sq != n)return false;

}

return isPrime(n);

}

int countDivisors(ll n) {

// ans will contain total number of distinct

// divisors

int ans = 1;

// Loop for counting factors of n

for (int i = 2; 1LL \* i \* i \* i <= n; i++) {

// Calculating power of i in n.

int cnt = 1; // cnt is power of prime i in n.

while (n % i == 0) // if i is a factor of n

{

n = n / i;

cnt = cnt + 1; // incrementing power

}

// Calculating the number of divisors

// If n = a^p \* b^q then total divisors of n

// are (p+1)\*(q+1)

ans = ans \* cnt;

}

// If i is greater than cube root of n

// First case

if (isPrime(n))

ans = ans << 1;

// Second case

else if (isPrimeSquare(n))

ans = ans \* 3;

// Third case

else if (n != 1)

ans = ans << 2;

return ans; // Total divisors

}

### Prime Factors

// return number of Divisors(n) using prime factorization

ll numOfDivisors(primeFactors mp) {

ll cnt = 1;

for (auto it : mp) cnt \*= (it.second + 1);

return cnt;

}

// return sum of Divisors(n) using prime factorization

ll sumOfDivisors(primeFactors mp) {

ll sum = 1;

for (auto it : mp) sum \*= sumPower(it.first, it.second);

return sum;

}

### Phi function

ll phi\_function(ll n) {

ll result = n;

primeFactors pf = prime\_factors(n);

for (auto &it : pf) {

ll p = it.first;

result -= (result / p);

}

return result;

}

void phi\_1\_to\_n(int n) {

for (int i = 0; i <= n; i++)

phi[i] = i;

for (int i = 2; i <= n; i++)

if (phi[i] == i)

for (int j = i; j <= n; j += i)

phi[j] -= phi[j] / i;

}

### Moebius function

char mob[N];

bool prime[N];

void moebius() {

memset(mob, 1, sizeof mob);

memset(prime + 2, 1, sizeof(prime) - 2);

mob[0] = 0;

mob[2] = -1;

for (int i = 4; i < N; i += 2) {

mob[i] \*= (i & 3) ? -1 : 0;

prime[i] = 0;

}

for (int i = 3; i < N; i += 2)

if (prime[i]) {

mob[i] = -1;

for (int j = 2 \* i; j < N; j += i) {

mob[j] \*= j % (1LL \* i \* i) ? -1 : 0;

prime[j] = 0;

}

}

}

## Extended Euclidean

ll egcd(ll a, ll b, ll& x, ll& y) {

if (a < 0) {

auto g = egcd(-a, b, x, y);

x \*= -1;

return g;

}

if (b < 0) {

auto g = egcd(a, -b, x, y);

y \*= -1;

return g;

}

if (!b) {

x = 1;

y = 0;

return a;

}

ll x1, y1;

ll g = egcd(b, a % b, x1, y1);

x = y1, y = x1 - y1 \* (a / b);

return g;

}

## Linear Diophantine Equation

// return false if there is no solution

// return true if there exist a solution

// x, y are the solutions and g is the gcd between a and b

bool Diophantine\_Solution(ll a, ll b, ll c, ll& x, ll& y, ll& g) {

if (!a && !b) {

if (c) return false;

x = y = g = 0;

return true;

}

g = egcd(a, b, x, y);

if (c % g) return false;

x \*= c / g;

y \*= c / g;

return true;

}

void shift\_solution(ll& x, ll& y, ll a, ll b, ll cnt) {

x += b \* cnt;

y -= a \* cnt;

}

// find all number of solutions of ax + by = c

// x in range {minx, maxx}

// y in range {miny, maxy}

ll Diophantine\_Solutions(ll a, ll b, ll c, ll minx, ll maxx, ll miny, ll maxy) {

if (minx > maxx || miny > maxy) return 0;

if (!a && !b && !c)

return (maxx - minx + 1) \* (maxy - miny + 1);

if (!a && !b) return 0;

if (!a) {

if (c % b) return 0;

ll num = c / b;

return (num >= miny && num <= maxy) \* (maxx - minx + 1);

}

if (!b) {

if (c % a) return 0;

ll num = c / a;

return (num >= minx && num <= maxx) \* (maxy - miny + 1);

}

ll x, y, g;

if (!Diophantine\_Solution(a, b, c, x, y, g))

return 0;

ll lx1, lx2, rx1, rx2;

// a \* x + b \* y = c

// (a / g) \* x + (b / g) \* y = c / g

a /= g, b /= g, c /= g;

g = 1;

int sign\_a = (a > 0 ? 1 : -1);

int sign\_b = (b > 0 ? 1 : -1);

// x + k \* b >= minx

// k \* b >= minx - x

// k >= (minx - x) / b

// k >= ceil((minx - x) / b)

shift\_solution(x, y, a, b, (minx - x) / b);

// if x is less than minx so we need to increase it by one step only

// from the upove equation x + k \* b >= minx

// if b is positive so choose k equal to 1 to increase x one

// if b is negattive so choose k equal to -1 because -1 \* -1 = 1 ans also increase x

if (x < minx)

shift\_solution(x, y, a, b, sign\_b);

if (x > maxx) return 0;

lx1 = x;

// x + k \* b <= maxx

// k \* b <= maxx - x

// k <= (maxx - x) / b

shift\_solution(x, y, a, b, (maxx - x) / b);

if (x > maxx)

shift\_solution(x, y, a, b, -sign\_b);

rx1 = x;

// y - k \* a >= miny

// y - miny >= k \* a

// k \* a <= y - miny

// k <= (y - miny) / a

shift\_solution(x, y, a, b, (y - miny) / a);

if (y < miny)

shift\_solution(x, y, a, b, -sign\_a);

if (y > maxy) return 0;

lx2 = x;

// y - k \* a <= maxy

// y - maxy <= k \* a

// k \* a >= y - maxy

// k >= (y - maxy) / a

shift\_solution(x, y, a, b, (y - maxy) / a);

if (y > maxy)

shift\_solution(x, y, a, b, sign\_a);

rx2 = x;

if (lx2 > rx2) swap(lx2, rx2);

// becuase we calculate the equations lx2, rx2 from shifting y

// not from shifting x directly

ll lx = max(lx1, lx2);

ll rx = min(rx1, rx2);

if (lx > rx) return 0;

return (rx - lx) / abs(b) + 1;

}

## Extended Euclidean for n variables

//O(n \* log(m)) Memory & Time; coefficients.size() <= n, coefficients[i] <= m

// 0-based implementation

template<typename T>

T extended\_euclidean(const deque<T>& cof, deque<T>& var) {

int n = cof.size();

if (!cof.back()) {

int cnt = 0, id = 0;

for (int i = 0; i < n; i++)

if (!cof[i]) {

cnt++;

var[i] = 0;

}

else id = i;

if (cnt >= n - 1) {

var[id] = 1;

return cof[id];

}

deque<T> new\_cof, new\_var;

for (int i = 0; i < n; i++)

if (cof[i]) {

new\_cof.push\_back(cof[i]);

new\_var.push\_back(var[i]);

}

T g = extended\_euclidean(new\_cof, new\_var);

for (int i = 0; !new\_var.empty(); i++)

if (cof[i]) {

var[i] = new\_var.front();

new\_var.pop\_front();

}

return g;

}

deque<T> new\_cof = cof;

for (int i = 0; i < n - 1; i++)

new\_cof[i] %= new\_cof.back();

new\_cof.push\_front(new\_cof.back());

new\_cof.pop\_back();

var.push\_front(var.back());

var.pop\_back();

T g = extended\_euclidean(new\_cof, var);

var.push\_back(var.front());

var.pop\_front();

for (int i = 0; i < n - 1; i++)

var.back() -= cof[i] / cof.back() \* var[i];

return g;

}

template<typename T>

vector<T> find\_any\_solution(const vector<T>& cof, T rhs) {

int n = cof.size();

if (!n)

return vector<T>();

deque<T> deque\_cof(cof.begin(), cof.end()), deque\_var(n);

T g = extended\_euclidean(deque\_cof, deque\_var);

if (g && rhs % g)

return vector<T>();

vector<T> var(deque\_var.begin(), deque\_var.end());

if (g) {

rhs /= g;

for (auto& it : var)

it \*= rhs;

}

return var;

}

## Sum Sequence

//return sum of sequence a, a+x , a+2x .... b

ll sumSequence(ll a, ll b, ll x) {

a = ((a + x - 1) / x) \* x;

b = (b / x) \* x;

return (b + a) \* (b - a + x) / (2 \* x);

}

## Sum Range Divisors

// return sum of divisors for all number from 1 to n //O(n)

ll sumRangeDivisors(int n) {

ll ans = 0;

for (int x = 1; x <= n; x++)

ans += (n / x) \* x;

return ans;

}

// calc 1e9 in 42ms,can calc more but need big integer

ll sumRangeDivisors(ll x) {

ll ans = 0, left = 1, right;

for (; left <= x; left = right + 1) {

right = x / (x / left);

ans += (x / left) \* (left + right) \* (right - left + 1) / 2;

}

return ans;

}

## Combinatorics

/\*

\* nCr = n!/((n-r)! \* r!)

\* nCr(n,r) = nCr(n,n-r)

\* nPr = n!/(n-r)!

\* nPr(circle) = nPr/r

\* nCr(n,r) = pascal[n][r]

\* catalan[n] = nCr(2n,n)/(n+1)

\*/

ull nCr(int n, int r) {

if (r > n)

return 0;

r = max(r, n - r);

ull ans = 1, div = 1, i = r + 1;

while (i <= n) {

ans \*= i++;

ans /= div++;

}

return ans;

}

ull nPr(int n, int r) {

if (r > n)

return 0;

ull p = 1, i = n - r + 1;

while (i <= n)

p \*= i++;

return p;

}

// return catalan number n-th using dp O(n^2)//max = 35 then overflow

vector<ull> catalanNumber(int n) {

vector<ull> catalan(n + 1);

catalan[0] = catalan[1] = 1;

for (int i = 2; i <= n; i++) {

ull& rt = catalan[i];

for (int j = 0; j < n; j++)

rt += catalan[j] \* catalan[n - j - 1];

}

return catalan;

}

// count number of paths in matrix n\*m

// go to right or down only

ull countNumberOfPaths(int n, int m) {

return nCr(n + m - 2, n - 1);

}

### nCr pre calculation

const int N = 1e5 + 100;

const int mod = 1e9 + 7;

ll fact[N];

ll inv[N]; //mod inverse for i

ll invfact[N]; //mod inverse for i!

void factInverse() {

fact[0] = inv[1] = fact[1] = invfact[0] = invfact[1] = 1;

for (long long i = 2; i < N; i++) {

fact[i] = (fact[i - 1] \* i) % mod;

inv[i] = mod - (inv[mod % i] \* (mod / i) % mod);

invfact[i] = (inv[i] \* invfact[i - 1]) % mod;

}

}

ll nCr(int n, int r) {

if (r > n) return 0;

return (((fact[n] \* invfact[r]) % mod) \* invfact[n - r]) % mod;

}

## Matrices

typedef vector<int> row;

typedef vector<row> matrix;

matrix initial(int n, int m, int val = 0) {

return matrix(n, row(m, val));

}

matrix identity(int n) {

matrix rt = initial(n, n);

for (int i = 0; i < n; i++)rt[i][i] = 1;

return rt;

}

matrix addIdentity(const matrix& a) {

matrix rt = a;

for (int i = 0; i < sz(a); i++)rt[i][i] += 1;

return rt;

}

matrix add(const matrix& a, const matrix& b) {

matrix rt = initial(sz(a), sz(a[0]));

for (int i = 0; i < sz(a); i++)for (int j = 0; j < sz(a[0]); j++)

rt[i][j] = a[i][j] + b[i][j];

return rt;

}

matrix multiply(const matrix& a, const matrix& b) {

matrix rt = initial(sz(a), sz(b[0]));

for (int i = 0; i < sz(a); i++) for (int k = 0; k < sz(a[0]); k++) {

if (a[i][k] == 0)continue;

for (int j = 0; j < sz(b[0]); j++)

rt[i][j] += a[i][k] \* b[k][j];

}

return rt;

}

matrix power(const matrix& a, ll k) {

if (k == 0)return identity(sz(a));

if (k & 1)return multiply(a, power(a, k - 1));

return power(multiply(a, a), k >> 1);

}

matrix power\_itr(matrix a, ll k) {

matrix rt = identity(sz(a));

while (k) {

if (k & 1)rt = multiply(rt, a);

a = multiply(a, a); k >>= 1;

}

return rt;

}

matrix sumPower(const matrix& a, ll k) {

if (k == 0)return initial(sz(a), sz(a));

if (k & 1)return multiply(a, addIdentity(sumPower(a, k - 1)));

return multiply(sumPower(a, k >> 1), addIdentity(power(a, k >> 1)));

}

/\* return matrix contains

a^k 0

a^1+a^2.. a^k I

\*/

matrix sumPowerV2(const matrix& a, ll k) {

int n = sz(a);

matrix rt = initial(2 \* n, 2 \* n);

for (int i = 0; i < 2 \* n; i++)

for (int j = 0; j < n; j++)

rt[i][j] = a[i % n][j];

for (int i = n; i < 2 \* n; i++)rt[i][i] = 1;

return power(rt, k);

}

## Matrix class

struct matrix {

using T = int;

using row = vector<T>;

vector<vector<T>> v;

matrix() {}

matrix(int n, int m, T val = 0) : v(n, row(m, val)) {

}

int size() const {

return v.size();

}

int cols() const {

return v[0].size();

}

matrix operator\*(T a) const {

matrix rt = \*this;

REP(i, rt.size())

REP(j, rt.cols())

rt.v[i][j] \*= a;

return rt;

}

friend matrix operator\*(T a, const matrix& b) {

return (b \* a);

}

friend matrix operator+(const matrix& a, const matrix& b) {

assert(a.size() == b.size() && a.cols() == b.cols());

matrix rt(a.size(), a.cols());

REP(i, rt.size()) REP(j, rt.cols())

rt.v[i][j] = a.v[i][j] + b.v[i][j];

return rt;

}

friend matrix operator\*(const matrix& a, const matrix& b) {

assert(a.cols() == b.size());

matrix rt(a.size(), b.cols());

REP(i, rt.size()) REP(k, a.cols()) {

if (a.v[i][k] == 0) continue;

REP(j, rt.cols()) rt.v[i][j] += a.v[i][k] \* b.v[k][j];

}

return rt;

}

};

matrix identity(int n) {

matrix r(n, n);

for (int i = 0; i < n; i++)

r.v[i][i] = 1;

return r;

}

matrix addIdentity(const matrix& a) {

matrix rt = a;

REP(i, a.size()) rt.v[i][i]++;

return rt;

}

matrix power(matrix a, long long y) {

assert(y >= 0 && a.size() == a.cols());

matrix rt = identity(a.size());

while (y > 0) {

if (y & 1)

rt = rt \* a;

a = a \* a;

y >>= 1;

}

return rt;

}

matrix sumPower(const matrix& a, ll k) {

if (k == 0)

return matrix(sz(a), sz(a));

if (k & 1)

return a \* addIdentity(sumPower(a, k - 1));

return (sumPower(a, k >> 1) \* addIdentity(power(a, k >> 1)));

}

/\* return matrix contains

a^k 0

a^1+a^2.. a^k I

\*/

matrix sumPowerV2(const matrix& a, ll k) {

int n = sz(a);

matrix rt(2 \* n, 2 \* n);

REP(i, n) REP(j, n) {

rt.v[i][j] = a.v[i][j];

rt.v[i + n][j] = a.v[i][j];

}

for (int i = n; i < 2 \* n; i++)

rt.v[i][i] = 1;

return power(rt, k);

}

## Mod

### Fast power

ll power(ll x, ll y, int mod) {

    if (y == 0) return 1;

    if (y == 1) return x % mod;

    ll r = power(x, y >> 1, mod);

    return (((r \* r) % mod) \* power(x, y & 1, mod)) % mod;

}

### Sum of powers

// return a ^ 1 + a ^ 2 + a ^ 3 + .... a ^ k

ll sumPower(ll a, ll k, int mod) {

if (k == 1) return a % mod;

ll half = sumPower(a, k / 2, mod);

ll p = half \* power(a, k / 2, mod) % mod;

p = (p + half) % mod;

if (k & 1) p = (p + power(a, k, mod)) % mod;

return p;

}

### Mod Inverse

ll modInverse(ll b, ll mod) { // if mod is Prime

    return power(b, mod - 2, mod);

}

ll modInverse(ll b, ll mod) { // if mod is not Prime,gcd(a,b) must be equal 1

    return power(b, phi\_function(mod) - 1, mod);

}

### (a^n)%p=result , return n

// (a^n)%p=result, return minimum n

int getPower(int a, int result, int mod) {

int sq = sqrt(mod);

map<int, int> mp;

ll r = 1;

for (int i = 0; i < sq; i++) {

if (mp.find(r) == mp.end())

mp[r] = i;

r = (r \* a) % mod;

}

ll tmp = modInverse(r, mod);

ll cur = result;

for (int i = 0; i <= mod; i += sq) {

if (mp.find(cur) != mp.end())

return i + mp[cur];

cur = (cur \* tmp) % mod;//val/(a^sq)

}

return INF;

}

// Returns minimum x for which a ^ x % m = b % m.

// a,m not not coprime

**int** **getPower**(**int** a, **int** b, **int** m) {

a %= m, b %= m;

**int** k = 1, add = 0, g;

**while** ((g = \_\_gcd(a, m)) > 1) {

**if** (b == k)

**return** add;

**if** (b % g)

**return** -1;

b /= g, m /= g, ++add;

k = (k \* 1ll \* a / g) % m;

}

**int** n = sqrt(m) + 1;

**int** an = 1;

**for** (**int** i = 0; i < n; ++i)

an = (an \* 1ll \* a) % m;

unordered\_map<**int**, **int**> vals;

**for** (**int** q = 0, cur = b; q <= n; ++q) {

vals[cur] = q;

cur = (cur \* 1ll \* a) % m;

}

**for** (**int** p = 1, cur = k; p <= n; ++p) {

cur = (cur \* 1ll \* an) % m;

**if** (vals.count(cur)) {

**int** ans = n \* p - vals[cur] + add;

**return** ans;

}

}

**return** -1;

}

### CRT

ll CRT(vector<ll>& a, vector<ll>& m){

ll lcm = m[0], rem = a[0];

int n = a.size();

for(int i = 1; i < n; i++){

ll x, y;

ll gcd = extended\_euclidean(lcm, m[i], x, y);

if((a[i] - rem) % gcd) return -1;

ll tmp = m[i] / gcd, f = (a[i] - rem) / gcd;

x = ((x % tmp) \* (f % tmp)) % tmp;

rem += lcm \* x;

lcm = lcm \* m[i] / gcd;

rem = (rem % lcm + lcm) % lcm;

}

return rem;

}

## FFT

typedef valarray<complex<double>> polynomial;

vector<complex<double>> CM1[3][LGN + 1];

const double PI = acos(-1);

void prepare() {

for (int sign = -1; sign <= 1; sign += 2) {

for (int i = 0; i <= LGN; i++) {

int N = 1 << i;

double theta = sign \* 2 \* PI / N;

complex<double> cm1 = 1;

complex<double> cm2(cos(theta), sin(theta));

for (int j = 0; j < N / 2; j++) {

CM1[sign + 1][i].push\_back(cm1);

cm1 \*= cm2;

}

}

}

}

void fft(polynomial &a, int sign = -1) {

int N = a.size();

int lgn = log2(N);

for (int m = N; m >= 2; m >>= 1, lgn--) {

int mh = m >> 1;

for (int i = 0; i < mh; i++) {

const complex<double> &w = CM1[sign + 1][lgn][i];

for (int j = i; j < N; j += m) {

int k = j + mh;

complex<double> x = a[j] - a[k];

a[j] += a[k];

a[k] = w \* x;

}

}

}

int i = 0;

for (int j = 1; j < N - 1; j++) {

for (int k = N >> 1; k > (i ^= k); k >>= 1)

;

if (j < i)

swap(a[i], a[j]);

}

}

void inv\_fft(polynomial &a) {

complex<double> N = a.size();

fft(a, 1);

a /= N;

}

valarray<int> mul(const valarray<int> &a, const valarray<int> &b) {

int adeg = (int) a.size() - 1, bdeg = (int) b.size() - 1;

int N = 1;

while (N <= adeg + bdeg)

N <<= 1;

polynomial A(N), B(N);

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

A[i] = a[i];

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

B[i] = b[i];

fft(A);

fft(B);

polynomial m = A \* B;

inv\_fft(m);

valarray<int> rt(N);

for (int i = 0; i < N; i++)

rt[i] = round(m[i].real());

return rt;

}

## NTT

typedef valarray<modint> polynomial;

vector<modint> CM1[2][LGN + 1];

bool validRoot(modint root) {

modint rootinv = modint(1) / root;

for (int invert = 0; invert <= 1; invert++) {

for (int i = 1; i <= LGN; i++) {

int N = 1 << i;

assert((MOD - 1) % N == 0);

int C = (MOD - 1) / N;

modint cm2 = modint::power(invert ? root : rootinv, C);

if (cm2.val <= 1) return false;

}

}

return true;

}

void prepare() {

modint root = 2;

while (!validRoot(root)) root += 1;

modint rootinv = modint(1) / root;

for (int invert = 0; invert <= 1; invert++) {

for (int i = 0; i <= LGN; i++) {

int N = 1 << i;

int C = (MOD - 1) / N;

modint cm2 = modint::power(invert ? root : rootinv, C);

modint cm1 = 1;

set<int> st;

for (int j = 0; j < N / 2; j++) {

CM1[invert][i].push\_back(cm1);

cm1 \*= cm2;

}

}

}

}

void fft(polynomial& a, bool invert = 0) {

int N = a.size();

int lgn = log2(N);

for (int m = N; m >= 2; m >>= 1, lgn--) {

int mh = m >> 1;

for (int i = 0; i < mh; i++) {

const modint& w = CM1[invert][lgn][i];

for (int j = i; j < N; j += m) {

int k = j + mh;

modint x = a[j] - a[k];

a[j] += a[k];

a[k] = w \* x;

}

}

}

int i = 0;

for (int j = 1; j < N - 1; j++) {

for (int k = N >> 1; k > (i ^= k); k >>= 1) continue;

if (j < i)swap(a[i], a[j]);

}

}

void inv\_fft(polynomial& a) {

int N = a.size();

fft(a, 1);

a /= N;

}

valarray<modint> mul(const polynomial& a, const polynomial& b) {

int adeg = (int)a.size() - 1, bdeg = (int)b.size() - 1;

int N = 1;

while (N <= adeg + bdeg)

N <<= 1;

polynomial A(N), B(N);

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

A[i] = a[i];

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

B[i] = b[i];

fft(A);

fft(B);

polynomial rt = A \* B;

inv\_fft(rt);

return rt;

}

# Misc

## Bitmask

bool getBit(ll num, int ind) {

return ((num >> ind) & 1);

}

ll setBit(ll num, int ind, bool val) {

return val ? (num | (1LL << ind)) : (num & ~(1LL << ind));

}

ll flipBit(ll num, int ind) {

return (num ^ (1LL << ind));

}

ll leastBit(ll num) {

return (num & -num);

}

template<class Int>

Int turnOnLastZero(Int num) {

return num | num + 1;

}

template<class Int>

Int turnOnLastConsecutiveZeroes(Int num) {

return num | num - 1;

}

template<class Int>

Int turnOffLastBit(Int num) {

return num & num - 1;

}

template<class Int>

Int turnOffLastConsecutiveBits(Int num) {

return num & num + 1;

}

//num%mod, mod is a power of 2

ll Mod(ll num, ll mod) {

return (num & mod - 1);

}

bool isPowerOfTwo(ll num) {

return (num & num - 1) == 0;

}

void genAllSubmask(int mask) {

for (int subMask = mask;; subMask = (subMask - 1) & mask) {

//code

if (subMask == 0)

break;

}

}

/\* \_\_builtin functions:

\* \_\_builtin\_popcount -> used to count the number of one’s

\* \_\_builtin\_clz -> used to count the leading zeros of the integer

\* \_\_builtin\_ctz -> used to count the trailing zeros of the integer

\*

int LOG2(int x) { //floor(log2(x))

return 31 - \_\_builtin\_clz(x);

}

int LOG2(long long x) { //floor(log2(x))

return 63 - \_\_builtin\_clzll(x);

}

## Coordinate Compress

void coordinateCompress(vector<int>& axes, vector<int>& iToV,

map<int, int> vToI, int start = 2, int step = 2) {

for (auto& it : axes) vToI[it] = 0;

iToV.resize(start + step \* vToI.size());

int idx = 0;

for (auto& it : vToI) {

it.second = start + step \* idx++;

iToV[it.second] = it.first;

}

}

## Random numbers

#include <chrono>

#include <random>

//write this line once in top

mt19937\_64 rng(chrono::steady\_clock::now().time\_since\_epoch().count()\* ((uint64\_t) new char | 1));

// use this instead of rand()

template<typename T> T Rand(T low, T high) {

return uniform\_int\_distribution<T>(low, high)(rng);

}

## Custom hash

struct custom\_hash {

static uint64\_t splitmix64(uint64\_t x) {

x += 0x9e3779b97f4a7c15;

x = (x ^ (x >> 30)) \* 0xbf58476d1ce4e5b9;

x = (x ^ (x >> 27)) \* 0x94d049bb133111eb;

return x ^ (x >> 31);

}

size\_t operator()(pair<uint64\_t, uint64\_t> x) const {

static const uint64\_t FIXED\_RANDOM = chrono::steady\_clock::now().time\_since\_epoch().count();

return splitmix64(x.first + FIXED\_RANDOM) ^ (splitmix64(x.second + FIXED\_RANDOM) >> 1);

}

size\_t operator()(uint64\_t x) const {

static const uint64\_t FIXED\_RANDOM = chrono::steady\_clock::now().time\_since\_epoch().count();

return splitmix64(x + FIXED\_RANDOM);

}

};

## Max histogram area

int maxHistogramArea(vector<int> v) {

stack<int> st;

int maxArea = 0, area = 0, i = 0;

while (i < sz(v)) {

if (st.empty() || v[st.top()] <= v[i])

st.push(i++);

else {

int top = st.top(); st.pop();

if (st.empty()) area = v[top] \* i;

else area = v[top] \* (i - st.top() - 1);

maxArea = max(maxArea, area);

}

}

while (!st.empty()) {

int top = st.top(); st.pop();

if (st.empty())

area = v[top] \* i;

else area = v[top] \* (i - st.top() - 1);

maxArea = max(maxArea, area);

}

return maxArea;

}

## Sorting

long long cnt = 0;

vector<int> v, temp;

// e the first index not have in range array

// like end()

template<class T = less<int>>

void merge\_sort(int s, int e, T cmp = less<int>()) {

if (s + 1 >= e) return;

int m = s + (e - s >> 1);

merge\_sort(s, m, cmp);

merge\_sort(m, e, cmp);

for (int i = s; i < e; i++)

temp[i] = v[i];

int i = s, j = m, k = s;

while (i < m && j < e)

if (cmp(temp[i], temp[j]))

v[k++] = temp[i++];

else

v[k++] = temp[j++], cnt += j - k;

while (i < m)

v[k++] = temp[i++];

while (j < e)

v[k++] = temp[j++];

}

// O(n\*log(n)/log(base))

// O(n + base) memory

void radix\_sort(vector<int>& v, int base) {

vector<int> tmp(v.size());

for (int it = 0, p = 1; it < 10; it++, p \*= base) {

vector<int> frq(base);

for (auto& it : v)

frq[(it / p) % base]++;

for (int i = 1; i < base; i++)

frq[i] += frq[i - 1];

for (int i = v.size() - 1; i >= 0; i--)

tmp[--frq[(v[i] / p) % base]] = v[i];

v = tmp;

}

}

void quick\_sort(int s, int e) {

if (s >= e) return;

int j = rand() % (e - s + 1) + s;

swap(v[s], v[j]);

j = s;

int pivot = v[s];

for (int i = s + 1; i <= e; i++)

if (v[i] <= pivot)

swap(v[i], v[++j]);

swap(v[s], v[j]);

quick\_sort(s, j - 1);

quick\_sort(j + 1, e);

}

## LIS binary Search

/\* without build

\* make upper\_bound if can take equal elements \*/

int LIS(const vector<int>& v) {

vector<int> lis(v.size());//put value less than zero if needed

int l = 0;

for (int i = 0; i < sz(v); i++) {

int idx = lower\_bound(lis.begin(), lis.begin() + l, v[i]) - lis.begin();

if (idx == l)

l++;

lis[idx] = v[i];

}

return l;

}

void LIS\_binarySearch(vector<int> v) {

int n = v.size();

vector<int> last(n), prev(n, -1);

int length = 0;

auto BS = [&](int val) {

int st = 1, ed = length, md, rt = length;

while (st <= ed) {

md = st + ed >> 1;

if (v[last[md]] >= val)

ed = md - 1, rt = md;

else st = md + 1;

}

return rt;

};

for (int i = 1; i < n; i++) {

if (v[i] < v[last[0]])

last[0] = i;

else if (v[i] > v[last[length]]) {

prev[i] = last[length];

last[++length] = i;

}

else {

int index = BS(v[i]);

prev[i] = last[index - 1];

last[index] = i;

}

}

cout << length + 1 << "\n";

vector<int> out;

for (int i = last[length]; i >= 0; i = prev[i])

out.push\_back(v[i]);

reverse(out.begin(), out.end());

for (auto it : out)

cout << it << endl;

}

## Mo algorithm

int sqrtN; //use a constent value

struct query {

int l, r, qIdx, block;

query(int l, int r, int qIdx) : l(l), r(r), qIdx(qIdx), block(l / sqrtN) {}

bool operator <(const query& o) const {

if (block != o.block) return block < o.block;

return (block % 2 == 0 ? r < o.r : r > o.r);

}

};

int curL, curR, ans;

vector<query> q;

void add(int index);

void remove(int index);

void solve(int l, int r) {

while (curL > l) add(--curL);

while (curR < r) add(++curR);

while (curL < l) remove(curL++);

while (curR > r) remove(curR--);

}

vector<int> MO(int n) {

vector<int> rt(q.size());

ans = curL = curR = 0;

add(0);

sort(q.begin(), q.end());

for (auto it : q) {

solve(it.l, it.r);

rt[it.qIdx] = ans;

}

return rt;

}

## floyd cycle detection algorithm

template<class IntFunction>

pair<int, int> find\_cycle\_floyd(IntFunction f, int x0) {

int tortoise = f(x0), hare = f(f(x0));

while (tortoise != hare) {

tortoise = f(tortoise);

hare = f(f(hare));

}

int start = 0;

tortoise = x0;

while (tortoise != hare) {

tortoise = f(tortoise);

hare = f(hare);

start++;

}

int length = 1;

hare = f(tortoise);

while (tortoise != hare) {

hare = f(hare);

length++;

}

return make\_pair(start, length);

}

## Convex Hull Trick (Line Container)

struct Line {

    ll m, c;

    mutable ll p; //p is intersection between cur and next

    bool operator<(const Line &o) const {

        //change to (m>o.m,c < o.c) to get min

        if (m != o.m)

            return m < o.m;

        return c > o.c;

    }

    bool operator<(ll x) const {

        return p < x;

    }

};

struct LineContainer: multiset<Line, less<>> {

    // (for doubles, use inf = INFINITY, div(a,b) = a/b)

    static const ll inf = LLONG\_MAX;

    ll div(ll a, ll b) { // floored division

        return a / b - ((a ^ b) < 0 && a % b);

    }

    bool isect(iterator x, iterator y) {

        if (y == end())

            return x->p = inf, 0;

        if (x->m == y->m)

            x->p = inf;

        else

            x->p = div(y->c - x->c, x->m - y->m);

        return x->p >= y->p;

    }

    void add(ll m, ll c) {

        auto z = insert( { m, c, 0 }), y = z++, x = y;

        while (isect(y, z))

            z = erase(z);

        if (x != begin() && isect(--x, y))

            isect(x, y = erase(y));

        while ((y = x) != begin() && (--x)->p >= y->p)

            isect(x, erase(y));

    }

    ll query(ll x) {

        assert(!empty());

        auto l = \*lower\_bound(x);

        return l.m \* x + l.c;

    }

};

## Java Scanner

static class MScanner {

StringTokenizer st;

BufferedReader br;

public MScanner(InputStream system) {

br = new BufferedReader(new InputStreamReader(system));

}

public MScanner(String file) throws Exception {

br = new BufferedReader(new FileReader(file));

}

public String next() throws IOException {

while (st == null !st.hasMoreTokens())

st = new StringTokenizer(br.readLine());

return st.nextToken();

}

public String nextLine() throws IOException {

return br.readLine();

}

public int nextInt() throws IOException {

return Integer.parseInt(next());

}

public double nextDouble() throws IOException {

return Double.parseDouble(next());

}

public char nextChar() throws IOException {

return next().charAt(0);

}

public long nextLong() throws IOException {

return Long.parseLong(next());

}

public int[] intArr(int n) throws IOException {

int[]in = new int[n]; for (int i = 0; in; i++)in[i] = nextInt();

return in;

}

public long[] longArr(int n) throws IOException {

long[]in = new long[n]; for (int i = 0; in; i++)in[i] = nextLong();

return in;

}

public int[] intSortedArr(int n) throws IOException {

int[]in = new int[n]; for (int i = 0; in; i++)in[i] = nextInt();

shuffle(in);

Arrays.sort(in);

return in;

}

public long[] longSortedArr(int n) throws IOException {

long[]in = new long[n]; for (int i = 0; in; i++)in[i] = nextLong();

shuffle(in);

Arrays.sort(in);

return in;

}

static void shuffle(int[]in) {

for (int i = 0; iin.length; i++) {

int idx = (int)(Math.random()in.length);

int tmp = in[i];

in[i] = in[idx];

in[idx] = tmp;

}

}

static void shuffle(long[]in) {

for (int i = 0; iin.length; i++) {

int idx = (int)(Math.random()in.length);

long tmp = in[i];

in[i] = in[idx];

in[idx] = tmp;

}

}

public Integer[] IntegerArr(int n) throws IOException {

Integer[]in = new Integer[n]; for (int i = 0; in; i++)in[i] = nextInt();

return in;

}

public Long[] LongArr(int n) throws IOException {

Long[]in = new Long[n]; for (int i = 0; in; i++)in[i] = nextLong();

return in;

}

public boolean ready() throws IOException {

return br.ready();

}

public void waitForInput() throws InterruptedException {

Thread.sleep(3000);

}

}

# Geometry

## point

#define ll long long

typedef long double ld;

typedef complex<double> point; // it can be long long not double

template<class T>

istream& operator>>(istream& is, complex<T>& p) {

T value;

is >> value;

p.real(value);

is >> value;

p.imag(value);

return is;

}

#define PI acos(-1.0)

#define EPS 1e-8

#define X real()

#define Y imag()

#define angle(a) (atan2((a).imag(), (a).real())) // angle with orignial

#define length(a) (hypot((a).imag(), (a).real()))

#define vec(a,b) ((b)-(a))

#define dp(a,b) ( (conj(a)\*(b)).real() )

#define cp(a,b) ( (conj(a)\*(b)).imag() ) product = area of parllelogram

#define normalize(a) (a)/length(a)

// norm(a) // return x^2 + y^2 //a is point //can use dp(a,a)

bool same(point p1, point p2) {// check to points same or not

return dp(vec(p1, p2), vec(p1, p2)) < EPS;

}

### rotate

point rotate(point p, double angle, point around = point(0, 0)) {

p -= around;

return (p \* exp(point(0, angle))) + around;

}

### reflect

// Refelect v around m and origin

point reflectO(point v, point m) {

return conj(v / m) \* m;

}

// Refelect point p around l1-l2

point reflect(point p, point l1, point l2) {

point z = p - l1, w = l2 - l1;

return conj(z / w) \* w + l1;

}

## Triangles

/\*

sin(A)/a = sin(B)/b = sin(C)/c

a^2 = b^2 + c^2 - 2b\*c\*cos(A

sin(A+B) = sin(A) \* cos(B) + sin(B) \* cos(A)

sin(A-B) = sin(A) \* cos(B) - sin(B) \* cos(A)

cos(A+B) = cos(A) \* cos(B) - sin(A) \* sin(B)

cos(A-B) = cos(A) \* cos(B) + sin(A) \* sin(B)

tan(A+B) = (tan(A) + tan(B))/(1 - tan(A) \* tan(B))

tan(A-B) = (tan(A) - tan(B))/(1 - tan(A) \* tan(B))

\*/

### Get Angles/Sides

double fixAngle(double A) {

return A > 1 ? 1 : (A < -1 ? -1 : A);

}

// return min angle: aOb / bOa

// dp(v1, v2) = |v1|\*|v2|\*cos(theta)

double angleO(point a, point O, point b) {

point v1(a - O), v2(b - O);

return acos(fixAngle(dp(v1, v2) / dist(v1) / dist(v2)));

}

double getSide\_a\_bAB(double b, double A, double B) {

return (sin(A) \* b) / sin(B);

}

double getAngle\_A\_abB(double a, double b, double B) {

return asin(fixAngle((a \* sin(B)) / b));

}

// give me wrong answer in team formation :D

double getAngle\_A\_abc(double a, double b, double c) {

return acos(fixAngle((b \* b + c \* c - a \* a) / (2 \* b \* c)));

}

double triangleArea(double a, double b, double c) {

double s = (a + b + c) / 2.0;

return sqrt((s - a) \* (s - b) \* (s - c) \* s);

}

double triangleArea(point p0, point p1, point p2) {

double a = length(vec(p1, p0)), b = length(vec(p2, p0)), c = length(

vec(p2, p1));

return triangleArea(a, b, c);

}

### Point In Triangle

bool pointInTriangle(point a, point b, point c, point pt) {

ll s1 = fabs(cp(vec(a,b), vec(a,c)));

ll s2 = fabs(cp(vec(pt,a), vec(pt,b))) + fabs(cp(vec(pt, b), vec(pt, c)))

+ fabs(cp(vec(pt, a), vec(pt, c)));

return s1 == s2;

}

### Get largest Circle Inside the triangle

//A triangle with area A and semi-perimeter s has an inscribed circle (incircle) with

//radius r = A/s

bool circleInTriangle(point a, point b, point c, point& ctr, double& r) {

double ab = length(a - b), bc = length(b - c),

ca = length(c - a);

double s = 0.5 \* (ab + bc + ca);

r = triangleArea(ab, bc, ca) / s;

if (fabs(r) < EPS) return 0; // no inCircle center

double ratio = length(a - b) / length(a - c);

point p1 = b + (vec(b, c) \* (ratio / (1 + ratio)));

ratio = length(b - a) / length(b - c);

point p2 = a + (vec(a, c) \* (ratio / (1 + ratio)));

return intersectSegments(a, p1, b, p2, ctr); // get their intersection point

}

## Lines

### IsCollinear

//o = anlge(a) - angle(b)

//if o = 0 || o = 180 isCollinear

//else not

bool isCollinear(point a, point b, point c) {

return fabs(cp(vec(a, b), vec(a, c))) < EPS;

}

### isPointOnRay

//o = anlge(a) - angle(b)

//if o = 0 isPointOnRay a->b

//else not

bool isPointOnRay(point a, point b, point c) {

if (!isCollinear(a, b, c))

return false;

return dcmp(dp(vec(a, b), vec(a, c)), 0) == 1;

}

bool isPointOnRay(point a, point b, point c) {

if (length(vec(a, c)) < EPS) return true;

return same(normalize(vec(a, b)), normalize(vec(a, c)));

}

### isPointOnSegment

bool isPointOnSegment(point a, point b, point c) {

return isPointOnRay(a, b, c) && isPointOnRay(b, a, c);

}

bool isPointOnSegment(point a, point b, point c) {

double acb = length(vec(b, a)), ac = length(vec(c, a)), cb = length(vec(c, b));

return dcmp(acb - (ac + cb), 0) == 0;

}

### distToLine

// dist point p2 to line p0-p1

double distToLine(point p0, point p1, point p2) {

return fabs(cp(vec(p0, p1), vec(p0, p2)) / length(vec(p1, p0)));

}

### distToSegment

//minimum distance from point p2 to segment p0-p1

double distToSegment(point p0, point p1, point p2) {

double d1, d2;

point v1 = p1 - p0, v2 = p2 - p0;

if ((d1 = dp(v1, v2)) <= 0) return length(vec(p0, p2));

if ((d2 = dp(v1, v1)) <= d1) return length(vec(p1, p2));

double t = d1 / d2;

return length(vec((p0 + v1 \* t), p2));

}

### pointToSegment

// minimum point in segment po-p1 to point p2

point pointToSegment(point p0, point p1, point p2) {

double d1, d2;

point v1 = p1 - p0, v2 = p2 - p0;

if ((d1 = dp(v1, v2)) <= 0) return p0;

if ((d2 = dp(v1, v1)) <= d1) return p1;

double t = d1 / d2;

return (p0 + v1 \* t);

}

### intersectSegments

// return point intersect in line a-b with c-d using parametric equations

bool intersectSegments(point a, point b, point c, point d, point& intersect) {

double d1 = cp(vec(b, a), vec(c, d)), d2 = cp(vec(c, a), vec(c, d)), d3 = cp(vec(b, a), vec(c, a));

if (fabs(d1) < EPS)

return false; // Parllel || identical

double t1 = d2 / d1, t2 = d3 / d1;

intersect = a + (b - a) \* t1;

if (t1 < -EPS || t2 < -EPS || t2 > 1 + EPS)

return false; //e.g ab is ray, cd is segment ... change to whatever

return true;

}

### CCW

// return 1 if point c is counter-clockwise about segment a-b

// -1 if point c is clockwise about segment a-b

// 0 if c is isCollinear about a-b

int ccw(point a, point b, point c) {

point v1(b - a), v2(c - a);

double t = cp(v1, v2);

if (t > EPS)

return 1;

if (t < -EPS)

return -1;

if (v1.X \* v2.X < -EPS || v1.Y \* v2.Y < -EPS)

return -1;

if (norm(v1) < norm(v2) - EPS)

return 1;

return 0;

}

## Circles

### Find circle passes with 3 points

// 2 points has infinite circles

// Find circle passes with 3 points, some times, there is no circle! (in case colinear)

// Draw two perpendicular lines and intersect them

pair<double, point> findCircle(point a, point b, point c) {

//create median, vector, its prependicular

point m1 = (b + a) \* 0.5, v1 = b - a, pv1 = point(v1.Y, -v1.X);

point m2 = (b + c) \* 0.5, v2 = b - c, pv2 = point(v2.Y, -v2.X);

point end1 = m1 + pv1, end2 = m2 + pv2, center;

intersectSegments(m1, end1, m2, end2, center);

return make\_pair(length(vec(center, a)), center);

}

### intersectLineCircle

// If line intersect cirlce at point p, and p = p0 + t(p1-p0)

// Then (p-c)(p-c) = r^2 substitute p and rearrange

// (p1-p0)(p1-p0)t^2 + 2(p1-p0)(p0-C)t + (p0-C)(p0-C) = r\*r; -> Quadratic

vector<point> intersectLineCircle(point p0, point p1, point C, double r) {

double a = dp(vec(p0, p1), vec(p0, p1)), b = 2 \* dp(vec(p0, p1), vec(C, p0)),

c = dp(vec(C, p0), vec(C, p0)) - r \* r;

double f = b \* b - 4 \* a \* c;

vector<point> v;

if (dcmp(f, 0) >= 0) {

if (dcmp(f, 0) == 0) f = 0;

double t1 = (-b + sqrt(f)) / (2 \* a);

double t2 = (-b - sqrt(f)) / (2 \* a);

v.push\_back(p0 + t1 \* (p1 - p0));

if (dcmp(f, 0) != 0) v.push\_back(p0 + t2 \* (p1 - p0));

}

return v;

}

### Circle Circle Intersection

vector<point> intersectCircleCircle(point c1, double r1, point c2, double r2) {

// Handle infinity case first: same center/radius and r > 0

if (same(c1, c2) && dcmp(r1, r2) == 0 && dcmp(r1, 0) > 0)

return vector<point>(3, c1); // infinity 2 same circles (not points)

// Compute 2 intersection case and handle 0, 1, 2 cases

double ang1 = angle(vec(c1, c2)), ang2 = getAngle\_A\_abc(r2, r1, length(vec(c1, c2)));

if (::isnan(ang2)) ang2 = 0; // if r1 or d = 0 => nan in getAngle\_A\_abc (/0)

vector<point> v(1, polar(r1, ang1 + ang2) + c1);

// if point NOT on the 2 circles = no intersection

if (dcmp(dp(vec(c1, v[0]), vec(c1, v[0])), r1 \* r1) != 0 ||

dcmp(dp(vec(c2, v[0]), vec(c2, v[0])), r2 \* r2) != 0)

return vector<point>();

v.push\_back(polar(r1, ang1 - ang2) + c1);

if (same(v[0], v[1])) // if same, then 1 intersection only

v.pop\_back();

return v;

}

### Circle Circle Intersection Area

ld circleCircleIntersectionArea(point cen1, ld r1, point cen2, ld r2) {

ld dis = hypot(cen1.X - cen2.X, cen1.Y - cen2.Y);

if (dis > r1 + r2)return 0;

if (dis <= fabs(r2 - r1) && r1 >= r2)

return PI \* r2 \* r2;

if (dis <= fabs(r2 - r1) && r1 < r2)

return PI \* r1 \* r1;

ld a = r1 \* r1, b = r2 \* r2;

ld ang1 = acos((a + dis \* dis - b) / (2 \* r1 \* dis)) \* 2;

ld ang2 = acos((b + dis \* dis - a) / (2 \* r2 \* dis)) \* 2;

ld ret1 = .5 \* b \* (ang2 - sin(ang2));

ld ret2 = .5 \* a \* (ang1 - sin(ang1));

return ret1 + ret2;

}

## Check if polygon is convex

// CCW function must return 0 if the 3 points are collinear

bool isConvex(vector<point>& v) {

int n = v.size(), m = n, sum = 0;

v.push\_back(v[0]);

v.push\_back(v[1]);

char tmp;

for (int i = 0; i < n; i++) {

tmp = ccw(v[i], v[i + 1], v[i + 2]);

if (tmp) sum += tmp;

else m--;

}

v.pop\_back();

v.pop\_back();

return abs(sum) == m;

}

## Convex hull

bool cmp(point a, point b) {

return a.X < b.X || (a.X == b.X && a.Y < b.Y);

}

bool cw(point a, point b, point c) {

return cp(vec(a, b), vec(b, c)) < 0;

}

bool ccw(point a, point b, point c) {

return cp(vec(a, b), vec(b, c)) > 0;

}

vector<point> convex\_hull(vector<point>& p) {

if (p.size() == 1) return p;

sort(p.begin(), p.end(), &cmp);

point p1 = p[0], p2 = p.back();

vector<point> up, down;

up.push\_back(p1);

down.push\_back(p1);

for (int i = 1; i < (int)p.size(); i++) {

if (i == p.size() - 1 || cw(p1, p[i], p2)) {

while (up.size() >= 2

&& !cw(up[up.size() - 2], up[up.size() - 1], p[i]))

up.pop\_back();

up.push\_back(p[i]);

}

if (i == p.size() - 1 || ccw(p1, p[i], p2)) {

while (down.size() >= 2

&& !ccw(down[down.size() - 2], down[down.size() - 1], p[i]))

down.pop\_back();

down.push\_back(p[i]);

}

}

vector<point> convex;

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

convex.push\_back(down[i]);

for (int i = up.size() - 2; i > 0; i--)

convex.push\_back(up[i]);

return convex;

}

## Point in polygon O(log(n))

void prepare(vector<point>& polygon) {

int pos = 0;

for (int i = 0; i < sz(polygon); i++) {

if (make\_pair(polygon[i].X, polygon[i].Y)

< make\_pair(polygon[pos].X, polygon[pos].Y))

pos = i;

}

rotate(polygon.begin(), polygon.begin() + pos, polygon.end());

}

int dcmp(double x, double y) {

if (fabs(x - y) <= EPS)

return 0;

return (x < y ? -1 : 1);

}

ll cross(point a, point b, point c) {

return cp(vec(a, b), vec(a, c));

}

bool isPointOnSegment(point a, point b, point c) {

double acb = length(a - b), ac = length(a - c), cb = length(b - c);

return dcmp(acb - (ac + cb), 0) == 0;

}

//call prepare(polygon) before start

bool pointInConvexPolygon(const vector<point>& polygon, point pt) {

if (isPointOnSegment(polygon[0], polygon.back(), pt))

return true;

if (cross(polygon[0], polygon.back(), pt) > 0)

return false;

if (cross(polygon[0], polygon[1], pt) < 0)

return false;

if (polygon.size() == 2)

return false;

int st = 2, ed = sz(polygon) - 2, ans = 1;

while (st <= ed) {

int md = st + ed >> 1;

if (cross(polygon[0], polygon[md], pt) >= 0)

st = md + 1, ans = md;

else

ed = md - 1;

}

return cross(polygon[ans], polygon[ans + 1], pt) >= 0;

}

## line sweep for lines intersections

struct segment {

point p, q;

int idx;

segment() {

}

segment(point a, point b, int idx) :

p(a), q(b), idx(idx) {

}

double CY(int x) const {

if (p.X == q.X)

return p.Y;

double t = 1.0 \* (x - p.X) / (q.X - p.X);

return p.Y + (q.Y - p.Y) \* t;

}

bool operator<(const segment& o) const {

if (p == o.p && q == o.q)

return idx < o.idx;

int maxX = max(p.X, o.p.X);

int res = dcmp(CY(maxX), o.CY(maxX));

if (res == 0)

return idx < o.idx;

return res < 0;

}

};

struct event {

bool entry;

point p;

int idx;

event(point p, bool entry, int idx) :

p(p), entry(entry), idx(idx) {

}

bool operator<(const event& o) const {

if (p.X != o.p.X)

return p.X < o.p.X;

if (entry != o.entry)

return entry > o.entry;

return idx < o.idx;

}

};

int ccw(point a, point b, point c) {

point v1(b - a), v2(c - a);

double t = cp(v1, v2);

if (t > EPS)

return 1;

if (t < -EPS)

return -1;

if (v1.X \* v2.X < -EPS || v1.Y \* v2.Y < -EPS)

return -1;

if (norm(v1) < norm(v2) - EPS)

return 1;

return 0;

}

bool intersect(point p1, point p2, point p3, point p4) {

// special case handling if a segment is just a point

bool x = (p1 == p2), y = (p3 == p4);

if (x && y) return p1 == p3;

if (x) return ccw(p3, p4, p1) == 0;

if (y) return ccw(p1, p2, p3) == 0;

return ccw(p1, p2, p3) \* ccw(p1, p2, p4) <= 0

&& ccw(p3, p4, p1) \* ccw(p3, p4, p2) <= 0;

}

pair<int, int> solve(vector<segment> v) {

int n = v.size();

vector<event> events;

for (int i = 0; i < n; i++) {

point& p = v[i].p;

point& q = v[i].q;

if (q.X < p.X || (q.X == p.X && q.Y < p.Y))

swap(p, q);

events.push\_back(event(p, true, i));

events.push\_back(event(q, false, i));

}

sort(all(events));

set<segment> st;

auto before = [&](set<segment>::iterator it) {

if (it == st.begin())

return st.end();

return --it;

};

auto check = [&](set<segment>::iterator x,

set<segment>::iterator y) -> bool {

if (x == st.end() || y == st.end())

return false;

return intersect(x->p, x->q, y->p, y->q);

};

for (auto& cur : events) {

if (cur.entry) {

auto it = st.insert(v[cur.idx]).first;

auto below = before(it);

auto above = next(it);

if (check(below, it)) return { below->idx,it->idx };

if (check(above, it)) return { above->idx,it->idx };

}

else {

auto it = st.find(v[cur.idx]);

auto below = before(it);

auto above = next(it);

if (check(below, above)) return { below->idx,above->idx };

st.erase(it);

}

}

return { -1,-1 };

}

## Pyramid Volume

ld pyramidVolume(ld ab, ld ac, ld ad, ld bc, ld bd, ld cd) {

ld w = ab, v = ac, u = ad, U = bc, V = bd, W = cd;

ld A = (w - U + v) \* (U + v + w);

ld x = (U - v + w) \* (v - w + U);

ld B = (u - V + w) \* (V + w + u);

ld y = (V - w + u) \* (w - u + V);

ld Z = (v - W + u) \* (W + u + v);

ld z = (W - u + v) \* (u - v + W);

ld a = sqrt(x \* B \* Z);

ld b = sqrt(A \* y \* Z);

ld c = sqrt(A \* B \* z);

ld d = sqrt(x \* y \* z);

ld volume = -a + b + c + d;

volume \*= a - b + c + d;

volume \*= a + b - c + d;

volume \*= a + b + c - d;

volume = sqrt(volume) / (192.0 \* u \* v \* w);

return volume;

}

## Suffix Automaton

class SuffixAutomaton {

private:

class SNode {

public:

int len, link, firstpos, is\_clone;

map<char, int> next;

int endpos;

vi inv\_next;

SNode() {

len = 0, link = firstpos = -1;

endpos = 1;

is\_clone = 0;

}

SNode(const SNode& other) :SNode() {

len = other.len;

link = other.link;

next = other.next;

firstpos = other.firstpos;

endpos = other.endpos;

is\_clone = other.is\_clone;

}

};

public:

int n, last, cur;

string str;

vector<SNode> nodes;

vector<ll> dist\_substr;

vi sz\_link\_tree;

SuffixAutomaton(string s) {

str = s;

n = sz(str);

nodes.resize(2 \* n);

dist\_substr = vector<ll>(2 \* n, -1);

sz\_link\_tree = vi(2 \* n);

last = 0, cur = 1;

for (auto& it : str)

add\_char(it);

build\_endpos();

build\_distinct\_substrings(0);

build\_inv\_next();

dfs\_linktree(0);

}

void add\_char(char ch) {

int newNode = cur++;

nodes[newNode].len = nodes[last].len + 1;

nodes[newNode].firstpos = nodes[newNode].len - 1;

int p = last;

last = newNode;

for (; p != -1 && nodes[p].next.find(ch) == nodes[p].next.end(); p = nodes[p].link)

nodes[p].next[ch] = newNode;

if (p == -1) {

nodes[newNode].link = 0;

return;

}

int q = nodes[p].next[ch];

if (nodes[p].len + 1 == nodes[q].len) {

nodes[newNode].link = q;

return;

}

int clone = cur++;

nodes[clone] = SNode(nodes[q]);

nodes[clone].len = nodes[p].len + 1;

nodes[clone].endpos = 0;

nodes[clone].is\_clone = true;

nodes[q].link = clone;

nodes[newNode].link = clone;

for (; p != -1 && nodes[p].next.find(ch) != nodes[p].next.end() && nodes[p].next[ch] == q; p = nodes[p].link)

nodes[p].next[ch] = clone;

}

void build\_endpos() {

vi tmp(cur - 1);

iota(all(tmp), 1);

sort(all(tmp), [&](int a, int b) {return nodes[a].len > nodes[b].len; });

for (auto& it : tmp)

nodes[nodes[it].link].endpos += nodes[it].endpos;

nodes[0].endpos = 0;

}

void build\_distinct\_substrings(int u) {

ll& ret = dist\_substr[u];

if (ret != -1) return;

ret = u > 0;

for (auto& it : nodes[u].next) {

build\_distinct\_substrings(it.second);

ret += dist\_substr[it.second];

}

}

void build\_inv\_next() {

// be attention this order is important to make occurrences sorted in increasing order

for (int i = 1; i < cur; i++)

nodes[nodes[i].link].inv\_next.push\_back(i);

}

void dfs\_linktree(int u) {

sz\_link\_tree[u] = !nodes[u].is\_clone;

for (auto& v : nodes[u].inv\_next) {

dfs\_linktree(v);

sz\_link\_tree[u] += sz\_link\_tree[v];

}

}

string lcs(const string& p) {

int u = 0, l = 0, best = 0, bestpos = 0;

for (int i = 0; i < sz(p); i++) {

while (u && nodes[u].next.find(p[i]) == nodes[u].next.end()) {

u = nodes[u].link;

l = nodes[u].len;

}

if (nodes[u].next.find(p[i]) != nodes[u].next.end()) {

l++;

u = nodes[u].next[p[i]];

}

if (l > best) {

best = l;

bestpos = i;

}

}

return p.substr(bestpos - best + 1, best);

}

int match(const string& p) {

int u = 0, idx = 0;

while (idx < sz(p) && nodes[u].next.find(p[idx]) != nodes[u].next.end())

u = nodes[u].next[p[idx]], idx++;

if (idx != sz(p))

return 0;

return u;

}

vi all\_occurrences\_of\_pattern(int u, int p\_len) {

vi ret; // 0 based all indices of p in automaton

queue<int> q;

q.push(u);

while (!q.empty()) {

int cur = q.front();

q.pop();

if (!nodes[cur].is\_clone)

ret.push\_back(nodes[cur].firstpos - p\_len + 1);

for (auto& v : nodes[cur].inv\_next)

q.push(v);

}

return ret;

}

vi get\_occurrences\_of\_pattern(const string& p) {

int u = match(p);

if (!u) return vi();

return all\_occurrences\_of\_pattern(u, sz(p));

}

int get\_cnt\_occurrences\_of\_pattern(const string& p) {

int u = match(p);

if (!u) return 0;

return sz\_link\_tree[u];

}

void kth\_distinct(int u, ll k, string& ret) {

if (u) k--;

if (!k) return;

for (auto& it : nodes[u].next) {

if (dist\_substr[it.second] >= k) {

ret.push\_back(it.first);

kth\_distinct(it.second, k, str);

return;

}

k -= dist\_substr[it.second];

}

}

string get\_kth\_distinct(ll k) {

if (k > dist\_substr[0]) return "-1";

string ret = "";

kth\_distinct(0, k, ret);

return ret;

}

};