Universidad de Chile

Índice

1. Estructuras de datos **1**
1.1. Disjoint Set Union (Union-Find) 1
1.2. Segment Tree 1
1.3. Segment Tree Lazy 2
1.4. Segment Tree Persistente 2
1.5. Segment Tree Iterativo (compacto) 3
1.6. Sparse Table 3
1.7. Li Chao Tree (dynamic, persistent) 4

2. Grafos **4**
2.1. LCA (Binary Lifting) 4
2.2. LCA (Sparse Table+ETT) 5
2.3. Heavy Light Decomposition 5
2.4. Dinic Max Flow 6
2.5. Centroid Decomposition 7

3. Matemáticas **7**
3.1. Exponenciación binaria 7
3.2. Logaritmo discreto 7
3.3. Lema de Burnside 8
3.4. Fórmula de inversión de Möbius 8

4. C++ **8**
4.1. Custom set/map hash 8
4.2. Policy Based Order Statistics Tree 8

1. Estructuras de datos

1.1. Disjoint Set Union (Union-Find)

```
// find y union en ~O(1) amortizado
struct DSU{
    vector <int> parent, sz; // sz = size
    DSU(int n){
        parent.resize(n);
        sz.resize(n);
        for(int i=0; i<n; i++){
            parent[i] = i;
            sz[i] = 1;
        }
    }
    int find_set(int v){
        if(v == parent[v]) return v;
        return parent[v] = find_set(parent[v]);
    }
    void union_set(int a, int b){
        a = find_set(a);
        b = find_set(b);
        if(a != b){
            if(sz[a] < sz[b])
                swap(a,b);
            parent[b] = a;
            sz[a] += sz[b];
        }
    }
};
```

1.2. Segment Tree

```
template <class T, T merge(T,T)>
struct segment_tree{
  int N;
  vector <T> tree;
  segment_tree(int _N){
    N = _N;
    tree.resize(4*N);
    build(0, 0, N-1);
  }

  segment_tree(vector <T> &A){
          N = int(A.size());
          tree.resize(4*N);
          build(0, 0, N-1, A);
  }

  void build(int n, int i, int j){
    if(i == j){
      tree[n] = T(); // initial value
      return;
    }
    int mid = (i+j)/2;
    build(2*n+1, i, mid);
    build(2*n+2, mid+1, j);
    tree[n] = merge(tree[2*n+1], tree[2*n+2]);
  }

  void build(int n, int i, int j, vector <T> &A){
    if(i == j){
      tree[n] = A[i]; // initial value
      return;
    }
    int mid = (i+j)/2;
    build(2*n+1, i, mid, A);
    build(2*n+2, mid+1, j, A);
    tree[n] = merge(tree[2*n+1], tree[2*n+2]);
  }

  T query(int l, int r){
    return query(0, 0, N-1, l, r);
  }

  T query(int n, int i, int j, int l, int r){
    if(l <= i && j <= r) return tree[n];
    int mid = (i+j)/2;
    if(mid < l || r < i)
      return query(2*n+2, mid+1, j, l, r);
    if(j < l || r < mid+1)
      return query(2*n+1, i, mid, l, r);
    return merge(
        query(2*n+1, i, mid, l, r),
        query(2*n+2, mid+1, j, l, r));
  }

  void update(int t, T val){
    update(0, 0, N-1, t, val);
  }

  void update(int n, int i, int j, int t, T val){
    if(t < i || j < t) return;
    if(i == j){
      tree[n] = val;
      return;
    }
    int mid = (i+j)/2;
    update(2*n+1, i, mid, t, val);
    update(2*n+2, mid+1, j, t, val);
    tree[n] = merge(tree[2*n+1], tree[2*n+2]);
  }

  int search(int from, T val){
    if(!from) return search(0, 0, N-1, val);
    return search(0, 0, N-1, val+query(0, from-1));
  }

  int search(int n, int i, int j, T val){
    if(tree[n] < val) return -1;
```

```
    if(i==j && tree[n] >= val) return i;
    int mid = (i+j)/2;
    if(tree[2*n+1] >= val) return search(2*n+1, i, mid, val);
    else return search(2*n+2, mid+1, j, val-tree[2*n+1]);
  }
};
```

1.3. Segment Tree Lazy

```
template <class T, T merge(T, T)>
struct segment_tree{
        int N;
        vector <T> tree, lazy;
        segment_tree(int _N){
                N = _N;
                tree.resize(4*N);
                lazy.assign(4*N, T()); // modify default value
                build(0, 0, N-1);
        }

        segment_tree(vector <T> &A){
                N = A.size();
                tree.resize(4*N);
                lazy.assign(4*N, T()); // modify default value
                build(0, 0, N-1, A);
        }

        void build(int n, int i, int j){
                if(i == j){
                        tree[n] = T(); // initial value
                        return;
                }
                int mid = (i+j)/2;
                build(2*n+1, i, mid);
                build(2*n+2, mid+1, j);
                tree[n] = merge(tree[2*n+1], tree[2*n+2]);
        }

        void build(int n, int i, int j, vector <T> &A){
                if(i == j){
                        tree[n] = A[i];
                        return;
                }
                int mid = (i+j)/2;
                build(2*n+1, i, mid, A);
                build(2*n+2, mid+1, j, A);
                tree[n] = merge(tree[2*n+1], tree[2*n+2]);
        }

        void push(int n, int i, int j){
                // modify this function
                if(lazy[n]){
                        tree[n] += lazy[n]*(j-i+1); // range increment
                        if(i != j){
                                lazy[2*n+1] += lazy[n];
                                lazy[2*n+2] += lazy[n];
                        }
                        lazy[n] = T();
                }
        }

        T query(int l, int r){
                return query(0, 0, N-1, l, r);
        }

        T query(int n, int i, int j, int l, int r){
                push(n, i, j);
                if(l <= i && j <= r) return tree[n];
                int mid = (i+j)/2;
                if(mid < l || r < i)
                        return query(2*n+2, mid+1, j, l, r);
                if(j < l || r < mid+1)
                        return query(2*n+1, i, mid, l, r);
                return merge(
                                query(2*n+1, i, mid, l, r),
                                query(2*n+2, mid+1, j, l, r));
        }

        void update(int l, int r, T val){
                update(0, 0, N-1, l, r, val);
        }

        void update(int n, int i, int j, int l, int r, T val){
                if(l <= i && j <= r){
                        lazy[n] += val; // modify this
                        push(n, i, j);
                        return;
                }
                push(n, i, j);
                if(r < i || j < l) return;
                int mid = (i+j)/2;
                update(2*n+1, i, mid, l, r, val);
                update(2*n+2, mid+1, j, l, r, val);
                tree[n] = merge(tree[2*n+1], tree[2*n+2]);
        }
};
```

1.4. Segment Tree Persistente

```
// Same time complexity as normal SegmentTree
// Additional O(log(n)) memory per update
template <typename T, T merge(T, T)>
struct st_node{
        st_node *left=0, *right=0;
        int i, j;
        T val;
        st_node() {}
        st_node(int _i, int _j) : i(_i), j(_j) {}
```

```
	st_node(vector <T> &A){
		int N = int(A.size());
		i = 0, j = N-1;
		build(A);
	}

	void build(vector <ll> &A){
		if(i == j){
			val = A[i];
			return;
		}
		int mid = (i+j)/2;
		left = new st_node<T,merge>(i, mid);
		right = new st_node<T,merge>(mid+1, j);
		left->build(A);
		right->build(A);
		val = merge(left->val, right->val);
	}

	st_node *update(int t, ll v){
		if(t < i || j < t){
			return this;
		}
		if(i == j){
			st_node *ret = new st_node<T,merge>(*this);
			ret->val = v;
			return ret;
		}
		st_node *ret = new st_node<T,merge>(i, j);
		ret->left = left->update(t, v);
		ret->right = right->update(t, v);
		ret->val = merge(ret->left->val, ret->right->val);
		return ret;
	}

	ll query(int l, int r){
		if(l <= i && j <= r) return val;
		int mid = (i+j)/2;
		if(mid < l || r < i) return right->query(l, r);
		else if(j < l || r < mid+1) return left->query(l, r);
		return merge(left->query(l, r), right->query(l, r));
	}
};
```

1.5. Segment Tree Iterativo (compacto)

```
template<class T, T m(T, T)> struct iter_seg_tree{
  int n; vector<T> ST;
  iter_seg_tree(vector<T> &a){
    n = a.size(); ST.resize(n << 1);
    for (int i=n;i<(n<<1);i++)ST[i]=a[i-n];
    for (int i=n-1;i>0;i--)ST[i]=m(ST[i<<1],ST[i<<1|1]);
  }
  void update(int pos, T val){ // replace with val
    ST[pos += n] = val;
    for (pos >>= 1; pos > 0; pos >>= 1)
      ST[pos] = m(ST[pos<<1], ST[pos<<1|1]);
  }
  T query(int l, int r){ // [l, r]
    T ansL, ansR; bool hasL = 0, hasR = 0;
    for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
      if (l & 1)
        ansL=(hasL?m(ansL,ST[l++]):ST[l++]),hasL=1;
      if (r & 1)
        ansR=(hasR?m(ST[--r],ansR):ST[--r]),hasR=1;
    }
    if (!hasL) return ansR; if (!hasR) return ansL;
    return m(ansL, ansR);
  }
};
// Example:
iter_seg_tree<int, my_merge_function> st;
```

1.6. Sparse Table

```
// O(nlogn) preprocesamiento, O(1) query en rango
// para función idempotente (como min, max, gcd, etc)
struct sparse_table{
    int n;
    vector <int> logs;
    vector <vector<ll>> table;
    sparse_table(vector <ll> &A){
        n = A.size();
        logs.resize(n+1);
        logs[1] = 0;
        for(int i=2; i<=n; i++){
            logs[i] = logs[i/2] + 1;
        }
        table.assign(logs[n]+1, vector<ll>(n,0));
        for(int i=0; i<=logs[n]; i++){
            int cur_len = 1 << i;
            for(int j=0; j+cur_len-1<n; j++){
                if(cur_len == 1){
                    table[i][j] = A[j];
                }
                else{
                    table[i][j] = min(table[i-1][j], table[i-1][j+cur_len
                        /2]);
                }
            }
        }
    }
    ll query(int i, int j){
        int p = logs[j-i+1];
        int len = 1 << p;
        return min(table[p][i], table[p][j-len+1]);
    }
};
```

1.7. Li Chao Tree (dynamic, persistent)

```
struct line{
        ll a,b;
        line(){}
        line(ll _a, ll _b) : a(_a), b(_b) {}
        ll eval(ll x) { return a*x+b; }
};

// Dynamic/persistent min Li Chao tree
// Tested on: https://codeforces.com/contest/319/problem/C (add_line,
    ↪ query)
// Tested on: https://www.acmicpc.net/problem/3319 (padd_line, query)
struct lc_node{
        lc_node *left=0, *right=0;
        ll i, j;
        line val;
        lc_node(ll _i, ll _j, line _val) : i(_i), j(_j), val(_val) {}

        // Non-persistent line add
        void add_line(ll a, ll b){
                line v(a,b);
                add_line(v);
        }

        void add_line(line &v){
                ll cur_left=val.eval(i), cur_right=val.eval(j);
                ll new_left=v.eval(i), new_right=v.eval(j);
                if(cur_left <= new_left && cur_right <= new_right) return;
                if(cur_left > new_left && cur_right > new_right){
                        val=v;
                        return;
                }
                ll mid = (i+j)>>1;
                if(cur_left > new_left) swap(val, v);
                if(val.eval(mid) < v.eval(mid)){
                        if(!right) right = new lc_node(mid+1, j, v);
                        else right->add_line(v);
                }
                else{
                        swap(val, v);
                        if(!left) left = new lc_node(i, mid, v);
                        else left->add_line(v);
                }
        }

        // Persistent line add
        lc_node *padd_line(ll a, ll b){
                line v(a,b);
                return padd_line(v);
        }

        lc_node *padd_line(line &v){
                ll cur_left=val.eval(i), cur_right=val.eval(j);
                ll new_left=v.eval(i), new_right=v.eval(j);
                if(cur_left <= new_left && cur_right <= new_right) return
                    ↪ this;
                lc_node *ret = new lc_node(*this);
                if(cur_left > new_left && cur_right > new_right){
                        ret->val = v;
                        return ret;
                }
                ll mid = (i+j)>>1;
                if(cur_left > new_left) swap(ret->val, v);
                if(ret->val.eval(mid) < v.eval(mid)){
                        if(!ret->right) ret->right = new lc_node(mid+1, j,
                            ↪ v);
                        else ret->right = ret->right->padd_line(v);
                }
                else{
                        swap(ret->val, v);
                        if(!ret->left) ret->left = new lc_node(i, mid, v);
                        else ret->left = ret->left->padd_line(v);
                }
                return ret;
        }
        ll query(ll x){
                if(i == j) return val.eval(x);
                ll mid = (i+j)>>1;
                if(x <= mid && left) return min(val.eval(x), left->query(x
                    ↪ ));
                else if(x >= mid+1 && right) return min(val.eval(x), right
                    ↪ ->query(x));
                return val.eval(x);
        }
};

/* Example:
lc_node *root = new lc_node(min_val, max_val, line(b[0],0));
for(int i=1; i<n; i++){
        dp[i] = root->query(a[i]);
        root->add_line(b[i],dp[i]);
}
*/
```

2. Grafos

2.1. LCA (Binary Lifting)

```
struct LCA{ // Uses Binary Lifting. O(nlogn) preprocessing, O(logn) query.
    int n, l, timer=0;
    vector <vector<int>> up;
    vector <int> enter, exit;
    LCA(vector <vector<int>> &adj, int root=0){
        n = adj.size();
        l = ceil(log2(n));
        enter.resize(n);
        exit.resize(n);
```

```
        up.resize(n, vector<int>(l+1));
        dfs(root, root, adj);
    }

    void dfs(int u, int p, vector<vector<int>> &adj){
        enter[u] = timer++;
        up[u][0] = p;
        for(int j=1; j<=l; j++){
            up[u][j] = up[up[u][j-1]][j-1];
        }
        for(int v : adj[u]){
            if(v != p) dfs(v, u, adj);
        }
        exit[u] = timer++;
    }

    bool is_ancestor(int u, int v){ // v is ancestor of u
        return enter[u] <= enter[v] && exit[u] >= exit[v];
    }

    int query(int u, int v){
        if(is_ancestor(u,v)) return u;
        if(is_ancestor(v,u)) return v;
        for(int i=l; i>=0; i--){
            if(!is_ancestor(up[u][i], v)){
                u = up[u][i];
            }
        }
        return up[u][0];
    }
};
```

2.2. LCA (Sparse Table+ETT)

```
// LCA with SparseTable and Euler Tour
// Requires sparse table of pair<int,int> with min operation
// O(n log n) preprocessing, O(1) query
struct LCA{
        SparseTable st;
        int time=0;
        vector <pair<int,int>> euler;
        vector <int> left, right;
        vector <bool> vis;
        LCA(vector <vector <int>> &adj, int root=0){
                int n = int(adj.size());
                left.resize(n);
                right.resize(n);
                vis.assign(n, false);
                dfs(root, adj);
                st = SparseTable(euler);
        }

        void dfs(int u, vector<vector<int>> &adj, int depth=0){
                vis[u] = 1;
                left[u] = int(euler.size());
                euler.push_back({depth,u});
                for(int v : adj[u]){
                        if(!vis[v]) dfs(v, adj, depth+1);
                        right[u] = int(euler.size());
                        euler.push_back({depth,u});
                }
        }

        int query(int u, int v){
                if(left[u] > right[v]) swap(u,v);
                return st.query(left[u], right[v]);
        }
};
```

2.3. Heavy Light Decomposition

```
// Heavy Light decomposition of a tree
// Queries in O(log²(n))
// requires: segment_tree
// querying on edges: store edge value in child, change enter[u] in query
template <class T, T merge(T, T)>
struct heavy_light{
        // depth: node depth;
        // sz: subtree size
        // enter: discovery time (index in euler tour)
        // par: parent node
        // head: head of node's chain
        vector <int> depth, sz, enter, par, head;
        segment_tree <T, merge> st;
        vector <T> euler;
        vector <vector <int>> &adj;
        vector <T> &val;
        int time=0;
        /* adj: adjacency list
         * val: value associated with each node
         * merge: merge function for queries
         */
        heavy_light(vector <vector <int>> &_adj, vector <T> &_val, int
            ↪ root=0) : adj(_adj), val(_val) {
                int n = int(adj.size());
                depth.resize(n); sz.resize(n);
                enter.resize(n); par.resize(n);
                euler.resize(n); head.resize(n);
                par[root] = -1;
                depth[root] = 0;
                dfs1(root);
                dfs2(root, root);
                st = segment_tree <T, merge>(euler);
        }
        void dfs1(int u){ // first dfs, computes depth and sz
                sz[u]=1;
                for(int v : adj[u]){
                        if(v != par[u]){
```

```
                        par[v] = u;
                        depth[v] = depth[u]+1;
                        dfs1(v);
                        sz[u] += sz[v];
                    }
                }
            }
        void dfs2(int u, int h){ // second dfs, computes hld
                head[u] = h;
                enter[u] = time++;
                euler[enter[u]] = val[u];
                int mx=-1;
                for(int v : adj[u]){
                        if(par[u] != v && (mx==-1 || sz[v]>sz[mx])) mx=v;
                }
                if(mx != -1) dfs2(mx, h);
                for(int v : adj[u]){
                        if(v != par[u] && v != mx)
                                dfs2(v, v);
                }
        }
        T query(int u, int v){
                T ans = T(); // identity element
                while(head[u] != head[v]){ // find LCA
                        if(depth[head[u]] > depth[head[v]]) swap(u, v);
                        ans = merge(ans, st.query(enter[head[v]], enter[v
                            ↪ ]));
                        v = par[head[v]];
                }
                if(depth[u] > depth[v]) swap(u, v); // make sure "u" is
                    ↪ LCA
                ans = merge(ans, st.query(enter[u], enter[v])); // enter[u
                    ↪ ]+1 for edge queriesk
                return ans;
        }
        void update(int u, T x){
                st.update(enter[u], x);
        }
};
```

2.4. Dinic Max Flow

```
// Dinic Max Flow O(V^2 E)
struct FlowEdge{
    int u,v;
    ll cap, flow = 0;
    FlowEdge(int u,int v,ll cap):u(u),v(v),cap(cap){}
};
struct Dinic{
    const ll flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector< vector<int> > gr;
    int s,t,n,m=0;
    vector<int> lvl,idx;
    Dinic (int n,int s,int t):n(n),s(s),t(t){
        gr.resize(n);
        lvl.resize(n);
        idx.resize(n);
    }
    void add_edge(int u,int v,ll cap){
        edges.emplace_back(u,v,cap);
        edges.emplace_back(v,u,0); //cap si bidireccional
        gr[u].push_back(m++);
        gr[v].push_back(m++);
    }
    bool run_bfs(){
        queue<int> bfs;
        bfs.push(s);
        fill(lvl.begin(),lvl.end(),-1);
        lvl[s] = 0;
        while (!bfs.empty()){
            int no = bfs.front();
            bfs.pop();
            for (int ne:gr[no]){
                if (lvl[edges[ne].v] == -1 && edges[ne].cap - edges[ne].
                    ↪ flow > 0){
                    lvl[edges[ne].v] = lvl[no] + 1;
                    bfs.push(edges[ne].v);
                }
            }
        }
        return lvl[t] != -1;
    }
    ll dfs(int u,ll cflow){
        if (cflow == 0 || u == t) return cflow;
        while (idx[u] < gr[u].size()){
            int edg = gr[u][idx[u]++];
            int v = edges[edg].v;
            if (lvl[u]+1 == lvl[v] && edges[edg].cap - edges[edg].flow >
                ↪ 0){
                ll rflow = dfs(v,min(cflow,edges[edg].cap - edges[edg].
                    ↪ flow));
                if (rflow){
                    edges[edg].flow += rflow;
                    edges[edg^1].flow -= rflow;
                    return rflow;
                }
            }
        }
        return 0;
    }
    ll flow(){
        ll f = 0;
        while (true){
            if (run_bfs()){
                fill(idx.begin(),idx.end(),0);
                ll cf;
                while (cf = dfs(s,flow_inf)) f += cf;
            } else break;
        }
        return f;
```

```
    }
};
```

2.5. Centroid Decomposition

```
struct cenDec{
    // cen_p[i]: el padre de i en el grafo de centroides
    // cen_d[i][j]: distancia entre j y su correspondiente centroide en el
        ↪ nivel i
    // cen_h[i] es la lista de hijos de i en el grafo de centroides
    vector<int> cen_p, path;
    vector< vector<int> > &gr, cen_d, cen_h;
    vector<bool> on;
    int n,cen_r;
    int pathm(int no,int p){
        path[no] = 1;
        for (int ne:gr[no]){
            if (on[ne] && ne != p){
                path[no] += pathm(ne,no);
            }
        }
        return path[no];
    }
    void dec(int c,int p,int l){
        pathm(c,-1);
        int cen = c, las = -1;
        while (cen != las){
            las = cen;
            for (int ne:gr[cen]){
                if (on[ne] && path[ne] > path[cen]/2){
                    cen = ne;
                    break;
                }
            }
            path[las] -= path[cen];
            path[cen] += path[las];
        }
        cen_p[cen] = p;
        if (p != -1) cen_h[p].push_back(cen);
        if (l > cen_d.size()) cen_d.push_back(vector<int>(n,-1));
        cen_d[l-1][cen] = 0;
        queue<int> bfs;
        bfs.push(cen);
        while (!bfs.empty()){
            int cno = bfs.front();
            bfs.pop();
            for (int ne:gr[cno]){
                if (on[ne] && cen_d[l-1][ne] == -1){
                    cen_d[l-1][ne] = cen_d[l-1][cno] + 1;
                    bfs.push(ne);
                }
            }
        }
        on[cen] = false;
        for (int ne:gr[cen]){
            if (on[ne]){
                dec(ne,cen,l+1);
            }
        }
    }
    cenDec(vector< vector<int> > &_gr):gr(_gr){
        n = gr.size();
        path.resize(n);
        cen_p.resize(n);
        cen_h.resize(n);
        on.assign(n,true);
        dec(0,-1,1);
        for (int i=0;i<n;i++){
            if (cen_p[i] == -1){
                cen_r = i;
                break;
            }
        }
    }
};
```

3. Matemáticas

3.1. Exponenciación binaria

```
const ll MOD; // MOD variable global
ll binpow(ll a, ll b){
  a %= MOD;
  ll ans=1;
  while(b > 0){
    if(b & 1)
      ans = ans * a % MOD;
    a = a * a % MOD;
    b >>= 1;
  }
  return ans;
}
```

3.2. Logaritmo discreto

```
// Si (a,b) coprimos, retorna mínimo x tal que a^x = b (mod m) o -1 si no
    ↪ existe.
// Asume 0^0 = 1
// fuente: cp-algorithms
int dlog(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;

    int an = 1;
    for (int i = 0; i < n; ++i)
```

```
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

    for (int p = 1, cur = 1; p <= n; ++p) {
        cur = (cur * 1ll * an) % m;
        if (vals.count(cur)) {
            int ans = n * p - vals[cur];
            return ans;
        }
    }
    return -1;
}
```

3.3. Lema de Burnside

El lema de Burnside sirve para problemas de conteo donde hay que contar solo una vez cada simetría.

Sea G un grupo finito actuando sobre un conjunto finito X. Para $g \in G$, denotamos como X^g los elementos de X que están fijos por g. El lema da una fórmula para el número de órbitas $|X/G|$:

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

Ejemplo: Contemos collares de n perlas, donde cada perla tiene m posibles colores. Dos collares son simétricos si son idénticos bajo alguna rotación. Así, cada órbita representa un collar, y el grupo G se compone de las n rotaciones posibles: $0, 1, \ldots, n-1$ pasos en algún sentido.

Entonces, contamos cuántos collares permanecen invariantes luego de aplicar una rotación de k pasos. Con cero pasos, todos los m^n collares permanecen fijos, y con 1 paso, los m collares donde todas las perlas tienen el mismo color permanecen fijos. En general, un total de $m^{\gcd(k,n)}$ collares están fijos con k pasos, porque bloques de tamaño $gcd(k, n)$ se reemplazan unos a los otros. Finalmente, por el Lema de Burnside, la cantidad de collares distintos es:

$$\frac{1}{n} \sum_{k=0}^{n-1} m^{\gcd(k,n)}.$$

3.4. Fórmula de inversión de Möbius

Sea $(P, \leq)$ un poset, la función de Möbius μ de P se define recursivamente para elementos de P como:

$$\mu(s,s) = 1$$
$$\mu(s,u) = -\sum_{s \leq t < u} \mu(s,t).$$

Si cada ideal principal de P es finito, $f\colon P \to R$ es una función, y existe una función g que cumple

$$g(y) = \sum_{x \leq y} f(x),$$

luego, se tiene

$$f(y) = \sum_{x \leq y} g(x)\mu(x,y).$$

4. C++

4.1. Custom set/map hash

```
struct custom_hash {
        static uint64_t splitmix64(uint64_t x) {
                // http://xorshift.di.unimi.it/splitmix64.c
                x += 0x9e3779b97f4a7c15;
                x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
                x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
                return x ^ (x >> 31);
        }
        size_t operator()(uint64_t x) const {
                static const uint64_t FIXED_RANDOM = chrono::steady_clock
                    ↪ ::now().time_since_epoch().count();
                return splitmix64(x + FIXED_RANDOM);
        }
};
unordered_map<ll, int, custom_hash> safe_map;
unordered_set<ll, custom_hash> safe_set;
```

4.2. Policy Based Order Statistics Tree

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
typedef tree<int,null_type,less<int>,rb_tree_tag,
                tree_order_statistics_node_update> indexed_set;
/*
usage:
indexed_set s;
```

```
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
cout << s.order_of_key(7) << "\n"; // 2
 */
```