Don't get Dijkstracted - ICPC Library

Contents

1 Data Structures

1.1 Segment Tree

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
const int ms = 1e6; // Max size of v

// Implementation of a Recursive Segment Tree (0-Based) (Sum Operation)
int seg[4*ms];
vector<int> v;

// SINGLE-ELEMENT MODIFY, RANGE QUERY [l,r]

// Call using build(1, 0, N - 1) (N = v.size())
void build(int p, int l, int r){
    if(l == r) seg[p] = v[l];
```

```
    else{
        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;
        build(lc, l, m);
        build(rc, m + 1, r);
        seg[p] = seg[lc] + seg[rc];
    }
}

// Call using update(1, 0, N - 1, idx, value)
void update(int p, int l, int r, int idx, int x){
    if(l == r) seg[p] = x;
    else{
        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;
        if(idx <= m) update(lc, l, m, idx, x);
        else update(rc, m + 1, r, idx, x);
        seg[p] = seg[lc] + seg[rc];
    }
}

// Call using query(1, 0, N - 1, ql, qr)
int query(int p, int l, int r, int ql, int qr){
    if(ql <= l && r <= qr) return seg[p];
    int m = (l+r)/2;
    int lc = 2*p;
    int rc = lc + 1;
    if(qr <= m) return query(lc, l, m, ql, qr);
    else if(ql > m) return query(rc, m + 1, r, ql, qr);
    else return query(lc, l, m, ql, qr) + query(rc, m + 1, r, ql, qr);
}

// Range update [L,R]. It only works if the result converges.
// If there is a common value for the whole range [L,R] use Lazy
    Propagation.
void updateRange(int p, int l, int r, int ql, int qr){
    if(l == r){
        seg[p] = value; // This value will vary according to each position
            of the "v" array
    }
    else{
        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;
        if(qr <= m) updateRange(lc, l, m, ql, qr);
        else if(ql > m) updateRange(rc, m + 1, r, ql, qr);
        else{
            updateRange(lc, l, m, ql, qr);
            updateRange(rc, m + 1, r, ql, qr);
        }
        seg[p] = seg[lc] + seg[rc];
    }
}

/*

Time Complexity:

build       -> O(n)
update      -> O(logn)
query       -> O(logn)
updateRange -> O(n)

Links:

https://cp-algorithms.com/data_structures/segment_tree.html

*/
```

1.2 SegTree with Lazy Propagation

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
vector<int> v;

// Implementation of a Recursive Segment Tree with Lazy Propagation (Sum
    Operation)
// RANGE UPDATE [l,r], RANGE QUERY [l,r]
class segTree{
    vector<int> seg, lazy;

public:
    segTree(int n){ // n = v.size()
        seg.assign(4*n, 0);
        lazy.assign(4*n, 0);
        build(1, 0, n - 1);
    }

    // Call using update(1, 0, n - 1, ql, qr, x)
    void update(int p, int l, int r, int ql, int qr, int x){
        propagate(p, l, r);
        if(r < ql || l > qr) return;
        if(ql <= l && r <= qr){
            lazy[p] = x;
            propagate(p,l,r);
        }
        else{
            int m = (l + r)/2;
            int lc = 2*p;
            int rc = lc + 1;
            update(lc, l, m, ql, qr, x);
            update(rc, m + 1, r, ql, qr, x);
            seg[p] = seg[lc] + seg[rc];
        }
    }
    // Call using query(1, 0, n - 1, ql, qr)
    int query(int p, int l, int r, int ql, int qr){
        propagate(p, l, r);
        if(r < ql || l > qr) return 0;
        if(ql <= l && r <= qr) return seg[p];
        int m = (l+r)/2;
        int lc = 2*p;
        int rc = lc + 1;
        return query(lc, l, m, ql, qr) + query(rc , m + 1, r, ql, qr);
    }

private:
    void build(int p, int l, int r){
        if(l == r){
            seg[p] = v[l];
            lazy[p] = 0;
        }
        else{
            int m = (l+r)/2;
            int lc = 2*p;
            int rc = lc + 1;
            build(lc, l, m);
            build(rc, m + 1, r);
            seg[p] = seg[lc] + seg[rc];
        }
    }
    void propagate(int p, int l, int r){
        if(lazy[p] == 0) return;
        seg[p] += (r - l + 1) * lazy[p];
        if(l != r){
            lazy[2*p] += lazy[p];
            lazy[2*p + 1] += lazy[p];
```

```
        }
        lazy[p] = 0;
    }
};

/*

Time Complexity:

build       -> O(n)
update      -> O(logn)
query       -> O(logn)

Links:

https://cp-algorithms.com/data_structures/segment_tree.html
https://www.youtube.com/watch?v=xuoQdt5pHj0
https://www.youtube.com/watch?v=UKH4Zgfa4kI
https://www.youtube.com/watch?v=3gPcs6PZPdk

*/
```

1.3 Sparse SegTree

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// In main we need to call create 2 times
// p == 0 -> NULL
// p == 1 -> Root

vector<int> answSeg, lNodes, rNodes;

// Returns the node index
int create(){
    answSeg.push_back(0);    // Must be a default value
    lNodes.push_back(0);     // The index "0" will always represent the null
        node
    rNodes.push_back(0);     // The index "0" will always represent the null
        node
    return answSeg.size() - 1;
}

void update(int p, int l, int r, int id, int val){
    if(id < l || id > r) return;
    if(l == r){
        answSeg[p] = val;
    }
    else{
        int m = (l + r) >> 1;
        if(id <= m){
            if(lNodes[p] == 0){
                int aux = create();
                lNodes[p] = aux;
            }
            update(lNodes[p], l, m, id, val);
        }
        else{
            if(rNodes[p] == 0){
                int aux = create();
                rNodes[p] = aux;
            }
            update(rNodes[p], m + 1, r, id, val);
        }
        answSeg[p] = answSeg[lNodes[p]] + answSeg[rNodes[p]];
    }
}

int query(int p, int l, int r, int ql, int qr){
    if(qr < l || ql > r) return 0;
    if(p == 0) return 0;
    if(l >= ql && r <= qr) return answSeg[p];
    int m = (l + r) >> 1;
    return query(lNodes[p], l, m, ql, qr) + query(rNodes[p], m + 1, r, ql,
        qr);
}

/*

Time Complexity

** n is the size of the range

query   -> O(logn)
update  -> O(logn)

Memory Complexity: O(Q * logn)

Links:

https://www.youtube.com/watch?v=omV7bTYdLHs&list=
    PLdyIeAAaboLsj9RpB4GJjAxIBXXnIrbGN&index=6

*/
```

1.4 Sparse SegTree with Lazy Propagation

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// In main we need to call create 2 times
// p == 0 -> NULL
// p == 1 -> Root

vector<int> answSeg, lazySeg, lNodes, rNodes;

// Returns the node index
int create(){
    answSeg.push_back(0);    // Must be a default value
    lazySeg.push_back(0);    // Must be a default value
    lNodes.push_back(0);     // The index "0" will always represent the null
        node
    rNodes.push_back(0);     // The index "0" will always represent the null
        node
    return answSeg.size() - 1;
}

void propagate(int p, int l, int r){
    if(p == 0 || lazySeg[p] == 0) return;
    answSeg[p] += (r - l + 1) * lazySeg[p];
    if(l != r){
        if(lNodes[p] == 0){
            int aux = create();
            lNodes[p] = aux;
        }
        if(rNodes[p] == 0){
            int aux = create();
            rNodes[p] = aux;
        }
        lazySeg[lNodes[p]] += lazySeg[p];
        lazySeg[rNodes[p]] += lazySeg[p];
    }
    lazySeg[p] = 0;
}

void update(int p, int l, int r, int ql, int qr, int val){
    propagate(p, l, r);
    if(l > qr || r < ql) return;
```

```
    if(ql <= l && r <= qr){
        lazySeg[p] = val;
        propagate(p, l, r);
    }
    else{
        int m = (l + r) >> 1;
        if(lNodes[p] == 0){
            int aux = create();
            lNodes[p] = aux;
        }
        if(rNodes[p] == 0){
            int aux = create();
            rNodes[p] = aux;
        }
        update(lNodes[p], l, m, ql, qr, val);
        update(rNodes[p], m + 1, r, ql, qr, val);
        answSeg[p] = answSeg[lNodes[p]] + answSeg[rNodes[p]];
    }
}

int query(int p, int l, int r, int ql, int qr){
    propagate(p, l, r);
    if(l > qr || r < ql) return 0;
    if(p == 0) return 0;
    if(l >= ql && r <= qr) return answSeg[p];
    int m = (l + r) >> 1;
    return query(lNodes[p], l, m, ql, qr) + query(rNodes[p], m + 1, r, ql,
        qr);
}

/*

Time Complexity

** n is the size of the range

query   -> O(logn)
update  -> O(logn)

Memory Complexity: O(Q * logn)

Links:

https://www.youtube.com/watch?v=omV7bTYdLHs&list=
    PLdyIeAAaboLsj9RpB4GJjAxIBXXnIrbGN&index=6

*/
```

1.5 Persistent SegTree

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// In main we need to call create 2 times
// p == 0 -> NULL
// p == 1 -> First version root

vector<int> roots, answSeg, lNodes, rNodes;

// On every update we need to save the root of the new version of the Seg
// roots[i] = update(roots[i - 1], -MAXN, MAXN, id, val)

// Returns the node index
int create(){
    answSeg.push_back(0);    // Must be a default value
    lNodes.push_back(0);     // The index "0" will always represent the null
        node
    rNodes.push_back(0);     // The index "0" will always represent the null
        node
    return answSeg.size() - 1;
}

// The update function needs to return the id of the newly created node.
int update(int p, int l, int r, int id, int val){
    int novo = create();
    answSeg[novo] = answSeg[p];
    lNodes[novo] = lNodes[p];
    rNodes[novo] = rNodes[p];

    if(l == r){
        answSeg[novo] = val;
        return novo;
    }
    int m = (l + r) >> 1;
    if(id <= m){
        int aux = update(lNodes[novo], l, m, id, val);
        lNodes[novo] = aux;
    }
    else{
        int aux = update(rNodes[novo], m + 1, r, id, val);
        rNodes[novo] = aux;
    }
    answSeg[novo] = answSeg[lNodes[novo]] + answSeg[rNodes[novo]];
    return novo;
}

int query(int p, int l, int r, int ql, int qr){
    if(qr < l || ql > r) return 0;
    if(p == 0) return 0;
    if(l >= ql && r <= qr) return answSeg[p];
    int m = (l + r) >> 1;
    return query(lNodes[p], l, m, ql, qr) + query(rNodes[p], m + 1, r, ql,
        qr);
}

/*

Time Complexity

** n is the size of the range

query   -> O(logn)
update  -> O(logn)

Memory Complexity: O(Q * logn)

Links:

https://www.youtube.com/watch?v=7_xncNind2w&list=
    PLdyIeAAaboLsj9RpB4GJjAxIBXXnIrbGN&index=8

*/
```

1.6 SegTree Beats

```
// Code based on brunomaletta's git
#include <bits/stdc++.h>
using namespace std;

#define int long long
const int INF = 0x3f3f3f3f3f3f3f3f;

struct node{
    int sum, lazy, sz; // Lazy just for sum
    int mx1, mx2, cntMx1;
    int mn1, mn2, cntMn1;
    node(int x = 0){
        sum = mx1 = mn1 = x;
        mx2 = -INF;
        mn2 = INF;
```

```
        cntMx1 = cntMn1 = sz = 1;
        lazy = 0;
    }
    node(const node & l, const node & r) {
        sum = l.sum + r.sum, sz = l.sz + r.sz;
        lazy = 0;
        if(l.mn1 > r.mn1){ // Min
            mn1 = r.mn1;
            cntMn1 = r.cntMn1;
            mn2 = min(l.mn1, r.mn2);
        } else if(l.mn1 < r.mn1){
            mn1 = l.mn1;
            cntMn1 = l.cntMn1;
            mn2 = min(r.mn1, l.mn2);
        } else{
            mn1 = l.mn1;
            cntMn1 = l.cntMn1 + r.cntMn1;
            mn2 = min(l.mn2, r.mn2);
        }
        if(l.mx1 < r.mx1){ // Max
            mx1 = r.mx1;
            cntMx1 = r.cntMx1;
            mx2 = max(l.mx1, r.mx2);
        } else if(l.mx1 > r.mx1){
            mx1 = l.mx1;
            cntMx1 = l.cntMx1;
            mx2 = max(r.mx1, l.mx2);
        } else{
            mx1 = l.mx1;
            cntMx1 = l.cntMx1 + r.cntMx1;
            mx2 = max(l.mx2, r.mx2);
        }
    }
    void setMin(int x){
        if(x >= mx1) return;
        sum += (x - mx1) * cntMx1;
        if(mn1 == mx1) mn1 = x;
        if(mn2 == mx1) mn2 = x;
        mx1 = x;
    }
    void setMax(int x){
        if(x <= mn1) return;
        sum += (x - mn1) * cntMn1;
        if(mx1 == mn1) mx1 = x;
        if(mx2 == mn1) mx2 = x;
        mn1 = x;
    }
    void setSum(int x){
        mx1 += x, mx2 += x;
        mn1 += x, mn2 += x;
        sum += x * sz;
        lazy += x;
    }
};

const int ms = 1e6;
node seg[4 * ms];
vector<int> v;

void build(int p, int l, int r){
    if(l == r) seg[p] = node(v[l]);
    else{
        int m = (l + r) >> 1;
        int lc = 2*p;
        int rc = lc + 1;
        build(lc, l, m);
        build(rc, m + 1, r);
        seg[p] = node(seg[lc], seg[rc]);
    }
}

void propagate(int p, int l, int r){
    if(l == r) return;
    for(int k = 0; k < 2; k++){
        if(seg[p].lazy) seg[2*p + k].setSum(seg[p].lazy);
        seg[2*p + k].setMin(seg[p].mx1);
        seg[2*p + k].setMax(seg[p].mn1);
    }
    seg[p].lazy = 0;
}

// v[ql..qr] -> v[i] = min(v[i], x)
void updateMin(int p, int l, int r, int ql, int qr, int x){
    if(l > qr || r < ql || seg[p].mx1 <= x) return;
    if(l >= ql && r <= qr && seg[p].mx2 < x){
        seg[p].setMin(x); return;
    }
    propagate(p, l, r);
    int m = (l + r) >> 1;
    int lc = 2*p;
    int rc = lc + 1;
    updateMin(lc, l, m, ql, qr, x);
    updateMin(rc, m + 1, r, ql, qr, x);
    seg[p] = node(seg[lc], seg[rc]);
}

// v[ql..qr] -> v[i] = max(v[i], x)
void updateMax(int p, int l, int r, int ql, int qr, int x){
    if(l > qr || r < ql || seg[p].mn1 >= x) return;
    if(l >= ql && r <= qr && seg[p].mn2 > x){
        seg[p].setMax(x); return;
    }
    propagate(p, l, r);
    int m = (l + r) >> 1;
    int lc = 2*p;
    int rc = lc + 1;
    updateMax(lc, l, m, ql, qr, x);
    updateMax(rc, m + 1, r, ql, qr, x);
    seg[p] = node(seg[lc], seg[rc]);
}

// v[ql..qr] -> v[i] += x
void updateSum(int p, int l, int r, int ql, int qr, int x){
    if(l > qr || r < ql) return;
    if(l >= ql && r <= qr){
        seg[p].setSum(x); return;
    }
    propagate(p, l, r);
    int m = (l + r) >> 1;
    int lc = 2*p;
    int rc = lc + 1;
    updateSum(lc, l, m, ql, qr, x);
    updateSum(rc, m + 1, r, ql, qr, x);
    seg[p] = node(seg[lc], seg[rc]);
}

//{ { min(v[ql..qr]), max(v[ql..qr]) }, sum(v[ql..qr]) }
pair<pair<int,int>, int> query(int p, int l, int r, int ql, int qr){
    if(l > qr || r < ql) return {{INF, -INF}, 0};
    if(l >= ql && r <= qr) return {{seg[p].mn1, seg[p].mx1}, seg[p].sum};
    propagate(p, l, r);
    int m = (l + r) >> 1;
    int lc = 2*p;
    int rc = lc + 1;
    auto L = query(lc, l, m, ql, qr), R = query(rc, m + 1, r, ql, qr);
    return { {min(L.first.first, R.first.first),
              max(L.first.second, R.first.second)},
              L.second + R.second };
}

/*

Time Complexity:

build   -> O(n)
query   -> O(logn)
update  -> O(logn) (using "updateSum" will be log^2(n) amortized)
```

```
Links:

https://codeforces.com/blog/entry/57319
https://www.youtube.com/watch?v=wFqKgrW1IMQ

*/
```

1.7 BIT (Fennwick Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Implementation of a Binary Indexed Tree (Fennwick Tree) (1-Based) (Sum
    Operation)
vector<int> bit;

// Array used to construct BIT (0-Based)
vector<int> v;

// Initialize Fennwick Tree
void build(){
    // Initialize all values with 0 (1-based)
    bit = vector<int>(v.size()+1, 0);

    // Putting the values of "v" in bit
    for(int i = 0; i < v.size(); i++) bit[i+1] = v[i];

    // Updating the values
    for(int i = 1; i < bit.size(); i++){
        int idx = i + (i & (-i));
        if(idx < bit.size()) bit[idx] += bit[i];
    }
}

// Return the sum of [0,idx] in "v"
int prefix_query(int idx){
    int result = 0;
    for(++idx; idx > 0; idx -= idx & -idx) result += bit[idx];
    return result;
}

// Computes the range sum between two indices (both inclusive) [l,r] in "v"
int range_query(int l, int r){
    if (l == 0) return prefix_query(r);
    else return prefix_query(r) - prefix_query(l - 1);
}

// Update bit adding "add" (idx represent the position in "v")
void update(int idx, int add) {
    for (++idx; idx < bit.size(); idx += idx & -idx) bit[idx] += add;
}

/*

Time Complexity:

build           -> O(n)
prefix_query    -> O(logn)
range_query     -> O(logn)
update          -> O(logn)

Links:

https://www.youtube.com/watch?v=uSFzHCZ4E-8
https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/
https://cp-algorithms.com/data_structures/fenwick.html
https://www.youtube.com/watch?v=v_wj_mOAlig
https://www.youtube.com/watch?v=CWDQJGaN1gY

*/
```

1.8 BIT 2D (Fennwick Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Implementation of a Fennwick Tree 2D (1-Based) (Sum Operation)
vector<vector<int>> bit;
// Matrix used to construct BIT (0-Based)
vector<vector<int>> v;

void build(){ // Initialize Fennwick Tree 2D
    bit.assign(v.size()+1, vector<int>(v.size()+1, 0)); // Bit[N+1][N+1]

    for(int i = 0; i < v.size(); i++)
        for(int j = 0; j < v[i].size(); j++)
            bit[i+1][j+1] = v[i][j];

    for(int i = 1; i < bit.size(); i++){
        for(int j = 1; j < bit[i].size(); j++){
            int idx_i = i + (i & (-i));
            int idx_j = j + (j & (-j));
            if(idx_i < bit.size() && idx_j < bit[i].size())
                bit[idx_i][idx_j] += bit[i][j];
        }
    }
}

// Returns the sum of [0,0] to [i,j]
int query(int i, int x){
    int result = 0;
    for(++i; i > 0; i -= i & -i)
        for(int j = x+1; j > 0; j -= j & -j)
            result += bit[i][j];
    return result;
}

// Returns the sum of (i_1,j_1) to (i_2,j_2) (both inclusive)
int range_query(int i_1, int j_1, int i_2, int j_2){
    return query(i_2,j_2) - query(i_1-1,j_2) - query(i_2,j_1-1) + query(i_1
        -1,j_1-1);
}

// Update bit adding "add" to position v[i][x]
void update(int i, int x, int add){
    for(++i; i < bit.size(); i += i & -i)
        for(int j = x+1; j < bit[i].size(); j += j & -j)
            bit[i][j] += add;
}

/*

Time Complexity:

build       -> O(n^2)
query       -> O((logn)^2)
range_query -> O((logn)^2)
update      -> O((logn)^2)

Links:

https://cp-algorithms.com/data_structures/fenwick.html#finding-minimum-of
    -0-r-in-one-dimensional-array
```

```
*/
```

1.9 Sparse Table

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

struct SparseTable{
    int n, LOG;
    vector<vector<int>> st;
    vector<int> lg2;
    int type;

    SparseTable(vector<int>& v, int type) : type(type) {
        n = v.size();
        LOG = 25;
        st.assign(LOG, vector<int>(n, 0));
        lg2.assign(n + 1, 0);

        for(int i = 0; i < n; i++) st[0][i] = v[i];
        for(int i = 2; i <= n; i++) lg2[i] = lg2[i/2] + 1;

        for(int i = 1; i < LOG; i++){
            for(int j = 0; j + (1 << i) - 1 < n; j++){
                st[i][j] = func(st[i - 1][j], st[i - 1][j + (1 << (i - 1))
                    ]);
            }
        }
    }

    int func(int x, int y){
        if(type == 0) return x + y;
        else if(type == 1) return min(x, y);
        else return max(x, y);
    }

    int query(int l, int r){
        if(type == 0){
            int answ = 0;
            for(int i = LOG - 1; i >= 0; i--){
                if((1 << i) <= r - l + 1){
                    answ += st[i][l];
                    l += (1 << i);
                }
            }
            return answ;
        }
        else{
            int i = lg2[r - l + 1];
            return func(st[i][l], st[i][r - (1 << i) + 1]);
        }
    }

    void update(int idx, int val){
        st[0][idx] = val;
        int k = 1;
        for(int i = 1; i < LOG; i++){
            for(int j = max(0LL, idx - 2*k + 1); (j <= idx) && (j + (1 << i
                ) - 1 < n); j++){
                st[i][j] = func(st[i - 1][j], st[i - 1][j + k]);
            }
            k *= 2;
        }
    }
};

/*
Time Complexity:

Build       -> O(n*logn)
QuerySum    -> O(logn)
QueryMin    -> O(1)
QueryMax    -> O(1)
QueryGCD    -> O(1)
Update      -> We don't use!

Links:

https://cp-algorithms.com/data_structures/sparse-table.html
https://www.youtube.com/watch?v=0jWeUdxrGm4

*/
```

1.10 Treap

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

typedef struct Node * pNode;
struct Node{
    int key, priority, sz;
    bool rev;
    pNode lc, rc;
    Node(){}
    Node(int x){
        key = x;
        priority = uniform_int_distribution<int>()(rng);
        sz = 1;
        rev = 0;
        lc = rc = NULL;
    }
};

int size(pNode it){ return it ? it->sz : 0; }

void upd_sz(pNode it){
    if(it) it->sz = size(it->lc) + size(it->rc) + 1;
}

void pushLazy(pNode it){
    if(it && it->rev){
        it->rev = false;
        swap(it->lc, it->rc);
        if(it->lc) it->lc->rev ^= 1;
        if(it->rc) it->rc->rev ^= 1;
    }
}

// Split Treap "t" in t1 = l and t2 = r by value "key"
void split(pNode t, pNode & l, pNode & r, int key){
    if(!t) return void(l = r = 0);
    pushLazy(t);
    int cur_key = size(t->lc); // cur_key = t->key if not implicit
    if(key <= cur_key){
        split(t->lc, l, t->lc, key);
        r = t;
    }
    else{
        split(t->rc, t->rc, r, key - (1 + size(t->lc))); // key if not
            implicit
        l = t;
    }
    upd_sz(t);
}
```

```cpp
// Merge treap "l" and "r" in "t".
// It works under the assumption that "l" and "r"
// are ordered (all keys in "l" are smaller than keys in "r")
void merge(pNode & t, pNode l, pNode r){
    pushLazy(l), pushLazy(r);
    if(!l || !r) t = (l ? l : r);
    else if(l->priority > r->priority){
        merge(l->rc, l->rc, r);
        t = l;
    }
    else{
        merge(r->lc, l, r->lc);
        t = r;
    }
    upd_sz(t);
}

// Only works when not an Implicit Treap
// To insert in a Implicit Treap use split + merge
void insert(pNode & t, pNode it){
    if(!t) t = it;
    else if(it->priority > t->priority){
        split(t, it->lc, it->rc, it->key);
        t = it;
    }
    else{
        insert(t->key <= it->key ? t->rc : t->lc, it);
    }
}

// Needs to change when using Implicit Treap
void erase(pNode & t, int key){
    if(t->key == key){
        pNode th = t;
        merge(t, t->lc, t->rc);
        delete th;
    }
    else{
        erase(key < t->key ? t->lc : t->rc, key);
    }
}

void reverse(pNode t, int l, int r){
    pNode t1, t2, t3;
    split(t, t1, t2, l);
    split(t2, t2, t3, r - l + 1);
    t2->rev ^= 1;
    merge(t, t1, t2);
    merge(t, t, t3);
}

void unite(pNode & t, pNode l, pNode r){
    if(!l || !r) return void( t = l ? l : r );
    if(l->priority < r->priority) swap(l, r);
    pNode lt, rt;
    split(r, lt, rt, l->key);
    unite(l->lc, l->lc, lt);
    unite(l->rc, l->rc, rt);
    t = l;
}

pNode kth_element(pNode t, int k) {
    if(!t) return NULL;
    if(t->lc) {
        if(t->lc->sz >= k) return kth_element(t->lc, k);
        else k -= t->lc->sz;
    }
    return (k == 1) ? t : kth_element(t->rc, k - 1);
}

// Show an array that corresponds to the current state of the implicit
    treap
void debugTreap(pNode t) {
    if(!t) return;
    pushLazy(t);
    debugTreap(t->lc);
    cout << t->key << " ";
    debugTreap(t->rc);
}

/*

Time Complexity:

Split   -> O(logn)
Merge   -> O(logn)
unite   -> O(m*log(n/m))

Links:

https://cp-algorithms.com/data_structures/treap.html
https://usaco.guide/adv/treaps?lang=cpp
https://www.youtube.com/watch?v=6x0UlIBLRsc
https://www.youtube.com/watch?v=erKlLEXLKyY

*/
```

1.11 DSU (Disjoint Set Union)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int ms = 1e6;
int par[ms], sz[ms];

void build(){
    for(int i = 0; i < ms; i++){
        par[i] = i;
        sz[i] = 1;
    }
}

// Path Compression Optimization
int find(int u){ return u == par[u] ? u : par[u] = find(par[u]); }

// Union by Size Optimization
void merge(int a, int b){
    a = find(a), b = find(b);
    if(a == b) return;
    if(sz[a] < sz[b]) swap(a, b);
    par[b] = a;
    sz[a] += sz[b];
}

/*

Time Complexity:

build   -> O(N)
find    -> O(logN)  (Average case is O(1))
merge   -> O(logN)  (Average case is O(1))

Links:

https://cp-algorithms.com/data_structures/disjoint_set_union.html
https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/

*/
```

1.12 DSU with Rollback

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ii = pair<int,int>;

struct Dsu{
    vector<int> parent, sz;
    vector<ii> changes;
    int comps;

    Dsu(int n){
        comps = n;
        parent.assign(comps, 0);
        iota(parent.begin(), parent.end(), 0);
        sz.assign(comps, 1);
    }

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

    void merge(int a, int b){
        a = find(a);
        b = find(b);
        if(a == b){
            changes.push_back({-1, -1});
            return;
        }
        if(sz[a] < sz[b]) swap(a, b);
        parent[b] = a;
        sz[a] += sz[b];
        changes.push_back({a, b});
        comps--;
    }

    void rollback(){
        auto [a, b] = changes.back();
        changes.pop_back();
        if(a == -1) return;
        sz[a] -= sz[b];
        parent[b] = b;
        comps++;
    }
};

/*

Time Complexity:

find        -> O(logn)
merge       -> O(logn)
rollback    -> O(1)

Links:

https://www.youtube.com/watch?v=AW_iea9xpFw

*/
```

1.13 Min Queue

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

struct MinQueue{
    queue<int> q;
    deque<int> mins;

    void push(int x){
        q.push(x);
        while(!mins.empty() && mins.back() > x){
            mins.pop_back();
        }
        mins.push_back(x);
    }

    void pop(){
        if(q.empty()) return;
        int x = q.front();
        q.pop();
        if(mins[0] == x) mins.pop_front();
    }

    int min(){
        return mins[0];
    }
};

/*

Time Complexity

push    -> O(1)
pop     -> O(1)
min     -> O(1)

Links:

https://cp-algorithms.com/data_structures/stack_queue_modification.html

*/
```

1.14 Mo

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int MAXN = 1e6;
const int block = sqrt(MAXN);  // Block size

struct Query{
    int l, r, idx;
    Query(int l, int r, int idx) : l(l), r(r), idx(idx) {}
    bool operator < (Query &other){
        if(l / block != other.l / block) return (l / block) < (other.l /
            block);
        return (l / block & 1) ? (r < other.r) : (r > other.r);
    }
};

vector<Query> queries;
int v[MAXN], answ[MAXN], curr_answ;

// Need to change the current answer according to the problem
void add(int x){}
void remove(int x){}

// Mo's Algorithm to answer offline queries
void Mo(){
    int curr_l = 0, curr_r = -1;
    sort(queries.begin(), queries.end());
```

```
    for(Query q : queries){
        while(curr_l > q.l){
            curr_l--;
            add(curr_l);
        }
        while(curr_r < q.r){
            curr_r++;
            add(curr_r);
        }
        while(curr_l < q.l){
            remove(curr_l);
            curr_l++;
        }
        while(curr_r > q.r){
            remove(curr_r);
            curr_r--;
        }
        answ[q.idx] = curr_answ;
    }
}

/*

Time Complexity

Mo's Algorithm  -> O( (N + Q) * sqrt(N) * F ) Where F is the complexity of
     add and remove function.

Links:

https://cp-algorithms.com/data_structures/sqrt_decomposition.html
https://www.youtube.com/watch?v=RENzmNgIZ4A&t=1s

*/
```

1.15 Mo (Hilbert)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int MAXN = 1e6;
const int block = sqrt(MAXN);  // Block size

// Ordering based on the Hilbert curve
inline int64_t hilbertOrder(int x, int y, int pow, int rotate){
    if(pow == 0) return 0;
    int hpow = 1 << (pow - 1);
    int seg = (x < hpow) ? ( (y < hpow) ? 0 : 3) : ( (y < hpow) ? 1 : 2);
    seg = (seg + rotate) & 3;
    const int rotateDelta[4] = {3, 0, 0, 1};
    int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
    int nrot = (rotate + rotateDelta[seg]) & 3;
    int64_t subSquareSize = int64_t(1) << (2*pow - 2);
    int64_t ans = seg * subSquareSize;
    int64_t add = hilbertOrder(nx, ny, pow - 1, nrot);
    ans += (seg == 1 || seg == 2) ? add : (subSquareSize - add - 1);
    return ans;
}

struct Query{
    int l, r, idx;
    int64_t ord;
    Query(int l, int r, int idx) : l(l), r(r), idx(idx) {
        ord = hilbertOrder(l, r, 21, 0);
    }
    bool operator < (Query &other){
        return ord < other.ord;
    }
};

vector<Query> queries;
int v[MAXN], answ[MAXN], curr_answ;

// Need to change the current answer according to the problem
void add(int x){}
void remove(int x){}

// Mo's Algorithm to answer offline queries
void Mo(){
    int curr_l = 0, curr_r = -1;
    sort(queries.begin(), queries.end());
    for(Query q : queries){
        while(curr_l > q.l){
            curr_l--;
            add(curr_l);
        }
        while(curr_r < q.r){
            curr_r++;
            add(curr_r);
        }
        while(curr_l < q.l){
            remove(curr_l);
            curr_l++;
        }
        while(curr_r > q.r){
            remove(curr_r);
            curr_r--;
        }
        answ[q.idx] = curr_answ;
    }
}

/*

Time Complexity

Mo's using Hilbert  -> O( N * sqrt(Q) )

Links:

https://cp-algorithms.com/data_structures/sqrt_decomposition.html
https://www.youtube.com/watch?v=RENzmNgIZ4A&t=1s
https://codeforces.com/blog/entry/61203

*/
```

1.16 Policy Based Struct

```
#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;

typedef tree<int, null_type, less<int>, rb_tree_tag,
        tree_order_statistics_node_update> indexed_set;

indexed_set s;

// Inserts "X" in set
s.insert(X);

// Deletes "X" in set
s.erase(X);

// Returns an iterator for the element at position "pos" (0-based)
s.find_by_order(pos);
```

```
// Returns the position of element "X", another function is to count the
     amount of elements strictly smaller than "X"
s.order_of_key(X);

/*

Time Complexity

All operations run on O(logn)

Note: When the element does not exist in the indexed_set, the "order_of_key
     ()" function returns the position it SHOULD BE,
if it existed, so it is useful to calculate the amount of elements strictly
      smaller than "X".

Links:

https://codeforces.com/blog/entry/11080
https://www.geeksforgeeks.org/policy-based-data-structures-g/

*/
```

1.17 Multi Indexed 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;

#define int long long // This line needs to be after imports

template<typename T>
class multi_indexed_set{
    tree<pair<T, int>, null_type, less<pair<T, int>>, rb_tree_tag,
            tree_order_statistics_node_update> miset;
    unordered_map<T, int> freq;

    public:
        void insert(T x){
            freq[x]++;
            miset.insert({x, freq[x]});
        }
        void erase(T x){
            if(!freq[x]) return;
            miset.erase({x, freq[x]});
            freq[x]--;
        }
        int order_of_key(T x){ return miset.order_of_key({x, 0}); }
        int count(T x) { return freq[x]; }
        int size(){ return miset.size(); }

};
multi_indexed_set<int> ms;

// Inserts "X" in multiset
ms.insert(X);

// Deletes "X" in multiset
ms.erase(X);

// Returns the position of element "X", another function is to count the
     amount of elements strictly smaller than "X"
ms.order_of_key(X);

// Returns the number of elements equal to "X"
ms.count(X);

// Returns the size of the multiset
ms.size();

/*

Time Complexity:

ms.insert(X)         -> O(logn)
ms.erase(X)          -> O(logn)
ms.order_of_key(X); -> O(logn)
ms.count(X)          -> O(1) (Average)
ms.size()            -> O(1)

Note: We can implement the "find_by_order" function

*/
```

2 Graph Algorithms

2.1 Dinic - Max Flow

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Dinic's Algorithm to find the Max Flow of a graph
class Dinic{
    int N;
    vector<int> level;
    vector<bool> dead;
    const int INF = 1e18;

public:
    struct Edge{
        Edge(int vertice, int capacity){
            v = vertice;
            cap = capacity;
        }
        int v;
        int cap;
    };
    int source, sink;
    vector<Edge> edge;
    vector<vector<int>> g;

    Dinic(int size, int u, int v){  // Initializing Dinic
        N = size;
        g.resize(N);
        level.resize(N);
        source = u;
        sink = v;
    }

    void addEdge(int u, int v, int cap){ // Making Residual Network
        g[u].push_back(edge.size());
        edge.push_back(Edge(v,cap));
        g[v].push_back(edge.size());
        edge.push_back(Edge(u,0));
    }

    int run(){
        int flow = 0;
        while(BFS())
            flow += maxflow(source, INF);

        return flow;
    }
```

```
    // Algorithm to find the edges from MinCut by dividing the graph into 2
         subgraphs.
    void mincut(){

        set<int> reachable; // 2 subgraphs
        set<int> unreachable;

        vector<bool> visited(N, false);
        queue<int> q;
        q.push(source);
        reachable.insert(source);

        while(!q.empty()){  // BFS to find all vertices that are reached by
             the source using positive-capacity edges

            int u = q.front();
            visited[u] = true;
            q.pop();

            for(auto x: g[u]){
                if(!visited[edge[x].v] && edge[x].cap > 0){
                    q.push(edge[x].v);
                    reachable.insert(edge[x].v);
                }
            }
        }

        for(int i = 1; i < N; i++){ // If the graph is 0-based you need to
             put i = 0
            if(reachable.find(i) == reachable.end())
                unreachable.insert(i);
        }

        bool flag = true;
        for(auto i: reachable){ // Printing edges responsible for
             connecting the 2 subgraphs
            for(auto j: g[i]){
                if(unreachable.find(edge[j].v) != unreachable.end()){
                    if(flag == true)cout << i << " " << edge[j].v << '\n';
                    flag = flag^true;
                }
            }
        }
    }

    // Algorithm to find edges from Maximum Matching of a bipartite graph (
         may have variations which capacity need not be 0)
    void max_matching(int n, int m){

        for(int i = 1; i <= n + m; i++){
            for(auto j: g[i]){
                if(j%2 == 0 && edge[j].cap == 0){
                    if(edge[j].v != 0 && edge[j].v != n+m+1)
                        cout << i << " " << edge[j].v - n << '\n';
                }
            }
        }
    }

private:
    bool BFS(){ // Construct the Augmenting Level Path
        level.assign(N, INF);
        dead.assign(N, false);
        level[source] = 0;
        queue<int> q;

        q.push(source);
        while(!q.empty()){

            int u = q.front();
            q.pop();

            if(u == sink) return true;

            for(auto x: g[u]){
                if(level[edge[x].v] == INF && edge[x].cap > 0){
                    level[edge[x].v] = level[u] + 1;
                    q.push(edge[x].v);
                }
            }
        }
        return false;
    }

    int maxflow(int u, int flow){
        if(dead[u] || flow == 0) return 0;
        if(u == sink) return flow;

        int answ = 0;
        for(auto i: g[u]){
            if(level[edge[i].v] != level[u] + 1) continue;
            int f = maxflow(edge[i].v, min(edge[i].cap, flow));
            int reversed_i = (i % 2 == 0 ? i + 1 : i - 1);    // Finding
                 the "even" edge of "i"
            flow -= f;
            answ += f;
            edge[i].cap -= f;
            edge[reversed_i].cap += f;
        }

        if(answ == 0) dead[u] = true;
        return answ;
    }
};

/*

Time Complexity

Dinic                                -> O(V^2 * E)
Dinic Maximum Matching               -> O(E * sqrt(V))
Dinic in Unit Capacities Networks    -> O(E * V^(2/3))

Links:

https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-
    problem/
https://www.youtube.com/watch?v=oHy3ddI9X3o
https://cp-algorithms.com/graph/edmonds_karp.html
https://cp-algorithms.com/graph/dinic.html
https://www.youtube.com/watch?v=M6cm8UeeziI
https://www.youtube.com/watch?v=duKIzgJQ1w8

*/
```

2.2 Min Cost Max Flow

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ii = pair<int,int>;
using ll = long long;
const int INF = 1e18;

// Implementation of the Min Cost Flow algorithm
class MinCostFlow{
private:
    struct Edge{
        int u,v,cap,cost,flow;
        Edge(int from, int to, int cap, int cost):
            u(from), v(to), cap(cap), cost(cost), flow(0) {}
    };
    int n, source, sink;
```

```
    vector<Edge> edge;
    vector<vector<int>> g;
    vector<int> dist;
    vector<int> parent;

public:
    MinCostFlow(int n, int source, int sink){ // Initializing
        this->n = n;
        this->source = source;
        this->sink = sink;
        g.assign(n, vector<int>());
    }

    void addEdge(int u, int v, int cap, int cost){
        int m = edge.size();
        edge.emplace_back(u, v, cap, cost);
        g[u].push_back(m);
        edge.emplace_back(v, u, 0, -cost);
        g[v].push_back(m+1);
    }

    ii run(int k){  // Running Min Cost Flow, to calculate Min Cost Max
         Flow put k = INF
        ii answ = {0,0};     // {flow, cost}
        while(answ.first < k && spfa()){
            ii aux = get_flow_and_cost();   // {flow, cost}
            int max_add = min(aux.first, k - answ.first);

            answ.first += max_add;
            answ.second += max_add * aux.second;
        }
        return answ;
    }

private:
    bool spfa(){ // Shortest Path Faster Algorithm
        dist.assign(n, INF);
        parent.assign(n, -1);
        queue<int> q;
        vector<bool> inqueue(n, false);
        vector<int> count(n, 0);

        dist[source] = 0;
        q.push(source);
        inqueue[source] = true;

        while(!q.empty()){
            int u = q.front();
            q.pop();
            inqueue[u] = false;

            for(auto id: g[u]){
                Edge aux = edge[id];

                int new_dist = dist[u] + aux.cost;
                if(aux.cap - aux.flow > 0 && new_dist < dist[aux.v]){

                    parent[aux.v] = id;
                    dist[aux.v] = new_dist;

                    if(!inqueue[aux.v]){
                        q.push(aux.v);
                        inqueue[aux.v] = true;

                        count[aux.v]++;
                        if(count[aux.v] > n) return false; // Found a
                             negative cycle
                    }
                }
            }
        }
        return dist[sink] < INF;
    }

    ii get_flow_and_cost(){
        ii flow_cost = {INF,0};

        int v = sink;
        while(v != source){
            Edge aux = edge[parent[v]];
            flow_cost.first = min(flow_cost.first, aux.cap - aux.flow);
            flow_cost.second += aux.cost;
            v = aux.u;
        }

        v = sink;
        while(v != source){
            edge[parent[v]].flow += flow_cost.first;
            edge[parent[v] ^ 1].flow -= flow_cost.first;
            v = edge[parent[v]].u;
        }
        return flow_cost;
    }
};

/*

Time Complexity

MinCostFlow -> O(n^2*m^2)

Links:

https://cp-algorithms.com/graph/min_cost_flow.html#algorithm
https://cp-algorithms.com/graph/bellman_ford.html#shortest-path-faster-
     algorithm-spfa
https://www.youtube.com/watch?v=AtkEpr7dsW4
https://blog.thomasjungblut.com/graph/mincut/mincut/

*/
```

2.3 Min Cost Max Flow (Dijkstra)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ii = pair<int,int>;

// Implementation of the Min Cost Max Flow algorithm using Dijkstra
struct MCMF{
    struct Edge{
        int from, to, cap, cost, flow;
        Edge(int from, int to, int cap, int cost) :
        from(from), to(to), cap(cap), cost(cost), flow(0) {}
    };
    int n, src, sink;
    const int INF = 1e18;
    vector<Edge> edgeList;
    vector<vector<int>> g;
    vector<int> dist, pot, parent;

    MCMF(int n, int src, int sink){
        this->n = n;
        this->src = src;
        this->sink = sink;
        g.assign(n, vector<int>());
        dist.assign(n, INF);
        pot.assign(n, 0);
    }

    void addEdge(int from, int to, int cap, int cost){
```

```
    int m = edgeList.size();
    edgeList.push_back(Edge(from, to, cap, cost));
    g[from].push_back(m);
    edgeList.push_back(Edge(to, from, 0, -cost));
    g[to].push_back(m + 1);
}

// {flow, cost} To calculate Min Cost Max Flow put k = INF
ii run(int k){
    ii answ = {0, 0};
    // If there are negative weights in the initial graph
    // it is necessary to run spfa once to configure the potentials
    // spfa();
    fixPot();
    while(answ.first < k && dijkstra()){
        ii flow = augment();
        int max_add = min(flow.first, k - answ.first);
        answ.first += max_add;
        answ.second += max_add * flow.second;
        fixPot();
    }
    return answ;
}

// {flow, cost}
ii augment(){
    ii flow = {INF, 0};

    int v = sink;
    while(v != src){
        Edge aux = edgeList[parent[v]];
        flow.first = min(flow.first, aux.cap - aux.flow);
        flow.second += aux.cost;
        v = aux.from;
    }

    v = sink;
    while(v != src){
        edgeList[parent[v]].flow += flow.first;
        edgeList[parent[v] ^ 1].flow -= flow.first;
        v = edgeList[parent[v]].from;
    }
    return flow;
}

bool spfa(){
    dist.assign(n, INF);
    parent.assign(n, -1);
    queue<int> q;
    vector<bool> inqueue(n, false);
    vector<int> count(n, 0);
    dist[src] = 0;
    q.push(src);
    inqueue[src] = true;
    while(!q.empty()){
        int u = q.front();
        q.pop();
        inqueue[u] = false;
        for(auto id : g[u]){
            Edge aux = edgeList[id];
            if(aux.cap - aux.flow == 0) continue;
            int new_dist = dist[u] + aux.cost + pot[u] - pot[aux.to];
            if(new_dist < dist[aux.to]){
                parent[aux.to] = id;
                dist[aux.to] = new_dist;
                if(!inqueue[aux.to]){
                    q.push(aux.to);
                    inqueue[aux.to] = true;
                    count[aux.to]++;
                    if(count[aux.to] > n) return false; // Found a
                        negative cycle
                }
            }
        }
    }
    return dist[sink] < INF;
}

    bool dijkstra(){
        dist.assign(n, INF);
        parent.assign(n, -1);
        priority_queue<ii, vector<ii>, greater<ii>> pq;
        pq.push({0, src});
        dist[src] = 0;
        while(!pq.empty()){
            auto [d, u] = pq.top();
            pq.pop();
            if(dist[u] < d) continue;
            for(auto id : g[u]){
                Edge aux = edgeList[id];
                if(aux.cap - aux.flow == 0) continue;
                int new_dist = dist[u] + aux.cost + pot[u] - pot[aux.to];
                if(new_dist < dist[aux.to]){
                    dist[aux.to] = new_dist;
                    parent[aux.to] = id;
                    pq.push({new_dist, aux.to});
                }
            }
        }
        return dist[sink] < INF;
    }

    void fixPot(){
        for(int i = 0; i < n; i++){
            if(dist[i] < INF) pot[i] += dist[i];
        }
    }
};

/*

Time Complexity

spfa        -> O(n*m)
dijkstra    -> O(m*log(n))
MCMF        -> O(n*m + F*(m*log(n))) or O(F*(m*log(n)))
               F = Max Flow
               F = min(F, n*max_cost)
Links:

https://cp-algorithms.com/graph/min_cost_flow.html
https://codeforces.com/blog/entry/95823
https://codeforces.com/blog/entry/105658
https://www.topcoder.com/thrive/articles/Minimum%20Cost%20Flow%20Part%20Two
    :%20Algorithms
https://usaco.guide/adv/min-cost-flow?lang=cpp

*/
```

2.4 Stoer Wagner (Min Cut)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Graph implementation using matrix
vector<vector<int>> g;

// Implementation of the Stoer-Wagner algorithm to find the global min cut
    of a graph
```

```cpp
// We'll use this algorithm for problems where we don't know exactly where
    the source or sink is
int stoer_wagner(int n){
    int answ = LONG_LONG_MAX;
    for(int i = 1; i < n; i++){   // Need to repeat only n-1 times
        vector<int> weight(n, 0);
        vector<bool> visited(n, false);

        int prev = -1, v = 0, cnt = 1, current_cut = 0;
        while(cnt <= n-i){ // Creating the subset until only 1 node is
            missing
            visited[v] = true;
            int nxt = -1;
            current_cut = 0;

            for(int j = 0; j < n; j++){ // Looking for edge with the
                greatest weight connected to current subset
                weight[j] += g[v][j];
                if(!visited[j] && weight[j] > current_cut){
                    nxt = j;
                    current_cut = weight[j];
                }
            }

            cnt++;
            prev = v;
            v = nxt;
        } // At the end of the loop "v" is the disconnected node and "prev"
            was the last node that reached it

        answ = min(answ, current_cut);
        for(int j = 0; j < n; j++){ // Joining the last 2 nodes, putting
            the edges of "v" in "prev"
            if(j != prev){
                g[j][prev] += g[j][v];
                g[prev][j] += g[v][j];
            }
            g[j][v] = 0; // Emptying the weights to "v" because "v" will no
                longer exist
        }
    }
    return answ;
}

/*

Time Complexity

stoer_wagner    -> O(V^3) can be optimized to O(|V|*|E| + |V|^2 * log|V|)

Links:

https://www.youtube.com/watch?v=AtkEpr7dsW4
https://blog.thomasjungblut.com/graph/mincut/mincut/

*/
```

2.5 Hungarian (Assignment)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// "n" needs to be the smallest dimension
template<typename T = int>
struct Hungarian{
    vector<vector<T>> a; // Matrix
    vector<T> u, v; // Potentials
    vector<int> p, way;
    int n, m;
    T inf;

    Hungarian(vector<vector<T>> & mat){
        n = mat.size();
        m = mat[0].size();
        a.assign(n + 1, vector<T>(m + 1, 0));
        u.assign(n + 1, 0);
        v.assign(m + 1, 0);
        p.assign(m + 1, 0);
        way.assign(m + 1, 0);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                a[i + 1][j + 1] = mat[i][j];
        inf = numeric_limits<T>::max();
    }

    // Returns {min assignment, v}
    // v[i] = Chosen column of the i-th row
    pair<T, vector<int>> assignment(){
        for(int i = 1; i <= n; i++){
            p[0] = i;
            int j0 = 0;
            vector<int> minv(m + 1, inf);
            vector<bool> used(m + 1, false);
            do{
                used[j0] = true;
                int i0 = p[j0], j1 = -1;
                T delta = inf;
                for(int j = 1; 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[p[j]] += delta, v[j] -= delta;
                    else minv[j] -= delta;
                }
                j0 = j1;
            } while(p[j0] != 0);
            do{
                int j1 = way[j0];
                p[j0] = p[j1];
                j0 = j1;
            } while(j0);
        }
        vector<int> answ(n + 1); // answ[1.. n]
        for(int j = 1; j <= m; j++) answ[p[j]] = j; // column in 1-based
        return make_pair(-v[0], answ);
    }
};

/*

Time Complexity:

assignment  -> O(n^2 * m)

Links:

https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html
https://e--maxx-ru.translate.goog/algo/assignment_hungary?_x_tr_sch=http&
    _x_tr_sl=ru&_x_tr_tl=en&_x_tr_hl=ru
https://en.wikipedia.org/wiki/Hungarian_algorithm

*/
```

2.6 Articulation Points (Tarjan)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>

vector<vector<int>> g;
vector<bool> visited, cutpoint;
vector<int> discovered, low;
int timer;

void dfs(int v, int p = -1){
    visited[v] = true;
    discovered[v] = low[v] = timer;
    timer++;
    int children = 0;
    for(auto to : g[v]){
        if(to == p) continue;
        if(visited[to]) low[v] = min(low[v], discovered[to]);
        else{
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if(p != -1 && low[to] >= discovered[v]){ // Caution: Can be
                called multiple times for "v"
                cutpoint[v] = true;
            }
            children++;
        }
    }
    if(p == -1 && children > 1) cutpoint[v] = true;
}

void find_cutpoints(int n){
    timer = 0;
    visited.assign(n, false);
    cutpoint.assign(n, false);
    discovered.assign(n, -1);
    low.assign(n, -1);
    for(int i = 0; i < n; i++){
        if(!visited[i]) dfs(i);
    }
}

/*

Time Complexity:

find_cutpoints  -> O(V + E)

Links:

https://cp-algorithms.com/graph/cutpoints.html
https://www.youtube.com/watch?v=iYJqgMKYsdI

*/
```

2.7 Articulation Points (DFS Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>
const int INF = 0x3f3f3f3f3f3f3f3f;

vector<vector<int>> g;
vector<int> lvl, dp; // dp[u] = min lvl that "u" can reach using backedge
vector<bool> cutpoint;

void dfs(int u, int p = -1){
    int children = 0;
    for(auto v : g[u]){
        if(lvl[v] == 0){ // Edge to child
            lvl[v] = lvl[u] + 1;
            dfs(v, u);
            if(dp[v] < lvl[u]){
                dp[u] = min(dp[u], dp[v]);
            }
            else if(p != -1){ // this subtree has no backedge to some "u"
                ancestor
                cutpoint[u] = true;
            }
            children++;
        }
        else if(lvl[v] < lvl[u]){ // Edge upwards
            dp[u] = min(dp[u], lvl[v]);
        }
    }
    if(p == -1 && children > 1) cutpoint[u] = true;
}

void find_cutpoints(int n){
    lvl.assign(n, 0);
    dp.assign(n, INF);
    cutpoint.assign(n, false);
    for(int i = 0; i < n; i++){
        if(!lvl[i]){
            lvl[i] = 1;
            dfs(i);
        }
    }
}

/*

Time Complexity

find_cutpoints  -> O(V + E)

Links:

https://codeforces.com/blog/entry/68138
https://cp-algorithms.com/graph/cutpoints.html
https://www.youtube.com/watch?v=iYJqgMKYsdI

*/
```

2.8 Block-Cut Tree

```cpp
// Code based on the Brunomont's github
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>

/*
Block-Cut Tree: It's a tree that each BCC and articulation points
is a unique vertex. We have a edge (BCCx, arty) if arty is inside BCCx

A 2-Coloring is such that one color will be the blocks and the other the
    articulation points

2-vertex-connected components = Biconnected components = Blocks
```

```
Blocks = maximal subset of E such that the induced subgraph is 2-vertex-
    connected

art[i] = number of new connected components formed after
removing vertex "i". if art[i] >= 1, "i" It's an articulation.

for all i < blocks.size()
blocks[i] it's a BCC
edgBlocks[i] it's the blocks[i] edges
tree[i] it's a vertex from tree that corresponds to block[i]

pos[i] = The vertex that "i" corresponds in the Block Cut Tree

The Block Cut Tree will have at most 2*n vertices
*/

struct BlockCutTree{
    vector<vector<int>> g, blocks, tree;
    vector<vector<pii>> edgBlocks;
    stack<int> stV; // Store the vertices in DFS order
    stack<pii> stE; // Store the edges in DFS order
    vector<int> disc, low, art, pos;
    int n, timer;

    BlockCutTree(vector<vector<int>> g) : g(g){
        n = g.size();
        disc.assign(n, -1);
        low.assign(n, -1);
        art.assign(n, 0);
        pos.assign(n, -1);
        build();
    }

    void dfs(int u, int p = -1){
        disc[u] = low[u] = timer++;
        stV.push(u);

        if(p != -1) stE.push({u, p});
        for(int v : g[u]){ // Backedges
            if(v != p && disc[v] != -1) stE.push({u, v});
        }
        bool hasFwd = false;
        for(int v : g[u]){
            if(v == p) continue;
            if(disc[v] == -1){ // Unvisited
                dfs(v, u);
                low[u] = min(low[u], low[v]);
                if(low[v] >= disc[u]){ // "u" is Articulation
                    art[u]++;
                    blocks.emplace_back(1, u);
                    while(blocks.back().back() != v){
                        blocks.back().push_back(stV.top());
                        stV.pop();
                    }

                    edgBlocks.emplace_back(1, stE.top());
                    stE.pop();
                    while(edgBlocks.back().back() != pair(v, u)){
                        edgBlocks.back().push_back(stE.top());
                        stE.pop();
                    }
                }
                hasFwd = true;
            }
            else low[u] = min(low[u], disc[v]);
        }
        if(p == -1){
            if(art[u]) art[u]--;
            if(!hasFwd){    // BCC = {u}
                blocks.emplace_back(1, u);
                edgBlocks.emplace_back();
            }
        }
    }

    void build(){
        int timer = 0;
        for(int i = 0; i < n; i++){
            if(disc[i] == -1) dfs(i);
        }

        tree.assign(blocks.size(), vector<int>());
        for(int i = 0; i < n; i++){
            if(art[i]){
                pos[i] = tree.size();
                tree.emplace_back();
            }
        }

        for(int i = 0; i < blocks.size(); i++){
            for(int j : blocks[i]){
                if(!art[j]) pos[j] = i;
                else{   // Edge BCC - Articulation
                    tree[i].push_back(pos[j]);
                    tree[pos[j]].push_back(i);
                }
            }
        }
    }
};

/*

Time Complexity:

BlockCutTree    -> O(V + E)

Links:

https://www.youtube.com/watch?v=iYJqgMKYsdI

*/
```

2.9 Bridges (Tarjan)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

using ii = pair<int,int>;

vector<vector<int>> g;
vector<bool> visited;
vector<int> discovered, low;
int timer = 0;

vector<ii> bridges;

// Tarjan's Algorithm for finding bridges in a graph
// Bridges = It's an edge which, when removed, increases the number of
    connected components in the graph.

/*
Note that this implementation malfunctions if the graph has multiple edges,
since it ignores them. Of course, multiple edges will never be a part of
    the answer,
so you can check additionally that the reported bridge is not a multiple
    edge.
Alternatively it's possible to pass to dfs the index of the edge used to
    enter the vertex
instead of the parent vertex (and store the indices of all vertices).
*/
```

```
void dfs(int v, int p = -1){
    visited[v] = true;
    discovered[v] = low[v] = timer;
    timer++;
    for(int to : g[v]){
        if(to == p) continue;
        if(visited[to]) low[v] = min(low[v], discovered[to]);
        else{
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if(low[to] > discovered[v]){    // It's a bridge
                bridges.push_back({v, to});
            }
        }
    }
}

void find_bridges(int n){
    timer = 0;
    visited.assign(n, false);
    discovered.assign(n, -1);
    low.assign(n, -1);
    for(int i = 0; i < n; i++){
        if(!visited[i]) dfs(i);
    }
}

/*

Time Complexity:

find_bridges    -> O(V + E)

Links:

https://cp-algorithms.com/graph/bridge-searching.html
https://www.youtube.com/watch?v=qrAub5z8FeA

*/
```

2.10 Bridges (DFS Tree)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>

vector<vector<int>> g;
vector<int> lvl, dp; // dp[u] = number of edges that traverses (parent[u],
    u)
vector<pii> bridges;

void dfs(int u, int p = -1){
    dp[u] = 0;
    for(auto v : g[u]){
        if(lvl[v] == 0){ // Edge to child
            lvl[v] = lvl[u] + 1;
            dfs(v, u);
            dp[u] += dp[v];
        }
        else if(lvl[v] < lvl[u]){ // Edge upwards
            dp[u]++;
        }
        else if(lvl[v] > lvl[u]){ // Edge downwards
            dp[u]--;
        }
    }
    if(p != -1) dp[u]--;    // Parent edge isn't a back-edge
    if(p != -1 && dp[u] == 0){  // (p, u) It's a bridge
        bridges.push_back({p, u});
    }
}

void find_bridges(int n){
    lvl.assign(n, 0);
    dp.assign(n, 0);
    for(int i = 0; i < n; i++){
        if(!lvl[i]){
            lvl[i] = 1;
            dfs(i);
        }
    }
}

/*

Time Complexity:

find_bridges    -> O(V + E)

Links:

https://codeforces.com/blog/entry/68138

*/
```

2.11 2-Edge-Connected Component

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>

/*
Code for compressing graph into 2-Edge-Connected Component
and finding Bridges online.
2CC it's a CC that remains connected whenever you remove fewer than 2 edges
    .
(Can't have Bridges inside)
*/

// 0 -> Connected Component
// 1 -> 2-Edge-Connected Component
vector<int> dsu[2], sz[2];
vector<int> parent, visited;
int bridges = 0;
int iteration;

// Initialize 2 DSU
void build(int n){
    iteration = 0;
    parent.resize(n);
    visited.resize(n);
    for(int i = 0; i < 2; i++){
        dsu[i].resize(n);
        sz[i].resize(n);
    }
    for(int i = 0; i < n; i++){
        dsu[0][i] = dsu[1][i] = i;
        sz[0][i] = sz[1][i] = 1;
        parent[i] = -1;
    }
    bridges = 0;
}

int find(int v, int type){
    if(v == -1) return -1;
```

```
    if(dsu[type][v] == v) return v;
    return dsu[type][v] = find(dsu[type][v], type);
}

// Makes "v" the new root of the CC tree that "v" belongs to
void reRoot(int v){
    v = find(v, 1);
    int root = v;
    int child = -1;
    while(v != -1){
        int p = find(parent[v], 1);
        parent[v] = child;
        dsu[0][v] = root;
        child = v;
        v = p;
    }
    sz[0][root] = sz[0][child];
}

void merge(int a, int b){
    iteration++;
    vector<int> path_a, path_b;
    int lca = -1;
    while(lca == -1){
        if(a != -1){
            a = find(a, 1);
            path_a.push_back(a);
            if(visited[a] == iteration){
                lca = a;
                break;
            }
            visited[a] = iteration;
            a = parent[a];
        }
        if(b != -1){
            b = find(b, 1);
            path_b.push_back(b);
            if(visited[b] == iteration){
                lca = b;
                break;
            }
            visited[b] = iteration;
            b = parent[b];
        }
    }

    for(int v : path_a){
        dsu[1][v] = lca;
        if(v == lca) break;
        bridges--;
    }
    for(int v : path_b){
        dsu[1][v] = lca;
        if(v == lca) break;
        bridges--;
    }
}

void addEdge(int a, int b){
    a = find(a, 1);
    b = find(b, 1);
    if(a == b) return;

    int ca = find(a, 0);
    int cb = find(b, 0);
    if(ca != cb){
        bridges++;
        if(sz[0][ca] > sz[0][cb]){
            swap(a, b), swap(ca, cb);
        }
        reRoot(a);
        parent[a] = dsu[0][a] = b;
        sz[0][cb] += sz[0][a];
    }
    else{
        merge(a, b);
    }
}

/*

Time Complexity:

build   -> O(n)
find    -> O(1) amortized
reRoot  -> O(nlogn) n = size of current tree
merge   -> O(|cycle|)

total   -> O(n*logn + m*logn) m = number of edges

Links:

https://cp-algorithms.com/graph/bridge-searching-online.html

*/
```

2.12 Kosaraju (Strongly Connected Component)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Kosaraju's Algorithm for finding Strongly Connected Components (SCC)
struct SCC {
    int n;
    vector<vector<int>> g, gt, condensation_g;
    vector<bool> visited;
    stack<int> order;
    vector<int> parent, root_nodes;

    SCC(vector<vector<int>> &graph){
        n = graph.size();
        this->g = graph;

        gt.assign(n, vector<int>());
        for(int i = 0; i < n; i++){
            for(auto v : g[i]){
                gt[v].push_back(i);
            }
        }
        build();
    }

private:
    void build(){
        visited.assign(n, false);
        for(int i = 0; i < n; i++){
            if(!visited[i]) dfs(i);
        }

        parent.assign(n, -1);

        while(!order.empty()){
            int u = order.top();
            order.pop();
            if(parent[u] == -1){
                findComponent(u, u);
                root_nodes.push_back(u);
            }
        }

        // Building the Condensation Graph
```

```
        condensation_g.assign(n, vector<int>());
        for(int u = 0; u < n; u++){
            for(auto v : g[u]){
                int root_u = parent[u];
                int root_v = parent[v];
                if(root_u != root_v)
                    condensation_g[root_u].push_back(root_v);
            }
        }
    }

    void dfs(int u){
        visited[u] = true;
        for(auto v : g[u]){
            if(!visited[v]) dfs(v);
        }
        order.push(u);
    }

    void findComponent(int u, int root){
        parent[u] = root;
        for(auto v : gt[u]){
            if(parent[v] == -1) findComponent(v, root);
        }
    }
};

/*

Time Complexity

SCC ->  O(E + V)

Links:

https://cp-algorithms.com/graph/strongly-connected-components.html
https://www.youtube.com/watch?v=RpgcYiky7uw

*/
```

2.13 2 SAT

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// 2-SAT solver using Kosaraju's Algorithm
struct TwoSAT{
    int n;
    vector<vector<int>> g, gt;
    vector<int> component;
    vector<bool> vis, assignment;
    stack<int> order;

    TwoSAT(int n) : n(n){
        g.assign(2 * n + 2, vector<int>());
        gt.assign(2 * n + 2, vector<int>());
        component.assign(2 * n + 2, -1);
        vis.assign(2 * n + 2, false);
    }

    // (a or b) = (!a -> b) and (!b -> a)
    void addOr(int a, int b){ add(-a, b); add(-b, a); }
    // (a and b) = (a or a) and (b or b)
    void addAnd(int a, int b){ addOr(a, a); addOr(b, b); }
    // (a -> b) = (!a or b)
    void addImp(int a, int b){ addOr(-a, b); }
    // (a xor b) = (a or b) and (!a or !b)
    void addXor(int a, int b){ addOr(a, b); addOr(-a, -b); }
    // (a equal b) = (a <--> b) = (a -> b) and (b -> a)
    void addEqual(int a, int b){ addImp(a, b); addImp(b, a); }
    // (a != b) = (a xor b)
    void addDiff(int a, int b){ addXor(a, b); }

    bool solve(){
        // 0 and 1 are dead nodes
        for(int i = 2; i < 2 * n + 2; i++){
            if(!vis[i]) dfs(i);
        }
        reverse();
        int idx = 0;
        while(!order.empty()){
            int u = order.top();
            order.pop();
            if(component[u] == -1){
                findComponent(u, idx++);
            }
        }
        assignment.resize(n + 1);    // 1-based
        for(int i = 1; i <= n; i++){
            if(component[trad(i)] == component[trad(-i)])
                return false;
            assignment[i] = component[trad(i)] > component[trad(-i)];
        }
        return true;
    }

private:
    // + -> xi (even), - -> !xi (odd)
    int trad(int v){ return v > 0 ? 2 * v : 2 * abs(v) + 1; }
    void add(int a, int b){ g[trad(a)].push_back(trad(b)); }

    void reverse(){
        for(int i = 0; i < 2 * n + 2; i++)
            for(auto v : g[i])
                gt[v].push_back(i);
    }

    void dfs(int u){
        vis[u] = true;
        for(auto v : g[u]){
            if(!vis[v]) dfs(v);
        }
        order.push(u);
    }

    void findComponent(int u, int root){
        component[u] = root;
        for(auto v : gt[u]){
            if(component[v] == -1) findComponent(v, root);
        }
    }
};

/*

Time Complexity:

2SAT    -> O(V + E)

Links:

https://cp-algorithms.com/graph/2SAT.html
https://www.youtube.com/watch?v=0nNYy3rltgA

*/
```

2.14 Euler Path

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp
#define int long long
#define pii pair<int,int>

/*

Hierholzer's Algorithm to find Euler path/circuit in a graph
To use directed graph: Euler<true> euler;

getPath(src): Try to find a Euler Path/circuit starting in "src"
the function returns a pair {possible, path}, path being a vector
of {vertex, edge id to reach vertex}
Obs.: In the first path's position id = -1

If you find a circuit the first and last vertex will be equal.

getCycle(): Find a Euler circuit if it exists.
Obs.: The first vertex doesn't repeat in the end

* See the conditions required to exist a path/circuit
and how to find the "src" in Theorems/Graph.txt

*/

template<bool directed = false>
struct Euler{
    vector<vector<pii>> g;
    vector<int> used;
    int n;

    Euler(int n) : n(n), g(n) {}

    void addEdge(int a, int b){
        int id = used.size();
        used.push_back(false);
        g[a].emplace_back(b, id);
        if(!directed) g[b].emplace_back(a, id);
    }

    // You need to pass the correct "src"
    pair<bool, vector<pii>> getPath(int src){
        if(!used.size()) return {true, {}};
        for(int & i : used) i = false;
        vector<int> point(n, 0);
        // { {vertex, prev vertex}, edge id}
        vector<pair<pii, int>> ret, st = { {{src, -1}, -1} };
        while(st.size()){
            int curr = st.back().first.first;
            int & it = point[curr];
            while(it < g[curr].size() && used[g[curr][it].second]) it++;
            if(it == g[curr].size()){ // no more edges out of "curr"
                if(ret.size() && ret.back().first.second != curr)
                    return {false, {}};
                ret.push_back(st.back());
                st.pop_back();
            }
            else{
                st.push_back({ {g[curr][it].first, curr}, g[curr][it].
                    second });
                used[g[curr][it].second] = true;
            }
        }
        if(ret.size() != used.size() + 1) return {false, {}};
        vector<pii> answ;
        for(auto i : ret) answ.emplace_back(i.first.first, i.second);
        reverse(answ.begin(), answ.end());
        return {true, answ};
    }

    pair<bool, vector<pii>> getCycle(){
        if(!used.size()) return {true, {}};
        int src = 0;
        while(!g[src].size()) src++;
        auto answ = getPath(src);
        if(!answ.first || answ.second[0].first != answ.second.back().first)
            {
            return {false, {}};
        }
        answ.second[0].second = answ.second.back().second;
        answ.second.pop_back();
        return answ;
    }
};

/*

Time Complexity:

getPath     -> O(V + E)
getCycle    -> O(V + E)

Links:

https://www.youtube.com/watch?v=8MpoO2zA2l4
https://usaco.guide/adv/eulerian-tours?lang=cpp
https://cp-algorithms.com/graph/euler_path.html

*/
```

2.15 Euler Tour technique

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Euler Tour technique for flattening a tree

const int ms = 2*1e5 + 100;
vector<vector<int>> g;
int timer = 0;
int st[ms], en[ms];

// For each vertex we assign a unique id,
// all vertices of its subtree will have contiguous ids
// We build the Seg Tree using the ids in st[u] and en[u]
void dfs(int u, int p = -1){
    st[u] = timer++;
    for(auto v : g[u]){
        if(v == p) continue;
        dfs(v, u);
    }
    en[u] = timer - 1; // This can change
}

/*

Time Complexity:

dfs -> O(V + E)

Links:

https://codeforces.com/blog/entry/81527

*/
```

2.16 HLD (Vertex)

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

```cpp
using namespace std;

#define int long long

const int ms = 1e5;
vector<vector<int>> g;
vector<int> val;
int sz[ms], par[ms], head[ms], pos[ms];
int v[ms], seg[4 * ms];
int t;

void dfs_sz(int u, int p = -1){
    sz[u] = 1;
    for(auto & v : g[u]){
        if(v == p) continue;
        dfs_sz(v, u);
        sz[u] += sz[v];
        if(g[u][0] == p || sz[v] > sz[g[u][0]]) swap(v, g[u][0]);
    }
}

void dfs_hld(int u, int p = -1){
    pos[u] = t++;
    v[pos[u]] = val[u];
    for(auto v : g[u]){
        if(v == p) continue;
        par[v] = u;
        head[v] = (v == g[u][0] ? head[u] : v);
        dfs_hld(v, u);
    }
}

// Call this function to initialize
void build_hld(int root = 0){
    dfs_sz(root);
    t = 0;
    head[root] = root;
    dfs_hld(root);
    build_seg(1, 0, t - 1); // We can use any data structure for range
        query
}

// Sum query between path from "a" to "b"
int query_path(int a, int b){
    if(pos[a] < pos[b]) swap(a, b);
    if(head[a] == head[b]) return query_seg(pos[b], pos[a]);
    return query_seg(pos[head[a]], pos[a]) + query_path(par[head[a]], b);
}

// Updates all vertices of the path from "a" to "b"
void update_path(int a, int b, int x){
    if(pos[a] < pos[b]) swap(a, b);
    if(head[a] == head[b]){
        update_seg(pos[b], pos[a], x);
        return;
    }
    update_seg(pos[head[a]], pos[a], x);
    update_path(par[head[a]], b, x);
}

// Queries all nodes in the subtree of "a"
int query_subtree(int a){
    return query_seg(pos[a], pos[a] + sz[a] - 1);
}

// Updates all nodes in the subtree of "a"
void update_subtree(int a, int x){
    update_seg(pos[a], pos[a] + sz[a] - 1, x);
}

int lca(int a, int b){
    if(pos[a] < pos[b]) swap(a, b);
    return head[a] == head[b] ? b : lca(par[head[a]], b);
}

/*

Time Complexity:

build           -> O(n)
update_path     -> O(log^2(n))
query_path      -> O(log^2(n))
query_subtree   -> O(log(n))
update_subtree  -> O(log(n))
lca             -> O(log(n))

Links:

https://www.youtube.com/watch?v=L8mxYASOU2E
https://usaco.guide/plat/hld?lang=cpp

*/
```

2.17 HLD (Edge)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int ms = 1e5;
vector<vector<pair<int,int>>> g;
vector<int> sobe;
int sz[ms], par[ms], head[ms], pos[ms];
int v[ms], seg[4 * ms];
int t;

void dfs_sz(int u, int p = -1){
    sz[u] = 1;
    for(auto & i : g[u]){
        auto [v, w] = i;
        if(v == p) continue;
        sobe[v] = w;
        dfs_sz(v, u);
        sz[u] += sz[v];
        if(g[u][0].first == p || sz[v] > sz[g[u][0].first]) swap(i, g[u
            ][0]);
    }
}

void dfs_hld(int u, int p = -1){
    pos[u] = t++;
    v[pos[u]] = sobe[u];
    for(auto i : g[u]){
        auto [v, w] = i;
        if(v == p) continue;
        par[v] = u;
        head[v] = (i == g[u][0] ? head[u] : v);
        dfs_hld(v, u);
    }
}

// Call this function to initialize
void build_hld(int root = 0){
    dfs_sz(root);
    t = 0;
    head[root] = root;
    dfs_hld(root);
    build_seg(1, 0, t - 1); // We can use any data structure for range
        query
}

// Sum query between path from "a" to "b"
int query_path(int a, int b){
    if(a == b) return 0;
```

```
    if(pos[a] < pos[b]) swap(a, b);
    if(head[a] == head[b]) return query_seg(pos[b] + 1, pos[a]);
    return query_seg(pos[head[a]], pos[a]) + query_path(par[head[a]], b);
}

// Updates all vertices of the path from "a" to "b"
void update_path(int a, int b, int x){
    if(a == b) return;
    if(pos[a] < pos[b]) swap(a, b);
    if(head[a] == head[b]){
        update_seg(pos[b] + 1, pos[a], x);
        return;
    }
    update_seg(pos[head[a]], pos[a], x);
    update_path(par[head[a]], b, x);
}

// Queries all nodes in the subtree of "a"
int query_subtree(int a){
    if(sz[a] == 1) return 0;
    return query_seg(pos[a] + 1, pos[a] + sz[a] - 1);
}

// Updates all nodes in the subtree of "a"
void update_subtree(int a, int x){
    if(sz[a] == 1) return;
    update_seg(pos[a] + 1, pos[a] + sz[a] - 1, x);
}

int lca(int a, int b){
    if(pos[a] < pos[b]) swap(a, b);
    return head[a] == head[b] ? b : lca(par[head[a]], b);
}

/*

Time Complexity:

build           -> O(n)
update_path     -> O(log^2(n))
query_path      -> O(log^2(n))
query_subtree   -> O(log(n))
update_subtree  -> O(log(n))
lca             -> O(log(n))

Links:

https://www.youtube.com/watch?v=L8mxYASOU2E
https://usaco.guide/plat/hld?lang=cpp

*/
```

2.18 DSU On Tree

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Smaller to Larger / DSU On Tree
// This code answers:
// "how many vertices in subtree of vertex v are colored with color c?"

vector<vector<int>> g;
const int ms = 1e5;
int sz[ms], color[ms], cnt[ms];

// Preprocesses the size of the subtrees
void dfsSize(int u, int p = -1){
    sz[u] = 1;
    for(auto v : g[u]){
        if(v == p) continue;
        dfsSize(v, u);
        sz[u] += sz[v];
    }
}

// For each problem we'll have a different add function
void add(int u, int p, int bigChild, int x){
    cnt[color[u]] += x;
    for(auto v : g[u]){
        if(v == p || v == bigChild) continue;
        add(v, u, bigChild, x);
    }
}

void DsuOnTree(int u, int p, bool keep){
    // Look for the largest child inside the subtree
    int maxSize = -1, bigChild = -1;
    for(auto v : g[u]){
        if(v != p && maxSize < sz[v]){
            maxSize = sz[v];
            bigChild = v;
        }
    }
    // Run a dfs on small childs and clear them from cnt
    for(auto v : g[u]){
        if(v == p || v == bigChild) continue;
        DsuOnTree(v, u, false);
    }
    // Calculates the answer of the greatest child and keeps his answer
    if(bigChild != -1) DsuOnTree(bigChild, u, true);
    // Merge the answers with small children
    add(u, p, bigChild, 1);
    // cnt[c] = # vertices in subtree of vertex u that has color c
    if(keep == false) add(u, p, -1, -1);
}

/*

Time Complexity:

DsuOnTree   -> O(nlogn)

Links:

https://codeforces.com/blog/entry/44351
https://codeforces.com/blog/entry/67696

*/
```

2.19 Centroid Decomposition

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int ms = 1e5;

vector<int> g[ms];
int dist[ms][30]; // Distance from the nodes to the centroid (represented
    by the level)
int parent[ms], sz[ms], removed[ms], height[ms];

// Preprocessing subtree sizes
void dfsSubtree(int u, int p){
    sz[u] = 1;
    for(auto v : g[u]){
        if(v != p && !removed[v]){
            dfsSubtree(v, u);
            sz[u] += sz[v];
```

```
        }
    }
}

// For each subtree we find its centroid
int getCentroid(int u, int p, int curr_size){
    for(auto v : g[u]){
        if(v != p && !removed[v] && sz[v] * 2 >= curr_size){
            return getCentroid(v, u, curr_size);
        }
    }
    return u;
}

// For each node, we calculate its distance
// to the centroid (represented by the level)
void setDist(int u, int p, int lvl){
    for(auto v : g[u]){
        if(v == p || removed[v]) continue;
        dist[v][lvl] = dist[u][lvl] + 1;
        setDist(v, u, lvl);
    }
}

// Algorithm to perform Centroid Decomposition
void decompose(int u, int p = -1, int lvl = 0){
    dfsSubtree(u, -1);
    int ctr = getCentroid(u, -1, sz[u]);
    parent[ctr] = p;
    height[ctr] = lvl;  // Assigning a level to the centroid
    removed[ctr] = 1;
    setDist(ctr, p, lvl);
    for(auto v : g[ctr]){
        if(v != p && !removed[v]){
            decompose(v, ctr, lvl + 1);
        }
    }
}

/*

Time Complexity:

decompose   -> O(nlogn)

Links:

https://www.quora.com/profile/Abbas-Rangwala-13/Centroid-Decomposition-of-a
    -Tree
https://medium.com/carpanese/an-illustrated-introduction-to-centroid-
    decomposition-8c1989d53308
https://www.youtube.com/watch?v=-DmMLQCmz94
https://usaco.guide/plat/centroid?lang=cpp

*/
```

2.20 LCA (Binary Lifting)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int ms = 10000;   // Number of vertices
const int LOG = 20;             // Log2(N)

vector<vector<int>> g;  // Graph (0-Based)
int up[ms][LOG];        // up[v][j] is 2^j-th ancestor of v
int depth[ms];

// Preprocessing
void dfs(int a, int p) {
        for(int b : g[a]) {
                if(b == p) continue;    // don't go back to the father
                depth[b] = depth[a] + 1;
                up[b][0] = a;                   // a is parent of b
                for(int j = 1; j < LOG; j++) {
                        up[b][j] = up[ up[b][j - 1] ][j - 1];
                }
                dfs(b, a);
        }
}

// Algorithm to find the Lowest Common Ancestor using Binary Lifting
int lca(int a, int b) {
        if(depth[a] < depth[b]) swap(a, b);
        // 1) Get same depth.
        int k = depth[a] - depth[b];
        for(int j = LOG - 1; j >= 0; j--) {
                if(k & (1 << j)) {
                        a = up[a][j]; // parent of a
                }
        }
        // 2) if b was ancestor of a then now a == b
        if(a == b) return a;
        // 3) move both a and b with powers of two
        for(int j = LOG - 1; j >= 0; j--) {
                if(up[a][j] != up[b][j]) {
                        a = up[a][j];
                        b = up[b][j];
                }
        }
        return up[a][0];
}

/*

Time Complexity:

dfs     -> O(N*logN)
lca     -> O(logN)

Links:

https://cp-algorithms.com/graph/lca_binary_lifting.html
https://www.youtube.com/watch?v=dOAxrhAUIhA
https://github.com/Errichto/youtube/blob/master/lca.cpp

*/
```

2.21 LCA (Segment Tree)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// LCA using Euler Tour technique to reduce in a RMQ problem.

const int ms = 2*1e5 + 100;
vector<vector<int>> g;
vector<int> euler;
int depth[ms], first[ms];
vector<int> seg; // After dfs, initialize with 4 * euler.size()

// We put in the crossing a certain vertex whenever we pass by it
void dfs(int u, int p = -1){
    first[u] = euler.size();
    euler.push_back(u);
    for(auto v : g[u]){
        if(v == p) continue;
```

```
        depth[v] = depth[u] + 1;
        dfs(v, u);
        euler.push_back(u);
    }
}

// Using a segmentation tree to answer the RMQ
// Obs.: We can use a Sparse Table to query in O(1)

// Call using build(1, 0, euler.size() - 1)
void build(int p, int l, int r){
    if(l == r){
        seg[p] = euler[l];
    }
    else{
        int m = (l + r) / 2;
        int lc = 2*p;
        int rc = lc + 1;

        build(lc, l, m);
        build(rc, m + 1, r);

        int vl = seg[lc];
        int vr = seg[rc];
        seg[p] = (depth[vl] < depth[vr]) ? vl : vr;
    }
}

int query(int p, int l, int r, int ql, int qr){
    if(l >= ql && r <= qr){
        return seg[p];
    }
    else{
        int m = (l + r) / 2;
        int lc = 2*p;
        int rc = lc + 1;

        if(qr <= m) return query(lc, l, m, ql, qr);
        else if(ql > m) return query(rc, m + 1, r, ql, qr);

        int vl = query(lc, l, m, ql, qr);
        int vr = query(rc, m + 1, r, ql, qr);

        return (depth[vl] < depth[vr]) ? vl : vr;
    }
}

int lca(int u, int v){
    int l = first[u];
    int r = first[v];
    if(l > r) swap(l, r);
    return query(1, 0, euler.size() - 1, l, r);
}

/*

Time Complexity:

dfs     -> O(V + E)
build   -> O(n)
query   -> O(logn)

Links:

https://cp-algorithms.com/graph/lca.html

*/
```

2.22 Tree Hashing

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

using ii = pair<int,int>;
const int MOD = 1e9 + 7;
const int ms = 1e5 + 100;
int h[ms], sz[ms];

int fexp(int x, int y){
    int answ = 1;
    x = x % MOD;
    while(y){
        if(y & 1) answ = answ * x % MOD;
        x = x * x % MOD;
        y >>= 1;
    }
    return answ;
}

// If the tree is not rooted you need to root the centroids
void getHash(int u, int p, vector<vector<int>>& g){
    sz[u] = 1;
    vector<ii> children;
    for(auto v : g[u]){
        if(v == p) continue;
        getHash(v, u, g);
        sz[u] += sz[v];
        children.push_back({h[v], sz[v]});
    }
    sort(children.begin(), children.end());

    h[u] = 1;
    for(auto [val, size] : children){
        int pw2 = fexp(2, 2 * size);
        h[u] = h[u] * pw2 % MOD;
        h[u] = (h[u] + val) % MOD;
    }
    h[u] = 2 * h[u] % MOD;
}

//------------Perfect Hash-------------//
// No collisions
map<vector<int>, int> hasher;

int hashify(vector<int> x){
    sort(x.begin(), x.end());
    if(!hasher[x]){
        hasher[x] = hasher.size();
    }
    return hasher[x];
}

int hashTree(int u, int p, vector<vector<int>>& g){
    vector<int> children;
    for(int v : g[u]){
        if(v == p) continue;
        children.push_back(hashTree(v, u, g));
    }
    return hashify(children);
}

/*

Time Complexity:

getHash     -> O(n*logn)
hashTree    -> O(n*logn)

Links:

https://codeforces.com/blog/entry/101010
https://codeforces.com/blog/entry/113465
```

```
*/
```

2.23 Kruskal

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

struct Edge{
    int u, v, w, id;
    Edge() {}
    Edge(int u, int v, int w = 0, int id = 0) : u(u), v(v), w(w), id(id) {}
    bool operator < (Edge & other) const { return w < other.w; }
};
vector<Edge> edges, mst;

// Kruskal's Algorithm for Minimum Spanning Tree using DSU (Returns the
    cost to build the MST)
int kruskal(int n){
    int answ = 0;
    sort(edges.begin(), edges.end()); // sort by weight
    initDSU(n);
    for(Edge e : edges){
        if(find(e.u) == find(e.v)) continue;
        answ += e.w;
        mst.push_back(e);
        merge(e.u, e.v);
        if(mst.size() == n - 1) break;
    }
    return answ;
}

/*

Time Complexity:

kruskal -> O(E*logN)

Links:

https://cp-algorithms.com/graph/mst_kruskal.html
https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html

*/
```

2.24 Prim

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>
const int INF = 0x3f3f3f3f3f3f3f3f;

vector<vector<pii>> g; // Graph (0-based) g[u] -> (v, w)

// Undirected Weighted Graph
void addEdge(int u, int v, int w){
    g[u].emplace_back(v, w);
    g[v].emplace_back(u, w);
}

// Prim's Algorithm for Minimum Spanning Tree (Returns the cost to build
    the MST)
int prim(int src){
    int n = g.size();
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    vector<int> costs(n, INF);
    costs[src] = 0;
    vector<int> parent(n, -1); // Store MST
    vector<bool> inMST(n, false);
    pq.emplace(0, src); // {w, u}
    int answ = 0;
    while(!pq.empty()){
        auto [wt, u] = pq.top();
        pq.pop();
        if(inMST[u] == true) continue;
        inMST[u] = true;
        answ += wt;
        for(auto [v, w] : g[u]){
            if(inMST[v] == false && costs[v] > w){
                costs[v] = w;
                pq.emplace(w, v);
                parent[v] = u;
            }
        }
    }
    /*
    // Print edges of MST using parent array (We start from 1 because the
        parent of the source is -1)
    for (int i = 1; i < N; ++i)
        printf("%d - %d\n", parent[i], i);
    */
    return answ;
}
```

2.25 Boruvka

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

struct Edge{
    int u, v, w, id;
    Edge() {}
    Edge(int u, int v, int w = 0, int id = 0) : u(u), v(v), w(w), id(id) {}
    bool operator < (Edge & other) const { return w < other.w; }
};

// Algorithm to find the Minimum Spanning Tree of a graph
vector<Edge> boruvka(vector<Edge> & edges, int n){
    vector<Edge> mst;
    vector<Edge> best(n);
    initDSU(n); // Use Path Compression & Union-by-size/rank
    bool f = true;
    while(f){
        f = false;
        for(int i = 0; i < n; i++) best[i] = Edge(i, i, INF);
        for(auto e : edges){
            int pu = find(e.u), pv = find(e.v);
            if(pu == pv) continue;
            if(e < best[pu]) best[pu] = e;
            if(e < best[pv]) best[pv] = e;
        }
        for(int i = 0; i < n; i++){
            Edge e = best[find(i)];
            if(find(e.u) == find(e.v)) continue;
            merge(e.u, e.v);
            mst.push_back(e);
            f = true;
        }
    }
    return mst;
}

/*
```

```
Time Complexity:

boruvka -> O(logV * (V + E))

Links:

https://codeforces.com/blog/entry/77760
https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

*/
```

2.26 Dijkstra

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>
const int INF = 0x3f3f3f3f3f3f3f3f;

vector<vector<pii>> g; // Graph (0-based) g[u] -> (v, w)
vector<int> dist; // Shortest distances (0-Based)
vector<int> predecessors; // Need to re-construct path (0-Based)

// Undirected Weighted Graph
void addEdge(int u, int v, int w){
    g[u].emplace_back(v, w);
    g[v].emplace_back(u, w);
}

// Dijkstra Algorithm (Single-source shortest paths problem)
void dijkstra(int src){
    int n = g.size();
    dist.assign(n, INF);
    predecessors.assign(n, -1);
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    pq.push({0, src}); // {Weight, vertex}
    dist[src] = 0;
    while(!pq.empty()){
        auto [d, u] = pq.top();
        pq.pop();
        if(dist[u] < d) continue;
        for(auto [v, w] : g[u]){
            if(dist[v] > dist[u] + w){
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
                predecessors[v] = u;
            }
        }
    }
}

/*

Time Complexity:

dijkstra -> O(E*LogV)

Links:

https://cp-algorithms.com/graph/dijkstra.html
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-
    priority_queue-stl/

*/
```

2.27 Floyd-Warshall

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
const int INF = 0x3f3f3f3f3f3f3f3f;
const int ms = 500;

int g[ms][ms], dist[ms][ms];

// Undirected Weighted Graph
void addEdge(int u, int v, int w){
    g[u][v] = w;
    g[v][u] = w;
}

// Initialize the distance matrix
void build(){
    for(int i = 0; i < ms; i++){
        for(int j = 0; j < ms; j++){
            if(i == j) dist[i][j] = 0;
            else if(g[i][j]) dist[i][j] = g[i][j];
            else dist[i][j] = INF;
        }
    }
}

// Floyd-Warshall Algorithm
// Find the length of the shortest path d[i][j] between each pair of
    vertices i and j.
void floydWarshall(){
    // We test all K vertices as intermediaries between (i -> j),
    // the shortest path between (i -> j) will be (i -> k -> j)
    for(int k = 0; k < ms; k++){
        for(int i = 0; i < ms; i++){
            for(int j = 0; j < ms; j++){
                // If our graph has negative weight edges it is necessary
                    to do this check
                if(dist[i][k] < INF && dist[k][j] < INF)
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }
}

/*

Time Complexity:

build           -> O(N^2)
floydWarshall   -> O(N^3)

Links:

https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html
https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

Obs.:

1) Com float, erros de precisao sao comuns, precisamos corrigir utilizando
    EPS (Ver o primeiro site).
2) Podemos guardar os predecessores utilizando uma matriz.
3) O grafo pode ter pesos negativos, mas nao ciclos negativos

*/
```

2.28 Bellman-Ford

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp
#define int long long

// Graph inplementation using Edge List (u,v,w) (0-Based)
vector<tuple<int,int,int>> edgeList;
vector<int> dist;

// Bellman-Ford Algorithm (Single source shortest path with negative weight edges)
// Returns true if valid answer exists.
bool bellmanFord(int src, int n){
    const int INF = 0x3f3f3f3f3f3f3f3f;
    const int NINF = -INF;
    dist.assign(n, INF);
    dist[src] = 0;

    // Relax all edges n - 1 times. If we're sure that we don't have
    // negative cycles, we can use a flag to stop when there isn't update
    for(int i = 0; i < n - 1; i++){
        for(int j = 0; j < edgeList.size(); j++){
            auto [u, v, w] = edgeList[j];
            if(dist[u] == INF) continue;
            dist[v] = min(dist[v], dist[u] + w);
            dist[v] = max(dist[v], NINF);
        }
    }

    // Check the existence of a negative cycle. If you have only one update we can return true.
    // We perform n - 1 steps to check if the cycle affects a specific node.
    for(int i = 0; i < n - 1; i++){
        for(int j = 0; j < edgeList.size(); j++){
            auto [u, v, w] = edgeList[j];
            if(dist[u] == INF) continue;
            if(dist[u] + w < dist[v]) dist[v] = NINF; // We can only return false here
        }
    }

    // There isn't negative cycle and our answer is in dist
    // change to "return dist[node] != NINF" to verify a specific node.
    return true;
}

/*

Time Complexity:

bellmanFord -> O(V*E)

Links:

https://cp-algorithms.com/graph/bellman_ford.html
https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

*/
```

2.29 SPFA

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>
const int INF = 0x3f3f3f3f3f3f3f3f;

vector<vector<pii>> g; // Graph (0-based) g[u] -> (v, w)
vector<int> dist; // Shortest distances (0-Based)

// Undirected Weighted Graph
void addEdge(int u, int v, int w){
    g[u].emplace_back(v, w);
    g[v].emplace_back(u, w);
}

// Shortest Path Faster Algorithm (SPFA) (Single source shortest path with negative weight edges)
// SPFA is a improvement of the Bellman-Ford algorithm.
bool spfa(int src){
    int n = g.size();
    dist.assign(n, INF);

    // count[i] = # of times the distance of "i" has changed
    // if it is greater than N-1 there's a negative cycle
    vector<int> count(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;
    dist[src] = 0;
    q.push(src);
    inqueue[src] = true;
    while(!q.empty()){
        int u = q.front();
        q.pop();
        inqueue[u] = false;
        for(auto [v, w] : g[u]){
            if(dist[u] + w < dist[v]){
                dist[v] = dist[u] + w;
                if(!inqueue[v]){
                    q.push(v);
                    inqueue[v] = true;
                    count[v]++;
                    if(count[v] > n) return false; // negative cycle
                }
            }
        }
    }
    return true; // There isn't negative cycle
}

/*

Time Complexity:

SPFA -> O(V*E) (In the worst case, on average it is O(E), so it's efficient)

Links:

https://cp-algorithms.com/graph/bellman_ford.html
https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

*/
```

2.30 DFS

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

vector<vector<int>> g; // Graph (0-based)
vector<int> visited;

// Undirected Graph
void addEdge(int u, int v){
    g[u].push_back(v);
    g[v].push_back(u);
}

void dfsUtil(int u){
```

```
    visited[u] = true;
    for(auto v : g[u]){
        if(visited[v]) continue;
        dfsUtil(v);
    }
}

// Traverse all the Graph (Disconnected Graph)
void dfs(){
    int n = g.size();
    visited.assign(n, false);
    for(int i = 0; i < n; i++){
        if(visited[i]) continue;
        dfsUtil(i);
    }
}

/*

Time Complexity:

addEdge -> O(1)
dfs     -> O(V+E)

Links:

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
https://www.geeksforgeeks.org/graph-implementation-using-stl-for-
    competitive-programming-set-1-dfs-of-unweighted-and-undirected/

*/
```

2.31 BFS

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
vector<vector<int>> g;  // Graph (0-based)
vector<int> visited;

// Undirected Graph
void addEdge(int u, int v){
    g[u].push_back(v);
    g[v].push_back(u);
}

void bfsUtil(int src){
    queue<int> q;
    q.push(src);
    visited[src] = true;
    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(auto v: g[u]){
            if(visited[v]) continue;
            visited[v] = true;
            q.push(v);
        }
    }
}

// Traverse all the Graph (Disconnected Graph)
void bfs(){
    int n = g.size();
    visited.assign(n, false);
    for(int i = 0; i < n; i++){
        if(visited[i]) continue;
        bfsUtil(i);
    }
}

// This function return true with a graph is bipartite
bool isBipartite(){
    int n = g.size();
    // -1 = unvisited, 0 = black, 1 = white
    vector<int> color(n, -1);
    queue<pair<int, int>> q;

    // loop incase graph is not connected
    for(int i = 0; i < n; i++){
        if(color[i] != -1) continue; // Already visited
        q.push({i, 0}); // {vertex, color}
        color[i] = 0;
        while(!q.empty()){
            auto[u, c] = q.front();
            q.pop();
            for(auto v : g[u]){
                if(color[v] == c) return false;
                if(color[v] == -1){
                    color[v] = 1 - c;
                    q.push({v, color[v]});
                }
            }
        }
    }
    return true;
}

/*

Time Complexity:

addEdge     -> O(1)
bfs         -> O(V+E)
isBipartite -> O(V+E)

Links:

https://www.geeksforgeeks.org/bfs-using-stl-competitive-coding/
https://www.geeksforgeeks.org/bipartite-graph/

*/
```

2.32 Topologial Sort

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
vector<vector<int>> g;

// Directed Graph
void addEdge(int u, int v){
    g[u].push_back(v);
}

// Kahn's algorithm for Topological Sorting (Directed Acyclic Graphs)
vector<int> topologicalSort(){
    int n = g.size();

    vector<int> inDegree(n, 0);
    for(int u = 0; u < n; u++) {
        for(auto v : g[u]){
            inDegree[v]++;
        }
    }

    queue<int> q;
    for(int i = 0; i < n; i++){
```

```
        if(inDegree[i]) continue;
        q.push(i);
    }
    int cnt = 0; // # of visited
    vector<int> answ;

    while(!q.empty()){
        int u = q.front();
        q.pop();
        answ.push_back(u);
        for(auto v : g[u]){
            if(--inDegree[v] == 0){
                q.push(v);
            }
        }
        cnt++;
    }
    if(cnt == n) return answ;
    else return {}; // There's a cycle, impossible to find topological sort
}

/*

Time Complexity:

topologicalSort -> O(V + E)

Links:

https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/
https://cp-algorithms.com/graph/topological-sort.html
https://www.geeksforgeeks.org/cpp-program-for-topological-sorting/

*/
```

3 Dynamic Programming

3.1 LCS

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
const int MAXN = 1e4;
int dp[MAXN][MAXN];
string s1, s2;

// Longest Common Subsequence
// Call lcs(0, 0)
int lcs(int i, int j){
    if(i == s1.size() || j == s2.size()) return 0;

    if(dp[i][j] != -1) return dp[i][j];

    if(s1[i] == s2[j])
        return dp[i][j] = 1 + lcs(i + 1, j + 1);

    return dp[i][j] = max(lcs(i + 1, j), lcs(i, j + 1));
}

string answ;

// Recovery lcs answer
void recovery(int i, int j){
    if(i == s1.size() || j == s2.size()) return;
    if(s1[i] == s2[j] && dp[i][j] == 1 + lcs(i + 1, j + 1)){
        answ.push_back(s1[i]);
        recovery(i + 1, j + 1);
    }
    else if(dp[i][j] == lcs(i + 1, j)) recovery(i + 1, j);
    else recovery(i, j + 1);
}

/*

Time Complexity

lcs -> O(n*m)

Links:

https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/
https://www.youtube.com/watch?v=sSno9rV8Rhg

*/
```

3.2 LIS

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

vector<int> v;

// Longest Increasing Subsequence
int lis(int n){
    vector<int> dp(n, 1);
    for(int i = 0; i < n; i++){
        for(int j = 0; j < i; j++){
            if(v[j] < v[i]) dp[i] = max(dp[i], dp[j] + 1);
        }
    }
    int answ = 0;
    for(int i = 0; i < n; i++) answ = max(answ, dp[i]);
    return answ;
}

// Longest Increasing Subsequence using Binary Search
vector<int> LISBS(int n){
    const int INF = 0x3f3f3f3f3f3f3f3f;
    vector<int> dp(n + 1, INF);
    vector<int> idx(n + 1, -1);
    vector<int> parent(n + 1, -1);
    dp[0] = -INF;

    for(int i = 0; i < n; i++){
        int j = upper_bound(dp.begin(), dp.end(), v[i]) - dp.begin();
        if(dp[j-1] < v[i] && v[i] < dp[j]){
            dp[j] = v[i];
            idx[j] = i;
            parent[j] = idx[j-1];
        }
    }

    vector<int> answ;
    int pos = 0;
    for(int i = 0; i <= n; i++){
        if(dp[i] < INF) pos = i;
    }
    while(pos != -1){
        answ.push_back(v[pos]);
        pos = parent[pos];
    }
    return answ;
}

/*

Time Complexity
```

```
lis      -> O(n^2)
LISBS    -> O(n*logn)

Links:

https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/
https://www.youtube.com/watch?v=1RpMc3fv0y4

*/
```

3.3 SOS

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// SOS DP
int SumOverSubsets(vector<int> & v, int n){
    int dp[(1 << n)][n + 1];
    for(int mask = 0; mask < (1 << n); mask++){ // Base case
        dp[mask][0] = v[mask];
    }
    for(int mask = 0; mask < (1 << n); mask++){
        for(int i = 1; i <= n; i++){
            if(mask & (1 << (i - 1))){ // bit at i - 1 is ON
                dp[mask][i] = dp[mask][i - 1] + dp[mask ^ (1 << (i - 1))][i
                    - 1];
            }
            else{ // bit at i - 1 is OFF
                dp[mask][i] = dp[mask][i - 1];
            }
        }
    }
    /*
    Memory optimization
    int dp[1 << n];
    for(int i = 0; i < (1 << n); i++) dp[i] = v[i];
    for(int i = 0; i < n; i++){
        for(int mask = 0; mask < (1 << n); mask++){
            if(mask & (1 << i)){
                dp[i] += dp[mask ^ (1 << i)];
            }
        }
    }
    */
    return dp[(1 << n) - 1][n];
}

/*

Time Complexity:

SOS DP  -> O(n * 2^n)

Links:

https://codeforces.com/blog/entry/45223
https://www.youtube.com/watch?v=mkiK_GCWX50
https://usaco.guide/adv/dp-sos?lang=cpp

*/
```

4 Number Theory

4.1 Sieve of Eratosthenes

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Sieve of Eratosthenes
vector<bool> sieve(int n){
    vector<bool> prime(n + 1, true);
    prime[0] = prime[1] = false;
    for(int i = 2; i * i <= n; i++){
        if(prime[i]){
            for(int j = i * i; j <= n; j += i){
                prime[j] = false;
            }
        }
    }
    return prime;
}

/*

Time Complexity:

sieve   ->  O(n*log(log(n)))

Links:

https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html

*/
```

4.2 Totient (phi)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Euler's totient function for one number
int phi(int n){
    int answ = n;
    for(int i = 2; i * i <= n; i++){
        if(n % i == 0){
            while(n % i == 0) n /= i;
            answ -= answ / i;
        }
    }
    if(n > 1) answ -= answ / n;
    return answ;
}

// Euler's totient function from 1 to n
vector<int> sievePhi(int n){
    vector<int> phi(n + 1);
    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;
            }
        }
    }
    return phi;
}

/*

Time Complexity:

phi ->  O(sqrt(n))
```

```
sievePhi    -> O(n*log(log(n)))

Links:

https://cp-algorithms.com/algebra/phi-function.html

*/
```

4.3 Fast Fourier transform

```
#include <bits/stdc++.h>
using namespace std;

typedef double ld;
const ld PI = acos(-1);

struct Complex {
    ld real, imag;
    Complex conj() { return Complex(real, -imag); }
    Complex(ld a = 0, ld b = 0) : real(a), imag(b) {}
    Complex operator + (const Complex &o) const { return Complex(real + o.
        real, imag + o.imag); }
    Complex operator - (const Complex &o) const { return Complex(real - o.
        real, imag - o.imag); }
    Complex operator * (const Complex &o) const { return Complex(real * o.
        real - imag * o.imag, real * o.imag + imag * o.real); }
    Complex operator / (ld o) const { return Complex(real / o, imag / o); }
    void operator *= (Complex o) { *this = *this * o; }
    void operator /= (ld o) { real /= o, imag /= o; }
};

typedef vector<Complex> CVector;
const int ms = 1 << 22;
int bits[ms];
Complex root[ms];

// Start by calling this function
void initFFT() {
    root[1] = Complex(1);
    for(int len = 2; len < ms; len += len) {
        Complex z(cos(PI / len), sin(PI / len));
        for(int i = len / 2; i < len; i++) {
            root[2 * i] = root[i];
            root[2 * i + 1] = root[i] * z;
        }
    }
}

void pre(int n) {
    int LOG = 0;
    while(1 << (LOG + 1) < n) {
        LOG++;
    }
    for(int i = 1; i < n; i++) {
        bits[i] = (bits[i >> 1] >> 1) | ((i & 1) << LOG);
    }
}

CVector fft(CVector a, bool inv = false) {
    int n = a.size();
    pre(n);
    if(inv) {
        reverse(a.begin() + 1, a.end());
    }
    for(int i = 0; i < n; i++) {
        int to = bits[i];
        if(to > i) {
            swap(a[to], a[i]);
        }
    }
    for(int len = 1; len < n; len *= 2) {
        for(int i = 0; i < n; i += 2 * len) {
            for(int j = 0; j < len; j++) {
                Complex u = a[i + j], v = a[i + j + len] * root[len + j];
                a[i + j] = u + v;
                a[i + j + len] = u - v;
            }
        }
    }
    if(inv) {
        for(int i = 0; i < n; i++)
            a[i] /= n;
    }
    return a;
}

void fft2in1(CVector &a, CVector &b) {
    int n = (int) a.size();
    for(int i = 0; i < n; i++) {
        a[i] = Complex(a[i].real, b[i].real);
    }
    auto c = fft(a);
    for(int i = 0; i < n; i++) {
        a[i] = (c[i] + c[(n-i) % n].conj()) * Complex(0.5, 0);
        b[i] = (c[i] - c[(n-i) % n].conj()) * Complex(0, -0.5);
    }
}

void ifft2in1(CVector &a, CVector &b) {
    int n = (int) a.size();
    for(int i = 0; i < n; i++) a[i] = a[i] + b[i] * Complex(0, 1);
    a = fft(a, true);
    for(int i = 0; i < n; i++) {
        b[i] = Complex(a[i].imag, 0);
        a[i] = Complex(a[i].real, 0);
    }
}

vector<long long> mod_mul(const vector<long long> &a, const vector<long
    long> &b, long long cut = 1 << 15) {
    int n = (int) a.size();
    CVector C[4];
    for(int i = 0; i < 4; i++) C[i].resize(n);
    for(int i = 0; i < n; i++) {
        C[0][i] = a[i] % cut;
        C[1][i] = a[i] / cut;
        C[2][i] = b[i] % cut;
        C[3][i] = b[i] / cut;
    }
    fft2in1(C[0], C[1]);
    fft2in1(C[2], C[3]);
    for(int i = 0; i < n; i++) {
        // 00, 01, 10, 11
        Complex cur[4];
        for(int j = 0; j < 4; j++) cur[j] = C[j/2+2][i] * C[j % 2][i];
        for(int j = 0; j < 4; j++) C[j][i] = cur[j];
    }
    ifft2in1(C[0], C[1]);
    ifft2in1(C[2], C[3]);
    vector<long long> ans(n, 0);
    for(int i = 0; i < n; i++) {
        // if there are negative values, care with rounding
        ans[i] += (long long) (C[0][i].real + 0.5);
        ans[i] += (long long) (C[1][i].real + C[2][i].real + 0.5) * cut;
        ans[i] += (long long) (C[3][i].real + 0.5) * cut * cut;
    }
    return ans;
}

// Function to perform the multiplication of polynomials
vector<int> mul(const vector<int> &a, const vector<int> &b) {
    int n = 1;
```

```
    while (n - 1 < (int) a.size() + (int) b.size() - 2) n += n;
    CVector poly(n);
    for(int i = 0; i < n; i++) {
        if(i < (int) a.size()) {
            poly[i].real = a[i];
        }
        if(i < (int) b.size()) {
            poly[i].imag = b[i];
        }
    }
    poly = fft(poly);
    for(int i = 0; i < n; i++) {
        poly[i] *= poly[i];
    }
    poly = fft(poly, true);
    vector<int> c(n, 0);
    for(int i = 0; i < n; i++) {
        c[i] = (int) (poly[i].imag / 2 + 0.5);
    }
    while (c.size() > 0 && c.back() == 0) c.pop_back(); // Attention here
    return c;
}

/*

Time Complexity:

fft  -> O(N*logN)

Links:

https://unacademy.com/class/fft-and-convulutions/AFIJV6BI
https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-
    multiplication/
https://cp-algorithms.com/algebra/fft.html

Applications:

1. All possible sums
2. All possible scalar products
3. Two stripes
4. String matching
5. String matching with wildcards

*/
```

4.4 Gauss

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const double eps = 1e-9;

// "a" is the system matrix. The last column of this matrix is vector b
// 0    -> 0 solutions
// 1    -> 1 solution
// 2    -> INF solutions
int gauss(vector<vector<double>> a, vector<double>& answ){
    int n = (int) a.size();         // # equations
    int m = (int) a[0].size() - 1;  // # variables

    vector<int> where(m, -1);
    for(int col = 0, row = 0; col < m && row < n; col++){
        int sel = row;
        for(int i = row; i < n; i++){
            if(abs(a[i][col]) > abs(a[sel][col])) sel = i;
        }
        if(abs(a[sel][col]) < eps) continue;
        for(int i = col; i <= m; i++){
            swap(a[sel][i], a[row][i]);
        }
        where[col] = row;

        for(int i = 0; i < n; i++){
            if(i != row){
                double c = a[i][col] / a[row][col];
                for(int j = col; j <= m; j++){
                    a[i][j] -= a[row][j] * c;
                }
            }
        }
        row++;
    }

    answ.assign(m, 0);
    for(int i = 0; i < m; i++){
        if(where[i] != -1)
            answ[i] = a[where[i]][m] / a[where[i]][i];
    }
    for(int i = 0; i < n; i++){
        double sum = 0;
        for(int j = 0; j < m; j++){
            sum += answ[j] * a[i][j];
        }
        if(abs(sum - a[i][m]) > eps) return 0;
    }
    for(int i = 0; i < m; i++){
        if(where[i] == -1) return 2;
    }
    return 1;
}


/*

Time Complexity:

gauss   -> O(n^3)

Links:

https://cp-algorithms.com/linear_algebra/linear-system-gauss.html
https://codeforces.com/blog/entry/60003

*/
```

4.5 Xor Gauss

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

template<class T = unsigned int>
class XorGauss{
    public:
    int d, sz;
    vector<T> basis;
    XorGauss(int n) : d(n){
        basis.assign(d, 0);
        sz = 0;
    }
    bool add(T mask){
        for(int i = d - 1; i >= 0; i--){
            if((mask & (((T)1) << i)) == 0) continue;
            if(basis[i]) mask ^= basis[i];
            else{
                basis[i] = mask;
                sz++;
```

```
                return true;
            }
        }
        return false;
    }
    // Returns the smallest possible value of a vector
    // by subtracting a linear combination from the basis
    T reduce(T mask){
        for(int i = d - 1; i >= 0; i--){
            mask = min(mask, mask ^basis[i]);
        }
        return mask;
    }
    // Returns the largest possible value of a vector
    T augment(T mask){
        return ~reduce(~mask);
    }
    // Checks whether the vector can be formed by a linear combination of
        the basis
    bool check(T mask){
        for(int i = d - 1; i >= 0; i--){
            if((mask & (((T)1) << i)) == 0) continue;
            if(!basis[i]) return false;
            mask ^= basis[i];
        }
        return true;
    }
};

/*

Time Complexity:

add ->  O(d)

Links:

https://codeforces.com/blog/entry/68953

*/
```

4.6 Extended GCD

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Extended Euclidean Algorithm
struct GCD_type { int x, y, d; };
GCD_type ex_GCD(int a, int b){
    if(b == 0) return {1, 0, a};
    auto answ = ex_GCD(b, a % b);
    return {answ.y, answ.x - a / b * answ.y, answ.d};
}

/*

Time Complexity:

ex_GCD ->  O(log(min(a, b))

Links:

https://cp-algorithms.com/algebra/extended-euclid-algorithm.html
https://www.math.cmu.edu/~bkell/21110-2010s/extended-euclidean.html
https://cp-algorithms.com/algebra/linear-diophantine-equation.html

*/
```

4.7 Chinese Remainder Theorem

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
using ii = pair<int,int>;

// Chinese Remainder Theorem
struct CRT{
    vector<int> a, m;
    struct GCD_type { int x, y, d; };

public:
    CRT(vector<int> remainders, vector<int> mods){
        int n = remainders.size();
        a.assign(n, 0);
        m.assign(n, 0);
        for(int i = 0; i < n; i++){
            a[i] = normalize(remainders[i], mods[i]);
            m[i] = mods[i];
        }
    }

    ii solve(){
        int n = a.size();
        int answ = a[0];
        int lcm = m[0];

        for(int i = 1; i < n; i++){
            auto gcd = ex_GCD(lcm, m[i]);
            int xi = gcd.x;
            int d = gcd.d;
            if((a[i] - answ) % d != 0) return {-1, -1}; // No solution
            answ = normalize(answ + xi * (a[i] - answ) / d % (m[i] / d) *
                lcm,
                             lcm * m[i] / d);
            lcm = LCM(lcm, m[i]);
        }

        return {answ, lcm}; // x = answ (mod lcm)
    }

private:
    int GCD(int a, int b){
        return (b == 0) ? a : GCD(b, a % b);
    }

    int LCM(int a, int b){
        return a / GCD(a, b) * b;
    }

    int normalize(int x, int mod){
        x %= mod;
        if(x < 0) x += mod;
        return x;
    }

    GCD_type ex_GCD(int a, int b){
        if(b == 0) return {1, 0, a};
        auto answ = ex_GCD(b, a % b);
        return {answ.y, answ.x - a / b * answ.y, answ.d};
    }
};

/*

Time Complexity:

CRT ->  O(nlog(LCM(n1,n2,...,nn)))
```

```
Links:

https://cp-algorithms.com/algebra/chinese-remainder-theorem.html
https://codeforces.com/blog/entry/61290

*/
```

4.8 Discrete Log

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
const int INF = 0x3f3f3f3f3f3f3f3f;

// Returns minimum "x" for which a^x % m = b % m , (gcd(a, m) = 1)
// Returns "INF" if there is no solution
// 0^0 = 1 is assumed here
int discreteLog(int a, int b, int m){
    a %= m, b %= m;
    int block = sqrt(m) + 1, an = 1;
    for(int i = 0; i < block; i++) an = an * a % m;
    map<int,int> vals;
    for(int q = 0, cur = b; q <= block; q++){ // b*a^q % m
        vals[cur] = q;
        cur = cur * a % m;
    }
    int answ = INF;
    for(int p = 1, cur = 1; p <= block; p++){
        cur = cur * an % m; // a^(block*p)
        if(vals.count(cur)){ // a^(block*p) % m = b*a^q % m
            answ = min(answ, block*p - vals[cur]);
            // return answ; // This value is already the minimum
        }
    }
    return answ;
}

// gcd(a, m) != 1
int discreteLog(int a, int b, int m){
    a %= m, b %= m;
    int k = 1, add = 0, g;
    while((g = gcd(a, m)) > 1){ // log^2(m) preprocess
        if(b == k) return add;
        if(b % g) return INF;
        b /= g, m /= g, add++;
        k = (k * a / g) % m;
    }
    int block = sqrt(m) + 1, an = 1;
    for(int i = 0; i < block; i++) an = an * a % m;
    map<int,int> vals;
    for(int q = 0, cur = b; q <= block; q++){
        vals[cur] = q;
        cur = cur * a % m;
    }
    int answ = INF;
    for(int p = 1, cur = k; p <= block; p++){
        cur = cur * an % m;
        if(vals.count(cur)){
            answ = min(answ, block*p - vals[cur] + add);
            // return answ; // This value is already the minimum
        }
    }
    return answ;
}

/*

Time Complexity:

discreteLog -> O(sqrt(m) * log(m))
discreteLog -> O(log^2(m) + sqrt(m) * log(m))

Links:

https://cp-algorithms.com/algebra/discrete-log.html
https://en.wikipedia.org/wiki/Baby-step_giant-step

*/
```

4.9 Fast Exponentiation

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Fast Modular Exponentiation
int fastModExp(int x, int y, int m){
    int ans = 1;
    x = x % m;
    while(y){
        if(y & 1) ans = ans * x % m;
        x = x * x % m;
        y = y >> 1;
    }
    return ans;
}

/*

Time Complexity:

fastModExp  -> O(logy)

Links:

https://www.youtube.com/watch?v=HN7ey_-A7o4
https://www.youtube.com/watch?v=-3Lt-EwR_Hw
https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-
    arithmetic/

Obs.:
We can reduce the exponent using Fermat's Little Theorem.
Example: a^(b^c), we can say that (b^c) = x*(m - 1) + y, then a^(b^c) = a^y
    . Being y = (b^c) % (m - 1)

*/
```

4.10 Multiplicative Inverse

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

// If we know MOD is prime, then we can use Fermats's little theorem to
    find the inverse.
int ModMultInv(int n){ return fastModExp(n, MOD-2, MOD); }

/*

Time Complexity:

ModMultInv -> O(logMOD)

Links:
```

```
https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/

*/
```

4.11 Miller Rabin - Pollard Rho

```
#include <bits/stdc++.h>
using namespace std;

// This code becomes inefficient for numbers greater to 10^20
// To numbers greater than 2^64 use __int128
typedef unsigned long long ull;

// Miller Rabin

ull mul(ull a, ull b, ull mod){
    ull answ = 0;
    for(a %= mod, b %= mod; b != 0;
        b >>= 1, a <<= 1, a = a >= mod ? a - mod : a){
        if(b & 1){
            answ += a;
            if(answ >= mod) answ -= mod;
        }
    }
    return answ;
}

ull mpow(ull a, ull b, ull mod){
    ull answ = 1;
    for(; b; b >>= 1, a = mul(a, a, mod))
        if(b & 1) answ = mul(answ, a, mod);
    return answ;
}

bool witness(ull a, ull k, ull q, ull n){
    ull t = mpow(a, q, n);
    if(t == 1 || t == n-1) return false;
    for(int i = 0; i < k - 1; i++){
        t = mul(t, t, n);
        if(t == 1) return true;
        if(t == n - 1) return false;
    }
    return true;
}

vector<ull> test = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
bool isPrime(ull n){
    if(n == 2) return true;
    if(n < 2 || !(n & 1)) return false;
    ull q = n - 1, k = 0;
    while(!(q & 1)) q >>= 1, k++;
    for(ull a : test){ // Maybe larger numbers than 1e9 is enough to
        generate 2 numbers rand()%(n-4) + 2;
        if(n == a) return true;
        if(witness(a, k, q, n)) return false;
    }
    return true;
}

// Pollard Rho

ull pollard_rho(ull n, ull c){
    ull x = 2, y = 2, i = 1, k = 2, d;
    while(1){
        x = (mul(x, x, n) + c);
        if(x >= n) x -= n;
        d = __gcd(x - y, n);
        if(d > 1) return d;
        if(++i == k) y = x, k <<= 1;
    }
    return n;
}

void factorize(vector<ull>& answ, ull n){
    if(n == 1) return;
    if(isPrime(n)){
        answ.push_back(n);
        return;
    }
    ull d = n;
    for(int i = 2; d == n; i++) d = pollard_rho(n, i);
    factorize(answ, d);
    factorize(answ, n/d);
}

// cin/cout to deal with __int128

istream& operator>>(istream& in, ull &x){
    static char s[40];
    in >> s;
    x = 0;
    for(char* p = s; *p; ++p) x = 10 * x + *p - '0';
    return in;
}

ostream& operator<<(ostream& out, ull x){
    static char s[40] = {};
    char* p = s + (sizeof(s) - 1);
    while (*--p = (char)(x % 10 + '0'), x /= 10, x);
    return out << p;
}

/*

Time Complexity

Miller_Rabin    -> O(K*log^3(N)) Where N is the number to be checked for
    primality, and K is the number of checks to get accuracy
Pollard_Rho     -> O(n^1/4)

Links:

Miller Rabin

https://cp-algorithms.com/algebra/primality_tests.html#deterministic-
    version
https://www.geeksforgeeks.org/multiply-large-integers-under-large-modulo/
https://www.youtube.com/watch?v=qdylJqXCDGs
https://www.youtube.com/watch?v=zmhUlVck3J0

Pollard Rho

https://cp-algorithms.com/algebra/factorization.html
https://www.youtube.com/watch?v=6khEMeU8Fck

*/
```

4.12 Miller - Rho Iterative

```
#include <bits/stdc++.h>
using namespace std;

// This code becomes inefficient for numbers greater to 10^20
// To numbers greater than 2^64 use __int128 and maybe __float128
typedef unsigned long long ull;
typedef long double ld;
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

// Miller Rabin Primality Test

// Just a multiplication to avoid overflow
ull fmul(ull a, ull b, ull m){
    ull q = (ld) a * (ld) b / (ld) m;
```

```
    ull r = a * b - q * m;
    return (r + m) % m;
}

// Fast Modular Exponentiation
ull fexp(ull x, ull y, ull m){
    ull answ = 1;
    x = x % m;
    while(y){
        if(y & 1) answ = fmul(answ, x, m);
        x = fmul(x, x, m);
        y = y >> 1;
    }
    return answ;
}

// Validation by Fermat's little Theorem
// a^(p-1) - 1 = 0 mod p
// (a^((p-1)/2) - 1)*(a^((p-1)/2) + 1) = 0 mod p
bool miller(ull p, ull a){
    ull s = p - 1;
    while(s % 2 == 0) s >>= 1;
    while(a >= p) a >>= 1;
    ull mod = fexp(a, s, p);
    while(s != p - 1 && mod != 1 && mod != p-1){
        mod = fmul(mod, mod, p);
        s <<= 1;
    }
    if(mod != p - 1 && s % 2 == 0) return false;
    else return true;
}

// Deterministic Miller Rabin algorithm
// We need to check for different bases "a" to increase the probability of
    hit
// For values greater than 2^64 add more bases
vector<ull> test = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
bool prime(ull p){
    if(p < 2) return false;
    if(p % 2 == 0) return (p == 2);
    for(ull a : test){
        if(p == a) return true;
        if(!miller(p, a)) return false;
    }
    return true;
}

// Pollard Rho

// Function used in Pollard Rho f(x) = x^2 + c
ull func(ull x, ull c, ull n){
    return (fmul(x, x, n) + c) % n;
}

ull gcd(ull a, ull b){
    if(!b) return a;
    else return gcd(b, a % b);
}

// Pollard Rho algorithm to discover a factor of n
ull rho(ull n){
    if(n % 2 == 0) return 2;
    if(prime(n)) return n;
    while(1){
        ull c;
        do {
            c = uniform_int_distribution<ull>(0, n - 1)(rng);
        } while(c == 0 || (c + 2) % n == 0);
        ull x = 2, y = 2, d = 1;
        ull pot = 1, lam = 1;
        do {
            if(pot == lam){
                x = y;
                pot <<= 1;
                lam = 0;
            }
            y = func(y, c, n);
            lam++;
            d = gcd(x >= y ? x - y : y - x, n);
        } while(d == 1);
        if(d != n) return d;
    }
}

// Return all the factors of n
vector<ull> factors(ull n){
    vector<ull> answ, rest, times;
    if(n == 1) return answ;
    rest.push_back(n);
    times.push_back(1);
    while(!rest.empty()){
        ull x = rho(rest.back());
        if(x == rest.back()){
            int freq = 0;
            for(int i = 0; i < rest.size(); i++){
                int cur_freq = 0;
                while(rest[i] % x == 0){
                    rest[i] /= x;
                    cur_freq++;
                }
                freq += cur_freq * times[i];
                if(rest[i] == 1){
                    swap(rest[i], rest.back());
                    swap(times[i], times.back());
                    rest.pop_back();
                    times.pop_back();
                    i--;
                }
            }
            while(freq--){
                answ.push_back(x);
            }
            continue;
        }
        ull e = 0;
        while(rest.back() % x == 0){
            rest.back() /= x;
            e++;
        }
        e *= times.back();
        if(rest.back() == 1){
            rest.pop_back();
                        times.pop_back();
        }
        rest.push_back(x);
                times.push_back(e);
    }
    return answ;
}

/*

Time Complexity

Miller_Rabin    -> O(K*log^3(N)) Where N is the number to be checked for
    primality, and K is the number of checks to get accuracy
Pollard_Rho     -> O(n^1/4)

Links:

Miller Rabin

https://cp-algorithms.com/algebra/primality_tests.html#deterministic-
    version
https://www.geeksforgeeks.org/multiply-large-integers-under-large-modulo/
https://www.youtube.com/watch?v=qdylJqXCDGs
https://www.youtube.com/watch?v=zmhUlVck3J0
```

```
Pollard Rho

https://cp-algorithms.com/algebra/factorization.html
https://www.youtube.com/watch?v=6khEMeU8Fck

*/
```

4.13 Pollard Rho with Montgomery

```
#include <bits/stdc++.h>
using namespace std;

// Algorithm for factoring numbers up to 10^29
// I don't know how it works yet

typedef unsigned long long u64;
typedef __int128_t i128;
typedef __uint128_t u128;

struct u256 {
  u128 hi, lo;
  static u256 mult(u128 x, u128 y) {
    u128 a = x >> 64, b = (u64)x;
    u128 c = y >> 64, d = (u64)y;
    u128 ac = a * c;
    u128 ad = a * d;
    u128 bc = b * c;
    u128 bd = b * d;
    u128 carry = (u128)(u64)ad + (u64)bc + (bd >> 64u);
    u128 h = ac + (ad >> 64u) + (bc >> 64u) + (carry >> 64u);
    u128 l = (ad << 64u) + (bc << 64u) + bd;
    return {h, l};
  }
};

struct Montgomery {
  u128 n, inv, r2;
  explicit Montgomery(u128 _n) : n(_n), inv(1), r2(-n % n) {
    assert(n & 1);
    for (int i = 0; i < 7; ++i) inv *= 2 - n * inv;
    for (int i = 0; i < 4; ++i) if ((r2 <<= 1) >= n) r2 -= n;
    for (int i = 0; i < 5; ++i) r2 = mult(r2, r2);
  }
  u128 init(u128 x) { return mult(x, r2); }
  u128 mult(u128 a, u128 b) { return reduce(u256::mult(a, b)); }
  u128 reduce(u256 x) {
    i128 a = x.hi - u256::mult(x.lo * inv, n).hi;
    return (a < 0) ? a + n : a;
  }
};

istream& operator>>(istream& in, u128 &x) {
  static char s[40];
  in >> s;
  x = 0;
  for (char* p = s; *p; ++p) x = 10 * x + *p - '0';
  return in;
}

ostream& operator<<(ostream& out, u128 x) {
  static char s[40] = {};
  char* p = s + (sizeof(s) - 1);
  while (*--p = (char)(x % 10 + '0'), x /= 10, x);
  return out << p;
}

#define rand() uid(rng)
mt19937 rng(chrono::high_resolution_clock::now().time_since_epoch().count()
    );
uniform_int_distribution<int> uid(0, numeric_limits<int>::max());

inline u128 gcd(u128 a, u128 b) {
  if (b != 0) while ((b ^= a ^= b ^= a %= b));
  return a;
}

inline u128 add(u128 a, u128 b, u128 m) { return (a += b) >= m ? a - m : a;
      }
inline u128 sub(u128 a, u128 b, u128 m) { return a < b ? a + m - b : a - b;
      }

u128 mult(u128 a, u128 b, u128 m) {
  u128 x = 0;
  while (b) {
    if (b & 1) x = add(x, a, m);
    b >>= 1;
    a = add(a, a, m);
  }
  return x;
}

u128 mpow(u128 a, u128 b, u128 mod) {
  u128 x = 1;
  while (b) {
    if (b & 1) x = mult(x, a, mod);
    b >>= 1;
    a = mult(a, a, mod);
  }
  return x;
}

u128 isqrt(u128 n) {
  u128 x = n, y = 1;
  while (x > y) {
    x = (x + y) >> 1;
    y = n / x;
  }
  return x;
}

bool isPrime(u128 n) {
  if (n < 2) return 0;
  if ((n & 1) == 0) return n == 2;
  u128 d = n - 1;
  int r = 0;
  while ((d & 1) == 0) {
    d >>= 1;
    ++r;
  }
  for (u128 a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) if ((a
      %= n)) {
    a = mpow(a, d, n);
    if (a == 1 || a == n - 1) continue;
    for (int i = 1; i < r && 1 < a && a != n - 1; ++i) a = mult(a, a, n);
    if (a != n - 1) return 0;
  }
  return 1;
}

u128 rho(u128 n) {
  if (isPrime(n)) return 1;
  if ((n & 1) == 0) return 2;
  u128 r = isqrt(n);
  if (r * r == n) return r;
  Montgomery mont(n);
  const int m = n < (1ull << 60) ? 32 : 512;
  const auto one = mont.init(1);
  const auto c = mont.init((rand() & 1023) + 1);
  auto f = [&](u128 x){ return add(mont.mult(x, x), c, n); };
  u128 x = 0, y = one;
  for (int l = 1; ; l <<= 1) {
    if (x == y) y = mont.init(rand());
    x = y;
    for (int i = 0; i < l; ++i) y = f(y);
```

```
    u128 ys = y, q = one, g = 1;
    for (int k = 0; k < l && g == 1; k += m) {
      ys = y;
      for (int i = min(m, l - k); i; --i) {
        y = f(y);
        q = mont.mult(q, sub(y, x, n));
      }
      g = gcd(n, q);
    }
    if (g == n) {
      y = ys;
      for (g = 1; g == 1; g = gcd(n, sub(y, x, n))) y = f(y);
    }
    if (g != 1 && g != n) return g;
  }
}

map<u128,int> findFactors(u128 n) {
  assert(n > 1);
  u128 x = rho(n);
  if (x == 1) return {{n, 1}};
  auto a = findFactors(x);
  auto b = findFactors(n / x);
  if (a.size() < b.size()) swap(a, b);
  for (auto i : b) a[i.first] += i.second;
  return a;
}

signed main() {
  // assert(freopen("in", "r", stdin));
  cin.sync_with_stdio(0), cin.tie(0), cout.tie(0);
  u128 n;
  while (cin >> n && n) {
    bool flag = 0;
    map<u128,int> aux = findFactors(n);
    for (auto i : aux) {
      if (flag) cout << ' ';
      flag = 1;
      cout << i.first << '^' << i.second;
    }
    cout << '\n';
  }
  cerr << (double)clock() / CLOCKS_PER_SEC << endl;
  return 0;
}
```

5 Geometry

5.1 Geometry

```
#include <bits/stdc++.h>
using namespace std;

const double inf = 1e100, eps = 1e-12;
const double PI = acos(-1.0L);

// Note: Whenever possible, work only with integers!
// If integers are used, the "cmp" function will no longer work.
// It's necessary to remove it from all locations!
int cmp(double a, double b = 0){
    if(abs(a-b) < eps) return 0;
    return (a < b) ? -1 : +1;
}

// Struct to represent a point/vector
struct PT{
    double x, y;
    PT(double x = 0, double y = 0) : x(x), y(y) {}
    PT operator + (const PT &p) const { return PT(x + p.x, y + p.y); }
    PT operator - (const PT &p) const { return PT(x - p.x, y - p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }
    bool operator < (const PT &p) const {
        if(cmp(x, p.x) != 0) return x < p.x;
        return cmp(y, p.y) < 0;
    }
    bool operator == (const PT &p) const { return !cmp(x, p.x) && !cmp(y, p
        .y); }
    bool operator != (const PT &p) const { return !(p == *this); }
};

// Debug function
ostream &operator << (ostream &os, const PT &p){
    return os << "(" << p.x << "," << p.y << ")";
}

// Function to calculate the dot product (u.v)
double dot (PT p, PT q) { return p.x * q.x + p.y * q.y; }

// Function to calculate the cross product (uXv) (2x2 determinant)
double cross (PT p, PT q) { return p.x * q.y - p.y * q.x; }

// Function to calculate the magnitude of the vector (|u|)
double norm (PT p) { return hypot(p.x, p.y); }

// Function to calculate the distance of 2 points
double dist (PT p, PT q) { return hypot(p.x - q.x, p.y - q.y); }

// Function to project point c onto segment a - b
PT projPtInSegment(PT a, PT b, PT c){
    double r = dot(b - a, b - a);
    if(cmp(r) == 0) return a;
    r = dot(b - a, c - a) / r;
    if(cmp(r, 0) < 0) return a;
    if(cmp(r, 1) > 0) return b;
    return a + (b - a) * r;
}

// Function to calculate the minimum distance between point c and segment a
    -b
double distPtSegment(PT a, PT b, PT c){
    return dist(c, projPtInSegment(a, b, c));
}

// Returns true if a point c is on segment a-b
bool ptInSegment(PT a, PT b, PT c){
    if(a == b) return a == c;
    a = a - c, b = b - c;
    return cmp(cross(a, b)) == 0 && cmp(dot(a, b)) <= 0;
}

// Returns true if 2 lines are parallel
bool parallel(PT a, PT b, PT c, PT d){
    return cmp(cross(b - a, c - d)) == 0;
}

// Returns true if two segments are collinear
bool collinear(PT a, PT b, PT c, PT d){
    return parallel(a, b, c, d) && cmp(cross(a - b, a - c)) == 0 && cmp(
        cross(c - d, c - a)) == 0;
}

// Returns true if two segments are intersecting at some point
bool segmentIntersection(PT a, PT b, PT c, PT d){
    if(collinear(a, b, c, d)){
        if(a == c || a == d || b == c || b == d) return true;
        if(cmp(dot(c - a, c - b)) > 0 && cmp(dot(d - a, d - b)) > 0 && cmp(
            dot(c - b, d - b)) > 0) return false;
        return true;
    }
    if(cmp(cross(d - a, b - a) * cross(c - a, b - a)) > 0) return false;
    if(cmp(cross(a - c, d - c) * cross(b - c, d - c)) > 0) return false;
    return true;
```

```
}

// Returns the position of the point relative to the simple polygon
// Polygon is simple if its boundary doesn't cross itself
// Returns -1 if the point is strictly inside the polygon
// Returns 0 if the point is on the boundary
// Returns 1 if the point is outside the polygon
int pointInPolygon(const vector<PT> &p, PT q){ // O(n)
    int windingNumber = 0;
    for(int i = 0; i < p.size(); i++){
        if(p[i] == q) return 0;
        int j = (i + 1) % p.size();
        if(p[i].y == q.y && p[j].y == q.y){ // "Throwing" the horizontal
             ray
            if(ptInSegment(p[i], p[j], q))
                return 0;
        }
        else{
            bool below = p[i].y < q.y;
            if(below == (p[j].y < q.y)) continue;
            auto orientation = cross(q - p[i], p[j] - p[i]);
            if(cmp(orientation) == 0) return 0;
            if(below == (orientation < 0)) windingNumber += below ? 1 : -1;
        }
    }
    return windingNumber == 0 ? 1 : -1;
}

// Polygon area using Shoelace Formula.
// Points need to be sorted in clockwise or counterclockwise.
double signedPolygonArea(const vector<PT> &p){
    double area = 0;
    for(int i = 0; i < p.size(); i++){
        int j = (i + 1) % p.size();
        area += p[i].x * p[j].y - p[j].x * p[i].y;
    }
    return area/2.0;
}

double polygonArea(const vector<PT> &p){ return abs(signedPolygonArea(p));
     }
```

5.2 Polar Sort

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

int quadrant(PT p){
    if(p.x < 0 && p.y < 0) return 0;
    if(p.x >= 0 && p.y < 0) return 1;
    if(p.x >= 0 && p.y >= 0) return 2;
    return 3;   // p.x < 0 && p.y >= 0
}

struct polar_sort{
    PT pivot;
    polar_sort (PT p): pivot(p) {}
    bool operator ()(PT a, PT b) const {
        a = a - pivot;
        b = b - pivot;
        if(quadrant(a) != quadrant(b)) return quadrant(a) < quadrant(b);
        return cross(a, b) > 0;
    }
};

// Call using
// sort(v.begin(), v.end(), polar_sort(pivot));
```

```
/*

Time Complexity:

sort    -> O(nlogn)

Links:

https://codeforces.com/blog/entry/72815

*/
```

5.3 Convex Hull

```
#include <bits/stdc++.h>
using namespace std;

#include "./Geometry.cpp"

// Monotone chain Algorithm to calculate Convex Hull
vector<PT> convexHull(vector<PT> p, bool needSort = 1){
    if(needSort) sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end()), p.end());
    int n = p.size(), k = 0;
    if(n <= 1) return p;
    vector<PT> answ(n + 2);   // Must be 2*n + 1 for collinear points

    // Lower hull
    for(int i = 0; i < n; i++){
        while(k >= 2 && cross(answ[k - 1] - answ[k - 2], p[i] - answ[k -
             2]) <= 0) k--; // If collinear points are allowed put only "<"
        answ[k++] = p[i];
    }

    // Upper hull
    for(int i = n - 2, t = k + 1; i >= 0; i--) {
        while(k >= t && cross(answ[k - 1] - answ[k - 2], p[i] - answ[k -
             2]) <= 0) k--; // If collinear points are allowed put only "<"
        answ[k++] = p[i];
    }

    answ.resize(k); // n+1 points where the first is equal to the last
    return answ;
}

/*

Time Complexity

convexHull  -> O(nlogn)

Links:

https://cp-algorithms.com/geometry/convex-hull.html#implementation_1
https://www.youtube.com/watch?v=JS-eBdqb1uM

*/
```

6 String Algorithms

6.1 Rabin Karp Hash

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
```

```cpp
// Implementation of Rabin-Karp algorithm
class RabinKarp{
    const uint64_t MOD = (1LL << 61) - 1;
    const int base = 31;
    int n;
    vector<uint64_t> h, p;

public:
    RabinKarp(string &s){ // Initializing
        this->n = s.size();
        p.assign(n, 0);
        h.assign(n, 0);

        p[0] = 1;
        h[0] = getInt(s[0]);
        for(int i = 1; i < n; i++){
            p[i] = modMul(p[i-1], base);
            h[i] = (modMul(h[i-1], base) + getInt(s[i])) % MOD;
        }
    }

    uint64_t getKey(int l, int r){    // [l, r]
        uint64_t answ = h[r];
        if(l > 0) answ = (answ + MOD - modMul(p[r - l + 1], h[l - 1])) %
            MOD;
        return answ;
    }

private:
    uint64_t getInt(char c){ // Attention: leave the subtraction if there
         are only letters
        return c - 'a' + 1;
    }

    uint64_t modMul(uint64_t a, uint64_t b) {
        uint64_t l1 = (uint32_t)a, h1 = a >> 32, l2 = (uint32_t)b, h2 = b
            >> 32;
        uint64_t l = l1 * l2, m = l1 * h2 + l2 * h1, h = h1 * h2;
        uint64_t ret = (l & MOD) + (l >> 61) + (h << 3) + (m >> 29) + ((m
            << 35) >> 3) + 1;
        ret = (ret & MOD) + (ret >> 61);
        ret = (ret & MOD) + (ret >> 61);
        return ret - 1;
    }
};

/*

Time Complexity:

RabinKarp   -> O(N)
getKey      -> O(1)

Links:

https://cp-algorithms.com/string/string-hashing.html#applications-of-
    hashing
https://cp-algorithms.com/string/rabin-karp.html
https://www.youtube.com/watch?v=qQ8vS2btsxI
https://codeforces.com/blog/entry/60445

*/
```

6.2 Z Algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Implementation of Z-function algorithm
struct ZFunction{
    vector<int> z;
    int n, patSz = -1;

    ZFunction(string str, string pattern = ""){ // Initializing
        if(!pattern.empty()){
            str = pattern + "$" + str;
            patSz = pattern.size();
        }
        n = str.size();
        z.assign(n, 0);

        int l = 0, r = 0;
        for(int i = 1; i < n; i++){ // Z-function
            if(i <= r){
                z[i] = min(r - i + 1, z[i - l]);
            }
            while(i + z[i] < n && str[z[i]] == str[i + z[i]]) z[i]++;
            if(i + z[i] - 1 > r){
                l = i;
                r = i + z[i] - 1;
            }
        }
    }

    vector<int> findPattern(){
        vector<int> answ;
        for(int i = 0; i < n; i++){
            if(z[i] == patSz) answ.push_back(i - patSz - 1);
        }
        return answ;
    }
};

/*

Time Complexity:

Z-function -> O(N)

Links:

https://cp-algorithms.com/string/z-function.html#efficient-algorithm-to-
    compute-the-z-function
https://www.youtube.com/watch?v=CpZh4eF8QBw

*/
```

6.3 KMP

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Implementation of Knuth-Morris-Pratt algorithm
template<typename T>
struct KMP{
    T pattern;
    int m;
    vector<int> lps;

public:
    KMP(T pattern){
        m = pattern.size();
        this->pattern = pattern;
        get_lps();
    }
```

```
    vector<int> findPattern(T text){
        vector<int> matches;
        int n = text.size(), last = 0;
        for(int i = 0; i < n; i++){
            int j = last;
            while(j > 0 && (j >= m || text[i] != pattern[j]))
                j = lps[j - 1];

            last = (j < m && text[i] == pattern[j]) ? j + 1 : 0;
            if(last == m) matches.push_back(i - m + 1);
        }
        return matches;
    }

private:
    // Find Longest proper prefix which is also suffix
    void get_lps(){
        lps.assign(m, 0);
        for(int i = 1; i < m; i++){
            int j = lps[i - 1];
            while(j > 0 && pattern[i] != pattern[j])
                j = lps[j - 1];
            if(pattern[i] == pattern[j]) lps[i] = j + 1;
        }
    }
};

/*

Time Complexity:

KMP -> O(n + m)

Links:

https://cp-algorithms.com/string/prefix-function.html
https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
https://www.youtube.com/watch?v=GTJr8OvyEVQ

*/
```

6.4 Trie

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int ms = 500010;
const int sigma = 26;

int trie[ms][sigma], terminal[ms], z = 1;

void insert(string &s){
    int curr = 0;
    for(int i = 0; i < s.size(); i++){
        int id = s[i] - 'a';
        if(!trie[curr][id]){
            trie[curr][id] = z++;
        }
        curr = trie[curr][id];
    }
    terminal[curr]++;
}

int count(string &s){
    int curr = 0;
    for(int i = 0; i < s.size(); i++){
        int id = s[i] - 'a';
        if(!trie[curr][id]) return false;
        curr = trie[curr][id];
    }
    return terminal[curr];
}

void remove(string &s){
    int curr = 0;
    for(int i = 0; i < s.size(); i++){
        int id = s[i] - 'a';
        if(!trie[curr][id]) return;
        curr = trie[curr][id];
    }
    terminal[curr]--;
}

// Print words in Trie with a given prefix
void print(int curr, string &answ){
    if(terminal[curr]) cout << answ << '\n';

    for(int i = 0; i < sigma; i++){
        if(trie[curr][i]){
            answ.push_back(i + 'a');
            print(trie[curr][i], answ);
            answ.pop_back();
        }
    }
}

/*

Time Complexity:

insert  -> O(n)
remove  -> O(n)
count   -> O(n)

Links:

https://www.youtube.com/watch?v=-urNrIAQnNo
https://www.youtube.com/watch?v=AXjmTQ8LEoI

*/
```

6.5 Aho Corasick

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

template<typename T = string, typename U = char, const int sigma = 26,
    const int OFFSET = 'a'>
struct AhoCorasick{
private:
    struct Node{
        int nxt[sigma];
        int link, slink, id;
        int parent;
        U label;

        Node(int parent = -1, U label = '?') : parent(parent), label(label)
            {
            memset(nxt, -1, sizeof nxt);
            link = slink = id = -1;
        }
        int & operator [](U label) { // Change according to the problem
            return nxt[label - OFFSET];
        }
    };

    int sz, patterns;
```

```
    vector<Node> trie;

public:
    AhoCorasick(){
        sz = 1;
        patterns = 0;
        trie.resize(1);
    }

    void insert(T & s){
        int node = 0;
        for(int i = 0; i < s.size(); i++){
            if(trie[node][s[i]] == -1){
                trie[node][s[i]] = sz++;
                trie.emplace_back(node, s[i]);
            }
            node = trie[node][s[i]];
        }
        trie[node].id = patterns++; // If it can repeat strings, it must be
             a list of IDs
    }

    int next(int node, U label){
        if(trie[node][label] == -1){
            if(!node) trie[node][label] = 0;
            else trie[node][label] = next(next_suffix(node), label);
        }
        return trie[node][label];
    }

    int next_suffix(int node){
        if(!node || !trie[node].parent) return 0;
        if(trie[node].link == -1){
            trie[node].link = next(next_suffix(trie[node].parent), trie[
                node].label);
        }
        return trie[node].link;
    }

    int next_terminal(int node){ // Super suffix link
        if(!node || !trie[node].parent) return 0;
        if(trie[node].slink == -1){
            trie[node].slink = next_suffix(node);
            if(trie[node].slink && trie[trie[node].slink].id == -1){
                trie[node].slink = next_terminal(trie[node].slink);
            }
        }
        return trie[node].slink;
    }
};

/*

Time Complexity:

add             -> O(|s|)
build           -> O(m * sigma) m = total length of its constituent strings
     (if using BFS)
next            -> O(1) amortized
next_suffix     -> O(1) amortized
next_terminal   -> O(1) amortized

Links:

https://cp-algorithms.com/string/aho_corasick.html
https://usaco.guide/adv/string-search?lang=cpp
https://www.youtube.com/watch?v=vpH5gSSapjI

*/
```

6.6 Suffix Array O(nlogn)

```
#include <bits/stdc++.h>
using namespace std;

// O(nlogn)
vector<int> suffix_array(string & s){
    s += '$';
    int n = s.size();
    int cntSz = max(n, 260);
    vector<int> sa(n), rnk(n);
    for(int i = 0; i < n; i++) sa[i] = i, rnk[i] = s[i];

    for(int k = 0; k < n; k ? k *= 2 : k++){
        vector<int> nsa(sa), nrnk(n), cnt(cntSz);

        for(int i = 0; i < n; i++) nsa[i] = (nsa[i] - k + n) % n, cnt[rnk[i
            ]]++;
        for(int i = 1; i < cntSz; i++) cnt[i] += cnt[i - 1];
        for(int i = n - 1; i >= 0; i--) sa[--cnt[rnk[nsa[i]]]] = nsa[i];

        for(int i = 1, r = 0; i < n; i++){
            r += rnk[sa[i]] != rnk[sa[i - 1]] || rnk[(sa[i] + k) % n] !=
                rnk[(sa[i - 1] + k) % n];
            nrnk[sa[i]] = r;
        }
        rnk = nrnk;
        if(rnk[sa[n - 1]] == n - 1) break;
    }
    return sa;
}

// Kasai lcp O(n)
// lcp[i] = lcp(sa[i], sa[i + 1])
vector<int> kasai(string & s, vector<int> & sa){
    int n = s.size(), h = 0;
    vector<int> rnk(n), lcp(n);
    for(int i = 0; i < n; i++) rnk[sa[i]] = i;
    for(int i = 0; i < n; i++){
        if(rnk[i] == n - 1) { h = 0; continue; }
        int nxt = sa[rnk[i] + 1];
        while(i + h < n && nxt + h < n && s[i + h] == s[nxt + h]) h++; //
            && s[i + h] != separator, if needed
        lcp[rnk[i]] = h;
        if(h) h--;
    }
    return lcp;
}
```

6.7 Suffix Array (alternative)

```
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define pii pair<int,int>

struct SuffixArray{
public:
    int n, realSz;
    string text;
    vector<int> sa, rank, lcp;
    vector<int> patStart, ps;
    char separator = 'A' - 1; // If you need, apply Suffix Array to
        integers
```

```
SuffixArray(string & s) : text(s), realSz(s.size()) {}

void add_pattern(string & s){
    n = text.size();
    text += separator + s;
    patStart.push_back(n + 1);
    separator--;
}

void build(){
    text += '$'; // Needs to be lexicographically smaller than all
        separators
    n = text.size();
    sa.resize(n);
    rank.resize(n);

    vector<pii> list;
    for(int i = 0; i < n; i++){
        list.push_back({text[i], i});
    }
    sort(list.begin(), list.end());
    for(int i = 0; i < n; i++) sa[i] = list[i].second;
    rank[sa[0]] = 0;
    int classe = 0;
    for(int i = 1; i < n; i++){
        if(text[sa[i]] != text[sa[i - 1]]) classe++;
        rank[sa[i]] = classe;
    }

    int k = 1;
    while(k < n){
        vector<int> aux(n);
        vector<int> count(n, 0);
        for(int i = 0; i < n; i++) count[rank[i]]++;
        for(int i = 1; i < n; i++) count[i] += count[i - 1];
        for(int i = n - 1; i >= 0; i--){
            int idx = (sa[i] - k + n) % n;
            aux[count[rank[idx]] - 1] = idx;
            count[rank[idx]]--;
        }
        swap(sa, aux);
        aux[sa[0]] = 0;
        classe = 0;
        for(int i = 1; i < n; i++){
            if(rank[sa[i]] != rank[sa[i - 1]] ||
               rank[(sa[i] + k) % n] != rank[(sa[i - 1] + k) % n]){
                classe++;
            }
            aux[sa[i]] = classe;
        }
        swap(rank, aux);
        k += k;
    }

    ps.assign(n, 0); // Prefix Sum of the suffixes that are part of the
        original text
    for(int i = 0; i < n; i++){
        ps[i] = sa[i] < realSz;
        if(i) ps[i] += ps[i - 1];
    }

    build_lcp();
}

// Kasai lcp O(n)
// lcp[i] = lcp(sa[i], sa[i + 1])
void build_lcp(){
    lcp.assign(n, 0);
    int h = 0;
    for(int i = 0; i < n; i++) rank[sa[i]] = i;
    for(int i = 0; i < n; i++){
        if(rank[i] == n - 1){
            h = 0;
            continue;
        }
        int nxt = sa[rank[i] + 1];
        while(i + h < n && nxt + h < n && text[i + h] == text[nxt + h])
            h++; // && text[i + h] != separator, if needed
        lcp[rank[i]] = h;
        if(h) h--;
    }
    // build_RMQ(lcp)
}

bool find(string & pat){
        int l = 0;
        int r = sa.size() - 1;
    for(int i = 0; i < pat.size(); i++){
        int st = l;
        int en = r;
        int answ = text.size();
        while(st <= en){
            int m = (st + en) >> 1;
            int idx = sa[m];
            if(text[idx + i] >= pat[i]){
                if(text[idx + i] == pat[i]){
                    answ = min(answ, m);
                }
                en = m - 1;
            }
            else{
                st = m + 1;
            }
        }
        if(answ == text.size()) return false;
        l = answ;
        st = answ;
        en = r;
        while(st <= en){
            int m = (st + en) >> 1;
            int idx = sa[m];
            if(text[idx + i] <= pat[i]){
                if(text[idx + i] == pat[i]){
                    answ = max(answ, m);
                }
                st = m + 1;
            }
            else{
                en = m - 1;
            }
        }
        r = answ;
    }
    return true;
}
};

/*

Time Complexity:

build   -> O(nlogn)
find    -> O(|pat|logn)

Links:

https://www.youtube.com/watch?v=Lu5sByCfPvE
https://usaco.guide/adv/suffix-array?lang=cpp
https://codeforces.com/edu/course/2/lesson/2/1

*/
```

7 Search Algorithms

7.1 Binary Search

```cpp
#include <bits/stdc++.h>
using namespace std;

int binarySearch(vector<int> & v, int l, int r, int x){
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (v[m] == x) return m;
        if (v[m] < x) l = m + 1;
        else r = m - 1;
    }
    return -1; // Can't find element "x"
}

// Finding the smallest solution
// Suppose that we wish to find the smallest value k that is a valid
    solution for a problem
// We know that check(x) is false when x < k and true when x >= k.
// The initial jump length "z" has to be large enough
int x = -1;
for(int b = z; b >= 1; b /= 2){
    while(!check(x+b)) x += b;
}
int k = x+1;

// Finding the maximum value
// BS can be used to find the maximun value for a function that is first
    increasing and then decreasing
// Not allowed that consecutive values of the function are equal.
int x = -1;
for(int b = z; b >= 1; b /=2){
    while(f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;

/*

Time Complexity:

Binary Search                   -> O(logn)
Finding the smallest solution   -> O(x*logz) x is the time complexity of
    function check();
Finding the maximum value       -> O(x*logz) x is the time complexity of
    function f();

Links:

https://www.geeksforgeeks.org/binary-search/

*/
```

7.2 Ternary Search

```cpp
#include <bits/stdc++.h>
using namespace std;

/*
Algorithm for finding the maximum of f(x)  which is unimodal on an interval
    [l,r]
Unimodal:
1. f strictly increases first, reaches a maximum (at a single point or over
    an interval), and then strictly decreases.
2. f strictly decreases first, reaches a minimum, and then strictly
    increases.
*/

// Real numbers [l, r]
double ternarySearch(double l, double r) {
    double eps = 1e-9;  // Set the error limit
    for(int i = 0; i < 200 && r-l > eps; i++){
        double m1 = (2*l + r)/3.0;
        double m2 = (l + 2*r)/3.0;
        if(f(m1) > f(m2)) l = m1;
        else r = m2;
    }
    return f(l);     // Return the maximum of f(x) in [l, r]
}

// Integers numbers [l, r]
int ternarySearch(int l, int r){
    int lo = l - 1;
    int hi = r;
    while(hi - lo > 1){
        int mid = (hi + lo) >> 1;
        if(f(mid) > f(mid + 1)) hi = mid;
        else lo = mid;
    }
    return f(lo + 1);
}

/*

Time Complexity:

ternarySearch   -> O(log3(N))

Links:

https://cp-algorithms.com/num_methods/ternary_search.html

*/
```

8 Miscellaneous

8.1 Kadane

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

// Kadane's Algorithm (Works in array's that have only negative numbers)
int kadane(int n, vector<int> &v){
    int answ = 0;    // If possible be an empty subarray
    int curr = 0;
    for(int i = 0; i < n; i++){
        curr = max(v[i], curr + v[i]); // We have two option, ours current
            sum is actually starting at v[i] or is the sum of previous
            subarray + v[i]
        answ = max(answ, curr);
    }
    return answ;
}

// Kadane circular array (Method 1)
int maxCircularSumMethod1(int n, vector<int> &v){

    // Ours answer don't have the corners (Just need to use Kadane)
    int max_kadane = kadane(n, v);

    // if maximum sum using standard kadane is less than 0
```

```
    if(max_kadane < 0)
      return max_kadane;

    // Ours answer can have the corners
    int max_wrap = 0;

    for(int i = 0; i < n; i++) {
        max_wrap += v[i];    // Calculate array-sum
        v[i] = -v[i];        // invert the array (change sign)
    }

    // Max sum with corner elements will be:
    // array-sum - (-max subarray sum of inverted array)
    max_wrap = max_wrap + kadane(n, v);

    // The maximum circular sum will be maximum of two sums
    return max(max_wrap, max_kadane);
}

// Kadane circular array (Method 2)
int maxCircularSumMethod2(int n, vector<int> &v){

    // Corner Case
    if (n == 1)
        return v[0];

    // Initialize sum variable which store total sum of the array.
    int totalSum = 0;
    for (int i = 0; i < n; i++) {
        totalSum += v[i];
    }

    // Initialize every variable with first value of array.
    int bestMax = v[0];
    int currSumMax = v[0];

    int bestMin = v[0];
    int currSumMin = v[0];

    // Concept of Kadane's Algorithm
    for (int i = 1; i < n; i++) {
        // Kadane's Algorithm to find Maximum subarray sum.
        currSumMax = max(v[i], currSumMax + v[i]);
        bestMax = max(bestMax, currSumMax);

        // Kadane's Algorithm to find Minimum subarray sum.
        currSumMin = min(v[i], currSumMin + v[i]);
        bestMin = min(bestMin, currSumMin);
    }

    // All values are negative, just return bestMax
    if (bestMin == totalSum)
        return bestMax;

    // Else, we will calculate the maximum value
    return max(bestMax, totalSum - bestMin);
}

/*

Time Complexity

Kadane's Algorithm -> O(n)

Links:

https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/
https://www.geeksforgeeks.org/maximum-contiguous-circular-sum/

*/
```

8.2 Submask Enumeration

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

void enumerate(int mask){
    for(int s = mask; ; s = (s - 1) & mask){
        // you can use submask "s"
        if(s == 0) break;
    }
}

/*

Time Complexity:

enumerate -> O(2^k) Being "k" the amount of active bits in mask
enumerate all masks -> O(3^n)

Links:

https://cp-algorithms.com/algebra/all-submasks.html

*/
```

8.3 Count Divisors

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int ms = 1e6 + 10;
vector<int> count_div(ms, 0);

// Preprocesses the number of divisors from all numbers up to ms
void countDivisors(){
    for(int i = 1; i < ms; i++){
        for(int j = i; j < ms; j += i){
            count_div[j]++;
        }
    }
}

/*

Time Complexity

countDivisors  -> O(n*logn) n = ms

Obs.:
If we have a very large number we need to use another algorithm that runs
    on O(n^(1/3))

*/
```

8.4 Next Great Element

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int inf = 0x3f3f3f3f;
```

```cpp
using ii = pair<int,int>;
using ll = long long;

const int N = 1e5; // Number of elements

vector<ll> v;

// NGE's answer from all elements in vector "v"
vector<ii> nxt(N,{inf,N});

// Find the Next Great Element for all array elements
// If an element does not exist it is defined as {inf, n}
void NGE(int n){

    // Push the first element {element, idx}
    stack<pair<ll,ll>> s;
    s.push({v[0],0});

    // Iterate for rest of the elements
    for (int i = 1; i < n; i++){

        if (s.empty()) {    // If the stack is empty just push the next
             element
            s.push({v[i],i});
            continue;
        }

        // if stack is not empty and the popped element is smaller than
             next (keep popping)
        while (s.empty() == false && s.top().first < v[i]){
            nxt[s.top().second] = {v[i],i};
            s.pop();
        }

        // Push "next" to stack so that we can find NGE for him
        s.push({v[i],i});
    }

    // The remaining elements in stack don't have NGE
    while (s.empty() == false) {
        nxt[s.top().second] = {inf,n};
        s.pop();
    }
}

// NGE with the fastest implementation using PGE's logic
// If an element does not exist it is defined as {inf, n}
void NGE(int n){

    // Push the first element {element, idx}
    stack<pair<ll,ll>> s;
    s.push({inf,n});    // The first element must be this

    // Traverse remaining elements (in reverse order of PGE)
    for(int i = n-1; i >= 0; i--){

        while (s.empty() == false && s.top().first < v[i])
            s.pop();

        if(s.empty())
            nxt[i] = {inf,n};
        else
            nxt[i] = {s.top().first, s.top().second};

        s.push({v[i],i});
    }
}

/*

Time Complexity

NGE -> O(n)

Links:

https://www.geeksforgeeks.org/next-greater-element/

*/
```

8.5 Previous Great Element

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int inf = 0x3f3f3f3f;
using ii = pair<int,int>;
using ll = long long;

const int N = 1e5; // Number of elements

vector<ll> v;

// PGE's answer from all elements in vector "v"
vector<ii> pre(N,{inf,-1});

// Find the previous great element for all array elements
// If an element does not exist it is defined as {inf,-1}
void PGE(int n){

    // Push the first element {element, idx}
    stack<pair<ll,ll>> s;
    s.push({v[0],0});

    // Traverse remaining elements
    for(int i = 1; i < n; i++){

        // Pop elements from stack while stack is not empty
        // and top of stack is smaller than arr[i]. We
        // always have elements in decreasing order in a
        // stack.
        while (s.empty() == false && s.top().first < v[i])
            s.pop();

        // If stack becomes empty, then no element is greater
        // on left side. Else top of stack is previous
        // greater.
        if(s.empty())
            pre[i] = {inf,-1};
        else
            pre[i] = {s.top().first, s.top().second};

        s.push({v[i],i});
    }
}

/*

Time Complexity

PGE -> O(n)

Links:

https://www.geeksforgeeks.org/previous-greater-element/

*/
```

8.6 Quick Select

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

vector<int> v;

int partition(int l, int r){
    int pivot = v[l + (r-l)/2];
    while(l < r){
        if(v[l] >= pivot){
            swap(v[l], v[r]);
            r--;
        }
        else
            l++;
    }

    if(v[l] < pivot) l++;

    return l;
}

// Algorithm to find the smallest "k"s elements using QuickSelect
// QuickSelect can be used to find the "k" element as well.
vector<int> smallest_k_elements(int k){
    int l = 0;
    int r = v.size()-1;
    int pivotIdx = v.size();

    while(pivotIdx != k){
        pivotIdx = partition(l,r);
        if(pivotIdx < k)
            l = pivotIdx;
        else
            r = pivotIdx - 1;
    }

    return vector<int>(v.begin(), v.begin() + k);
}

/*

Time Complexity:

QuickSelect -> O(n) In the average case, in the worst case it runs on O(n
    ^2) but it is rare.

Links:

https://www.youtube.com/watch?v=ooLKYx1TlSE

*/
```

8.7 Spiral Traversal

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

vector<ll> answ;

// Spiral Traversal (m = number of rows && n = number of columns)
// "v" is our matrix
void SpiralTraversal(int m, int n){

    int i, k = 0, l = 0;

    /*
        k - starting row index
        m - ending row index    (exclusive)
        l - starting column index
        n - ending column index (exclusive)
        i - iterator
    */

    while (k < m && l < n) {

        // Left to right
        for (i = l; i < n; ++i) {
            answ.push_back(v[k][i]);
        }
        k++;

        // From top to bottom
        for (i = k; i < m; ++i) {
            answ.push_back(v[i][n-1]);
        }
        n--;

        // Right to Left
        if (k < m) {
            for (i = n - 1; i >= l; --i) {
                answ.push_back(v[m-1][i]);
            }
            m--;
        }

        // From bottom to top
        if (l < n) {
            for (i = m - 1; i >= k; --i) {
                answ.push_back(v[i][l]);
            }
            l++;
        }
    }
}

/*

Time Complexity

SpiralTraversal -> O(m*n)

Links:

https://www.geeksforgeeks.org/print-a-given-matrix-in-spiral-form/

*/
```

8.8 Unordered Map Tricks

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;

struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    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);
    }
};
```

```
unordered_map<long long, int, custom_hash> safe_map;
gp_hash_table<long long, int, custom_hash> safe_hash_table;

/*---------------another way-----------------*/
/**
 * Author: Simon Lindholm, chilli
 * Date: 2018-07-23
 * License: CC0
 * Source: http://codeforces.com/blog/entry/60737
 * Description: Hash map with mostly the same API as unordered\_map, but \
     tilde
 * 3x faster. Uses 1.5x memory.
 * Initial capacity must be a power of 2 (if provided).
 */
#pragma once

#include <bits/extc++.h> /** keep-include */
// To use most bits rather than just the lowest ones:
struct chash { // large odd number for C
    const uint64_t C = (long long)(4e18 * acos(0)) | 71;
    long long operator()(long long x) const { return __builtin_bswap64(x*C)
        ; }
};
__gnu_pbds::gp_hash_table<long long,int,chash> h({},{},{},{},{1<<16});

/* For CodeForces, or other places where hacking might be a problem:

const int RANDOM = chrono::high_resolution_clock::now().time_since_epoch().
    count();
struct chash { // To use most bits rather than just the lowest ones:
    const uint64_t C = (long long)(4e18 * acos(0)) | 71; // large odd
        number
    long long operator()(long long x) const { return __builtin_bswap64((x^
        RANDOM)*C); }
};
__gnu_pbds::gp_hash_table<long long, int, chash> h({},{},{},{}, {1 << 16});

*/

// pair<int, int> hash function
struct HASH{
  size_t operator()(const pair<int,int>&x)const{
    return (size_t) x.first * 37U + (size_t) x.second;
  }
};
unordered_map<int,int> mp;
mp.reserve(1024);
mp.max_load_factor(0.25);

/*

Links:

https://codeforces.com/blog/entry/60737
https://codeforces.com/blog/entry/62393

*/
```

9 Util

9.1 Structure for matrix

```
template<typename T = int>
struct Matrix{
    int r, c;
    vector<vector<T>> mat;

    Matrix(int r, int c): r(r), c(c){
        mat.assign(r, vector<T>(c, 0));
    }

    Matrix(vector<vector<T>> m, int r, int c): r(r), c(c){
        mat.assign(r, vector<T>(c, 0));
        for(int i = 0; i < r; i++)
            for(int j = 0; j < c; j++)
                this->mat[i][j] = m[i][j];
    }

    Matrix operator *(Matrix &other){
        Matrix answ(this->r, other.c);
        for(int i = 0; i < r; i++){
            for(int j = 0; j < other.c; j++){
                for(int k = 0; k < c; k++){ // MOD only if necessary
                    answ.mat[i][j] = (answ.mat[i][j] + mat[i][k] * other.
                        mat[k][j]%MOD)%MOD;
                }
            }
        }
        return answ;
    }

    vector<T> operator [](int r){
        return mat[r];
    }
};

/*

Time Complexity:

fastModExp  -> O(d^3*logy)  d is the dimension of the square matrix

Links:

https://zobayer.blogspot.com/2010/11/matrix-exponentiation.html
https://www.geeksforgeeks.org/matrix-exponentiation/

*/
```

9.2 Coordinate Compression

```
template<class T>
struct CoordinateCompression{
    vector<T> v;
    void push(const T & a) { v.push_back(a); }
    int build(){
        sort(v.begin(), v.end());
        v.erase(unique(v.begin(), v.end()), v.end());
        return (int) v.size();
    }
    int operator[] (const T & a){
        auto it = lower_bound(v.begin(), v.end(), a);
        return (int) it - v.begin();
    }
};

/*

Time Complexity:

push    -> O(1)
build   -> O(nlogn)
[]      -> O(logn)

*/
```

10 Theorems

10.1 Graph

```
Flow:

-> Edge-disjoint path:
    Conjunto de arestas de caminhos que levam do source ao sink sem repetir
         arestas entre os caminhos.
    Max Edge-disjoint path = Max Flow, com arestas de peso 1 no grafo.

-> Node-disjoint path:
    Conjunto de nos de caminhos que levam do source ao sink sem repetir nos
         entre os caminhos.
    Max Node-disjoint path = Max Flow.
    Criamos um grafo onde cada no u ira existir uma copia u'. O no u tera
         todas as arestas de chegadas e o no u' todas as de saida e
         adicionamos a aresta u -> u'. Todas as arestas com capacidade 1.

-> Hall's Theorem:
    Em grafos bipartidos, existe um perfect matching no lado X se para cada
         subconjunto Y de X,
    |Y| <= |vizinhosY|, onde |Y| eh quantos vertices tem em Y.

-> Konig's Theorem:
    Em um grafo bipartido, Minimum Node Cover = Maximum Matching (Minimum
         Node Cover eh o conjunto minimo de nos onde cada aresta do grafo
         tem pelo menos um endpoint no conjunto).
    O complemento, n - Maximum Matching = Maximum Independent Set (Maior
         conjunto de nos onde nenhum dos nos possuem arestas conectando
         outro).

-> Node-disjoint path cover eh quando cada no so pertence a um unico
     caminho.
    Minimum Node-disjoint path cover = n - Maximum Match
    Criamos o grafo onde todo vertice u sera ligado ao source e criamos uma
         copia u' que sera ligada ao sink. Existira uma aresta v -> y'
         caso exista a aresta v -> y.

-> General Path Cover eh um path cover onde um no pode pertencer a mais de
     um caminho.
    Mesma coisa do Node-disjoint path cover mas existira uma aresta u -> v'
         caso exista caminho de u para v no grafo.

-> Dilworh's Theorem:
    Antichain = conjunto de nos de um grafo onde nao tem caminho de
         qualquer no para outro dentro do conjunto.
    Em um grafo direcionado aciclico, Minimum General Path Cover = Maximum
         antichain

Planar Graph:

-> Euler's formula for a connected planar graph:
    v + f - e = 2

-> Euler's formula for a disconnected planar graph:
    v - e + f = c + 1

-> Kuratowski's Theorem:
    A Graph is planar if and only if it contains no subdivision of K5 or K3
         ,3.

Euler Path/Circuit:

-> Undirected Graph:
    -> Existe quando todas as arestas pertencem ao mesmo componente conexo
         e quando acontece um dos 2 casos:
    1. Grau de todo vertice eh par. (Euler Path = Euler Circuit)
    2. Grau de 2 vertices sao impar e o resto eh par. (Nao existe Euler
         Circuit e o "src" pode ser qualquer um dos impares)

-> Directed Graph:
    -> Existe quando todas as arestas pertencem ao mesmo componente conexo
         e quando acontece um dos 2 casos:
    1. Em todo vertice "u", inDegree[u] == outDegree[u] (Euler Path = Euler
         Circuit)
    2. Em um vertice "u", outDegree[u] == inDegree[u] + 1 e em outro
         vertice "v" inDegree[v] == outDegree[v] + 1 e nos outros
        "x" inDegree[x] == outDegree[x] (Nao existe Euler Circuit e o "src"
             eh o "u" com o final do Eulerian Path sendo "v")
```

10.2 Math

```
-> Numero de fatores/divisores:
    Dado um numero n em sua fatoracao prima n = (p_1^alfa_1) * (p_2^alfa_2)
         * ...
    O numero de fatores eh dado pelo produtorio i = 1,...,k de (alfa_i + 1)

-> Soma dos fatores/divisores:
    Dado um numero n em sua fatoracao prima n = (p_1^alfa_1) * (p_2^alfa_2)
         * ...
    A soma dos fatores eh dada pelo produtorio i = 1,...,k de (p_i^(alfa_i
         + 1) - 1) / (p_i - 1)

-> Produto dos fatores/divisores:
    Dado por n^( numerodefatores(n) / 2)

-> Goldbach's conjecture:
    Todo inteiro par n > 2 pode ser representado como n = a + b onde a e b
         sao primos

-> Twin prime conjecture:
    Existe infinitos pares da forma {p, p + 2} onde p e p + 2 sao primos

-> Legendre's conjecture:
    Existe sempre um primo entre n^2 e (n + 1)^2

-> Fermat's little theorem:
    Se p eh primo e a e p sao coprimos, entao a^(p - 1) mod p = 1
    a^k mod p = a^(k mod (p - 1)) mod p (maneira de reduzir o expoente)

-> Euler's theorem:
    a^phi(p) mod p = 1, onde phi(p) eh o totiente de Euler

-> Modular Inverse:
    Existe quando a e p sao coprimos.
    (a^-1) = a^(phi(p) - 1)

-> Diophantine Equations:
    Equacoes de forma aX + bY = c, existem solucao caso c seja divisivel
         por gcd(a, b)
    Para acharmos uma solucao usa Extended gcd.

-> Lagrange's Theorem:
    Todo inteiro positivo pode ser inscrito como a soma de 4 quadrados (n =
         a^2 + b^2 + c^2 + d^2)

-> Zeckendorf's Theorem:
    Todo inteiro positivo possui uma representacao unica da soma de numeros
         de Fibonacci onde
    nenhum par de numeros sao iguais ou numeros de Fibonacci consecutivos.

-> Euclid's formula:
    Usada para formar todas as triplas (a,b,c) primitivas pitagoricas. (
         primitiva = a,b,c sao coprimos)
```

```
    (n^2 - m^2, 2*n*m, n^2 + m^2), onde 0 < m < n e n e m sao coprimos e
        pelo menos um eh par.
    Ex.: m = 1 e n = 2, (3,4,5)

-> Wilson's Theorem:
    n eh primo quando (n-1)! mod n = n - 1

-> Catalan numbers:
    C_n = (2n escolhe n) / (n + 1)
    ou C_0 = 1, C_n = somatorio de i = 0 -> n-1 de C_i*C_(n-1+1)
    1. Number of Balanced bracket sequences (n = pairs of brackets)
    2. There are C_n binary trees of n nodes
    3. There are C_(n-1) rooted trees of n nodes

-> Burnside's lemma
    Usado para saber a quantidade de objetos onde apenas um representante
        seja contado em um grupo simetrico.
    Somatorio de k = 1,2,...,n de c(k) / n
    onde n eh a quantidade de maneiras que podemos "rotacionar" o objeto e
        temos c(k) combinacoes
    que permanecem inalteradas quando a rotacao k-esima eh aplicada.
    Ex.: quantidade de colares de n perolas onde cada perola possui m
        possiveis cores:
        Somatorio de i = 0,1,...,n-1 de m^(gcd(i, n)) / n

-> Cayley's formula
    Existem n^(n - 2) labeled trees que possuem n vertices.
    Prova vem do Prufer Code.

-> Matriz:

    -> Counting paths
        Seja G uma matriz de adjacencia de um grafo sem pesos, onde [u][v]
            possui a quantidade de arestas de u -> v,
        a matriz G^n mostra a quantidade de caminhos com n arestas do
            vertice "u" para o "v".

    -> Shortest paths
        Seja G uma matriz de adjacencia de um grafo com pesos, onde [u][v]
            eh o minimo peso de u -> v,
        a matriz G^n mostra o menor caminho com n arestas do vertice u -> v
            .
        Obs.: Modificar a operacao de multiplicacao: Ao inves de somatorio
            ser minimo e ao inves de produto ser soma.
        Obs.: no fexpo no primeiro (y & 1) nao fazer a multiplicacao e sim
            so uma atribuicao sendo answ = matrix por conta da matrix
            identidade perder o sentido.

    -> Kirchhoff's Theorem
        Diz a quantidade de spanning trees de um grafo g.
        Seja a matriz M construida de tal maneira:
        M[i][i] = grau do vertice i em g.
        M[i][j] = -1 se existe aresta em g entre i e j.
        M[i][j] = 0 se nao existe aresta em g entre i e j.
        Seja M' a matriz M apos remover qualquer linha e qualquer coluna.
        A quantidade de Spanning trees de g eh dado por DET(M').

-> Probability:

    -> We can either use combinatorics or simulate the process that
        generates the event.

    -> Conditional Probability
        P(A|B) = P(A AND B) / P(B)
        is the probability of A assuming that B happens.
        We can calculate the intersection of A and B using this formula.

    -> Event A is independent if P(A|B) = P(A)

    -> Random Variables
        We denote P(X = x) the probability that the value of a random
            variable X is x.

    -> Expected value
        Indicates the average value of a random variable X.
        somatorio x de P(X = x) * x
        Where x goes through all possible values of X.
        Linearity: E[X1 + X2 + ... + Xn] = E[X1] + E[X2] + ... + E[Xn]

    -> Distributions
        1. Uniform Distribution: the random variable X has n possible
            values a, a + 1,...,b and the probability of each value is 1/n
            .
           E[X] = (a + b) / 2
        2. Binominal Distribution: n attempts are made and the probability
            that a single attempt succeeds is p.
           P(X = x) = p^x * (1 - p)^(n - x) * Comb(n, x)
           E[X] = p * n
        3. Geometric Distribution: the probability that an attempt succeeds
            is p, and we continue ultil the first success happens.
           E[X] = 1 / p

-> Basico:

    -> Progressao Aritmetica:
        a_n = a_1 + (n - 1) * r
        S_n = ((a_1 + a_n) * n) / 2

    -> Progressao Geometrica:
        a_n = a_1 * q^(n - 1)
        S_n = (a_1 * (q^n - 1)) / (q - 1)
        S_inf = (a_1) / (1 - q)    // Se -1 < q < 1

    -> Somatorio de i = 1 -> n de i * i = n * (n + 1) * (2*n + 1) / 6
```

10.3 Geometry

```
-> Lattice Points:
    Lattice point is a point whose coordinates are integers.
    The amount of lattice points of a segment a-b will be gcd( abs((b - a).
        x), abs((b - a).y))

-> Pick's Theorem:
    The area of a polygon = Number of Lattice points inside + Lattice
        points on the boundary / 2 - 1.
    Area = i + b/2 - 1
```