Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1 Segtree **2**
1.1 Standard SegTree 2
1.2 Lazy SegTree 2

2 Graph **3**
2.1 Dinic 3
2.2 Dinic Min cost 4
2.3 Dkistra 5
2.4 Dominator tree 5
2.5 Bellman Ford 6
2.6 Dfs tree 6
2.7 Topological Sort 6
2.8 Kth Ancestor 7

3 Parser **7**
3.1 Parsing Functions 7

4 String **8**
4.1 Z function 8
4.2 Suffix Array 8
4.3 String hash 8

5 Algorithms **9**
5.1 Sparse table 9

6 Set **9**
6.1 Ordered Set 9
6.2 Set 9
6.3 Multiset 9

1 Segtree

1.1 Standard SegTree

Complexity: O(log(n)) query and update
Sum segment tree with point update.

```
// TITLE: Standard SegTree
// COMPLEXITY: O(log(n)) query and update
// DESCRIPTION: Sum segment tree with point update.

using type = int;

type iden = 0;
vector<type> seg;
int segsize;

type func(type a, type b)
{
    return a + b;
}

// query do intervalo [l, r)
type query(int l, int r, int no = 0, int lx = 0, int rx = segsize)
{
    // l lx rx r
    if (r <= lx or rx <= l)
        return iden;
    if (l <= lx and rx <= r)
        return seg[no];

    int mid = lx + (rx - lx) / 2;
    return func(query(l, r, 2 * no + 1, lx, mid),
                query(l, r, 2 * no + 2, mid, rx));
}

void update(int dest, type val, int no = 0, int lx = 0, int rx = segsize)
{
    if (dest < lx or dest >= rx)
        return;
    if (rx - lx == 1)
    {
        seg[no] = val;
        return;
    }

    int mid = lx + (rx - lx) / 2;
    update(dest, val, 2 * no + 1, lx, mid);
    update(dest, val, 2 * no + 2, mid, rx);
    seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
}

signed main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int n;
    cin >> n;
    segsize = n;
    if (__builtin_popcount(n) != 1)
    {
        segsize = 1 + (int)log2(segsize);
        segsize = 1 << segsize;
    }
    seg.assign(2 * segsize - 1, iden);

    loop(i, 0, n)
    {
        int x;
        cin >> x;
        update(i, x);
    }
}
```

1.2 Lazy SegTree

Complexity: O(log(n)) query and update
Sum segment tree with range sum update.

```
// TITLE: Lazy SegTree
// COMPLEXITY: O(log(n)) query and update
// DESCRIPTION: Sum segment tree with range sum update.
vector<int> seg, lazy;
int segsize;

// change 0s to -1s if update is
// set instead of add. also,
// remove the +=s
void prop(int no, int lx, int rx) {
    if (lazy[no] == 0) return;

    seg[no]+=(rx-lx)*lazy[no];
    if(rx-lx>1) {
        lazy[2*no+1] += lazy[no];
        lazy[2*no+2] += lazy[no];
    }

    lazy[no]=0;
}

void update(int l, int r, int val,int no=0, int lx=0, int rx=segsize) {
    // l r lx rx
    prop(no, lx, rx);
    if (r <= lx or rx <= l) return;
    if (l <= lx and rx <= r) {
        lazy[no]=val;
        prop(no,lx,rx);
        return;
    }

    int mid=lx+(rx-lx)/2;
    update(l,r,val,2*no+1,lx,mid);
    update(l,r,val,2*no+2,mid,rx);
    seg[no] =seg[2*no+1]+seg[2*no+2];
}

int query(int l,int r,int no=0,int lx=0, int rx=segsize) {
    prop(no,lx,rx);
    if (r <= lx or rx <= l) return 0;
    if (l <= lx and rx <= r) return seg[no];

    int mid=lx+(rx-lx)/2;
    return query(l,r,2*no+1, lx, mid)+
           query(l,r,2*no+2,mid,rx);
}

signed main() {
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);

    int n;cin>>n;
    segsize=n;
    if(__builtin_popcount(n) != 1) {
        segsize=1+(int)log2(segsize);
        segsize= 1<<segsize;
    }

    seg.assign(2*segsize-1, 0);
    // use -1 instead of 0 if
```

```cpp
    // update is set instead of add
    lazy.assign(2*segsize-1, 0);
}
```

2 Graph

2.1 Dinic

Complexity: O(V*V*E), Bipartite is O(sqrt(V) E)
Dinic

```cpp
// TITLE: Dinic
// COMPLEXITY: O(V*V*E), Bipartite is O(sqrt(V) E)
// DESCRIPTION: Dinic

const int oo = 0x3f3f3f3f3f3f3f3f;
// Edge structure
struct Edge
{
    int from, to;
    int flow, capacity;

    Edge(int from_, int to_, int flow_, int capacity_)
        : from(from_), to(to_), flow(flow_), capacity(capacity_)
    {}
};

struct Dinic
{
    vector<vector<int>> graph;
    vector<Edge> edges;
    vector<int> level;
    int size;

    Dinic(int n)
    {
        graph.resize(n);
        level.resize(n);
        size = n;
        edges.clear();
    }

    void add_edge(int from, int to, int capacity)
    {
        edges.emplace_back(from, to, 0, capacity);
        graph[from].push_back(edges.size() - 1);

        edges.emplace_back(to, from, 0, 0);
        graph[to].push_back(edges.size() - 1);
    }

    int get_max_flow(int source, int sink)
    {
        int max_flow = 0;
        vector<int> next(size);
        while(bfs(source, sink)) {
            next.assign(size, 0);
            for (int f = dfs(source, sink, next, oo); f != 0; f = dfs(source, sink, next, oo)) {
                max_flow += f;
            }
        }
        return max_flow;
    }

    bool bfs(int source, int sink)
    {
        level.assign(size, -1);
        queue<int> q;
        q.push(source);
        level[source] = 0;

        while(!q.empty()) {
            int a = q.front();
            q.pop();

            for (int & b: graph[a]) {
                auto edge = edges[b];
                int cap = edge.capacity - edge.flow;
                if (cap > 0 && level[edge.to] == -1) {
                    level[edge.to] = level[a] + 1;
                    q.push(edge.to);
                }
            }
        }
        return level[sink] != -1;
    }

    int dfs(int curr, int sink, vector<int> & next, int flow)
    {
        if (curr == sink) return flow;
        int num_edges = graph[curr].size();

        for (; next[curr] < num_edges; next[curr]++) {
            int b = graph[curr][next[curr]];
            auto & edge = edges[b];
            auto & rev_edge = edges[b^1];

            int cap = edge.capacity - edge.flow;
            if (cap > 0 && (level[curr] + 1 == level[edge.to])) {
                int bottle_neck = dfs(edge.to, sink, next, min(flow, cap));
                if (bottle_neck > 0) {
                    edge.flow += bottle_neck;
                    rev_edge.flow -= bottle_neck;
                    return bottle_neck;
                }
            }
        }
        return 0;
    }

    vector<pair<int, int>> mincut(int source, int sink)
    {
        vector<pair<int, int>> cut;
        bfs(source, sink);
        for (auto & e: edges) {
            if (e.flow == e.capacity && level[e.from] != -1 && level[e.to] == -1 && e.capacity > 0) {
                cut.emplace_back(e.from, e.to);
            }
        }
        return cut;
    }
};

// Example on how to use
void solve()
{
    int n, m;
    cin >> n >> m;
    int N = n + m + 2;

    int source = N - 2;
    int sink = N - 1;

    Dinic flow(N);

    for (int i = 0; i < n; i++) {
        int q; cin >> q;
        while(q--) {
            int b; cin >> b;
            flow.add_edge(i, n + b - 1, 1);
        }
    }
    for (int i =0; i < n; i++) {
        flow.add_edge(source, i, 1);
    }
    for (int i =0; i < m; i++) {
        flow.add_edge(i + n, sink, 1);
    }

    cout << m - flow.get_max_flow(source, sink) << endl;

    // Getting participant edges
    for (auto & edge: flow.edges) {
        if (edge.capacity == 0) continue; // This means is a reverse edge
        if (edge.from == source || edge.to == source) continue;
        if (edge.from == sink   || edge.to == sink) continue;
        if (edge.flow == 0) continue; // Is not participant

        cout << edge.from + 1 << " " << edge.to -n + 1 << endl;
    }
}
```

2.2 Dinic Min cost

Complexity: O(V*V*E), Bipartite is O(sqrt(V) E)
Gives you the max_flow with the min cost

```
// TITLE: Dinic Min cost
// COMPLEXITY: O(V*V*E), Bipartite is O(sqrt(V) E)
// DESCRIPTION: Gives you the max_flow with the min cost

// Edge structure
struct Edge
{
    int from, to;
    int flow, capacity;
    int cost;

    Edge(int from_, int to_, int flow_, int capacity_, int cost_)
        : from(from_), to(to_), flow(flow_), capacity(capacity_), cost(cost_)
    {}
};

struct Dinic
{
    vector<vector<int>> graph;
    vector<Edge> edges;
    vector<int> dist;
    vector<bool> inqueue;
    int size;
    int cost = 0;

    Dinic(int n)
    {
        graph.resize(n);
        dist.resize(n);
        inqueue.resize(n);
        size = n;
        edges.clear();
    }

    void add_edge(int from, int to, int capacity, int cost)
    {
        edges.emplace_back(from, to, 0, capacity, cost);
        graph[from].push_back(edges.size() - 1);

        edges.emplace_back(to, from, 0, 0, -cost);
        graph[to].push_back(edges.size() - 1);
    }

    int get_max_flow(int source, int sink)
    {
        int max_flow = 0;
        vector<int> next(size);
        while(spfa(source, sink)) {
            next.assign(size, 0);
            for (int f = dfs(source, sink, next, oo); f != 0; f = dfs(source, sink, next, oo)) {
                max_flow += f;
            }
        }
        return max_flow;
    }

    bool spfa(int source, int sink)
    {
        dist.assign(size, oo);
        inqueue.assign(size, false);
        queue<int> q;
        q.push(source);
        dist[source] = 0;
        inqueue[source] = true;

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

            for (int & b: graph[a]) {
                auto edge = edges[b];
                int cap = edge.capacity - edge.flow;
                if (cap > 0 && dist[edge.to] > dist[edge.from] + edge.cost) {
                    dist[edge.to] = dist[edge.from] + edge.cost;
                    if (not inqueue[edge.to]) {
                        q.push(edge.to);
                        inqueue[edge.to] = true;
                    }
                }
            }
        }
        return dist[sink] != oo;
    }

    int dfs(int curr, int sink, vector<int> & next, int flow)
    {
        if (curr == sink) return flow;
        int num_edges = graph[curr].size();

        for (; next[curr] < num_edges; next[curr]++) {
            int b = graph[curr][next[curr]];
            auto & edge = edges[b];
            auto & rev_edge = edges[b^1];
            int cap = edge.capacity - edge.flow;
            if (cap > 0 && (dist[edge.from] + edge.cost == dist[edge.to])) {
                int bottle_neck = dfs(edge.to, sink, next, min(flow, cap));
                if (bottle_neck > 0) {
                    edge.flow += bottle_neck;
                    rev_edge.flow -= bottle_neck;
                    cost += edge.cost * bottle_neck;
                    return bottle_neck;
                }
            }
        }
        return 0;
    }

    vector<pair<int, int>> mincut(int source, int sink)
    {
        vector<pair<int, int>> cut;
        spfa(source, sink);
        for (auto & e: edges) {
            if (e.flow == e.capacity && dist[e.from] != oo && level[e.to] == oo && e.capacity > 0) {
                cut.emplace_back(e.from, e.to);
            }
        }
        return cut;
    }
};

// Example on how to use
void solve()
{

    int N = 10;

    int source = 8;
    int sink = 9;

    Dinic flow(N);
    flow.add_edge(8, 0, 4, 0);
    flow.add_edge(8, 1, 3, 0);
    flow.add_edge(8, 2, 2, 0);
    flow.add_edge(8, 3, 1, 0);

    flow.add_edge(0, 6, oo, 3);
    flow.add_edge(0, 7, oo, 2);
    flow.add_edge(0, 5, oo, 0);

    flow.add_edge(1, 4, oo, 0);

    flow.add_edge(4, 9, oo, 0);
    flow.add_edge(5, 9, oo, 0);
    flow.add_edge(6, 9, oo, 0);
    flow.add_edge(7, 9, oo, 0);

    int ans = flow.get_max_flow(source, sink);
    debug(ans);
    debug(flow.cost);
}

int32_t main()
{
    solve();
}
```

2.3 Dkistra

Complexity: O(E + V.log(v))

```
// TITLE: Dkistra
// COMPLEXITY: O(E + V.log(v))
// DESCRIPION: Finds to shortest path from start

int dist[mxN];
bool vis[mxN];
vector<pair<int, int>> g[mxN];

void dikstra(int start)
{
    fill(dist, dist + mxN, oo);
    fill(vis, vis + mxN, 0);
    priority_queue<pair<int, int>> q;
    dist[start] = 0;
    q.push({0, start});

    while(!q.empty()) {
        auto [d, a] = q.top();
        q.pop();
        if (vis[a]) continue;
        vis[a] = true;
        for (auto [b, w]: g[a]) {
            if (dist[a] + w < dist[b]) {
                dist[b] = dist[a] + w;
                q.push({-dist[b], b});
            }
        }
    }
}
```

2.4 Dominator tree

Complexity: O(E + V)

```
// TITLE: Dominator tree
// COMPLEXITY: O(E + V)
// DESCRIPION: Builds dominator tree

vector<int> g[mxN];
vector<int> S, gt[mxN], T[mxN];
int dsu[mxN], label[mxN];
int sdom[mxN], idom[mxN], id[mxN];
int dfs_time = 0;

vector<int> bucket[mxN];
vector<int> down[mxN];

void prep(int a)
{
    S.pb(a);
    id[a] = ++dfs_time;
    label[a] = sdom[a] = dsu[a] = a;

    for (auto b: g[a]) {
        if (!id[b]) {
            prep(b);
            down[a].pb(b);
        }
        gt[b].pb(a);
    }
}

int fnd(int a, int flag = 0)
{
    if (a == dsu[a]) return a;
    int p = fnd(dsu[a], 1);
    int b = label[ dsu[a] ];
    if (id [ sdom[b] ] < id[ sdom[ label[a] ] ]) {
        label[a] = b;
    }
    dsu[a] = p;
    return (flag ? p: label[a]);
}

void build_dominator_tree(int root)
{
    prep(root);
    reverse(all(S));

    int w;
    for (int a: S) {
        for (int b: gt[a]) {
            w = fnd(b);
            if (id[ sdom[w] ] < id[ sdom[a] ]) {
                sdom[a] = sdom[w];
            }
        }
        gt[a].clear();
        if (a != root) {
            bucket[ sdom[a] ].pb(a);
        }
        for (int b: bucket[a]) {
            w = fnd(b);
            if (sdom[w] == sdom[b]) {
                idom[b] = sdom[b];
            }
            else {
                idom[b] = w;
            }
        }
        bucket[a].clear();
        for (int b: down[a]) {
            dsu[b] = a;
        }
        down[a].clear();
    }
    reverse(all(S));
    for (int a: S) {
        if (a != root) {
            if (idom[a] != sdom[a]) {
                idom[a] = idom[ idom[a] ];
            }
            T[ idom[a] ].pb(a);
        }
    }
    S.clear();
}
```

2.5 Bellman Ford

Complexity: O(n * m) | n = |nodes|, m = |edges|
Finds shortest paths from a starting node to all nodes of the graph. The node can have negative cycle and belman-ford will detected

~~~
// TITLE: Bellman Ford
// COMPLEXITY: O(n * m) | n = |nodes|, m = |edges|
// DESCRIPTION: Finds shortest paths from a starting node to all nodes of the graph. The node can have negative cycle and belman-ford will detected

// a and b vertices, c cost
// [{a, b, c}, {a, b, c}]
vector<tuple<int, int, int>> edges;
int N;

void bellman_ford(int x){
    for (int i = 0; i < N; i++){
        dist[i] = oo;
    }
    dist[x] = 0;

    for (int i = 0; i < N - 1; i++){
        for (auto [a, b, c]: edges){
            if (dist[a] == oo) continue;
            dist[b]= min(dist[b], dist[a] + w);
        }
    }
}
// return true if has cycle
bool check_negative_cycle(int x){
    for (int i = 0; i < N; i++){
        dist[i] = oo;
    }
    dist[x] = 0;

    for (int i = 0; i < N - 1; i++){
        for (auto [a, b, c]: edges){
            if (dist[a] == oo) continue;
            dist[b]= min(dist[b], dist[a] + w);
        }
    }

    for (auto [a, b, c]: edges){
        if (dist[a] == oo) continue;
        if (dist[a] + w < dist[b]){
            return true;
        };
    }
    return false;
}
```
~~~

2.6 Dfs tree

Complexity: O(E + V)

```
// TITLE: Dfs tree
// COMPLEXITY: O(E + V)
// DESCRIPION: Create dfs tree from graph

int desce[mxN], sobe[mxN];
int backedges[mxN], vis[mxN];
int pai[mxN], h[mxN];

void dfs(int a, int p) {
    if(vis[a]) return;
    pai[a] = p;
    h[a] = h[p]+1;
    vis[a] = 1;

    for(auto b : g[a]) {
        if (p == b) continue;
        if (vis[b]) continue;
        dfs(b, a);
        backedges[a] += backedges[b];
    }
    for(auto b : g[a]) {
        if(h[b] > h[a]+1)
            desce[a]++;
        else if(h[b] < h[a]-1)
            sobe[a]++;
    }
    backedges[a] += sobe[a] - desce[a];
}
```

2.7 Topological Sort

Complexity: O(N + M), N: Vertices, M: Arestas
Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

```cpp
// TITLE: Topological Sort
// COMPLEXITY: O(N + M), N: Vertices, M: Arestas
// DESCRIPTION: Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

typedef vector<vector<int>> Adj_List;
typedef vector<int> Indegree_List; // How many nodes depend on him
typedef vector<int> Order_List;    // The order in which the nodes appears

Order_List kahn(Adj_List adj, Indegree_List indegree)
{
    queue<int> q;
    // priority_queue<int> q; // If you want in lexicografic order
    for (int i = 0; i < indegree.size(); i++) {
        if (indegree[i] == 0)
            q.push(i);
    }
    vector<int> order;

    while (not q.empty()) {
        auto a = q.front();
        q.pop();

        order.push_back(a);
        for (auto b: adj[a]) {
            indegree[b]--;
            if (indegree[b] == 0)
                q.push(b);
        }
    }
    return order;
}

int32_t main()
{

    Order_List = kahn(adj, indegree);
    if (Order_List.size() != N) {
        cout << "IMPOSSIBLE" << endl;
    }
    return 0;
}
```

2.8 Kth Ancestor

Complexity: O(n * log(n))
Preprocess, then find in log n

```cpp
// TITLE: Kth Ancestor
// COMPLEXITY: O(n * log(n))
// DESCRIPTION: Preprocess, then find in log n

const int LOG_N = 30;
int get_kth_ancestor(vector<vector<int>> & up, int v, int k)
{
    for (int j = 0; j < LOG_N; j++) {
        if (k & ((int)1 << j)) {
            v = up[v][j];
        }
    }
    return v;
}

void solve()
{
    vector<vector<int>> up(n, vector<int>(LOG_N));

    for (int i = 0; i < n; i++) {
        up[i][0] = parents[i];
        for (int j = 1; j < LOG_N; j++) {
            up[i][j] = up[up[i][j-1]][j-1];
        }
    }
    cout << get_kth_ancestor(up, x, k) << endl;

}
```

3 Parser

3.1 Parsing Functions

Complexity:

```cpp
// TITLE: Parsing Functions

vector<string> split_string(const string & s, const string & sep = " ") {
    int w = sep.size();
    vector<string> ans;
    string curr;

    auto add = [&](string a) {
        if (a.size() > 0) {
            ans.push_back(a);
        }
    };

    for (int i = 0; i + w < s.size(); i++) {
        if (s.substr(i, w) == sep) {
            i += w-1;
            add(curr);
            curr.clear();
            continue;
        }
        curr.push_back(s[i]);
    }
    add(curr);
    return ans;
}

vector<int> parse_vector_int(string & s)
{
    vector<int> nums;
    for (string x: split_string(s)) {
        nums.push_back(stoi(x));
    }
    return nums;
}

vector<float> parse_vector_float(string & s)
{
    vector<float> nums;
    for (string x: split_string(s)) {
        nums.push_back(stof(x));
    }
    return nums;
}

void solve()
{
    cin.ignore();
    string s;
    getline(cin, s);

    auto nums = parse_vector_float(s);
    for (auto x: nums) {
        cout << x << endl;
    }
}
```

4 String

4.1 Z function

Complexity: Z function complexity
z function

```
// TITLE: Z function
// COMPLEXITY: Z function complexity
// DESCRIPTION: z function

void z_function(string& s)
{
    return;
}
```

4.2 Suffix Array

Complexity: O(n log(n)), contains big constant (around 25).
Computes a sorted array of the suffixes of a string.

```
// TITLE: Suffix Array
// COMPLEXITY: O(n log(n)), contains big constant (around 25).
// DESCRIPTION: Computes a sorted array of the suffixes of a string.

void countingsort(vi& p, vi& c) {
    int n=p.size();
    vi count(n,0);
    loop(i,0,n) count[c[i]]++;

    vi psum(n); psum[0]=0;
    loop(i,1,n) psum[i]=psum[i-1]+count[i-1];

    vi ans(n);
    loop(i,0,n)
        ans[psum[c[p[i]]]++]=p[i];

    p = ans;
}

vi sfa(string s) {
    s += "$";

    int n=s.size();
    vi p(n);
    vi c(n);
    {
        vector<pair<char, int>> a(n);
        loop(i,0,n) a[i]={s[i],i};
        sort(all(a));

        loop(i,0,n) p[i]=a[i].second;

        c[p[0]]=0;
        loop(i,1,n) {
            if(s[p[i]] == s[p[i-1]]) {
                c[p[i]]=c[p[i-1]];
            }
            else c[p[i]]=c[p[i-1]]+1;
        }
    }

    for(int k=0; (1<<k) < n; k++) {
        loop(i, 0, n)
            p[i] = (p[i] - (1<<k) + n) % n;

        countingsort(p,c);

        vi nc(n);
        nc[p[0]]=0;
        loop(i,1,n) {
            pii prev = {c[p[i-1]], c[(p[i-1]+(1<<k))%n]};
            pii cur = {c[p[i]], c[(p[i]+(1<<k))%n]};

            if (prev == cur)
                nc[p[i]]=nc[p[i-1]];
            else nc[p[i]]=nc[p[i-1]]+1;
        }
        c=nc;
    }

    return p;
}
```

4.3 String hash

Complexity: O(n) preprocessing, O(1) query
Computes the hash of arbitrary substrings of a given string s.

```
// TITLE: String hash
// COMPLEXITY: O(n) preprocessing, O(1) query
// DESCRIPTION: Computes the hash of arbitrary substrings of a given string s.

struct hashs
{
    string s;
    int m1, m2, n, p;
    vector<int> p1, p2, sum1, sum2;

    hashs(string s) : s(s), n(s.size()), p1(n + 1), p2(n + 1), sum1(n + 1), sum2(n + 1)
    {
        srand(time(0));
        p = 31;
        m1 = rand() / 10 + 1e9; // 1000253887;
        m2 = rand() / 10 + 1e9; // 1000546873;

        p1[0] = p2[0] = 1;
        loop(i, 1, n + 1)
        {
            p1[i] = (p * p1[i - 1]) % m1;
            p2[i] = (p * p2[i - 1]) % m2;
        }

        sum1[0] = sum2[0] = 0;
        loop(i, 1, n + 1)
        {
            sum1[i] = (sum1[i - 1] * p) % m1 + s[i - 1];
            sum2[i] = (sum2[i - 1] * p) % m2 + s[i - 1];
            sum1[i] %= m1;
            sum2[i] %= m2;
        }
    }

    // hash do intervalo [l, r)
    int gethash(int l, int r)
    {
        int c1 = m1 - (sum1[l] * p1[r - l]) % m1;
        int c2 = m2 - (sum2[l] * p2[r - l]) % m2;
        int h1 = (sum1[r] + c1) % m1;
        int h2 = (sum2[r] + c2) % m2;
        return (h1 << 30) ^ h2;
    }
};
```

5 Algorithms

5.1 Sparse table

Complexity: O(n log(n)) preprocessing, O(1) query
Computes the minimum of a half open interval.

```
// TITLE: Sparse table
// COMPLEXITY: O(n log(n)) preprocessing, O(1) query
// DESCRIPTION: Computes the minimum of a half open interval.

struct sptable {
    vector<vi> table;

    int ilog(int x) {
        return (__builtin_clzll(1ll) - __builtin_clzll(x));
    }

    sptable(vi& vals) {
        int n = vals.size();
        int ln= ilog(n)+1;
        table.assign(ln, vi(n));

        loop(i,0,n) table[0][i]=vals[i];

        loop(k, 1, ln) {
            loop(i,0,n) {
                table[k][i] = min(table[k-1][i],
                table[k-1][min(i + (1<<(k-1)), n-1)]);
            }
        }
    }

    // returns minimum of vals in range [a, b)
    int getmin(int a, int b) {
        int k = ilog(b-a);
        return min(table[k][a], table[k][b-(1<<k)]);
    }
};
```

6 Set

6.1 Ordered Set

Complexity: log n
Worst set with adtional operations

```
// TITLE: Ordered Set
// COMPLEXITY: log n
// DESCRIPTION: Worst set with adtional operations


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

int32_t main() {
    ordered_set<int> oset;

    oset.insert(5);
    oset.insert(1);
    oset.insert(2);
    // o_set = {1, 2, 5}
    5 == *(oset.find_by_order(2)); // Like an array index
    2 == oset.order_of_key(4); // How many elements are strictly less than 4
}
```

6.2 Set

Complexity: Insertion Log(n)
Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

```
// TITLE: Set
// COMPLEXITY: Insertion Log(n)
// Description: Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

int main() {
  set<int> set1;

  set1.insert(1);       // O(log(n))
  set1.erase(1);        // O(log(n))

  set1.upper_bound(1);  // O(log(n))
  set1.lower_bound(1);  // O(log(n))
  set1.find(1);         // O(log(n))
  set1.count(1);        // O(log(n))

  set1.size();          // O(1)
  set1.empty();         // O(1)

  set1.clear()          // O(1)
  return 0;
}
```

6.3 Multiset

Complexity: O(log(n))
Same as set but you can have multiple elements with same values

```
// TITLE: Multiset
// COMPLEXITY: O(log(n))
// DESCRIPTION: Same as set but you can have multiple elements with same values

int main() {
  multiset<int> set1;
}
```