Competitive-programming

Algoritmos e ideias de programação competitiva

Table of Contents

9. ModularArithmetic
    1. Overloading Operations Struct
        1. Basic operations with combinatorics
10. Number-Theory
    1. Combinatorics Theory
    2. Crivo de Eratóstenes
    3. Factorization
        1. Trial Division with precomputed primes
        2. Pollard Rho
11. OrderedSet
    1. Policy Based Data Structures (PBDS)
12. Searching-Sorting
    1. Binary search
    2. Merge sort
    3. Ternary Search
13. Segtree
    1. Segtree with sum, max, min
    2. Implicit Segtree or Sparse Segtree
    3. Iterative P-sum Classic Segtree with MOD
    4. Inverted Segtree
    5. Recursive Segtree with Lazy propagation
        1. Sum range query, increase range query
        2. Range Minimum Query, Update (Assignment) Query
        3. Complex Lazy Problems
    6. Recursive Classic Segtree
14. Strings
    1. Booth's Algorithm
    2. Knuth–Morris–Pratt algorithm (KMP)
    3. SUFFIX ARRAY
    4. KASAI's ALGORITHM FOR LCP (longest common prefix)
    5. TRIE
    6. Z-function
15. Structures
    1. BIT (Fenwick Tree or Binary indexed tree)
16. Tree
    1. Binary lifting
    2. Find the Centroid of a Tree
    3. Find the Diameter
    4. Find the lenght of the longest path from all nodes
    5. Heavy Light Decomposition

Tips

- Invert the problem; fix the answer; maybe you don't need to simulate all steps; overflow problems; unitialized global variable; analize complexity; change approach; skip question!

Flags for compilation:

```
g++ -Wall -Wextra -Wshadow -ggdb3 -D_GLIBCXX_ASSERTIONS -fmax-errors=2 -std=c++17 -O3
test.cpp -o test
```

Template:

```
// Needed
#include <bits/stdc++.h>
using namespace std;
#define sws cin.tie(0)->sync_with_stdio(0)
// Life Quality
#define endl '\n'
#define ll long long
#define vll vector<ll>
#define pb push_back
#define ld long double
#define vld vector<ld>
#define pll pair<ll, ll>
#define vpll vector<pll>
#define ff first
#define ss second
#define tlll tuple<ll, ll, ll>
// Remainders
#define teto(a, b) ((a+b-1)/(b))
#define LSB(i) ((i) & -(i))
// Debugging
#define db(a) cerr << " [ " << #a << " = " << a << " ] " << endl;
#define debug(a...) cerr<<#a<<": ";for(auto &b:a)cerr<<b<<" ";cerr<<endl;
template <typename... A> void dbg(A const&... a) { ((cerr << "{" << a << "} "), ...);
cerr << endl; }
// Constants
const int MAX = 2e5+10;
const long long MOD = 1e9+7;
const int INF = 0x3f3f3f3f;
const long long LLINF = 0x3f3f3f3f3f3f3f3f;
const long double EPS = 1e-7;
const long double PI = acos(-1);

int32_t main(){ sws;

}
```

DP

Bitmask DP

use a bitmask of chosen itens to be a state of the DP

Example:

https://cses.fi/problemset/task/1653/

```
int32_t main(){
    ll n, x; cin >> n >> x;
```

```
    vll a(n);

    for(ll i=0; i<n; i++) cin >> a[i];

    // dp[bitmask of selected people] -> {elevator rides, weight occupied}
    vpll dp( (1 << n) , {INF, INF});
    dp[0] = {1, 0};

    for(ll mask=0; mask< (1<<n); mask++) {
        for(ll j=0; j<n; j++) {
            if (mask & (1 << j)) {
                ll bit = mask ^ (1 << j);

                // there is room for one more weight
                if (dp[bit].ss + a[j] <= x)
                    dp[mask] = min( dp[mask], {dp[bit].ff, dp[bit].ss + a[j]} );

                // add an elevator ride, and create a new one with just one person
                else
                    dp[mask] = min( dp[mask], {dp[bit].ff + 1, a[j]} );
            }
        }
    }
    cout << dp[(1 << n) - 1].ff << endl;
}
```

Broken Profile

Solves problem where is needed to count the ways of filling a *n* x *m* grid with dominos/tilings of specific size.

Example: Fill *n* x *m* with 2x1 dominos or 1x2 dominos -> https://cses.fi/problemset/task/2181

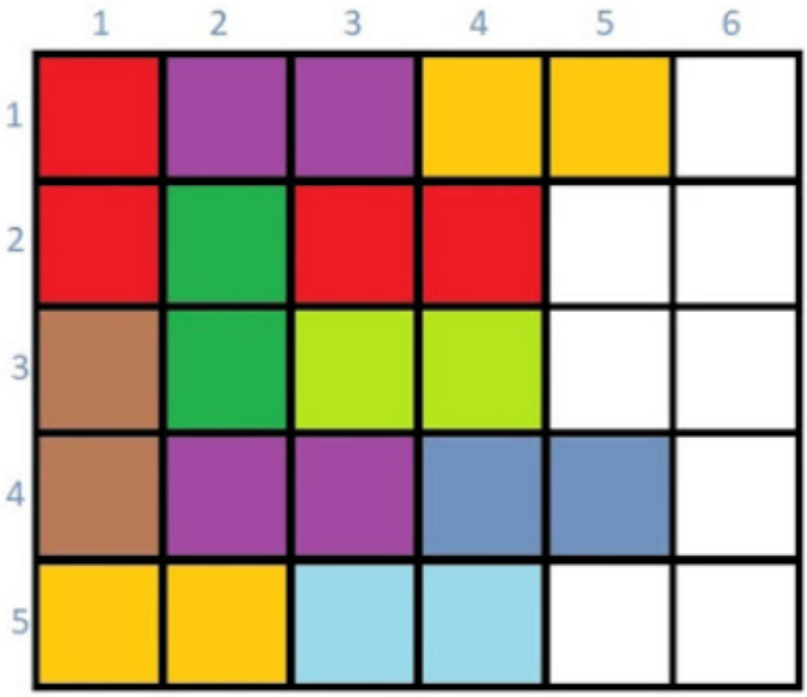
1 ≤ n ≤ 10 (rows) 1 ≤ m ≤ 1000 (collums)

States

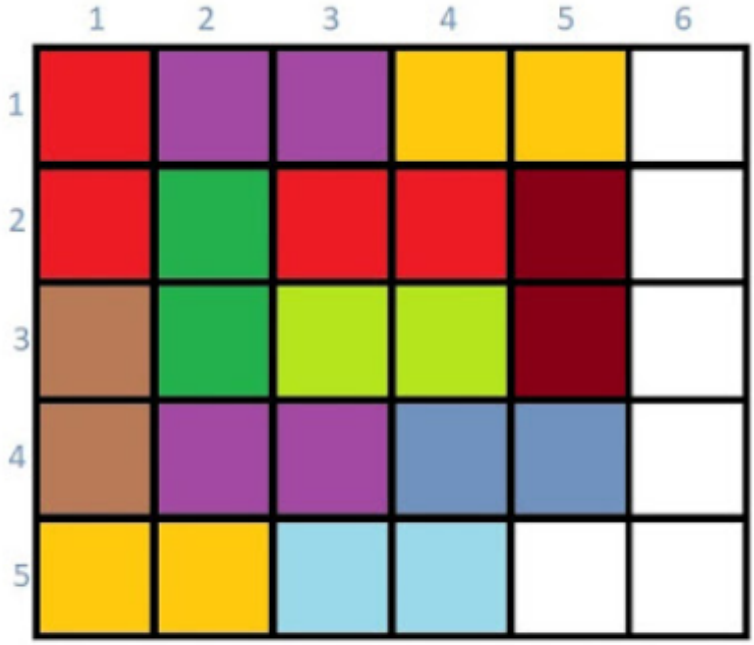
dp[j][p] -> numbers of ways of filling first *j* columns completely with dominoes (without leaving any block as empty) and leaving the profile *p* for the *j+1* collum.

j = collums completely filled. *p* = bitmask representation of the "profile" for the j+1 collum.

Note that, the (j+1) th column should not contain a complete domino (in the vertical), those types will be included in the dp transition from j+1 to j+2.



Valid filling of first 4 columns leaving the 5th column with profile 9 i.e. 01001



Invalid filling that cannot be counted into dp[4][15] as the brown domino is not used in filling any of the first four columns.
Note: 15 represents 01111 i.e. the profile of 5th columns

Transitions

Initial States:

```
dp[i = 0][p = 0] = 1
dp[i = 0][p != 0] = 0
```

From a specific **dp[j][q]**, with profile *q* for the *j-th* collum and all the collums before filled, it's possible to generate several **dp[j+1][p]** just by adding vertical and horizontal tiles.

Final Answer: dp[m][0]

```
// grid size
ll dp[1010][1 << 11];
ll n, m;
// n is the num of rows, m is the num of collums

// check if i'th bit of q is occupied
bool occupied(ll i, ll q) {
    return q & (1 << (i-1));
}

void solveBlock(ll i, ll j, ll p, ll q) {
    // from profile q -> generate profile p
    // OBS: q represents the profile of the j col, and p represents the profile of the
j+1 col when j is filled;
    // i <= 10 (row); j <= 1000 (col)

    // found a new way to fill j, add this possibility
    if (i == n+1) {
        dp[j+1][p] = (dp[j+1][p] + dp[j][q]) % MOD;
        return;
    }

    // skip occupied block
    if ( occupied(i, q) ) {
        solveBlock(i+1, j, p, q);
        return;
    }

    // insert vertical tile
    if(i+1 <= n and !occupied(i+1, q)){
        solveBlock(i+2, j, p, q);
    }

    // insert horizontal tile
    if (j+1 <= m) {
        solveBlock(i+1, j, p^(1<<(i-1)), q);
    }
}

int32_t main(){ sws;
    cin >> n >> m;
    memset(dp, 0, sizeof(dp));
    dp[0][0] = 1; // Initial Condition
    for(ll j=0; j<m; j++) { // each collum
        for(ll q=0; q < (1 << n); q++){ // each collum profile
            solveBlock(1, j, 0, q);
        }
    }
    cout << dp[m][0] << endl;
}
```

Digit DP

Use each digit position as state and also is the considered number is already smaller than the reference. The rest of the states are defined by the problem

Example1:

Calculate the quantity of numbers with no consective equal digits

```
string s; // number
ll tab[20][2][2][2][20];

// * returns the qtd of numbers with no consective equal digits
ll dp(ll i, bool smaller, bool consec, bool significantDigit, ll lastDigit){
    if (i >= (ll) s.size()) {
        if (consec) return 0;
        else return 1;
    }

    if (tab[i][smaller][consec][significantDigit][lastDigit] != -1)
        return tab[i][smaller][consec][significantDigit][lastDigit];

    ll limit = (s[i] - '0');
    ll ans = 0;

    for(ll a=0; a<=9; a++){
        bool tmp = consec;
        bool tmp2 = significantDigit; // avoid left zeros: 00001

        if (a > 0) tmp2 = 1;
        if (a == lastDigit and significantDigit) tmp = 1;

        if (smaller){
            ans += dp(i+1, 1, tmp, tmp2, a);
        }
        else if (a < limit){
            ans += dp(i+1, 1, tmp, tmp2, a);
        }
        else if (a == limit){
            ans += dp(i+1, 0, tmp, tmp2, a);
        }
    }
    return tab[i][smaller][consec][significantDigit][lastDigit] = ans;
}


int32_t main(void){ sws;
    ll a, b; cin >> a >> b;

    memset(tab, -1, sizeof(tab));
    s = to_string(b);
    ll ansr = dp(0, 0, 0, 0, 15); // 15 is simply a not valid number

    memset(tab, -1, sizeof(tab));
```

```
    s = to_string(a-1);
    ll ansl = dp(0, 0, 0, 0, 15);

    cout << ansr - ansl << endl;
}
```

Example2:

Classy numbers are the numbers than contains no more than 3 non-zero digit

```
string s;
ll tab[20][2][5];

// * returns qtd of classy numbers
ll dp(ll i, bool smaller, ll dnn){
    if (dnn > 3) return 0;
    if (i >= s.size()) return 1;

    if (tab[i][smaller][dnn] != -1) return tab[i][smaller][dnn];

    ll limit = (s[i] - '0');
    ll ans = 0;

    for(ll a=0; a<=9; a++){
        ll dnn2 = dnn;
        if (a > 0) dnn2 += 1;

        if (smaller){
            ans += dp(i+1, 1, dnn2);
        }
        else if (a < limit){
            ans += dp(i+1, 1, dnn2);
        }
        else if (a == limit){
            ans += dp(i+1, 0, dnn2);
        }
    }
    return tab[i][smaller][dnn] = ans;
}

int32_t main(void){ sws;
    ll t; cin >> t;
    while(t--){
        ll l, r;
        cin >> l >> r;

        memset(tab, -1, sizeof(tab));
        s = to_string(r);
        ll ansr = dp(0, 0, 0);

        memset(tab, -1, sizeof(tab));
        s = to_string(l-1);
        ll ansl = dp(0, 0, 0);

        cout << ansr - ansl << endl;
```

```
    }
}
```

Knapsack

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1

Capacity=6

i \ w	0	1	2	3	4	5	6
0							
1							
2							
3							
4							

- Use int instead of long long for 10^8 size matrix

```
    int n; cin >> n; // quantity of items to be chosen
    int x; cin >> x; // maximum capacity or weight
    vector<int> cost(n+1);
    vector<int> value(n+1);
    for(int i=1; i<=n; i++) cin >> cost[i];
    for(int i=1; i<=n; i++) cin >> value[i];

    vector<vector<int>> dp(n+1, vector<int>(x+1, 0));

    for(int i=1; i<=n; i++){
        for(int j=1; j<=x; j++){
            // same answer as if using -1 total capacity (n pega)
            dp[i][j] = max(dp[i][j], dp[i-1][j]);
            // use the item with index i (pega)
            if (j-cost[i] >= 0)
                dp[i][j] = max(dp[i][j], dp[i-1][j-cost[i]] + value[i]);
        }
    }

    cout << dp[n][x] << endl;
```

LIS (Longest Increasing Sequence)

Strictly Increasing: ans_i < ans_(i+1)

Requires a vector *x* with size *n*

```
vll d(n+1, LLINF);
d[0] = -LLINF;
for(ll i=0; i<n; i++){
    ll idx = upper_bound(d.begin(), d.end(), x[i]) - d.begin();
    if (d[idx-1] < x[i])
        d[idx] = min(d[idx], x[i]);
}
ll lis = (lower_bound(d.begin(), d.end(), LLINF) - d.begin() - 1);
```

DSU

Disjoint Set Union

```
struct DSU{
    vll group;
    vll card;
    DSU (long long n){
        group = vll(n);
        iota(group.begin(), group.end(), 0);
        card = vll(n, 1);
    }
    long long find(long long i){
        return (i == group[i]) ? i : (group[i] = find(group[i]));
    }
    void join(long long a ,long long b){
        a = find(a);
        b = find(b);
        if (a == b) return;
        if (card[a] < card[b]) swap(a, b);
        card[a] += card[b];
        group[b] = a;
    }
};
```

Avisos

Possui a optimização de **Compressão** e **Balanceamento**

Both are: *O(a(N)) ~ O(1)*:

find(i): finds the representative of an element and returns it

join(a, b): finds both representatives and unites them, remaining only one for all. No return value

Flow

Fluxo

```
const ll N = 505; // number of nodes, including sink and source
struct Dinic {  // O( Vertices^2 * Edges)
    struct Edge {
        ll from, to, flow, cap;
    };
    vector<Edge> edges;

    vll g[N];
    ll ne = 0, lvl[N], vis[N], pass;
    ll qu[N], px[N], qt;

    ll run(ll s, ll sink, ll minE) {
        if (s == sink) return minE;
```

```
        ll ans = 0;
        for(; px[s] < (int)g[s].size(); px[s]++){
            ll e = g[s][ px[s] ];
            auto &v = edges[e], &rev = edges[e^1];
            if( lvl[v.to] != lvl[s]+1 || v.flow >= v.cap) continue;
            ll tmp = run(v.to, sink, min(minE, v.cap - v.flow));
            v.flow += tmp, rev.flow -= tmp;
            ans += tmp, minE -= tmp;
            if (minE == 0) break;
        }
        return ans;
    }

    bool bfs(ll source, ll sink) {
        qt = 0;
        qu[qt++] = source;
        lvl[source] = 1;
        vis[source] = ++pass;
        for(ll i=0; i<qt; i++) {
            ll u = qu[i];
            px[u] = 0;
            if (u == sink) return 1;
            for(auto& ed :g[u]) {
                auto v = edges[ed];
                if (v.flow >= v.cap || vis[v.to] == pass) continue;
                vis[v.to] = pass;
                lvl[v.to] = lvl[u]+1;
                qu[qt++] = v.to;
            }
        }
        return false;
    }

    ll flow(ll source, ll sink) { // max_flow
        reset_flow();
        ll ans = 0;
        while(bfs(source, sink))
            ans += run(source, sink, LLINF);
        return ans;
    }

    void addEdge(ll u, ll v, ll c, ll rc) { // c = capacity, rc = retro-capacity;
        Edge e = {u, v, 0, c};
        edges.pb(e);
        g[u].pb(ne++);
        e = {v, u, 0, rc};
        edges.pb(e);
        g[v].pb(ne++);
    }

    void reset_flow() {
        for (ll i=0; i<ne; i++) edges[i].flow = 0;
        memset(lvl, 0, sizeof(lvl));
        memset(vis, 0, sizeof(vis));
        memset(qu, 0, sizeof(qu));
        memset(px, 0, sizeof(px));
        qt = 0; pass = 0;
```

```
    }

    vpll cut() { // OBS: cut set cost is equal to max flow
        vpll cuts;
        for (auto [from, to, flow, cap]: edges)
            if (flow == cap and vis[from] == pass and vis[to] < pass and cap > 0)
                cuts.pb({from, to});
        return cuts;
    }
};
```

How to use?

Set an unique id for all nodes

Remember to include the sink vertex and the source vertex. Usually *n+1* and *n+2*, *n* = max number of normal vertices

use **dinic.addEdge** to add edges -> (from, to, normal way capacity, retro-capacity)

use **dinic.flow(source_id, sink_id)** to receive maximum flow from source to sink through the network

OBS: It's possible to access *dinic.edges*, which is a vector that contains all edges and also its respective properties, like the **flow** passing through each edge. This can be used to **matching problems** with a bipartite graph and *1 capacity* for example.

Example

```
int32_t main(){sws;
    ll n, m; cin >> n >> m;
    Dinic dinic;

    for(ll i=1; i<=n; i++){
        ll k; cin >> k;
        for(ll j=0; j<k; j++){
            ll empresa; cin >> empresa;
            empresa += n;
            dinic.addEdge(i, empresa, 1, 0);
        }
    }

    ll source = n + m + 1;
    ll sink = n + m + 2;

    for(ll i=1; i<=n; i++){
        dinic.addEdge(source, i, 1, 0);
    }

    for(ll j=1; j<=m; j++){
        dinic.addEdge(j+n, sink, 1, 0);
    }

    cout << m - dinic.flow(source, sink) << endl;
}
```

Minimum Cut

Another problem solved by network flow is the **minimum cut**.

Let's define an **s-t cut C** = *(S-component, T-component)* as a partition of $V \in G$ such that source $s \in S\text{-component}$ and sink $t \in T\text{-component}$. Let's also define a cut-set of C to be the set {(u, v) ∈ E | u ∈ S-component, v ∈ T-component} such that if all edges in the cut-set of C are removed, the Max Flow from s to t is 0 (i.e., s and t are disconnected). The cost of an s-t cut C is defined by the sum of the capacities of the edges in the cut-set of C.

The by-product of computing Max Flow is Min Cut! After Max Flow algorithm stops, we run graph traversal (DFS/BFS) from source s again. All reachable vertices from source s using positive weighted edges in the residual graph belong to the S-component. All other unreachable vertices belong to the T-component. All edges connecting the S-component to the T-component belong to the cut-set of C. The Min Cut value is equal to the Max Flow value mf. This is the minimum over all possible s-t cuts values.

Example:

https://cses.fi/problemset/task/1695/

```
int32_t main(){ sws;
    ll n, m; cin >> n >> m;
    Dinic dinic;
    for(ll i=0; i<m; i++) {
        ll u, v; cin >> u >> v;
        dinic.addEdge(u, v, 1, 1);
    }
    dinic.flow(1, n);
    vpll ans = dinic.cut();
    cout << ans.size() << endl;
    for(auto [u, v] : ans) cout << u << ' ' << v << endl;
}
```

Geometry

Closest-point (Divide and conquer)

```
int solve(vector<point> x_s, vector<point> y_s){
    int n = x_s.size();

    if(n < 4){
        int d = x_s[0].dist(x_s[1]);
        for(int i=0; i<n; i++){
            for(int j=i+1; j<n; j++)
                d = min(d, x_s[i].dist(x_s[j]));
        }
        return d;
    }

    int mid = n/2;
    vector<point> x_sl(x_s.begin(), x_s.begin()+mid);
```

```
    vector<point> x_sr(x_s.begin()+mid, x_s.end());
    vector<point> y_sl, y_sr;
    for(auto p: y_s){
        if(p.x <= x_s[mid].x)
            y_sl.push_back(p);
        else
            y_sr.push_back(p);
    }

    int dl = solve(x_sl, y_sl);
    int dr = solve(x_sr, y_sr);


    // Merge !!!
    int d = min(dl, dr);

    vector<point> possible;
    for(auto p: y_s){
        if(x_s[mid].x-d < p.x and p.x < x_s[mid].x+d)
            possible.push_back(p);
    }

    n = possible.size();
    for(int i=0; i<n; i++){
        for(int j=1; (j<7 and j+i<n); j++){
            d = min(d, possible[i].dist(possible[i+j]));
        }
    }

    return d;
}
```

Convex Hull

Complexity: O(n * log (n))

```
struct point{
    int x, y;
    int ind;

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

    int operator ^(const point& b) const{ // cross product
        return x*b.y - y*b.x;
    }

    int cross(const point& b, const point&c) const{ // cross product with diferent base
        return (b - *this) ^ (c - *this);
    }

    bool operator <(const point& b) const{
        return make_pair(x,y) < make_pair(b.x,b.y);
    }
```

```
};

vector<point> convex_hull(vector<point>& v){
    vector<point> hull;
    sort(v.begin(), v.end());

    for(int rep=0; rep<2; rep++){
        int S = hull.size();
        for(point next : v){

            while(hull.size() - S >= 2){
                point prev = hull.end()[-2]; // hull[size - 2]
                point mid = hull.end()[-1]; // hull[size - 1]
                if(prev.cross(mid, next) <=0) // 0 collinear
                    break;
                hull.pop_back();
            }

            hull.push_back(next);
        }

        hull.pop_back();
        reverse(v.begin(), v.end());
    }
    return hull;
}
```

Point Struct

```
struct Point{
    int x, y;
    int ind; // idx

    Point(){
        this->x = 0;
        this->y = 0;
    }

    Point(int x, int y){
        this->x = x;
        this->y = y;
    }

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

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

    int operator *(const Point& b) const{ // dot product
        return x*b.y + y*b.x;
    }
```

```
    int operator ^(const Point& b) const{ // cross product
        return x*b.y - y*b.x;
    }

    int dot(const Point& b, const Point&c) const{ // dot product with diferent base
        return (b - *this) * (c - *this);
    }

    int cross(const Point& b, const Point&c) const{ // cross product with diferent base
        return (b - *this) ^ (c - *this);
    }

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

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

};
```

Teoria:

Por definição, o produto escalar define o cosseno entre dois vetores:

cos(a, b) = (a * b) / (||a|| * ||b||)

a * b = cos(a, b) * (||a|| * ||b||)

O sinal do produto vetorial de A com B indica a relação espacial entre os vetores A e B.

cross(a, b) > 0 -> ***B*** está a esquerda de ***A***.

cross(a, b) = 0 -> ***B*** é colinear ao ***A***.

cross(a, b) > 0 -> ***B*** está a direita de ***A***.

A magnitude do produto vetorial de A com B é a área do paralelogramo formado por A e B. Logo, a metade é a área do triângulo formado por A e B.

Área de qualquer polígono, convexo ou não.

Definindo um vértice como 0, e enumerando os demais de [1 a N), calcula-se a área do polígono como o somatório da metade de todos os produtos vetorias entre o 0 e os demais.

```
For i in [1, N) :
    Area += v0 ^ vi
Area = abs(Area)
```

Lembre-se de pegar o módulo da área para ignorar o sentido escolhido.

Graph

2-SAT (2-satisfiability)

SAT (Boolean satisfiability problem) is NP-Complete.

2-SAT is a restriction of the SAT problem, in 2-SAT every clause has exactly two literals.

Can be solved with graphs in *O(Vertices + Edges)*.

```
// 0-idx graph !!!!
struct TwoSat {
    ll N; // needs to be the twice of the number of variables
    // node with idx 2x => variable x
    // node with idx 2x+1 => negation of variable x

    vector<vll> g, gt;
    // g = graph; gt = transposed graph (all edges are inverted)

    TwoSat(ll n) { // number of variables
        N = 2*n;
        g.assign(N, vll());
        gt.assign(N, vll());
    }

    vector<bool> used;
    vll order, comp;
    vector<bool> assignment;
    //  assignment[x] == 1 -> x was assigned
    //  assignment[x] == 0 -> !x was assigned

    // dfs1 and dfs2 are part of kosaraju algorithm
    void dfs1(ll u) {
        used[u] = true;
        for (ll v : g[u]) if (!used[v]) dfs1(v);
        order.pb(u); // topological order
    }

    void dfs2(ll u, ll timer) {
        comp[u] = timer;
        for (ll v : gt[u]) if (comp[v] == -1) dfs2(v, timer);
    }

    bool solve_2SAT() {
        order.clear();
        used.assign(N, false);
        for (ll i = 0; i < N; i++) if (!used[i]) dfs1(i);

        comp.assign(N, -1);
        for (ll i = 0, j = 0; i < N; i++) {
            ll u = order[N - i - 1]; // reverse order
            if (comp[u] == -1) dfs2(u, j++);
        }

        assignment.assign(N/2, false);
```

```
        for (ll i = 0; i < N; i += 2) {
            if (comp[i] == comp[i + 1]) return false; // x and !x contradiction
            assignment[i / 2] = comp[i] > comp[i + 1];
        }
        return true;
    }

    void add_disjunction(ll a, bool na, ll b, bool nb) {
        // disjunction of (a, b) => if one of the two variables is false, then the other
one must be true
        // na and nb signify whether a and b are to be negated
        // na == 1 => !a ; na == 0 => a
        // nb == 1 => !b ; nb == 0 => b
        a = 2*a ^ na;
        b = 2*b ^ nb;
        ll neg_a = a ^ 1;
        ll neg_b = b ^ 1;
        g[neg_a].pb(b);
        g[neg_b].pb(a);
        gt[b].pb(neg_a);
        gt[a].pb(neg_b);
    }
};
```

Example of Application:

https://cses.fi/problemset/task/1684/ (Giant Pizza)

```
int32_t main(){ sws;
    ll m, n; cin >> m >> n;

    TwoSat twoSat(n);

    for(ll i=0; i<m; i++) {
        char charA, charB;
        ll a, b;
        cin >> charA >> a >> charB >> b;
        // at least one => (!a 'disjoint' !b)
        bool na = (charA == '-');
        bool nb = (charB == '-');
        twoSat.add_disjunction(a-1, na, b-1, nb);
    }

    if (!twoSat.solve_2SAT()) cout << "IMPOSSIBLE" << endl;
    else {
        for(ll i=0; i<n; i++) {
            if (twoSat.assignment[i]) cout << "+ ";
            else cout << "- ";
        }
        cout << endl;
    }
}
```

Extended Version

Possibly faster implementation, with redundant functions to facilitate logic.

```
struct TwoSat { // copied from kth-competitive-programming/kactl
    ll N;
    vector<vll> gr;
    vll values; // 0 = false, 1 = true
    TwoSat(ll n = 0) : N(n), gr(2*n) {} // crazy constructor, n = number of variables
    ll addVar() { // (optional)
        gr.emplace_back();
        gr.emplace_back();
        return N++;
    }
    void either(ll f, ll j) {
        f = max(2*f, -1-2*f);
        j = max(2*j, -1-2*j);
        gr[f].push_back(j^1);
        gr[j].push_back(f^1);
    }
    void atMostOne(const vll& li) { // (optional)
        if ((ll)li.size() <= 1) return;
        ll cur = ~li[0];
        for(ll i=2; i<(ll)li.size(); i++) {
            ll next = addVar();
            either(cur, ~li[i]);
            either(cur, next);
            either(~li[i], next);
            cur = ~next;
        }
        either(cur, ~li[1]);
    }
    vll _val, comp, z; ll time = 0;
    ll dfs(ll i) {
        ll low = _val[i] = ++time, x; z.push_back(i);
        for(ll e : gr[i]) if (!comp[e])
            low = min(low, _val[e] ?: dfs(e));
        if (low == _val[i]) do {
            x = z.back(); z.pop_back();
            comp[x] = low;
            if (values[x>>1] == -1)
                values[x>>1] = x&1;
        } while (x != i);
        return _val[i] = low;
    }
    bool solve() {
        values.assign(N, -1);
        _val.assign(2*N, 0); comp = _val;
        for(ll i=0; i<2*N; i++) if (!comp[i]) dfs(i);
        for(ll i=0; i<N; i++) if (comp[2*i] == comp[2*i+1]) return 0;
        return 1;
    }
};
```

BFS

```
vector<vll> g(MAX, vll());
queue<ll> fila;
bool vis[MAX];

void bfs(ll i){
    memset(vis, 0, sizeof(vis));
    fila.push(i);
    vis[i] = 1;

    while(!fila.empty()){
        ll u = fila.front(); fila.pop();

        for(auto v : g[u]) if (!vis[v]) {
            vis[v] = 1;

            d[v] = d[u] + 1;

            fila.push(v);
        }
    }
}
```

Bridges (Cut Edges)

Theory: After constructing a DFS Tree, an edge (u, v) is a bridge if and only if there is no back-edge from *v, or a descendent of v,* to *u, or an ancestor of u*.

To do this efficiently, it's used *tin[i] (entry time of node **i**)* and *low[i] (minimum entry time of all nodes that can be reached from node **i**)*.

```
vector<vll> g(MAX, vll());
bool vis[MAX];
ll tin[MAX], low[MAX];
ll timer;
vpll bridges;

void dfs(ll u, ll p = -1){
    vis[u] = 1;
    tin[u] = low[u] = timer++;
    for(auto v : g[u]) if (v != p) {
        if (vis[v]) low[u] = min(low[u], tin[v]);
        else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u])
                bridges.pb( {u, v} );
        }
    }
}

void find_bridges(ll n) {
    timer = 1;
    memset(vis, 0, sizeof(vis));
    memset(tin, 0, sizeof(tin));
```

```
    memset(low, 0, sizeof(low));
    for(ll i=1; i<=n; i++) if (!vis[i]) dfs(i);
}
```

Articulation Points and Bridges

Finds all Cut-Vertices and Cut-Edges in a single dfs tranversal O(V+E)

Maybe is working, maybe it's not, needs testing for exquisite graphs, like cliques

```
vector<vll> g(MAX, vll());
vll tin(MAX, -1), low(MAX, 0);
// tin[] = the first time a node is visited ("time in")
// if tin[u] != -1, u was visited
// low[] = lowest first_time of any node reachable by the current node

ll root = -1, rootChildren = 0, timer = 0;
// root = the root of a dfs transversal, rootChildren = number of direct descedentes of
the root

vector<bool> isArticulation(MAX, 0); // this vector exists, because we can define
several time if a node is a cut vertice
vll articulations; // cut vertices
vpll bridges; // cut edges

void dfs(ll u, ll p) {
    low[u] = tin[u] = timer++;

    for(auto v : g[u]) if (v != p) {
        if (tin[v] == -1) { // not visited
            if (u == root) rootChildren += 1;

            dfs(v, u);

            if (low[v] >= tin[u]) isArticulation[u] = 1;
            if (low[v] > tin[u]) bridges.pb({u, v});
        }

        low[u] = min(low[u], low[v]);
    }
}

void findBridgesAndPoints(ll n) {
    timer = 0;
    for(ll i=1; i<=n; i++) if (tin[i] == -1) {
        root = i; rootChildren = 0;
        dfs(i, -1);
        if (rootChildren > 1) isArticulation[i] = 1;
    }
    for(ll i=1; i<=n; i++) if (isArticulation[i]) articulations.pb(i);
}
```

Cycles

Find a Cycle

vis[] array stores the current state of a node: **-1** -> not visited **0** -> explored, not ended (still need to end edge transversals) **1** -> visited, totally explored (no more edges to transverse)

p[] array stores the descedent of each node, to reconstruct cycle components

```
vector<vll> g(MAX, vll());
vll vis(MAX, -1);
vll p(MAX, -1);
ll cycle_end, cycle_start;

bool dfs(ll u) {
    vis[u] = 0;
    for(auto v : g[u]) if (vis[v] != 1) {
        if (vis[v] == 0){
            cycle_end = u;
            cycle_start = v;
            return 1;
        }
        p[v] = u;
        if (dfs(v)) return 1;
    }
    vis[u] = 1;
    return 0;
}

bool find_first_cycle(ll n) {
    for(ll i=1; i<=n; i++) if (vis[i] == -1) {
        if (dfs(i)){
            stack<ll> ans;
            ans.push(cycle_start);

            ll j = cycle_end;
            while(j != cycle_start){
                ans.push(j);
                j = p[j];
            }
            ans.push(cycle_start);

            cout << ans.size() << endl;
            while(!ans.empty()){
                cout << ans.top() << ' ';
                ans.pop();
            }
            cout << endl;
            return 1;
        }
    }
    return 0
}
```

DFS Tree

A *Back Edge* existence means that there is a cycle.

```
bool visited[MAX];
vector<vll> g(MAX, vll());
map<ll, ll> spanEdges;
map<ll, ll> backEdges; // children to parent
ll h[MAX];
ll p[MAX];

void dfs(ll u=1, ll parent=0, ll layer=1){
    if (visited[u]) return;
    visited[u] = 1;
    h[u] = layer;
    for(auto v : g[u]){
        if (v == parent) spanEdges[u] = v;
        else if (visited[v] and h[v] < h[u]) backEdges[u] = v;
        else dfs(v, u, layer+1);
    }
}
```

Euler Path

Definitions:

An **Eulerian Path** or **Eulerian Trail** (*Caminho Euleriano*) consists of a path that transverses all **Edges**.

A special case is the closed path, which is an **Eulerian Circuit** or **Eulerian Cycle** (*Circuito/Ciclo Euleriano*). A graph is considered *eulerian* (**Eulerian Graph**) if it has an Eulerian Circuit.

Similarly, a **Hamiltonian Path** consists of a path that transverses all **Vertices**.

Conditions for Eulerian Path existence

To check if it is possible, there is a need for connectivity:

connectivity, all vertices (that contains at least 1 edge) are connected. But there is no need for it to be strongly connected. To check connectivity, you can consider a directed graph as undirected and do a dfs.

and also:

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of the Euler path/circuit problem we care about:

	Eulerian Circuit	Eulerian Path
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

Hierholzer Algorithm

Find a **Eulerian Path/Circuit** with a linear complexity of *O(Edges)*.

Using an *ordered set* on **Undirected Graphs** increases complexity by *log2(Edges)*. This can be optimized using a *list* with references to each bidirectional edge so that any reversed edge can be erased in *O(1)*.

Example 1:

Generating an **Eulerian Path** with Hierholzer in a *Directed Graph*, starting on node *1* and ending on node *n*.

https://cses.fi/problemset/task/1693

```
vector<vll> g(MAX, vll());
vector<vll> ug(MAX, vll()); // undirected graph

vll inDegree(MAX, 0);
vll outDegree(MAX, 0);

vector<bool> vis(MAX, 0);

ll dfsConnected(ll u) {
    ll total = 1; vis[u] = 1;
    for(auto v : ug[u]) if (!vis[v]) {
        total += dfsConnected(v);
    }
```

```
    return total;
}

// O(n) -> O(Vertices)
bool checkPossiblePath(ll start, ll end, ll n, ll nodes) {

    // check connectivity
    vis.assign(n+1, 0);
    ll connectedNodes = dfsConnected(1);
    if (connectedNodes != nodes) return 0;

    // check degrees
    for(ll i=1; i<=n; i++) {
        if (i == start) { // start node needs to have 1 more outDegree than inDegree
            if (inDegree[i]+1 != outDegree[i]) return 0;
        }
        else if (i == end) { // end node needs to have 1 more inDegree than outDegree
            if (inDegree[i] != outDegree[i]+1) return 0;
        }
        else {
            if (inDegree[i] != outDegree[i]) return 0;
        }
    }

    return 1;
}

// O(m) -> O(Edges)
// Hierholzer function can be used directly if there is already a garanted existance of
an eulerian path/circuit.
vll hierholzer(ll start, ll n) { // generate an eulerian path, assuming there is only 1
end node
    vll ans, pilha, idx(n+1, 0);

    pilha.pb(start);
    while(!pilha.empty()) {
        ll u = pilha.back();
        if (idx[u] < (ll) g[u].size()) {
            pilha.pb( g[u][idx[u]] );
            idx[u] += 1;
        }
        else { // no more outEdge from node u, backtracking
            ans.pb(u);
            pilha.pop_back();
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
}

int32_t main(){ sws;
    ll n, m; cin >> n >> m;

    // OBS: some nodes are isolated and don't contribute to the eulerian path
    ll participantNodes = 0;

    for(ll i=0; i<m; i++) {
```

```
        ll a, b; cin >> a >> b;

        g[a].pb(b);
        ug[a].pb(b); ug[b].pb(a);

        outDegree[a] += 1;
        inDegree[b] += 1;

        if (!vis[a]) {
            vis[a] = 1;
            participantNodes += 1;
        }
        if (!vis[b]) {
            vis[b] = 1;
            participantNodes += 1;
        }
    }

    if ( !checkPossiblePath(1, n, n, participantNodes) ) {
        cout << "IMPOSSIBLE" << endl;
        return 0;
    }

    for(auto elem : hierholzer(1, n)) cout << elem << ' ';
    cout << endl;
}
```

Example 2:

Generating an **Eulerian Circuit** with Hierholzer in an *Undirected Graph*, starting on node *1* and also ending on node *1*.

https://cses.fi/problemset/task/1691

```
// adding log2(m) complexity due to ordered_set structure required for not using a same
bidirectional edge twice
#include <bits/extc++.h>
using namespace __gnu_pbds;
template <class T> using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;

vector<ordered_set<ll>> g(MAX, ordered_set<ll>()); // undirected graph

vll degree(MAX, 0);

vector<bool> vis(MAX, 0);

ll dfsConnected(ll u) {
    ll total = 1; vis[u] = 1;
    for(auto v : g[u]) if (!vis[v]) {
        total += dfsConnected(v);
    }
    return total;
}

// O(n log2(m)) -> O(Vertices * log2(Edges))
```

```cpp
bool checkPossiblePath(ll n, ll nodes) {

    // check connectivity
    vis.assign(n+1, 0);
    ll connectedNodes = dfsConnected(1);
    if (connectedNodes != nodes) return 0;

    // check degrees
    for(ll i=1; i<=n; i++) {
        // all degrees need to be even
        if (degree[i] % 2 == 1) return 0;
    }

    return 1;
}

// O(m * log2(m)) -> O(Edges * log2(m))
// Hierholzer function can be used directly if there is already a garanted existance of an eulerian path/circuit.
vll hierholzer(ll start, ll n) { // generate an eulerian path, assuming there is only 1 end node
    vll ans, pilha, idx(n+1, 0);

    pilha.pb(start);
    while(!pilha.empty()) {
        ll u = pilha.back();
        if (idx[u] < (ll) g[u].size()) {
            ll v = *(g[u].find_by_order(idx[u]));

            pilha.pb( v );
            g[v].erase(u);

            idx[u] += 1;
        }
        else { // no more outEdge from node u, backtracking
            ans.pb(u);
            pilha.pop_back();
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
}

int32_t main(){ sws;
    ll n, m; cin >> n >> m;

    // OBS: some nodes are isolated and don't contribute to the eulerian circuit
    ll participantNodes = 0;

    for(ll i=0; i<m; i++) {
        ll a, b; cin >> a >> b;

        g[a].insert(b);
        g[b].insert(a);

        degree[a] += 1;
        degree[b] += 1;
```

```
        if (!vis[a]) {
            vis[a] = 1;
            participantNodes += 1;
        }
        if (!vis[b]) {
            vis[b] = 1;
            participantNodes += 1;
        }
    }

    if ( !checkPossiblePath(n, participantNodes) ) {
        cout << "IMPOSSIBLE" << endl;
        return 0;
    }

    for(auto elem : hierholzer(1, n)) cout << elem << ' ';
    cout << endl;
}
```

Euler Tour Technique (ETT)

AKA: Preorder time , DFS time.

Flattening a tree into an array to easily query and update subtrees. This is achieved by doing a *Pre Order Tree Transversal:* (childs -> node), a simple *dfs* marking *entry times* and *leaving times*.

Creates an array that can have some properties, like all child vetices are ordered after their respective roots.

```
vector<vector<int>> g(MAX, vector<int>());
int timer = 1; // to make a 1-indexed array
int st[MAX]; // L index
int en[MAX]; // R index

void dfs_time(int u, int p) {
    st[u] = timer++;
    for (int v : g[u]) if (v != p) {
        dfs_time(v, u);
    }
    en[u] = timer-1;
}
```

Problems

https://cses.fi/problemset/task/1138 -> change value of node and calculate sum of the path to root of a tree

Strongly Connected Components

Kosaraju

Used for **Finding Strongly Connected Somponents** (SCCs) in a *directed graph* (digraph).

Complexity O(1) -> O(V + E), linear on number of edges and vertices.

Remember to also construct the inverse graph (*gi*).

```
vector<vll> g(MAX, vll());
vector<vll> gi(MAX, vll()); // inverted edges
bool vis[MAX]; // visited vertice?
ll component[MAX]; // connected component of each vertice
stack<ll> pilha; // for inverting order of transversal

void dfs(ll u) {
    vis[u] = 1;
    for(auto v : g[u]) if (!vis[v]) dfs(v);
    pilha.push(u);
}

void dfs2(ll u, ll c) {
    vis[u] = 1; component[u] = c;
    for(auto v : gi[u]) if (!vis[v]) dfs2(v, c);
}

// 1 - idx
void kosaraju(ll n){
    memset(vis, 0, sizeof(vis));
    for(ll i=1; i<=n; i++) if (!vis[i]) dfs(i);

    memset(vis, 0, sizeof(vis));
    memset(component, 0, sizeof(component));

    while(!pilha.empty()) {
        ll u = pilha.top(); pilha.pop();
        if (!vis[u]) dfs2(u, u);
    }
}
```

Can be extended to generate a Condensation Graph

AKA: condensate/convert all SCC's into single vertices and create a new graph

```
vector<vll> gc(MAX, vll()); // Condensation Graph

void condensate(ll n){
    for(ll u=1; u<=n; u++)
        for(auto v : g[u]) if (component[v] != component[u])
            gc[ component[u] ].pb( component[v] );
}
```

Single-Source Shortest Paths (SSSP)

Bellman-Ford for shortest paths

Supports Negative edges!

Solves: Finds all shortest paths from a initial node *x* to every other node

Complexity: O(n * m) = O(vertices * edges) -> quadratic

Conjecture: After **at most** *n-1 (Vertices-1) iterations*, all shortest paths will be found.

```
#define tlll tuple<ll, ll, ll>
vector<tlll> edges(MAX, tlll() );
vll d(MAX, INF);

void BellmanFord(ll x, ll n) {
    d[x] = 0;
    for(ll i=0; i<n-1; i++) { // n-1 iterations will suffice
        for(auto [u, v, w] : edges) if (d[u] + w < d[v]) {
              d[v] = d[u] + w;
        }
    }
}
```

Variation of Bellman-Ford to find a negative cycle

Iterate *n* (number of Vertices) times and if in the last iteration a distance if reduced, it means that there is a negative cycle. Save this last node, whose distance was reduced, and, which a parent array, reconstruct the negative cycle.

```
#define tlll tuple<ll, ll, ll>
vector<tlll> edges;
vll d(5050, INF);
vll p(5050, -1);

// modification of bellman-ford algorithm to detect negative cycle
void BellmanFord_Cycle(ll start, ll n){ // O (Vertices * Edges)
    d[start] = 0;
    ll x = -1; // possible node inside a negative cycle

    for(ll i=0; i<n; i++) { // n-iterations to find a cycle in the last iteration
        x = -1; // default value
        for(auto [u, v, w] : edges) if (d[u] + w < d[v]) {
            d[v] = d[u] + w;
            p[v] = u;
            x = v;
        }
    }

    if (x != -1) { // Negative cycle found
        for(ll i=0; i<n; i++) x = p[x]; // set x to a node, contained in a cycle in p[]

        vll cycle = {x};
        for(ll tmp = p[x]; tmp != x; tmp = p[tmp]) cycle.pb(tmp);
        cycle.pb(x);
        reverse(cycle.begin(), cycle.end());

        //output
        for(auto elem : cycle) cout << elem << ' ';
        cout << endl;
        return;
    }
```

```
    // No Negative cycles
    return;
}
```

Dijkstra

Only Works for Non-Negative Weighted Graph

```
priority_queue<pll, vpll, greater<pll>> pq;
vector<vpll> g(MAX, vpll());
vll d(MAX, INF);

void dijkstra(ll start){
    pq.push({0, start});
    d[start] = 0;

    while( !pq.empty() ){
        auto [p1, u] = pq.top(); pq.pop();
        if (p1 > d[u]) continue;
        for(auto [v, p2] : g[u]){
            if (d[u] + p2 < d[v]){
                d[v] = d[u] + p2;
                pq.push({d[v], v});
            }
        }
    }
}
```

OBS Dijkstra can be modified for the opposite operation: *longest paths*.

Modified Dijkstra for K-Shortest Paths

```
priority_queue<pll, vpll, greater<pll>> pq;
vector<vpll> g(MAX, vpll());
vll cnt(MAX, 0);

// modified Dijkstra for K-Shortest Paths (not necessarily the same distance)
vll dijkstraKSP(ll start, ll end, ll k){ // O(K * M) = O(K * Edges)

    vll ans;
    pq.push({0, start});

    while( cnt[end] < k ){
        auto [dis, u] = pq.top(); pq.pop();

        if (cnt[u] == k) continue;
        cnt[u] += 1;

        if (u == end) { // found a shortest path
            ans.pb(dis); // adding the distance of this path
        }

        for(auto [v, w] : g[u]){
```

```
            pq.push({dis+w, v});
        }
    }

    return ans; // not ordered!
}
```

Extended Dijkstra

Besides the **Shortest Path Distance**,

Also *Computes*:

- **how many shortest paths**;
- **what is the minimum number of edges transversed in any shortest path**;
- **what is the maximum number of edges transversed in any shortest path**;

https://cses.fi/problemset/task/1202

```
priority_queue<pll, vector<pll>, greater<pll>> pq;
vector<vpll> g(MAX, vpll());
vll d(MAX, LLINF);
vll ways(MAX, 0);
vll mn(MAX, LLINF);
vll mx(MAX, -LLINF);

void dijkstra(ll start){
    pq.push({0, start});

    ways[start] = 1;
    d[start] = 0, mn[start] = 0, mx[start] = 0;

    while( !pq.empty() ){
        auto [p1, u] = pq.top(); pq.pop();

        if (p1 > d[u]) continue;

        for(auto [v, p2] : g[u]){
            // reset info, shorter path found, previous ones are discarted
            if (d[u] + p2 < d[v]){

                ways[v] = ways[u];
                mn[v] = mn[u]+1;
                mx[v] = mx[u]+1;
                d[v] = d[u] + p2;

                pq.push({d[v], v});

            }
            // same distance, another path, update info
            else if (d[u] + p2 == d[v]) {
                ways[v] = (ways[v] + ways[u]) % MOD;
                mn[v] = min(mn[v], mn[u]+1);
                mx[v] = max(mx[v], mx[u]+1);
            }
```

```
        }
    }
}
```

Graph Terminology:

Clique: a subset of vertices of an *undirected graph* such that every two distinct vertices in the clique are adjacent.

Articulation Points (Cut Vertices): *A vertex* whose remotion would split a connected component and create more *connected components*.

Bridges (Cut Edges): *An Edge* whose remotion would split a connected component and increase the number of *connected components* present. Also called *isthmus* or *cut arc*.

Topological Sort

Sort a directed graph with no cycles in an order which each source of an edge is visited before the sink of this edge.

Cannot have cycles, because it would create a contradition of which vertices whould come before.

It can be done with a DFS, appending in the reverse order of transversal.

```
vector<vll> g(MAX, vll());
vector<bool> vis;
vll topological;

void dfs(ll u) {
    vis[u] = 1;
    for(auto v : g[u]) if (!vis[v]) dfs(v);
    topological.pb(u);
}

// 1 - indexed
void topological_sort(ll n) {
    vis.assign(n+1, 0);
    topological.clear();
    for(ll i=1; i<=n; i++) if (!vis[i]) dfs(i);
    reverse(topological.begin(), topological.end());
}
```

Math

Matrix

```
struct Matrix{
    vector<vll> M, IND;

    Matrix(vector<vector<int>> mat){
        M = mat;
    }
```

```
    Matrix(int row, int col, bool ind=0){
        M = vector<vector<int>>(row, vector<int>(col, 0));
        if(ind){
            vector<int> aux(row, 0);
            for(int i=0; i<row; i++){
                aux[i] = 1;
                IND.push_back(aux);
                aux[i] = 0;
            }
        }
    }

    Matrix operator +(const Matrix &B) const{ // A+B (sizeof(A) == sizeof(B))
        vector<vector<int>> ans(M.size(), vector<int>(M[0].size(), 0));
        for(int i=0; i<(int)M.size(); i++){
            for(int j=0; j<(int)M[i].size(); j++){
                ans[i][j] = M[i][j] + B.M[i][j];
            }
        }
        return ans;
    }

    Matrix operator *(const Matrix &B) const{ // A*B (A.column == B.row)
        vector<vector<int>> ans;
        for(int i=0; i<(int)M.size(); i++){
            vector<int> aux;
            for(int j=0; j<(int)M[i].size(); j++){
                int sum=0;
                for(int k=0; k<(int)B.M.size(); k++){
                    sum = sum + (M[i][k]*B.M[k][j]);
                }
                aux.push_back(sum);
            }
            ans.push_back(aux);
        }
        return ans;
    }

    Matrix operator ^(const int n) const{ // Need identity Matrix
        if (n == 0) return IND;
        if (n == 1) return (*this);
        Matrix aux = (*this) ^ (n/2);
        aux = aux * aux;
        if(n % 2 == 0)
            return aux;
        else{
            return (*this) * aux;
        }
    }
};
```

Another Version with long long and MOD

```cpp
struct Matrix{
    vector<vll> M, Identity;

    Matrix(vector<vll> mat) {
        M = mat;
    }

    // identity == 0 => Empty matrix constructor
    // identity == 1 => also generates a Identity Matrix
    Matrix(ll row, ll col, bool identity = 0){
        M.assign(row, vll(col, 0));

        if (identity) { // row == col
            Identity.assign(row, vll(col, 0));
            for(ll i=0; i<row; i++)
                Identity[i][i] = 1;
        }
    }

    // A+B ; needs (sizeof(A) == sizeof(B))
    Matrix operator +(const Matrix &B) const{
        ll row = M.size(); ll col = M[0].size();

        Matrix ans(row, col);

        for(ll i=0; i<row; i++){
            for(ll j=0; j<col; j++){
                ans.M[i][j] = (M[i][j] + B.M[i][j]) % MOD;
            }
        }
        return ans;
    }
    // A*B (A.column == B.row)
    Matrix operator *(const Matrix &B) const{
        ll rowA = M.size();
        ll colA; ll rowB = colA = M[0].size();

        Matrix ans(rowB, colA);

        for(ll i=0; i<rowA; i++){
            for(ll j=0; j<colA; j++){
                ll sum=0;
                for(ll k=0; k<rowB; k++){
                    sum += (M[i][k] * B.M[k][j]) % MOD;
                    sum %= MOD;
                }
                ans.M[i][j] = sum;
            }
        }
        return ans;
    }

    Matrix operator ^(const ll n) const{ // Need identity Matrix
        if (n == 0) return Identity;
        if (n == 1) return (*this);
        Matrix aux = (*this) ^ (n/2);
        aux = aux * aux;
```

```
        if(n % 2 == 0) return aux;
        else return (*this) * aux;
    }
};
```

Usage

For faster linear recurrence computation with matrix exponentiation.

Base * Operator^(n) = Result[n]

Example:

Recorrence: dp[i] = dp[i-1] + dp[i-2] + dp[i-3] + dp[i-4] + dp[i-5] + dp[i-6]

Base Matrix [dp[5], dp[4], dp[3], dp[2], dp[1], dp[0]]

- Operator Matrix ^ 1 [1, 1, 0, 0, 0, 0] [1, 0, 1, 0, 0, 0] [1, 0, 0, 1, 0, 0] [1, 0, 0, 0, 1, 0] [1, 0, 0, 0, 0, 1] [1, 0, 0, 0, 0, 0]

= Result Matrix [dp[n+5], dp[n+4], dp[n+3], dp[n+2], dp[n+1], dp[n]]

```
int32_t main(){ sws;
    ll n; cin >> n;
    Matrix op(6, 6, 1);
    op.M[0] = {1, 1, 0, 0, 0, 0};
    op.M[1] = {1, 0, 1, 0, 0, 0};
    op.M[2] = {1, 0, 0, 1, 0, 0};
    op.M[3] = {1, 0, 0, 0, 1, 0};
    op.M[4] = {1, 0, 0, 0, 0, 1};
    op.M[5] = {1, 0, 0, 0, 0, 0};

    Matrix base(vector(1, vll({16, 8, 4, 2, 1, 1})));
    if (n <= 5) cout << base.M[0][5-n] << endl;
    else {
        op = op^(n-5);
        Matrix ans = base * op;
        cout << ans.M[0][0] << endl;
    }
}
```

Series Theory

Closed formulas for some sequences

Natural Number Summation (PA):

$ 1 + 2 + 3 + 4 + 5 + ... + n-1 + n $

$ = \sum_{i=1}^n i $

= $ \frac{ n(n+1) }{ 2 } $

Natural Number Quadratic Summation:

$ 1 + 4 + 9 + 16 + 25 + ... + (n-1)^2 + n^2 $

$ = \sum_{i=1}^n i^2 $

= $ \frac{ n(n+1)(2n+1) }{ 6 } $

Triangular Numbers Summation:

$ 1 + 3 + 6 + 10 + 15 + ... + \frac{(n-1)(n)}{2} + \frac{(n)(n+1)}{2} $

$ = \sum_{i=1}^n \frac{i(i+1)}{2} = \frac{1}{2}(\sum_{i=1}^n i^2 + \sum_{i=1}^n i) $

$ = \frac{1}{2} (\frac{ n(n+1) }{ 2 } + \frac{ n(n+1)(2n+1) }{ 6 }) $

Misc

Minimum Excluded (MEX)

! TODO 😦

let x be an array:

$ MEX(x) <= len(x) $

ModularArithmetic

Overloading Operations Struct

```
const int MOD = 1e9+7;

struct intM{
    long long val = 0;

    intM(long long n=0){
        val = n%MOD;
        if (val < 0) val += MOD;
    }

    bool operator ==(const intM& b) const{
        return (val == b.val);
    }

    intM operator +(const intM& b) const{
        return (val + b.val) % MOD;
    }

    intM operator -(const intM& b) const{
        return (val - b.val + MOD) % MOD;
    }

    intM operator *(const intM& b) const{
        return (val*b.val) % MOD;
```

```
    }

    intM operator ^(const intM& b) const{ // fast exp [(val^b) mod M];
        if (b == 0) return 1;
        if (b == 1) return (*this);
        intM tmp = (*this)^(b.val/2); // diria que não vale a pena definir "/", "/" já é
a multiplicação pelo inv
        if (b.val % 2 == 0) return tmp*tmp; // diria que não vale a pena definir "%",
para não confidir com o %MOD
        else return tmp * tmp * (*this);
    }

    intM operator /(const intM& b) const{
        return (*this) * (b ^ (MOD-2));
    }
};
```

Basic operations with combinatorics

Also contains combinatorics operations

```
struct OpMOD{
    vll fact, ifact;

    OpMOD () {}

    // overloaded constructor that computes factorials
    OpMOD(ll n){ // from fact[0] to fact[n]; O(n)
        fact.assign(n+1 , 1);
        for(ll i=2; i<=n; i++) fact[i] = mul(fact[i-1], i);

        ifact.assign(n+1, 1);
        ifact[n] = inv(fact[n]);
        for(ll i=n-1; i>=0; i--) ifact[i] = mul(ifact[i+1], i+1);
    }

    ll add(ll a, ll b){
        return ( (a%MOD) + (b%MOD) ) % MOD;
    }

    ll sub(ll a, ll b){
        return ( ((a%MOD) - (b%MOD)) + MOD ) % MOD;
    }

    ll mul(ll a, ll b){
        return ( (a%MOD) * (b%MOD) ) % MOD;
    }

    ll fast_exp(ll n, ll i){ // n ** i
        if (i == 0) return 1;
        if (i == 1) return n;
        ll tmp = fast_exp(n, i/2);
        if (i % 2 == 0) return mul(tmp, tmp);
        else return mul( mul(tmp, tmp), n );
    }
```

```
    ll inv(ll n){
        return fast_exp(n, MOD-2);
    }

    ll div(ll a, ll b){
        return mul(a, inv(b));
    }

    // n! / (n! (n-k)! )
    ll combination(ll n, ll k){ // "Combinação/Binomio de Newton"
        return mul( mul(fact[n], ifact[k]) , ifact[n-k]);
    }

    // n! / (n-k)!
    ll disposition(ll n, ll k){ // "Arranjo Simples"
        return mul(fact[n], ifact[n-k]);
    }

    // n!
    ll permutation(ll n){ // "Permutação Simples"
        return fact[n];
    }

    // n! / (k1! k2! k3!)
    ll permutationRepetition(ll n, vll x) { // "Permutação com Repetição"
        ll tmp = fact[n];
        for(auto k : x) tmp = mul(tmp, ifact[k]);
        return tmp;
    }

    // (n+m-1)! / ((n-1)! (m!))
    ll starBars(ll n, ll m) { // "pontos e virgulas"
        // n Groups -> n-1 Bars
        // m Stars
        return combination(n+m-1, m);
    }

    // !n = (n-1) * ( !(n-1) + !(n-2) )
    vll subfactorial; // derangements
    void computeSubfactorials(ll n) {
        subfactorial.assign(n+1, 0);
        subfactorial[0] = 1;
        // !0 = 1
        // !1 = 0
        for(ll i=2; i<=n; i++) {
            subfactorial[i] = mul( (i-1) , add(subfactorial[i-1], subfactorial[i-2]) );
        }
    }
};

// remember to pass a number delimeter (n) to precompute factorials
OpMOD op;
```

Number-Theory

Combinatorics Theory

Stars and Bars

Also called "sticks and stones", "balls and bars", and "dots and dividers"

$ x_1 + x_2 + ... + x_n = m $

Example: (n = 3, m = 7)

★ ★ ★ ★ | ★ | ★ ★

n Groups; *n-1* Bars; *m* Stars;

Solution

C(n+m-1, n-1) = (n+m-1)! / ((n-1)! (m)!)

Proof

Elements = Bars + Stars = (n-1) + m = n+m-1; Repetition of Bars = n-1 Repetition of Stars = m

Therefore, it's a simple *permutation with repetition*.

$ P^{(n+m-1)}_{(n-1,m)} = C (n+m-1, m)$

Derangement

In combinatorial mathematics, a derangement is a permutation of the elements of a set, such that no element appears in its original position. In other words, a derangement is a permutation that has no fixed points.

Counting derangements

The number of derangements of a set of size n is known as the subfactorial of n or the n-th derangement number or n-th de Montmort number.

A subfactorial is noted as:

!n = (n-1) * (!(n-1) + !(n-2)), for n >= 2.

!1 = 0 *!0* = 1

Crivo de Eratóstenes

```
vector<int> crivo(int n){
    int max = 1e6;
    vector<int> primes {2};
    bitset<max> sieve;
    sieve.set();

    for(int i=3; i<=n; i+=2){
        if(sieve[i]){ // i is prime
            primes.push_back(i);

            for(int j= i*i; j<=n; j += 2*i) // sieving all odd multiples of i >= i*i
```

```
                sieve[j] = false;
        }
    }

    return primes;
}
```

Optimized

```
// O (N log^2(N) ) -> Teorema de Merten
vll primes {2, 3};
set<ll> isPrime = {2, 3};
void eratostenes(ll n){
    bitset<MAX> sieve;
    sieve.set();
    for(ll i=5, step=2; i<=n; i+=step, step = 6 - step){
        if(sieve[i]){ // i is prime
            primes.push_back(i);
            isPrime.insert(i);
            for(ll j= i*i; j<=n; j += 2*i) // sieving all odd multiples of i >= i*i
                sieve[j] = false;
        }
    }
}
```

Factorization

Trial Division with precomputed primes

Complexity: O(sqrt(N))

Returns: a vector containing all the primes that divides *N* (There can be multiples instances of a prime and it is ordered)

```
vll eratostenes(ll n){
    vll primes {2, 3};
    bitset<MAX> sieve;
    sieve.set();

    for(ll i=5, step=2; i<=n; i+=step, step = 6 - step){
        if(sieve[i]){ // i is prime
            primes.push_back(i);

            for(ll j= i*i; j<=n; j += 2*i) // sieving all odd multiples of i >= i*i
                sieve[j] = false;
        }
    }

    return primes;
}

vll primes = eratostenes(MAX);
```

```
vector<ll> factorization(ll n){
    vll factors;

    for(ll p : primes){
        if (p*p > n) break;
        while(n % p == 0){
            factors.pb(p);
            n /= p;
        }
    }

    if (n > 1) factors.pb(n);

    return factors;
}
```

Pollard Rho

Complexity: better than O(sqrt(N)) 😄

Returns: a vector containing all the primes that divides *N* (There can be multiples instances of a prime and it is *not* ordered)

```
ll mul(ll a, ll b, ll m) {
    ll ret = a*b - (ll)((ld)1/m*a*b+0.5)*m;
    return ret < 0 ? ret+m : ret;
}

ll pow(ll a, ll b, ll m) {
    ll ans = 1;
    for (; b > 0; b /= 2ll, a = mul(a, a, m)) {
        if (b % 2ll == 1)
            ans = mul(ans, a, m);
    }
    return ans;
}

bool prime(ll n) {
    if (n < 2) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0) return 0;

    ll r = __builtin_ctzll(n - 1), d = n >> r;
    for (int a : {2, 325, 9375, 28178, 450775, 9780504, 795265022}) {
        ll x = pow(a, d, n);
        if (x == 1 or x == n - 1 or a % n == 0) continue;

        for (int j = 0; j < r - 1; j++) {
            x = mul(x, x, n);
            if (x == n - 1) break;
        }
        if (x != n - 1) return 0;
    }
    return 1;
```

```
}

ll rho(ll n) {
    if (n == 1 or prime(n)) return n;
    auto f = [n](ll x) {return mul(x, x, n) + 1;};

    ll x = 0, y = 0, t = 30, prd = 2, x0 = 1, q;
    while (t % 40 != 0 or gcd(prd, n) == 1) {
        if (x==y) x = ++x0, y = f(x);
        q = mul(prd, abs(x-y), n);
        if (q != 0) prd = q;
        x = f(x), y = f(f(y)), t++;
    }
    return gcd(prd, n);
}

vector<ll> fact(ll n) {
    if (n == 1) return {};
    if (prime(n)) return {n};
    ll d = rho(n);
    vector<ll> l = fact(d), r = fact(n / d);
    l.insert(l.end(), r.begin(), r.end());
    return l;
}
```

OrderedSet

Policy Based Data Structures (PBDS)

Ordered Set

```
// * Ordered Set and Map
// find_by_order(i) -> iterator to elem with index i; O(log(N))
// order_of_key(i) -> index of key; O(log(N))

#include <bits/extc++.h>
using namespace __gnu_pbds;
template <class T> using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
```

Ordered Map

```
// * Ordered Set and Map
// find_by_order(i) -> O(log(N))
// order_of_key(i) -> O(log(N))

#include <bits/extc++.h>
using namespace __gnu_pbds;
template <class K, class V> using ordered_map = tree<K, V, less<K>, rb_tree_tag,
tree_order_statistics_node_update>;
```

Ordered Multiset

Ordered Set pode ser tornar um multiset se utilizar um pair do valor com um index distinto. pll{val, t}, 1 <= t <= n

Observação:

O set não precisa conter a chave sendo buscada pelo *order_of_key()*.

order_of_key() returns index starting from 0; [0, n)

Problemas

Consegue computar em O(log(N)), quantos elementos são menores que K, utilizando o index.

Searching-Sorting

Binary search

Finds the first element that changes value in any monotonic function

Maximum

Monotonically Decreasing [1, 1, 1, 1, 0, 0, 0, 0]

```
bool f(ll a){
    // Add desired function here
    return true;
}

ll search(ll l=0, ll r=1e9, ll ans=0){
    while(l <= r) { // [l, r]
        ll mid = (l+r)/2;
        if(f(mid)) { // (mid, r]
            ans = mid;
            l = mid+1;
        }
        else { // [l; mid)
            r = mid-1;
        }
    }
    return ans;
}
```

Minimum

Monotonically Increasing [0, 0, 0, 0, 1, 1, 1, 1]

```
bool f(ll mid){
    // Add desired function here
    return true;
```

```
}

ll bSearch(ll l=0, ll r=1e9, ll ans=0){
    while(l <= r) { // [l, r]
        ll mid = (l+r)/2;
        if (f(mid) ) { // [l, mid)
            ans = mid;
            r = mid-1;
        }
        else { // (mid, r]
            l = mid+1;
        }
    }
    return ans;
}
```

Merge sort

Merge Sort with number of inversions counter.

```
int merge(vector<int> &v, int l, int mid, int r){
    int i=l, j=mid+1, swaps=0;
    vector<int> ans;

    while(i <= mid or j <= r){

        if(j > r or (v[i] <= v[j] and i<=mid)){
            ans.push_back(v[i]);
            i++;
        }
        else if(i > mid or (v[j] < v[i] and j <= r)){
            ans.push_back(v[j]);
            j++;
            swaps = swaps + abs(mid+1-i);
        }
    }

    for(int i=l; i<=r; i++)
        v[i] = ans[i-l];

    return swaps;
}

int merge_sort(vector<int> &v, vector<int> &ans, int l, int r){
    if(l==r){
        ans[l] = v[l];
        return 0;
    }


    int mid = (l+r)/2, swaps = 0;
    swaps += merge_sort(v, ans, l, mid);
    swaps += merge_sort(v, ans, mid+1, r);
    swaps += merge(ans, l, mid, r);
```

```
    return swaps;
}
```

Updated

Directly updates the *v* vector. Also return the number of swaps (inversions).

O(N log(N))

```
// O(N)
ll merge(vll &v, ll l, ll r) {
    ll i = l, mid = (l+r)/2, j = mid+1, swaps = 0;
    vll ans;

    while(i <= mid or j <= r) {

        if(j > r or (v[i] <= v[j] and i <= mid)) {
            ans.pb(v[i]);
            i += 1;
        }

        else if(i > mid or (v[j] < v[i] and j <= r)){
            ans.pb(v[j]);
            j += 1;
            swaps += (mid-1)+1; // mid-i+1 = elements remaining in the left subarray
(same number of elements that will be swaped to the right)
        }
    }

    for(ll k=l; k<=r; k++) v[k] = ans[k-l];

    return swaps;
}

// O(log2(N))
ll merge_sort(vll &v, ll l, ll r){
    if(l == r) return 0;

    ll mid = (l+r)/2, swaps = 0;
    swaps += merge_sort(v, l, mid);
    swaps += merge_sort(v, mid+1, r);
    swaps += merge(v, l, r);

    return swaps;
}
```

Ternary Search

Complexity: O(log(n))

Requires EPS!, precision usually defined in the question text

```
ld f(ld d){
    // function here
}

ld ternary_search(ld l, ld r){ // for min value
    while(r - l > EPS){
        // divide into 3 equal parts and eliminate one side
        ld m1 = l + (r - l) / 3;
        ld m2 = r - (r - l) / 3;

        if (f(m1) < f(m2)){
            r = m2;
        }
        else{
            l = m1;
        }
    }
    return f(l);
}
```

Segtree

Segtree with sum, max, min

```
#define int long long // need long long ?
// !  Initialize N  !
int L = 1, N; // L = 1 = left limit; N = right limit
// 1 - indexed
class SegmentTree {
    public:
        struct node{
            int psum, mx, mn;
        };

        node merge(node a, node b){
            node tmp;
            // merge operaton:
            tmp.psum = a.psum + b.psum;
            tmp.mx = max(a.mx, b.mx);
            tmp.mn = min(a.mn, b.mn);
            return tmp;
        }

        vector<node> tree;
        vector<int> v;

        SegmentTree() {
            v.assign(N+2, 0);
            tree.assign(N*4 + 10, node{0, 0, 0});
        }

        void build (int l=L, int r=N, int i=1) {
```

```
            if (l == r){
                // leaf element
                node tmp{v[l], v[l], v[l]};
                tree[i] = tmp;
            }
            else{
                int mid = (l+r)/2;
                build(l, mid, 2*i);
                build(mid+1, r, 2*i+1);
                tree[i] = merge(tree[2*i], tree[2*i+1]);
            }
        }

        void point_update(int idx=1, int val=0, int l=L, int r=N, int i=1){
            if (l == r){
                // update operation to leaf
                node tmp{val, val, val};
                tree[i] = tmp;
            }
            else{
                int mid = (l+r)/2;
                if (idx <= mid) point_update(idx, val, l, mid, 2*i);
                else point_update(idx, val, mid+1, r, 2*i+1);
                tree[i] = merge(tree[2*i], tree[2*i+1]);
            }
        }
        node range_query(int left=L, int right=N, int l=L, int r=N, int i=1){
            // left/right are the range limits for the update query
            // l / r are the variables used for the vertex limits
            if (right < l or r < left){ // out of bounds
                // null element
                node tmp{0, -INF, INF};
                return tmp;
            }
            else if (left <= l and r <= right){ // contained interval
                return tree[i];
            }
            else{ // partially contained
                int mid = (l+r)/2;
                node ansl = range_query(left, right, l, mid, 2*i);
                node ansr = range_query(left, right, mid+1, r, 2*i+1);
                return merge(ansl, ansr);
            }
        }
};
```

Implicit Segtree or Sparse Segtree

Creates vertices only when needed. Uncreated vertices are considered to have default values.

TODO Needs testing!

```
// Remember to set R value !!
ll L=1, R;
struct SegImplicit {
```

```
    struct Node{
        ll ps = 0, Lnode = 0, Rnode = 0;
    };
    ll idx = 2; // 1-> root / 0-> zero element
    Node tree[4*MAX];

    ll merge(Node a, Node b){
        return a.ps + b.ps;
    }

    void increase(ll pos, ll x, ll l=L, ll r=R, ll i=1) {
        if (l == r) {
            tree[i].ps += x;
            return;
        }

        ll mid = (l+r)/2;

        if (pos <= mid) {
            if (tree[i].Lnode == 0) tree[i].Lnode = idx++; // new vertex
            increase(pos, x, l, mid, tree[i].Lnode);
        }
        else {
            if (tree[i].Rnode == 0) tree[i].Rnode = idx++;
            increase(pos, x, mid+1, r, tree[i].Rnode);
        }

        tree[i].ps = merge(tree[ tree[i].Lnode ], tree[ tree[i].Rnode ]);
    }

    Node query(ll left, ll right, ll l=L, ll r=R, ll i=1) {
        if (right < l or r < left)
            return Node{};

        if (left <= l and r <= right)
            return tree[i];

        ll mid = (l+r)/2;
        Node ansl, ansr;
        if (tree[i].Lnode != 0) ansl = query(left, right, l, mid, tree[i].Lnode);
        if (tree[i].Rnode != 0) ansr = query(left, right, mid+1, r, tree[i].Rnode);

        return Node{merge(ansl, ansr), 0, 0};
    }
};
```

Iterative P-sum Classic Segtree with MOD

```
struct Segtree{
    vector<ll> t;
    int n;

    Segtree(int n){
        this->n = n;
        t.assign(2*n, 0);
```

```
    }

    ll merge(ll a, ll b){
        return (a + b) % MOD;
    }

    void build(){
        for(int i=n-1; i>0; i--)
            t[i]=merge(t[i<<1], t[i<<1|1]);
    }

    ll query(int l, int r){ // [l, r]
        ll resl=0, resr=0;
        for(l+=n, r+=n+1; l<r; l>>=1, r>>=1){
            if(l&1) resl = merge(resl, t[l++]);
            if(r&1) resr = merge(t[--r], resr);
        }
        return merge(resl, resr);
    }

    void update(int p, ll value){
        p+=n;
        for(t[p]=(t[p] + value)%MOD; p >>= 1;)
            t[p] = merge(t[p<<1], t[p<<1|1]);
    }

};
```

Inverted Segtree

Range_increase -> using delta encoding

Point_update -> adding all values during transversal

```
int L = 1, N; // L = 1 = left limit; N = right limit
class SegmentTree {
    public:
        struct node{
            int psum;
        };

        node tree[4*MAX];
        int v[MAX];

        // requires minimum index and maximum index
        SegmentTree() {
            memset(v, 0, sizeof(v));
        }

        node merge(node a, node b){
            node tmp;
            // merge operaton:
            tmp.psum = a.psum + b.psum;
            //
            return tmp;
```

```
        }

        void build(int l=L, int r=N, int i=1) {
            if (l == r){
                node tmp;
                // leaf element
                tmp.psum = v[l];
                //
                tree[i] = tmp;
            }
            else{
                int mid = (l+r)/2;
                build(l, mid, 2*i);
                build(mid+1, r, 2*i+1);
                tree[i] = node{0};
            }
        }
        node point_query(int idx=1, int l=L, int r=N, int i=1){
            if (l == r){
                return tree[i];
            }
            else{
                int mid = (l+r)/2;
                if (idx <= mid)
                    return merge(tree[i], point_query(idx, l, mid, 2*i));
                else
                    return merge(tree[i], point_query(idx, mid+1, r, 2*i+1));
            }
        }
        void range_increase(int val, int left=L, int right=N, int l=L, int r=N, int i=1)
{
            // left/right are the range limits for the update query
            // l / r are the variables used for the vertex limits
            if (right < l or r < left){
                return;
            }
            else if (left <= l and r <= right){
                tree[i] = merge(tree[i], node{val});
            }
            else{
                int mid = (l+r)/2;
                range_increase(val, left, right, l, mid, 2*i);
                range_increase(val, left, right, mid+1, r, 2*i+1);
            }
        }
};
```

Recursive Segtree with Lazy propagation

Sum range query, increase range query

```
ll L = 1, R;
struct SegtreeLazy{
    vector<ll> tree, lazy, v;
```

```
    SegtreeLazy() {
        tree.assign(4*(R-L+2), 0);
        lazy.assign(4*(R-L+2), 0);
        v.assign((R-L+2), 0);
    }

    void build(ll l=L, ll r=R, ll i=1) {
        if (l == r) tree[i] = v[l];
        else{
            ll mid = (l+r)/2;
            build(l, mid, 2*i);
            build(mid+1, r, 2*i+1);
            tree[i] = tree[2*i] + tree[2*i+1];
        }
        lazy[i] = 0;
    }

    void propagate(ll l, ll r, ll i){
        if(lazy[i]) {
            tree[i] += lazy[i] * (r-l+1);
            if(l != r){
                lazy[2*i] += lazy[i];
                lazy[2*i+1] += lazy[i];
            }
            else v[l] += lazy[i]; // update array
            lazy[i] = 0;
        }
    }

    // [left, right] = (selected interval for the query)
    // l, r = the variables used for the vertex limits

    // increase function adds 'val' to [left, right]
    void increase(ll left=L, ll right=R, ll val=0, ll l=L, ll r=R, ll i=1){
        propagate(l, r, i);

        if (right < l or r < left) return;

        else if (left <= l and r <= right){
            lazy[i] += val;
            propagate(l, r, i);
        }

        else{
            ll mid = (l+r)/2;
            increase(left, right, val, l, mid, 2*i);
            increase(left, right, val, mid+1, r, 2*i+1);
            tree[i] = tree[2*i] + tree[2*i+1];
        }
    }

    ll query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1){
        propagate(l, r, i);

        if (right < l or r < left) return 0;

        else if (left <= l and r <= right) return tree[i];
```

```
        else{
            ll mid = (l+r)/2;
            return (
                query(left, right, l, mid, 2*i) +
                query(left, right, mid+1, r, 2*i+1)
            );
        }
    }
};
```

Range Minimum Query, Update (Assignment) Query

```
ll L = 1, R;
struct SegtreeLazy{
    vll tree, lazy, v;

    SegtreeLazy() {
        tree.assign(4*(R-L+2), 0);
        lazy.assign(4*(R-L+2), 0);
        v.assign((R-L+2), 0);
    }

    void build(ll l=L, ll r=R, ll i=1) {
        if (l == r) tree[i] = v[l];
        else{
            ll mid = (l+r)/2;
            build(l, mid, 2*i);
            build(mid+1, r, 2*i+1);
            tree[i] = min(tree[2*i], tree[2*i+1]);
        }
        lazy[i] = LLINF; // min query default value
    }

    void propagate(ll l, ll r, ll i){
        if(lazy[i] != LLINF) { // need to propagate lazy
            tree[i] = lazy[i];
            if(l != r)
                lazy[2*i] = lazy[2*i+1] = lazy[i];
            else
                v[l] = lazy[i]; // update 'v' vector
            lazy[i] = LLINF;
        }
    }

    // [left, right] = (selected interval for the query)
    // l, r = the variables used for the vertex limits

    // update function changes all elements in [left, right] to val
    void update(ll left=L, ll right=R, ll val=0, ll l=L, ll r=R, ll i=1){
        propagate(l, r, i);

        if (right < l or r < left) return;

        else if (left <= l and r <= right){
```

```
            lazy[i] = val;
            propagate(l, r, i);
        }

        else{
            ll mid = (l+r)/2;
            update(left, right, val, l, mid, 2*i);
            update(left, right, val, mid+1, r, 2*i+1);
            tree[i] = min(tree[2*i], tree[2*i+1]);
        }
    }

    ll query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1){
        propagate(l, r, i);

        if (right < l or r < left) return LLINF;

        else if (left <= l and r <= right) return tree[i];

        else{
            ll mid = (l+r)/2;
            return min(
                query(left, right, l, mid, 2*i),
                query(left, right, mid+1, r, 2*i+1)
            );
        }
    }
};
```

For MAX Query

Use the same code as min segtree, change:

min() -> **max() LLINF** -> **-LLINF**

Complex Lazy Problems

Requirements: to be able to *propagate/push* the lazy stored updates. In other words, the property of **Aggregation:** to transfer the data saved in *lazy[i]* to *tree[i]* and also the property of **Composition:** to push the updates to the children (*lazy[i]* to *lazy[2*i]* and *lazy[i]* to *lazy[2*i+1]*).

Example1:

range increase query by x range sum of the squares = a[l]^2 + ... + a[r]^2

```
// import this struct
struct intM{};

ll L = 1, R; // Declare R = n and also use build() afterwards -_- macake
struct SegtreeLazy{

    struct Node {
        intM val;
        intM sqr;
```

```
    };

    Node merge(Node a, Node b) {
        return Node{
            a.val + b.val,
            a.sqr + b.sqr
        };
    }

    vector<intM> v, lazy;
    vector<Node> tree;

    SegtreeLazy() {
        tree.assign(4*(R-L+2), Node{});
        lazy.assign(4*(R-L+2), intM{});
        v.assign((R-L+2), intM{});
    }

    void build(ll l=L, ll r=R, ll i=1) {
        if (l == r) {
            tree[i] = Node{
                v[l],
                v[l] * v[l]
            };
        }
        else{
            ll mid = (l+r)/2;
            build(l, mid, 2*i);
            build(mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
        lazy[i] = intM{};
    }

    void propagate(ll l, ll r, ll i){
        if(lazy[i].val) {
            tree[i].sqr = tree[i].sqr + (intM(2) * lazy[i] * tree[i].val);
            tree[i].sqr = tree[i].sqr + (lazy[i] * lazy[i] * intM(r-l+1));

            tree[i].val = tree[i].val + (lazy[i] * intM(r-l+1));

            if(l != r){
                lazy[2*i] = lazy[2*i] + lazy[i];
                lazy[2*i+1] = lazy[2*i+1] + lazy[i];
            }

            lazy[i] = intM{};
        }
    }

    // [left, right] = (selected interval for the query)
    // l, r = the variables used for the vertex limits

    // increase function adds 'val' to [left, right]
    void increase(ll left=L, ll right=R, ll val=0, ll l=L, ll r=R, ll i=1){
        propagate(l, r, i);
```

```
        if (right < l or r < left) return;

        else if (left <= l and r <= right){
            lazy[i] = lazy[i] + intM(val);
            propagate(l, r, i);
        }

        else{
            ll mid = (l+r)/2;
            increase(left, right, val, l, mid, 2*i);
            increase(left, right, val, mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }

    Node query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1){
        propagate(l, r, i);

        if (right < l or r < left) return Node{};

        else if (left <= l and r <= right) return tree[i];

        else{
            ll mid = (l+r)/2;
            return merge(
                query(left, right, l, mid, 2*i),
                query(left, right, mid+1, r, 2*i+1)
            );
        }
    }
};

int32_t main(){ sws;
    ll n, q; cin >> n >> q;
    R = n;

    SegtreeLazy st;

    for(ll i=1; i<=n; i++) {
        ll x; cin >> x;
        st.v[i] = intM(x);
    }

    st.build();

    while(q--) {
        char c; cin >> c;
        if (c == 'u') {
            ll l, r, x; cin >> l >> r >> x;
            st.increase(l, r, x);
        }
        else {
            ll l, r; cin >> l >> r;
            cout << st.query(l, r).sqr.val << endl;
        }
    }
}
```

Recursive Classic Segtree

Data structure that creates parent vertices for a linear array to do faster computation with binary agregation.

<table>
<tr><td colspan="16">1: [0, 16)</td></tr>
<tr><td colspan="8">2: [0, 8)</td><td colspan="8">3: [8, 16)</td></tr>
<tr><td colspan="4">4: [0, 4)</td><td colspan="4">5: [4, 8)</td><td colspan="4">6: [8, 12)</td><td colspan="4">7: [12, 16)</td></tr>
<tr><td colspan="2">8:
[0, 2)</td><td colspan="2">9:
[2, 4)</td><td colspan="2">10:
[4, 6)</td><td colspan="2">11:
[6, 8)</td><td colspan="2">12:
[8, 10)</td><td colspan="2">13:
[10, 12)</td><td colspan="2">14:
[12, 14)</td><td colspan="2">15:
[14, 16)</td></tr>
<tr><td>16:
0</td><td>17:
1</td><td>18:
2</td><td>19:
3</td><td>20:
4</td><td>21:
5</td><td>22:
6</td><td>23:
7</td><td>24:
8</td><td>25:
9</td><td>26:
10</td><td>27:
11</td><td>28:
12</td><td>29:
13</td><td>30:
14</td><td>31:
15</td></tr>
</table>

Clearer version (min-seg)

```
// 1 indexed segtree for minimum
ll L=1, R;
struct Segtree {
    struct Node {
        ll mn;
    };

    vector<Node> tree;
    vll v;

    Segtree(ll n) {
        v.assign(n+1, 0);
        tree.assign(4*(n+1), Node{});
        R = n;
    }

    Node merge(Node a, Node b) {
        Node tmp;
        // merge operaton:
        tmp.mn = min(a.mn, b.mn);
        //
        return tmp;
    }

    void build( ll l=L, ll r=R, ll i=1 ) {
        if (l == r) {
            Node tmp;
            // leaf element:
            tmp.mn = v[l];
            //
            tree[i] = tmp;
        }
        else {
            ll mid = (l+r)/2;
            build(l, mid, 2*i);
            build(mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }
```

```
    void point_update(ll idx=1, ll val=0, ll l=L, ll r=R, ll i=1 ) {
        if (l == r) {
            // update operation:
            Node tmp{val};
            //
            tree[i] = tmp;
        }
        else {
            ll mid = (l+r)/2;
            if (idx <= mid) point_update(idx, val, l, mid, 2*i);
            else point_update(idx, val, mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }

    Node range_query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1) {
        // left/right are the range limits for the update query
            // l / r are the variables used for the vertex limits
            if (right < l or r < left){
                // null element
                Node tmp{INF};
                //
                return tmp;
            }
            else if (left <= l and r <= right) return tree[i];
            else{
                int mid = (l+r)/2;
                Node ansl = range_query(left, right, l, mid, 2*i);
                Node ansr = range_query(left, right, mid+1, r, 2*i+1);
                return merge(ansl, ansr);
            }
    }
};
```

Even more polished (sum-seg):

```
// 1 indexed segtree for sum
ll L=1, R;
struct Segtree {
    struct Node {
        // null element:
        ll ps = 0;
    };

    vector<Node> tree;
    vll v;

    Segtree(ll n) {
        v.assign(n+1, 0);
        tree.assign(4*(n+1), Node{});
        R = n;
    }

    Node merge(Node a, Node b) {
        return Node{
```

```
            // merge operaton:
            a.ps + b.ps
        };
    }

    void build( ll l=L, ll r=R, ll i=1 ) {
        if (l == r) {
            tree[i] = Node {
                // leaf element:
                v[l]
            };
        }
        else {
            ll mid = (l+r)/2;
            build(l, mid, 2*i);
            build(mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }

    void update(ll idx=1, ll val=0, ll l=L, ll r=R, ll i=1 ) {
        if (l == r) {
            tree[i] = Node{
                // update operation:
                val
            };
        }
        else {
            ll mid = (l+r)/2;
            if (idx <= mid) update(idx, val, l, mid, 2*i);
            else update(idx, val, mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }

    Node query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1) {
        // left/right are the range limits for the update query
            // l / r are the variables used for the vertex limits
            if (right < l or r < left){
                // null element:
                return Node{};
            }
            else if (left <= l and r <= right) return tree[i];
            else{
                int mid = (l+r)/2;
                return merge(
                    query(left, right, l, mid, 2*i),
                    query(left, right, mid+1, r, 2*i+1)
                );
            }
    }
};
```

Avisos

Details

0 or 1-indexed, depends on the arguments used as default value

Uses a **struct node** to define node/vertex properties. *Default:* psum

Uses a **merge function** to define how to join nodes

Parameters

left and **right**: parameters that are the range limits for the range query

l and **r**: are auxilary variables used for delimiting a vertex boundaries

idx: index of the leaf node that will be updated

val: value that will be inserted to the idx node

Atributes

Tree: node array

v: vector that are used for leaf nodes

Methods

O(n):

build(l, r, i): From **v** vector, constructs Segtree

O(log(N))

point_update(idx, l, r, i, val): updates leaf node with *idx* index to *val* value. No return value

range_query(left, right, l, r, i): does a range query from *left* to *right* (inclusive) and returns a node with the result

Requires

MAX variable

Problems

- Range Sum Query, point update
- Range Max/Min Query, point update
- Range Xor Query, point update

Strings

Booth's Algorithm

An efficient algorithm which uses a modified version of KMP to compute the **least amount of rotation needed** to reach the **lexicographically minimal string rotation**.

A *rotation* of a string can be generated by moving characters one after another from beginning to end. For example, the rotations of *acab* are *acab*, *caba*, *abac*, and *baca*.

```cpp
// Booth Algorithm
ll least_rotation(string s) { // O(n)
    ll n = s.length();
    vll f(2*n, -1);
    ll k = 0;
    for(ll j=1; j<2*n; j++) {
        ll i = f[j-k-1];
        while(i != -1 and s[j % n] != s[(k+i+1) % n] ) {
            if (s[j % n] < s[(k+i+1) % n])
                k = j - i - 1;
            i = f[i];
        }
        if (i == -1 and s[j % n] != s[(k+i+1) % n] ) {
            if (s[j % n] < s[(k+i+1) % n])
                k = j;
            f[j - k] = -1;
        }
        else
            f[j - k] = i + 1;
    }
    return k;
}

int32_t main(){ sws;
    string s; cin >> s;
    ll n = s.length();
    ll ans_idx = least_rotation(s);
    string tmp = s + s;
    cout << tmp.substr(ans_idx, n) << endl;
}
```

Knuth–Morris–Pratt algorithm (KMP)

AKA Prefix function

todo 😄

SUFFIX ARRAY

Complexity: O(n * log (n))

Returns: An array with size *n*, whose values are the indexes from the longest substring (0) to the smallest substring (n) after ordering it lexicographically. Example:

```
Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ---------------->     1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana
```

```
So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

Solves: Finding the number of all distint substrings of a string. Done by adding all sizes of the substrings (size[i] = *total_size* - sa[i]) and subtracting all lcp's.

```
vector<int> suffix_array(string s) {
    s += "$";
    int n = s.size(), N = max(n, 260);
    vector<int> sa(n), ra(n);
    for (int i = 0; i < n; i++) sa[i] = i, ra[i] = s[i];

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

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

        for (int i = 1, r = 0; i < n; i++) nra[sa[i]] = r += ra[sa[i]] !=
            ra[sa[i-1]] or ra[(sa[i]+k)%n] != ra[(sa[i-1]+k)%n];
        ra = nra;
        if (ra[sa[n-1]] == n-1) break;
    }
    return vector<int>(sa.begin()+1, sa.end());
}
```

KASAI's ALGORITHM FOR LCP (longest common prefix)

Complexity: O(log (n))

Returns: An array of size *n* (like the suffix array), whose values indicates the lenght of the longest common prefix beetwen *sa[i]* and *sa[i+1]*

```
vector<int> kasai(string s, vector<int> sa) {
    int n = s.size(), k = 0;
    vector<int> ra(n), lcp(n);
    for (int i = 0; i < n; i++) ra[sa[i]] = i;

    for (int i = 0; i < n; i++, k -= !!k) {
        if (ra[i] == n-1) { k = 0; continue; }
        int j = sa[ra[i]+1];
        while (i+k < n and j+k < n and s[i+k] == s[j+k]) k++;
        lcp[ra[i]] = k;
    }
    return lcp;
}
```

TRIE

AKA: prefix tree, digital tree

MAX Should be the number of maximum nodes to be created.

```
struct Trie{
    ll trie[MAX][26];
    bool isWordEnd[MAX];
    ll nxt = 1, wordsCnt = 0;

    void add(string s){ // O(n)
        ll node = 0;
        for(auto c: s) {
            if(trie[node][c-'a'] == 0)
                node = trie[node][c-'a'] = nxt++;
            else
                node = trie[node][c-'a'];
        }
        if(!isWordEnd[node]){
            isWordEnd[node] = true;
            wordsCnt++;
        }
    }

    bool find(string s, bool remove=false){ // O(n)
        ll node = 0;
        for(auto c: s) {
            if(trie[node][c-'a'] == 0)
                return false;
            else
                node = trie[node][c-'a'];
        }
        if(remove and isWordEnd[node]){
            isWordEnd[node] = false;
            wordsCnt--;
        }
        return isWordEnd[node];
    }
};
```

Z-function

Suppose we are given a string s of length n. The Z-function for this string is an array of length n where the i-th element is equal to the greatest number of characters starting from the position i that coincide with the first characters of s.

The first element of the Z-function, *z[0]*, is generally not well defined. This implementation assumes it as *z[0] = 0*. But it can also be interpreted as *z[0] = n* (all characters coincide).

```
vll z_function(string s) { // O(n)
    ll n = (ll) s.length();
    vll z(n);
    for (ll i=1, l=0, r=0; i<n; i++) {
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);

        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
```

```
        if (r < i + z[i] - 1) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

Solves

- Find occurrences of pattern string (*pattern*) in the main string (*str*):

```
int32_t main() { sws;
    string str, pattern; cin >> str >> pattern;
    string s = pattern + '$' + str;
    vll z = z_function(s);
    ll ans = 0;
    ll n = pattern.size();
    for(ll i=0; i< (ll) str.size(); i++){
        if( z[i + n + 1] == n)
            ans += 1;
    }
    cout << ans << endl;
}
```

- Find all border lengths of a given string.

OBS: A border of a string is a prefix that is also a suffix of the string but not the whole string. For example, the borders of *abcababcab* are *ab* and *abcab*.

Works because *z[i] == j* is the condition when the common characters of z[i] reaches the end of the string. For example:

abcababcab

z[8] = 2

ab is the border;

```
int32_t main(){ sws;
    string s; cin >> s;
    vll z = z_function(s);
    ll n = s.length();
    for(ll i=n-1, j=1; i>=0; i--) {
        if (z[i] == j) cout << j << ' ';
        j += 1;
    }
    cout << endl;
}
```

- Find all period lengths of a string.

OBS: A period of a string is a prefix that can be used to generate the whole string by repeating the prefix. The last repetition may be partial. For example, the periods of *abcabca* are *abc*, *abcabc* and *abcabca*.

Works because *z[i] + i >= n* is the condition when the common characters of z[i] in addition to the elements already passed, exceeds or is equal to the end of the string. For example:

abaababaab

z[8] = 2

abaababa is the period; the remaining (z[i] characters) are a prefix of the period; and when all these characters are combined, it can form the string (which has *n* characters).

```
int32_t main(){ sws;
    string s; cin >> s;
    vll z = z_function(s);
    ll n = s.length();
    for(ll i=1; i<n; i++) {
        if (z[i] + i >= n) {
            cout << i << ' ';
        }
    }
    cout << n << endl;
}
```

Structures

BIT (Fenwick Tree or Binary indexed tree)

Complexity O(log(n)): point update, range query

0-indexed:

```
struct FenwickTree {
    vector<ll> bit;  // binary indexed tree
    ll n;

    FenwickTree(ll n) { // all zero constructor
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<ll> a) : FenwickTree(a.size()) { // vector constructor
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    ll sum(ll r) { // prefix sum [1, r]
        ll ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }
```

```
    ll query(ll l, ll r) { // range sum [l, r]
        return sum(r) - sum(l - 1);
    }

    void add(ll idx, ll delta) { // add delta to current value
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

1-indexed

```
struct FenwickTree {
    vector<ll> bit;  // binary indexed tree
    ll n;

    FenwickTree(ll n) { // all zero constructor
        this->n = n + 2;
        bit.assign(n + 2, 0);
    }

    FenwickTree(vector<ll> a) : FenwickTree(a.size()) { // vector constructor
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    ll sum(ll idx) { // sum from 1 to idx [inclusive] (prefix sum)
        ll ret = 0;
        for (++idx; idx > 0; idx -= idx & -idx)
            ret += bit[idx];
        return ret;
    }

    ll query(ll l, ll r) { // sum from l to r [inclusive]
        return sum(r) - sum(l - 1);
    }

    void add(ll idx, ll delta) { // add delta to current value
        for (++idx; idx < n; idx += idx & -idx)
            bit[idx] += delta;
    }
};
```

Tree

Binary lifting

Solves: LCA, O(log) travelling in a tree

OBS: log2(1e5) ~= 17; log2(1e9) ~= 30 ; log2(1e18) ~= 60

Use for deep trees: LLOGMAX = 62;

```
const ll LOGMAX = 32;

vector<vll> g(MAX, vll());
ll depth[MAX]; // depth[1] = 0
ll jump[MAX][LOGMAX]; // jump[v][k] -> 2^k antecessor of v
// 1 points to 0 and 0 is the end point loop
ll N; // quantity of vertices of the tree

void binary_lifting(ll u = 1, ll p = -1){ // DFS, O(N)
    for(auto v : g[u]) if (v != p){
        depth[v] = depth[u] + 1;

        jump[v][0] = u;
        for(ll k=1; k < LOGMAX; k++)
            jump[v][k] = jump[ jump[v][k-1] ][k-1];
        binary_lifting(v, u);
    }
}

ll go(ll v, ll dist){ // O(log(N))
    for(ll k = LOGMAX-1; k >= 0; k--)
        if (dist & (1 << k))
            v = jump[v][k];
    return v;
}

ll lca(ll a, ll b){  // O(log(N))
    if (depth[a] < depth[b]) swap(a, b);

    a = go(a, depth[a] - depth[b]);
    if (a == b) return a;

    for(ll k = LOGMAX-1; k >= 0; k--){
        if (jump[a][k] != jump[b][k]){
            a = jump[a][k];
            b = jump[b][k];
        }
    }
    return jump[a][0];
}


int32_t main(){ sws;
    ll n; cin >> n;
    N = n;

    binary_lifting();
}
```

Find the Centroid of a Tree

A centroid of a tree is defined as a node such that when the tree is rooted at it, no other nodes have a subtree of size greater than *N/2*.

We can find a centroid in a tree by starting at the root. Each step, loop through all of its children. If all of its children have subtree size less than or equal to *N/2*, then it is a centroid. Otherwise, move to the child with a subtree size that is more than *N/2* and repeat until you find a centroid.

```
vector<vll> g(MAX, vll());
vll subtreeSize(MAX, 1);
ll N; // <- initialize N = n !!

void getSizes(ll u = 1, ll p = -1) {
    for(auto v : g[u]) if (v != p) {
        getSizes(v, u);
        subtreeSize[u] += subtreeSize[v];
    }
}

ll centroid(ll u = 1, ll p = -1) {
    for(auto v : g[u]) if (v != p) {
        if (subtreeSize[v] * 2 > N) return centroid(v, u);
    }
    return u;
}
```

Find the Diameter

From any node X find a node A which is the farthest away from X. Then, from node A, find a node B which is the farthest away from A.

Path from (A - B) is a diameter.

It can be proven by drawing a diameter line. If any node is further than any of the diameter extremities, then it should be switched (so the first line wasn't a diameter at all).

From any node, the fasthest node is a diameter extremity. Then from this extremity, the fasthest node is the other diameter extremity.

```
vector<vll> g(MAX, vll());

pll dfs(ll u, ll p){
    pll ans = {u, 0};
    for(auto v : g[u]) if (v != p) {
        auto [node, comp] = dfs(v, u);
        if (comp+1 > ans.ss){
            ans = {node, comp+1};
        }
    }
    return ans;
}

int32_t main(){sws;
    ll n; cin >> n;
    for(ll i=1; i<n; i++){
        ll a, b; cin >> a >> b;
        g[a].pb(b);
```

```
        g[b].pb(a);
    }
    ll ans1 = dfs(1, 1).ff;
    cout << dfs(ans1 , ans1).ss << endl;
}
```

Find the lenght of the longest path from all nodes

It can be proven that to any node X, the maximum distance is either dist(X, A) or dist(X, B), which are the extremities of a diameter.

```
vector<vll> g(MAX, vll());
vll distA(MAX, 0);
vll distB(MAX, 0);

pll dfs(ll u, ll p, ll op) {
    pll ans = {u, 0};
    for(auto v : g[u]) if (v != p) {
        if (op == 1) distA[v] = distA[u]+1;
        else if (op == 2) distB[v] = distB[u]+1;

        auto [node, length] = dfs(v, u, op);

        if (length + 1 > ans.ss)
            ans = {node, length+1};
    }
    return ans;
}

int32_t main() { sws;
    ll n; cin >> n;
    for(ll i=0; i<n-1; i++) {
        ll u, v; cin >> u >> v;
        g[u].pb(v);
        g[v].pb(u);
    }

    auto [nodeA, _t1] = dfs(1, -1, 0);
    auto [nodeB, _t2] = dfs(nodeA, -1, 1);
    dfs(nodeB, -1, 2);

    for(ll i=1; i<=n; i++) {
        cout << max(distA[i], distB[i]) << ' ';
    }
    cout << endl;
}
```

Heavy Light Decomposition

Features:

- Update all nodes along the path from node *x* to node *y*.

- Find the sum, maximum, minimum (or any other operation that satisfies the associative property) along the path from node *x* to node *y*.

Each query takes *O(log(N))* time. So the total complexity should be *O(Q log(N))*

Definitions:

- A heavy child of a node is the child with the largest subtree size rooted at the child.
- A light child of a node is any child that is not a heavy child.
- A heavy edge connects a node to its heavy child.
- A light edge connects a node to any of its light children.
- A heavy path is the path formed by a collection heavy edges.
- A light path is the path formed by a collection light edges.

```
ll N, v[MAX]; // Set N = n !
vector<vll> g(MAX, vll());

ll sz[MAX], p[MAX], dep[MAX], id[MAX], tp[MAX];
ll st[1 << 19];

void update_node(ll idx, ll val) { // O(log^2(N))
    st[idx += N] = val;
    for (idx /= 2; idx; idx /= 2)
        st[idx] = max(st[2 * idx], st[2 * idx + 1]);
}

ll query(ll lo, ll hi) {
    ll ra = 0, rb = 0;
    for (lo += N, hi += N + 1; lo < hi; lo /= 2, hi /= 2) {
        if (lo & 1)
            ra = max(ra, st[lo++]);
        if (hi & 1)
            rb = max(rb, st[--hi]);
    }
    return max(ra, rb);
}

ll dfs_sz(ll cur, ll par) {
    sz[cur] = 1;
    p[cur] = par;
    for(ll chi : g[cur]) {
        if(chi == par) continue;
        dep[chi] = dep[cur] + 1;
        p[chi] = cur;
        sz[cur] += dfs_sz(chi, cur);
    }
    return sz[cur];
}

ll ct = 1; // counter
void dfs_hld(ll cur, ll par, ll top) {
    id[cur] = ct++;
    tp[cur] = top;
    update_node(id[cur], v[cur]);
    ll h_chi = -1, h_sz = -1;
    for(ll chi : g[cur]) {
```

```
        if(chi == par) continue;
        if(sz[chi] > h_sz) {
            h_sz = sz[chi];
            h_chi = chi;
        }
    }
    if(h_chi == -1) return;
    dfs_hld(h_chi, cur, top);
    for(ll chi : g[cur]) {
        if(chi == par || chi == h_chi) continue;
        dfs_hld(chi, cur, chi);
    }
}

// returns the max_value of a node in the path from X to Y
ll path(ll x, ll y){ // O(log^2(N))
    ll ret = 0;
    while(tp[x] != tp[y]){
        if(dep[tp[x]] < dep[tp[y]])swap(x,y);
        ret = max(ret, query(id[tp[x]],id[x]));
        x = p[tp[x]];
    }
    if(dep[x] > dep[y])swap(x,y);
    ret = max(ret, query(id[x],id[y]));
    return ret;
}


int32_t main(){ sws;
    ll n, q; cin >> n >> q;
    N = n;
    for(ll i=1; i<=n; i++) cin >> v[i];
    for(ll i=2; i<=n; i++) {
        ll a, b; cin >> a >> b;
        g[a].pb(b);
        g[b].pb(a);
    }
    dfs_sz(1, 1);
    dfs_hld(1, 1, 1);
    while(q--) {
        ll t; cin >> t;
        if (t == 1) {
            ll s, x; cin >> s >> x;
            v[s] = x;
            update_node(id[s], v[s]);
        }
        else {
            ll a, b; cin >> a >> b;
            cout << path(a, b) << endl;
        }
    }
}
```