# Competitive-programming

Algoritmos e ideias de programação competitiva

## Flags for compilation:

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

# BIT-FenwickTree

## 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;
    }
};
```

# Divide-and-conquer

## Merge sort

```
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++;
        }
        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;
}
```

---

# 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;
}
```

## 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

---

# Fluxo

## 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;
    }
};
```

## 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;
}
```

# 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

```cpp
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:

$$ cos(a, b) = ( a \cdot b ) / ( ||a|| \cdot||b|| ) $$

$$ a \cdot b = cos(a, b) \cdot ( ||a|| \cdot||b|| ) $$

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

## BFS

```cpp
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);
        }
    }
}
```

## Binary lifting

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

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

```
const int LOGMAX = 32;
const int LLOGMAX = 62;

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();
}
```

## 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);
}
```

## 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.

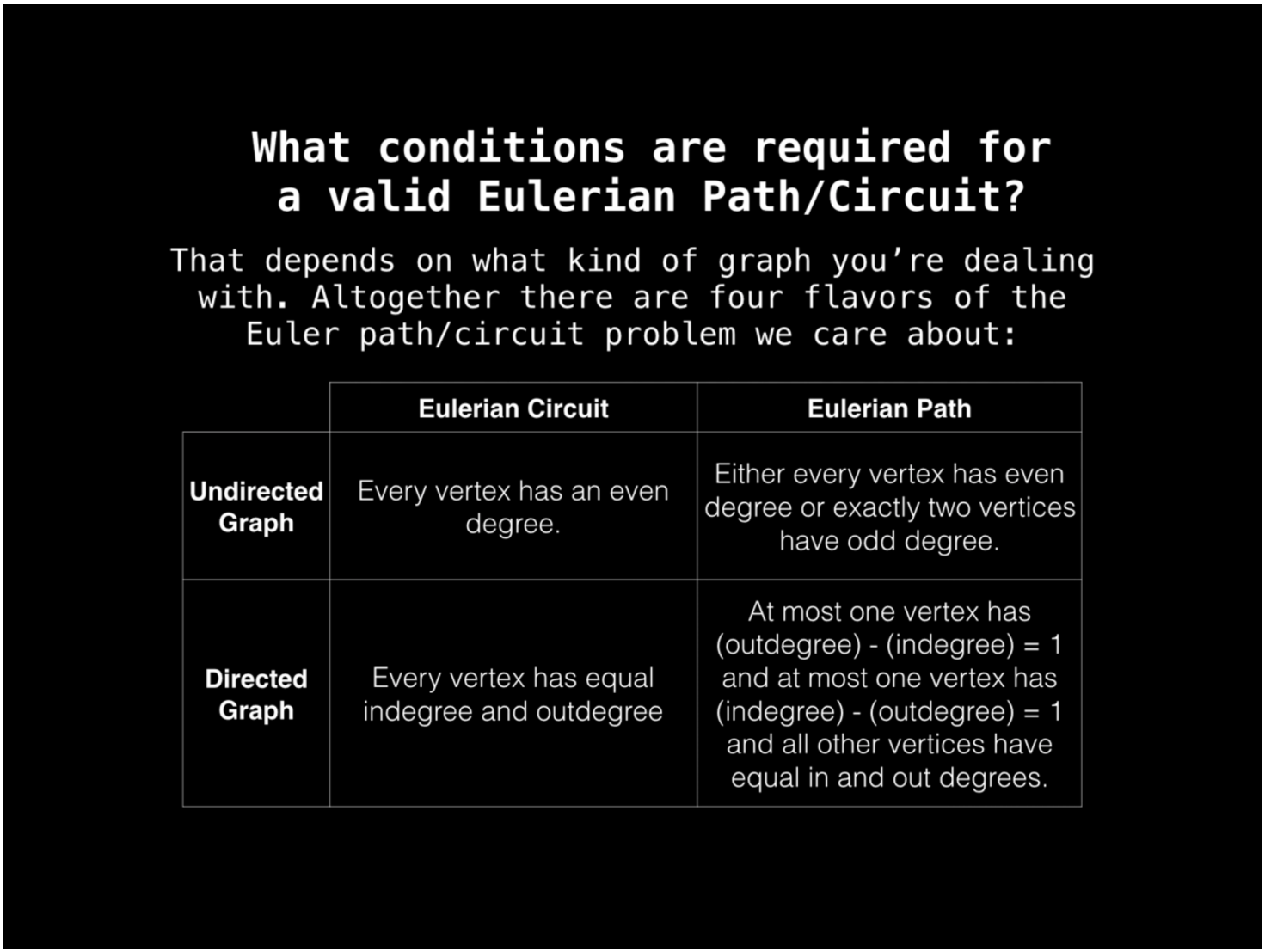
Similarly, a **Hamiltonian Path** consists of a path that transverses all **Vertices**.

## 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:




# 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)
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;
```

```cpp
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))
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))
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

## 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});
            }
        }
    }
}
```

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);
            }
        }
    }
}
```

## 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

Matrix operations

```
struct Matrix{
    vector<vector<int>> 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;
        }
    }
};
```

# ModularArithmetic

## Overloading Operations Struct

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

struct intM{
    long long val;

    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));
    }

    ostream& operator <<(ostream& os, const intM& a){
        os << a.val;
        return os;
    }
};
```

## Modular Arithmetic

Basic operations with redundant MOD operators

Also contains combinatorics operations

```
struct OpMOD{
    vector<long long> fact, ifact;

    OpMOD () {}

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

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

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

    long long sub(long long a, long long b){
        long long tmp = (a%MOD) - (b%MOD) % MOD;
        if (tmp < 0) tmp += MOD;
        return tmp;
    }

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

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

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

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

    long long combination(long long n, long long k){ // n! / (n! (n-k)! )
        return mul( mul(fact[n], ifact[k]) , ifact[n-k]);
    }

    long long disposition(long long n, long long k){ // n! / (n-k)!
        return mul(fact[n], ifact[n-k]);
    }
};
```

```
OpMOD op;
```

# Number-Theory

## 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;
        }
    }
}
```

## 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

```
bool attribute(int a){
    // add code here!!!!!
    return true;
}

int search(int l=0, int r=1e9, int ans=0){
    while(l <= r) { // [l; r]
        int mid = (l+r)/2;

        if(attribute(mid)) { // [mid; r]
            ans = mid;
            l = mid+1;
        }
```

```
        else { // [l; mid]
            r = mid-1;
        }
    }
    return ans;
}
```

- 

## 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);
            }
        }
};
```

## 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

```
ll L=1, N; // L=1=left delimiter; N=right delimiter
class SegmentTreeLazy {
    public:
        struct node{
            int psum = 0;
        };

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

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

        SegmentTreeLazy() {
            memset(lazy, 0, sizeof(lazy));
            memset(v, 0, sizeof(v));
        }

        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;
                lazy[i] = 0;
            }
            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]);
                lazy[i] = 0;
            }
        }
        void range_update(int left=L, int right=N, int val=0, int l=L, int r=N, int i=1)
{
            // left/right are the range limits for the update query (can be chosen)
            // l / r are the variables used for the vertex limits
            if (lazy[i]){
```

```
                tree[i].psum += lazy[i] * (r-l+1);
                if (l != r){
                    lazy[2*i] += lazy[i];
                    lazy[2*i+1] += lazy[i];
                }
                lazy[i] = 0;
            }

            if (right < l or r < left) return;
            else if (left <= l and r <= right){
                tree[i].psum += val * (r-l+1);
                if (l != r){
                    lazy[2*i] += val;
                    lazy[2*i+1] += val;
                }
            }
            else{
                int mid = (l+r)/2;
                range_update(left, right, val, l, mid, 2*i);
                range_update(left, right, 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 (lazy[i]){
                tree[i].psum += lazy[i] * (r-l+1);
                if (l != r){
                    lazy[2*i] += lazy[i];
                    lazy[2*i+1] += lazy[i];
                }
                lazy[i] = 0;
            }

            if (right < l or r < left){
                node tmp{0};
                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);
            }
        }
};
```

Details

**0 or 1-indexed**, depends on the arguments passed on to the default variables

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

**Lazy**: array containing lazy updates

### Methods

**O(n):**

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

**O(log(N))**

**range_update(left, right, l, r, i, val)**: updates all element from *left* to *right* (inclusive) with *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, range update
- Range Max/Min Query, range update
- Range Xor Query, range update

## 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);
            }
    }
};
```

## Old (sum-seg):

```
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] = 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};
                //
                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){
                // null element
                node tmp{0};
                //
                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);
            }
        }
};
```

## 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

## 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

Todo 😦

## Z function

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
```

```
    for (int 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*):

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