Team Reference
"testMCL"

Lázaro Raúl Iglesias Vera riglesiasvera@gmail.com
Carmen Irene Cabrera Rodríguez kr1000a@gmail.com
Manuel Santiago Fernandez Arias manfedezar@gmail.com

Índice

1 C++ Details

1.1 Basic include and using

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

1.2 Fast I/O

```
ios_base::sync_with_stdio(0);
cin.tie(0);
```

1.3 R/W Files

```
freopen("file.in", "r", stdin);
freopen("file.out", "w", stdout);
```

1.4 Set output size

```
//C++ style
cout << setw(10) << setfill('0') << n << endl;

//C style
printf("%010lld\n", n);
```

1.5 Numerical limits

```
numeric_limits<ll>::max();
numeric_limits<ll>::min();
```

2 Well know problems

2.1 Sieve

```
vector<int> primes;
vector<bool> mark(n + 1);
for(ll i = 2; i * i <= n; ++i)
    if(!mark[i])
        for(ll j = i * i; j <= n; j += i)
            mark[i] = true
```

2.2 LIS Longest Increasing Subsequences

```
int lis(vector<int> &v) {
    set<int> p;
    for(int i = 0; i < v.size(); ++i) {
        auto it = lower_bound(p.begin(), p.end(), v[i]);
        if(it != p.end()) p.erase(it);
        p.insert(v[i]);
    }
    return p.size();
}
```

2.3 Combinations DP

```
vector<vectot<int>> cb(n + 1, vector<int>(n + 1));
for(int i = 0; i <= n; ++i){
    for(int j = 0; j <= i; ++j){
        if(j == 0 || i == j)
            cb[i][j] = 1;
        else
            cb[i][j] = (cb[i - 1][j - 1] + cb[i - 1][j]) % MOD;
    }
}
```

2.4 Binary Exponentation

```
typedef long long ll;

ll BinExp(ll num, ll pot){
    ll x = 1;
    for(; pot > 0; pot /= 2){
        if(pot & 1)
            x = (x * num) % MOD;
        num = (num * num) % MOD;
    }
    return x;
}
//If MOD is a prime number is possible to find
//the modular inverse of ‘num‘ with ‘pot = MOD - 2‘
```

3 Data Structures

3.1 C++ AVL

```
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp>     // Including tree_order_statistics_node_update

#define select find_by_order
#define rank order_of_key

using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
                tree_order_statistics_node_update> ordered_set;
```

3.2 Trie

```
struct Trie{
    string s;
    int node = 0;
    int trie[MAX][30];
    long long counter[MAX];
    long long fin[MAX];

    int PrefixQuery(int &k, int v){
        int pos = 0;
        while(k < s.size() && trie[pos][s[k] - 'a'] != 0){
            pos = trie[pos][s[k++] - 'a'];
            counter[pos] += v;
        }
        return pos;
    }

    void Add(int k, int v){
        ++counter[0];
        int pos = PrefixQuery(k, v);
        while(k < s.size()){
            trie[pos][s[k++] - 'a'] = ++node;
            pos = node;
            counter[pos] += v;
        }
        fin[pos] += v;
    }

    void Remove(int k, int v){
        int safe = k;
        int pos = PrefixQuery(k, 0);
        if(k == s.size()){
            fin[pos] -= v;
            fin[pos] = max(0, fin[pos]);
            k = safe;
            PrefixQuery(k, -1 * v);
        }
    }
};
```

3.3 Fenwick Tree

```
typedef long long ll;
struct Fenwick{
    vector<ll> tree;
    int n;

    Fenwick(int n): n(n), tree(n + 1, 0){}

    //Point Update
    void update(int i, ll val){
        if(i > 0)
            while (i <= n) {
                tree[i] += val;
                i += i & -i;
            }
    }

    //Point query
    ll query(int i){
        ll sum = 0;
        while(i > 0){
            sum += tree[i];
            i -= i & -i;
        }
        return sum;
    }

    //Range update
    void rangeUpdate(int l, int r, ll val){
        update(l, val);
        update(r + 1, -val);
    }

    ll count(){ return query(n); }
    ll from(ll i){ return count() - query(i - 1); }
};

//Range update when Range Query
void rangeUpdate(Fenwick &f1, Fenwick &f2, int l, int r, ll val){
    f1.update(l, val);
    f1.update(r + 1, -val);
    f2.update(l, val * (l - 1));
    f2.update(r + 1, -val * r);
}

void mergeQuery(Fenwick &f1, Fenwick &f2, int i){
    return (f1.query(i) * i) - f2.query(i)
}

//Range Query
void rangeQuery(Fenwick &f1, Fenwick &f2, int l, int r){
    return mergeQuery(f1, f2, r) - mergeQuery(f1, f2, l - 1)
}
```

3.4 Segment Tree with lazy propagation

```
struct segment_tree{
    int n;
    vector<ll> data;
    vector<ll> st;
    vector<ll> lazy;

    segment_tree(int n) : n(n), data(n), st(4 * n), lazy(4 * n) {}

    void push_lazy(int node){
        int l = node * 2, r = l + 1;
        //Set son's lazy and new answer
        lazy[l] += lazy[node];
        st[l] += lazy[node];
        lazy[r] += lazy[node];
        st[r] += lazy[node];
        //clean node lazy
        lazy[node] = 0;
    }

    void build() { build(1, 1, n + 1); }

    void build(int node, int a, int b){
        if( a == b - 1){
            //set initial value
            st[node] = data[a - 1];
            return;
        }
        int mid = (a + b) / 2, l = node * 2, r = l + 1;
        build(l, a, mid);
        build(r, mid, b);
        //set node's answer
        st[node] = min(st[l], st[r]);
    }

    void update(int x, int y, ll v) { update(1, 1, n + 1, x, y + 1, v); }

    ll update(int node, int a, int b, int x, int y, ll v){
        if(x <= a && b <= y){
            //Add Lazy
            lazy[node] += v;
            st[node] += v;
        }
        else{
            push_lazy(node);
            int mid = (a + b) / 2, l = node * 2, r = l + 1;
            if(y <= mid)
                st[node] = min(st[r], update(l, a, mid, x, y, v));
            else if(x >= mid)
                st[node] = min(st[l], update(r, mid, b, x, y, v));
            else {
                st[node] = update(l, a, mid, x, y, v);
                st[node] = min(st[node], update(r, mid, b, x, y, v));
            }
        }
        return st[node];
    }
```

```
        ll query(int x, int y) { return query(1, 1, n + 1, x, y + 1); }

        ll query(int node, int a, int b, int x, int y){
            if(x <= a && b <= y){
                return st[node];
            }
            push_lazy(node);
            int mid = (a + b) / 2, l = node * 2, r = l + 1;
            ll val;
            if(y <= mid)
                val = query(l, a, mid, x, y);
            else if(x >= mid)
                val = query(r, mid, b, x, y);
            else {
                val = query(l, a, mid, x, y);
                val = min(val, query(r, mid, b, x, y));
            }
            return val;
        }
    };
```

3.5 Persistent Segment Tree

```
struct segment_tree{
    struct st_node{
        int val;
        int r, l;
    };

    int n;
    int identifier = 0;
    vector<st_node> container;
    vector<int> versions;

    segment_tree(ll n, int v):
            n(n), versions(v + 1), container(v * (ceil(log2(n + 1)) + 1)){}

    void CopieAndInsert(int original, int &copie, int l, int r, int element, int cant){
        copie = ++identifier;
        container[copie] = container[original];
        container[copie].val += cant;
        if(l + 1 == r)
            return;
        int middle = (l + r) / 2;
        if(element < middle){
            CopieAndInsert(container[original].l, container[copie].l,
                                l, middle, element, cant);
        }
        else{
            CopieAndInsert(container[original].r, container[copie].r,
                                middle, r, element, cant);
        }
    }
```

```
    void add(int old, int cur, int x, int cant) {
         CopieAndInsert(versions[old], versions[cur], 1, n + 1, x, cant);
    }

    int query(int v, int x) {return query(versions[v], 1, n + 1, x);}

    int query(int root, int l, int r, int element){
        if(l + 1 == r){
            return container[root].val;
        }
        int middle = (l + r) / 2;
        if(element < middle)
            return query(container[root].l, l, middle, element);
        else
            return query(container[root].r, middle, r, element);
    }
};
```

3.6 Aho-Corasick

```
typedef pair<int, int> pi;

struct AhoCorasick {
    int alpha;
    int node = -1;
    vector<vector<int>> trie;
    vector<bool> fin;

    AhoCorasick(int top, int alpha):
            alpha(alpha), trie(top, vector<int>(alpha)), fin(top) {}

    AhoCorasick(int alpha): alpha(alpha) { NewNode(); }

    int NewNode(){
        trie.emplace_back(vector<int>(alpha));
        fin.push_back(false);
        return ++node;
    }

    void Add(string& s, int v = 1) {
        int pos = 0;
        for(auto c:s) {
            if(trie[pos][c - 'a'] == 0)
                trie[pos][c - 'a'] = NewNode();
            pos = trie[pos][c - 'a'];
        }
        fin[pos] = true;
    }

    void Fails() {
        queue<pi> q;
        int a, b;
        for (int i = 0; i < alpha; ++i)
            if (trie[0][i] != 0)
                q.push({trie[0][i], 0});
```

```
        pi v;
        while (!q.empty()) {
            v = q.front();
            q.pop();
            for (int i = 0; i < alpha; ++i) {
                a = trie[v.first][i];
                b = trie[v.second][i];
                if (a != 0) {
                    fin[a] = fin[a] || fin[b];
                    q.push({a, b});
                }
                else
                    trie[v.first][i] = b;
            }
        }
    }
};
```

3.7 Disjoin Set

```
struct DS{
    vector<int> p;
    vector<int> c;

    DS(int n): p(n, 0), c(n, 1){
        for(int i = 0; i < n; ++i)
            p[i] = i;
    }

    int SetOf(int x){
        return (p[x] == x)? x : p[x] = SetOf(p[x]);
    }

    bool Merge(int x, int y){
        x = SetOf(x);
        y = SetOf(y);
        if(x == y)
            return false;
        if(c[x] < c[y])
            swap(x, y);
        p[y] = x;
        c[x] += c[y];
        retuen true;
    }
};
```

3.8 Indexed Heap

```
struct Heap{
    int cant;
    vector<int> heap, pos, idx;

    Heap(): heap(1, 0), pos(1, 0), idx(1, 0), cant(0){}
    Heap(int n): heap(n + 1, 0), pos(n + 1, 0), idx(n + 1, 0), cant(0){}

    void Add(int x){
        if(++cant >= (int)heap.size()){
            heap.push_back(0);
            idx.push_back(0);
```

```
            pos.push_back(0);
        }
        heap[cant] = x;
        idx[cant] = pos[cant] = cant;
        HeapifyUp(cant);
    }

    void Insert(int x){
        heap[++cant] = x;
        HeapifyUp(cant);
    }

    int SuprimeAt(int x){
        return Suprime(pos[x]);
    }

    int Suprime(int x = 1){
        int result = heap[x];
        heap[x] = heap[cant--];
        HeapifyDown(x);
        HeapifyUp(x);
        return result;
    }

    int Peek(){ return heap[1]; }

    void Swap(int x, int y){
        pos[idx[x]] = y;
        pos[idx[y]] = x;
        swap(idx[x], idx[y]);
        swap(heap[x], heap[y]);
    }

    void HeapifyUp(int x){
        while(x > 1 && heap[x] < heap[x / 2]){
            Swap(x, x / 2);
            x = x / 2;
        }
    }

    void HeapifyDown(int x){
        while(2 * x <= cant){
            int i = 2 * x;
            i += (i < cant && heap[i + 1] < heap[i]);
            if(heap[x] <= heap[i])
                break;
            Swap(x, i);
            x = i;
        }
    }

    bool empty() {return cant == 0;}
};
```

4 Algorithms

4.1 Articulation Points and Bridge Edges

```
struct articulation_points{
    int V, dt;
    vector<bool> ap;
    vector<int> d, low;
    vector<vector<int>> G;

    articulation_points(int n): V(n), dt(0), d(n, 0), low(n, 0), G(n), ap(n) {}

    void add_edge(int u, int v){
        G[u].push_back(v);
    }

    void dfs(int u){
        d[u] = low[u] = ++dt;
        for (int v : G[u])
        if (!d[v]){
            dfs(v);
            low[u] = min(low[u], low[v]);
            if((d[u] == 1 && d[v] > 2) || (d[u] != 1 && low[v] >= d[u]))
                ap[u] = true;
        }
        else low[u] = min(low[u], d[v]);
    }

    void solve(){
        for (int i = 0; i < V; ++i)
            if (!d[i]) dt = 0, dfs(i);
    }
};

struct bridge{
    int V, dt;
    vector<pair<int, int>> bridges;
    vector<int> low, d, parent;
    vector<vector<int>> G;

    bridge(int n) :
        V(n), dt(0), low(n, 0), d(n, 0), parent(n), bridges(0), G(n, vector<int>(0)) {}

    void add_edge(int u, int v){
        G[u].push_back(v);
    }

    void dfs(int u){
        d[u] = low[u] = ++dt;
        for (int v : G[u])
            if (!d[v]){
                parent[v] = u;
                dfs(v);
                low[u] = min(low[u], low[v]);
                if(d[u] < low[v])
                    bridges.emplace_back(min(u, v), max(u, v));
```

```
            }
            else if (v != parent[u])
                low[u] = min(low[u], d[v]);
    }

    void solve(){
        for (int i = 0; i < V; ++i)
            if (!d[i]) dfs(i);
    }
};
```

4.2 Z-Function

```
vector<int> z_function(string &s) {
    int n = (int) s.length();
    vector<int> z(n, 0);
    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 (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

4.3 RMQ Range Minimun Query

```
struct RMQ{
    int n, lg;
    vector<vector<ll>> dp;

    RMQ(vector<ll> &data, ll top):
        n(data.size()), lg((int)log2(n)), dp(n + 1, vector<ll>(lg + 1, top))
    {
        for (int i = 1; i <= n; ++i)
            dp[i][0] = data[i - 1];

        for (int p = 1; p <= lg; ++p)
            for (int i = 1; i <= n - (1 << p) + 1; ++i)
                dp[i][p] = min(dp[i][p - 1], dp[i + (1 << (p - 1))][p - 1]);
    }

    ll query(int a, int b){
        //in arguments for refer x is necessary to pass x + 1
        int p = log2(b - a + 1);
        return min(dp[a][p], dp[b - (1 << p) + 1][p]);
    }
};
```

4.4 Matix Exponentation

```
typedef long long ll;
typedef vector<vector<ll> > matrix;

matrix MatrixMult(matrix x, matrix y){
    ll v;
    matrix m(DIM, vector<ll>(DIM));
    for(int i = 0; i < DIM; ++i)
        for(int j = 0; j < DIM; ++j)
            for(int k = 0 ; k < DIM; ++k) {
                v = (x[i][k] * y[k][j]) % MOD;
                m[i][j] = (m[i][j] + v) % MOD;
            }
    return m;
}

matrix BinExp(matrix num, ll pot){
    matrix x(DIM, vector<ll>(DIM));
    for(int i = 0; i < DIM; ++i)
        x[i][i] = 1;
    for(; pot > 0; pot /= 2){
        if(pot & 1)
            x = MatrixMult(x, num);
        num = MatrixMult(num, num);
    }
    return x;
}
```

4.5 KMP Knuth-Morris-Pratt

```
vector<int> prefixFunction(string &p){
    int k = 0, m = (int)p.size();
    vector<int> prefix(m, 0);
    for(int i = 1; i < m; ++i){
        while(k > 0 && p[i] != p[k])
            k = prefix[k - 1];
        if(p[i] == p[k]) ++k;
        prefix[i] = k;
    }
    return prefix;
}

int kmp(string &t, string &p){
    int k = 0;
    auto prefix = prefixFunction(p);
    int cont = 0, n = (int)t.size(), m = (int)p.size();
    for(int i = 0; i < n; ++i){
        while(k > 0 && t[i] != p[k])
            k = prefix[k - 1];
        if(t[i] == p[k]) ++k;
        if(k == m){
            ++cont;
            k = prefix[k - 1];
        }
    }
    return cont;
}
```

4.6 Bellman Ford

```
struct bellman_ford{
    int V, E;
    vector<ll> d;
    vector<int> parent, cycle;
    struct edge { int u, v; ll w; };
    vector<edge> G;

    bellman_ford(int n) :
        V(n), E(0), d(n, oo), parent(n, -1), cycle(0), G(0) {}

    void add_edge(int u, int v, ll w){
        G.push_back({ u, v, w });
    }

    void negative_cycle(int u){
        for (int i = 0; i < V; ++i)
        u = parent[u];

        cycle.push_back(u);
        for (int v = parent[u]; v != u; v = parent[v])
            cycle.push_back(v);
    }

    bool solve(int source = 0){
        d[source] = 0;
        E = G.size();
        bool r = true;

        for (int i = 1; i <= V && r; ++i){
            r = false;
            for (auto &e : G)
                if (d[e.u] != oo && d[e.u] + e.w < d[e.v]){
                    r = true;
                    parent[e.v] = e.u;
                    d[e.v] = d[e.u] + e.w;

                    if (i == V){
                        negative_cycle(e.v);
                        return true;
                    }
                }
        }
        return false;
    }
};
```

4.7 Floyd Warshall

```
vector<vector<int>> c(V, vector<int>(V, oo));
c[0][0] = 0;
for (int k = 0; k < V; ++k)
    for (int i = 0; i < V; ++i)
        if (c[i][k] < oo)
            for (int j = 0, w; j < V; ++j)
                if ((w = c[i][k] + c[k][j]) < c[i][j])
                    c[i][j] = w;
```

4.8 SCC Strongly Connected Components

```
struct strongly_connected_component{
    int V, dt;
    vector<bool> del;
    vector<int> dfsnum, low, S;
    vector<vector<int>> G, SCC;

    strongly_connected_component(int n) :
        V(n), dt(0), del(n, 0), dfsnum(n, 0), low(n, 0),
        S(0), G(n, vector<int>(0)), SCC(0) {}

    void add_edge(int u, int v){
        G[u].push_back(v);
    }

    void dfs(int u){
        S.push_back(u);
        dfsnum[u] = low[u] = ++dt;

        int v;
        for (auto v : G[u]){
            if(!dfsnum[v]){
                dfs(v);
                v = low[v];
            }
            else
                v = !del[v] ? dfsnum[v] : low[u];
            low[u] = min(low[u], v);
        }

        if (low[u] == dfsnum[u]){
            SCC.push_back(vector<int>(0));
            while (!del[u]){
                SCC.back().push_back(S.back());
                del[S.back()] = true;
                S.pop_back();
            }
        }
    }

    vector<vector<int>> solve(){
        for (int i = 0; i < V; ++i)
            if (!dfsnum[i]) dfs(i);
        return SCC;
    }
};

//single method
void SCC(int u, int id, int SC, int time){
    d[u] = ++t;
    int v;
    for(int i = 0; i < vecinos.size(); ++i){
        v = (id == 1)? ayd[u] : adyT[u];
        if(d[v] < time)
        SCC(v, id, SC, time);
    }
    if(id == 1)
        s.push(u);
```

```
        else
            scc[u] = SC;
    }
```

4.9 Kruskal

```
struct kruskal{
    DS d;
    vector<pair<ll, pair<int, int>>> edge;

    kruskal(int n): d(n + 1){}

    void add_edge(int a, int b, ll c){ edge.push_back({c , {a, b}}); }

    vector<pair<ll, pair<int, int>>> solve(){
        vector<pair<int, pair<int,int>>> R;
        sort(edge.begin(), edge.end());
        for(auto e:edge)
            if(d.Merge(e.second.first, e.second.second))
                R.push_back(e);
        return R;
    }
};
```

4.10 Prim

```
struct Prim{
    int n, oo;
    vector<int> d, p;
    vector<bool> flag;
    vector<vector<pair<int, int>>> G;
    priority_queue<pair<int, int>> cola;

    Prim(int n, int inf):
        n(n), oo(inf), d(n + 1, inf), p(n + 1, 0), flag(n + 1, true), G(n + 1){}

    void add_edge(int a, int b, int cost){
        G[a].push_back({b, cost});
    }

    void solve(int start = 1){
        int u, v, c;
        pair<int, int> aux;

        d[start] = 0;
        cola.push({0, start});

        while(!cola.empty()){
            aux = cola.top();
            cola.pop();
            u = aux.second;
            if(flag[u]){
                for(auto &edge:G[u]){
                    v = edge.first;
                    c = edge.second;
                    if(flag[v] && c < d[v]){
                        d[v] = c;
                        p[v] = u;
```

```
                    cola.push({-c, v});
                }
            }
            flag[u] = false;
        }
    }
}
};
```

4.11 Dijkstra

```
struct Dijkstra{
    int n, oo;
    vector<int> d, p;
    vector<bool> flag;
    vector<vector<pair<int, int>>> G;
    queue<pair<int, int>> cola;

    Dijkstra(int n, int inf):
        n(n + 1), oo(inf), d(n + 1, inf),
        p(n + 1, 0), flag(n + 1, true), G(n + 1){}

    void add_edge(int a, int b, int cost){
        G[a].push_back({b, cost});
    }

    void solve(int start){
        int u, v, c;

        d[start] = 0;
        cola.push_back({0, start});
        while(!cola.empty()){
            u = cola.front().second;
            cola.pop();
            if(flag[u]){
                for(auto &edge:graph[u]){
                    v = edge.first;
                    c = edge.second;
                    if(flag[v] && c + d[u] < d[v]){
                        d[v] = c + d[u];
                        p[v] = u;
                        cola.push({-(c + d[u]), v});
                    }
                }
                flag[u] = false;
            }
        }
    }
};
```

4.12 Hash

```
struct Hash{
    ll base = 31;
    //ll base1 = 43;ll base2 =47;
    ll MOD = 1000000007;
    //ll MOD1 = 1e15 + 7; ll MOD2 =998244353;
    int len;
    vector<ll> p, h, fh, bh;
```

```
        string s;

        void Calc_Hash(){
            fh[0] = s[0];
            int pos = len - 1;
            bh[pos] = s[pos];
            pos--;
            for( int i = 1 ; i < len ; ++i){
                fh[i] = ( (fh[i - 1] * base ) % MOD + s[i] ) % MOD;
                bh[pos] = ( (bh[pos + 1] * base) % MOD + s[pos]) % MOD;
                pos--;
            }
        }

        Hash(string s,int maxsize):
            s(s), len(s.size()), p(maxsize, 0),
            h(maxsize, 0), fh(maxsize, 0), bh(maxsize, 0){
                p[0] = 1;
                for(int i = 1; i < maxsize; ++i){
                    p[i] = (p[i - 1] * base) % MOD;
                }
                Calc_Hash();
        }

        ll Get_Hash_B(int s, int e){
            if(e == len - 1)
                return bh[s];
            return (bh[s] + MOD - (bh[e + 1] * p[e - s + 1]) % MOD) % MOD;
        }
        ll Get_Hash_F(int s, int e){
            if(s == 0)
                return fh[e];
            return (fh[e] + MOD - (fh[s - 1] * p[e - s + 1]) % MOD) % MOD;
        }
        bool Palin(int s, int e){
            return Get_Hash_F(s,e) == Get_Hash_B(s,e);
        }
    };
```

4.13 Manacher

```
    vector<int> Manacher(string &text, int tsize ){
        int n = tsize;
        n = 2 * n + 1;
        vector<int> man(n,0);
        int c = 0 ,r = 0;
        vector<char> s;
        s.push_back('$');

        for(int i = 0 ; i < tsize ; ++i){
            s.push_back(text[i]);
            s.push_back('$');
        }

        s.push_back('*');

        for(int i = 1 ; i < n-1 ; i++){
            int j = c - ( i - c);
```

```
        if(r > i)
            man[i] = min(r - i , man[j]);
        while(s[i + 1 + man[i]] == s[i - 1 - man[i]])
            man[i]++;
        if( i + man[i] > r){
            c = i;
            r = i + man[i];
        }
    }
    return man;
}
```

4.14 DSU on Tree

```
struct DSU {
    int n;
    vector<int> sz, h;
    vector<vector<int>> G;
    vector<bool> big;
    //other containers

    DSU(int n):
        n(n + 1), G(n + 1), sz(n + 1), h(n + 1), big(n + 1) {}

    void add_edge(int a, int b){
        G[a].push_back(b);
    }

    void getSize(int v = 1, int dad = -1){
        sz[v] = 1;
        for(auto &u : G[v]){
            if(u != dad){
                h[u] = h[v] + 1;
                getSize(u, v);
                sz[v] += sz[u];
            }
        }
    }

    void add(int upd, int v, int dad, int val){
        check(upd, v, val);
        for(auto &u : G[v])
            if(!big[u] && u != dad)
                add(upd, u, v, val);
    }

    //update values needed to compute the answer
    void check(int upd, int v, int val){
        return;
    }

    void dfs(int v = 1, int dad = -1, bool clean = false){
        int heavy = -1, top = -1;
        //set heavy
        for(auto &u : G[v]){
            if(u != dad && sz[u] > top){
                heavy = u;
                top = sz[u];
```

```
                }
            }
            //update all
            for(auto &u : G[v])
                if(u != dad && u != heavy)
                    dfs(u, v, true);
            //safe heavy
            if(heavy != -1) {
                big[heavy] = true;
                dfs(heavy, v);
                //use heavy
            }
            add(v, v, dad, 1);
            //update node answer
            if(heavy != -1)
                big[heavy] = 0;
            if(clean)
                add(v, v, dad, -1);
        }
    };
```

4.15 SQRT Descomposition MO's algorithm

```
struct MO{
    struct range{
        int l, r, id;
    };

    ll block;
    vector<ll> answers, data;
    vector<range> queries;

    //other necessary variables
    ll cnt[1000001], ans = 0;

    MO(ll n, ll q): answers(q), data(n), queries(q), block((ll)sqrt(n)){}

    void add_query(int a, int b, int i){
        queries[i] = {--a, --b, i};
    }

    // Try to update in O(1) complexity
    void add(ll x){
        ans += (cnt[x]++ == 0);
    }

    // Try to update in O(1) complexity
    void remove(ll x){
        ans -= (--cnt[x] == 0);
    }

    void solve(){
        // comparer to order all queries
        auto mo_cmp = [&](const range &x, const range &y){
            ll blockX = x.l / block;
            ll blockY = y.l / block;
            if(blockX != blockY)
                return blockX < blockY;
```

```
                return (blockX & 1) ? (x.r < y.r) : (x.r > y.r);
            };
            sort(all(queries), mo_cmp);

            int l, r, left = 0, right = -1;
            for(auto &q:queries){
                while(right < q.r)
                    add(data[++right]);
                while(right > q.r)
                    remove(data[right--]);
                while(left < q.l)
                    remove(data[left++]);
                while(left > q.l)
                    add(data[--left]);
                answers[q.id] = ans;
            }
        }
    };
```

4.16 Dinic Flow

```
template<typename T>
struct dinic{
    struct edge{
        int src, dst;
        T cap, flow;
        int rev;
    };

    int n;
    vector<vector<edge>> adj;
    vector<int> level, iter;
    const T oo = numeric_limits<T>::max();

    dinic(int n) : n(n), adj(n) {} //initialize with n+1

    void add_edge(int src, int dst, T cap){
        adj[src].push_back({ src, dst, cap, 0, (int) adj[dst].size() });
        if (src == dst)
            adj[src].back().rev++;
        adj[dst].push_back({ dst, src, 0, 0, (int) adj[src].size() - 1 });
    }

    T augment(int u, int t, T cur){
        if (u == t)
            return cur;
        for (int &i = iter[u]; i < (int)adj[u].size(); ++i){
            edge &e = adj[u][i];
            if (e.cap - e.flow > 0 && level[u] > level[e.dst]){
                T f = augment(e.dst, t, min(cur, e.cap - e.flow));
                if (f > 0){
                    e.flow += f;
                    adj[e.dst][e.rev].flow -= f;
                    return f;
                }
            }
        }
        return 0;
```

```
    }

    int bfs(int s, int t)
    {
        level.assign(n, n);
        level[t] = 0;
        queue<int> Q;
        for (Q.push(t); !Q.empty(); Q.pop()){
            int u = Q.front();
            if (u == s)
                break;
            for (edge &e : adj[u]){
                edge &erev = adj[e.dst][e.rev];
                if (erev.cap - erev.flow > 0 && level[e.dst] > level[u] + 1){
                    Q.push(e.dst);
                    level[e.dst] = level[u] + 1;
                }
            }
        }
        return level[s];
    }

    T max_flow(int s, int t)
    {
        for (int u = 0; u < n; ++u) // initialize
            for (auto &e : adj[u])
                e.flow = 0;

        T flow = 0;
        while (bfs(s, t) < n){
            iter.assign(n, 0);
            for (T f; (f = augment(s, t, oo)) > 0;)
                flow += f;
        } // level[u] == n ==> s-side
        return flow;
    }
};
```

4.17 Matching

```
struct Matching{
    int L, R;
    vector<vector<int>> adj;

    Matching(int L, int R) : L(L), R(R), adj(L + R) {}

    void add_edge(int u, int v){
        adj[u].push_back(v + L);
        adj[v + L].push_back(u);
    }

    int maximum_matching(){
        vector<int> level(L), mate(L + R, -1);

        function<bool(void)> levelize = [&](){
            queue<int> Q;
            for (int u = 0; u < L; ++u){
                level[u] = -1;
```

```
                if (mate[u] < 0){
                    level[u] = 0;
                    Q.push(u);
                }
            }
            while (!Q.empty()){
                int u = Q.front(); Q.pop();
                for (int w : adj[u]){
                    int v = mate[w];
                    if (v < 0)
                        return true;
                    if (level[v] < 0){
                        level[v] = level[u] + 1;
                        Q.push(v);
                    }
                }
            }
            return false;
        };

        function<bool(int)> augment = [&](int u){
            for (int w : adj[u]){
                int v = mate[w];
                if (v < 0 || (level[v] > level[u] && augment(v))){
                    mate[u] = w;
                    mate[w] = u;
                    return true;
                }
            }
            return false;
        };
        int match = 0;
        while (levelize())
            for (int u = 0; u < L; ++u)
                if (mate[u] < 0 && augment(u))
                    ++match;
        return match;
    }
};
```

4.18 Miller Rabin

```
//return if n is prime with high percent of veracity
// 'a' is a random between 2 and n-2
// O((ln n)**4) complexy
bool Miller_Rabin_Test(ll n, ll a){
    ll k = 0, a, m, temp = n - 1, pot = 1, b;
    while(temp % 2 == 0){
        temp /= 2;
        ++k;
        pot *= 2;
    }
    m = temp / pot;
    b = BinExp(a, m, n); // a = base, m = pot, n = mod
    if(b == 1 || b == n - 1) return true;
    while(k--){
        b = (b * b) % n;
        if(b == n - 1) return true;
```

```
        if (k == 0 || b == 1) return false;
    }
}
```

4.19 Pollard-Rho

```
/* method to return prime divisor for n */
long long int PollardRho(long long int n)
{
    /* initialize random seed */
    srand (time(NULL));

    /* no prime divisor for 1 */
    if (n==1) return n;

    /* even number means one of the divisors is 2 */
    if (n % 2 == 0) return 2;

    /* we will pick from the range [2, N) */
    long long int x = (rand()%(n-2))+2;
    long long int y = x;

    /* the constant in f(x).
    * Algorithm can be re-run with a different c
    * if it throws failure for a composite. */
    long long int c = (rand()%(n-1))+1;

    /* Initialize candidate divisor (or result) */
    long long int d = 1;

    /* until the prime factor isn't obtained.
    If n is prime, return n */
    while (d==1)
    {
        /* Tortoise Move: x(i+1) = f(x(i)) */
        x = (modular_pow(x, 2, n) + c + n)%n;

        /* Hare Move: y(i+1) = f(f(y(i))) */
        y = (modular_pow(y, 2, n) + c + n)%n;
        y = (modular_pow(y, 2, n) + c + n)%n;

        /* check gcd of |x-y| and n */
        d = __gcd(abs(x-y), n);

        /* retry if the algorithm fails to find prime factor
        * with chosen x and c */
        if (d==n) return PollardRho(n);
    }

    return d;
}
```