

Contents

1 Data Structures

1.1	Bit 2d	1
1.2	DSU - Disjoint Set Union	1
1.3	DSU - Binary Tree	1
1.4	Segment Tree	1

2 Graphs

2.1	BFS 0-1	2
2.2	Dijkstra	2
2.3	Dinic - Flow/matchings	2
2.4	Floyd-Warshall	3
2.5	Hopcroft-Karp - Bipartite Matching	3
2.6	Hungarian	4
2.7	Kosaraju - SCCs	4
2.8	MST - Kruskal	4
2.9	Kuhn - Bipartite Matching	4
2.10	Min cost flow	4
2.11	MST - Prim	5

3 DP

3.1	Bin Packing	5
3.2	Broken Profile DP	5
3.3	Convex Hull Trick (CHT)	5
3.4	Edit Distance (Levenshtein)	6
3.5	Knapsack - 1D	6
3.6	Knapsack - 2D	6
3.7	LCS - Longest Common Subsequence	6
3.8	LiChao Tree	6
3.9	LIS - Longest Increasing Subsequence	6
3.10	SOSDP	6
3.11	Subset Sum	6

4 Trees

4.1	Sum of distances	7
4.2	Edge HLD	7
4.3	HLD - Heavy light decomposition	7
4.4	LCA - RMQ	7
4.5	LCA - binary lifting	8

5 Problemas clássicos

5.1	2sat	8
5.2	Next Greater Element	8

6 Strings

6.1	Hashing	10
-----	-------------------	----

6.2	KMP	11
6.3	Suffix Array	11
6.4	Z	11

7 Math

7.1	Combinatorics (Pascal's Triangle)	12
7.1.1	Combinatorial Analysis	12
7.2	Convolutions	12
7.2.1	AND convolution	12
7.2.2	GCD convolution	12
7.2.3	LCM convolution	12
7.2.4	OR convolution	12
7.2.5	XOR convolution	12
7.3	Extended Euclid	12
7.4	Factorization	12
7.5	FFT - Fast Fourier Transform	12
7.6	Inclusion-Exclusion Principle	12
7.7	Mint	12
7.8	Modular Inverse	12
7.9	Euler's Totient	12

8 Geometry

8.1	Convex hull - Graham Scan	15
8.2	Basic elements - geometry lib	15
8.2.1	Polygon Area	15
8.2.2	Point in polygon	15

1 Data Structures

1.1 Bit 2d

2D Sum BIT, update and sum. The problem must be 1-indexed.

Query/update time: $\mathcal{O}((\log n)^2)$

Construction time: $\mathcal{O}(n^2(\log n)^2)$

Space: $\mathcal{O}(n^2)$

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 #define MAX 1123
6
7 int bit[MAX][MAX], x, y;
8 void setbit(int i, int j, int delta) {
9     int j_;
10    while(i <= x) {
11        j_ = j;
12        while(j_ <= y) {
13            bit[i][j_] += delta;
14            j_ += j_ & -j_;
15        }
16        i += i & -i;
17    }
18 }
19 int getbit(int i, int j) {
20     int ans = 0;
21     int j_;
22     while(i) {
23         j_ = j;
24         while(j_) {
25             ans += bit[i][j];
26             j_ -= j_ & -j_;
27         }
28         i -= i & -i;
29     }
30     return ans;
31 }
32
33 int main(void) {
34     int p;
35     while (scanf("%d %d %d", &x, &y, &p), x || y || p)
36     {
37         for(int i = 0 ; i <= x ; i++)
38             for(int j = 0 ; j <= y ; j++)
39                 bit[i][j] = 0;
40         int q;
41         scanf("%d", &q);
42         while(q--) {
43             char c;
44             scanf(" %c", &c);
45             int n, xi, yi, zi, wi;
46             if(c == 'A') {
47                 scanf(" %d %d %d", &n, &xi, &yi);
48                 xi++; yi++;
49                 setbit(xi, yi, n);
50             }
51             else {
52                 scanf(" %d %d %d %d", &xi, &yi, &zi, &wi);
53                 xi++; yi++; zi++; wi++;
54                 if(xi > zi) swap(xi, zi);
55                 if(yi > wi) swap(yi, wi);
56                 ll ans = getbit(zi, wi) - getbit(zi, yi
57                     - 1)
58                     - getbit(xi - 1, wi) + getbit(xi - 1,
59                     yi - 1);
60                 printf("%lld\n", ans * (ll) p);
61             }
62         }
63     }
64     return 0;
65 }
```

1.2 DSU - Disjoint Set Union

Query/update time: $\mathcal{O}(1)$

Construction time: $\mathcal{O}(n)$

Space: $\mathcal{O}(n)$

```

1 vi _p, _rank;
2
3 int _find(int u) { return _p[u] == u ? u : _p[u] = _find
(_p[u]); }
```

```

4 void _union(int u, int v){
5     u = _find(u);
6     v = _find(v);
7     if(_rank[u] < _rank[v]){
8         _p[u] = v;
9     }
10    else{
11        _p[v] = u;
12        if(_rank[u] == _rank[v]) _rank[u]++;
13    }
14 }
15 void _make(int u){
16     _rank = vi(u, 0);
17     _p = vi(u);
18     for(int i = 0; i < u; i++) _p[i] = i;
19 }

```

1.3 DSU - Binary Tree

Specific code to find maximum path sums between pairs of vertices. Uses Kruskal-style MST. Query/update time: possibly $\mathcal{O}(n)$ Construction time: $\mathcal{O}(n)$ Space: $\mathcal{O}(n)$

```

1 vi d;
2 vi_ii e;
3 vi ans;
4
5 int merged;
6 vi _p, _leaf, _wei;
7 vvi adj;
8 int _find(int u) { return _p[u] == u ? u: _p[u] = _find(_p[u]); }
9 void _union(int u, int v, int w){
10    u = _find(u);
11    v = _find(v);
12    int merge_ind = merged+n;
13    _p[u] = merge_ind;
14    _p[v] = merge_ind;
15    _leaf[merge_ind] = _leaf[u] + _leaf[v];
16    _wei[merge_ind] = max(_wei[u], _wei[v]);
17    adj[u].push_back(merge_ind);
18    adj[merge_ind].push_back(u);
19    adj[v].push_back(merge_ind);
20    adj[merge_ind].push_back(v);
21    merged++;
22 }
23 void make(){
24     _p = vi(2*n);
25     for(int i = 0; i < 2*n; i++) _p[i] = i;
26     _leaf = vi(2*n, 1);
27     _wei = vi(2*n);
28     for(int i = 0; i < n; i++) _wei[i] = d[i];
29     merged = 0;
30     adj = vvi(2*n);
31 }
32
33 void dfs(int u, int p){
34     for(auto &v: adj[u]){
35         if(v == p) continue;
36         ans[v] = ans[u] + (_leaf[u] - _leaf[v])*_wei[u];
37         dfs(v, u);
38     }
39 }

```

1.4 Segment Tree

Segment tree with lazy propagation. Here the interval convention is $[l, r]$, with 0-based indexing. The example solves Kadane (max subarray sum) with point/range updates.

Query/update time: $\mathcal{O}(\log n)$

Construction time: $\mathcal{O}(n)$

Space: $\mathcal{O}(n)$

```

1 struct segtree {
2     int size;
3     vector<node> nodes;
4     vector<bool> hasLazy;
5     vector<int> lazy;
6
7     struct node {
8         int seg, pre, suf, sum;
9     };
10
11    node NEUTRAL = {0,0,0,0};
12
13    void debug(){
14        if (nodes.empty() || size == 0) {
15            cout << "[Empty Tree]\n"; return;
16        }
17
18        string indent = "...";
19        function<void(int, int, int, string)> print_dfs
20        ;
21
22        print_dfs = [&](int x, int lx, int rx, string
23                         prefix) {
24            cout << prefix << " [" << lx << ", " << rx
25                         << ")";
26
27            // debug node
28            node a = nodes[x];
29            cout << "{ ";
30            cout << "seg: " << a.seg << ", ";
31            cout << "pre: " << a.pre << ", ";
32            cout << "suf: " << a.suf << ", ";
33            cout << "sum: " << a.sum << ", ";
34            cout << "hasLazy: " << hasLazy[x] << ", ";
35            cout << "lazy: " << lazy[x] << ", ";
36            cout << "}";
37            cout << endl;
38
39            if (rx-lx <= 1) return;
40
41            int mx = (lx+rx)/2;
42            print_dfs(2*x+1, lx, mx, prefix + indent);
43            print_dfs(2*x+2, mx, rx, prefix + indent);
44        };
45
46        print_dfs(0, 0, size, "");
47
48        node single(int v){
49            return {v,v,v,v};
50        }
51
52        node merge(node a, node b){
53            return {
54                max(max(a.seg, b.seg), a.suf + b.pre),
55                max(a.pre, a.sum + b.pre),
56                max(b.suf, b.sum + a.suf),
57                a.sum+b.sum
58            };
59        }
60
61    void init (vi &a){
62        int n = a.size();
63        size = 1;
64        while (size < n) size *= 2;
65        nodes.assign(2*size-1, NEUTRAL);
66        hasLazy.assign(2*size-1, false);
67        lazy.assign(2*size-1, 0);
68        build(0,0,size,a);
69
70    void build(int x, int lx, int rx, vi &a){
71        if (rx-lx == 1){
72            if (lx < a.size()) nodes[x] = single(a[lx])
73            ;
74            return;
75        }
76        int mx = (lx+rx)/2;
77        build(2*x+1, lx, mx, a);
78        build(2*x+2, mx, rx, a);
79        nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
80
81    void set(int i, int v, int x, int lx, int rx){
82        if (rx-lx == 1){
83            nodes[x] = single(v);
84            return;
85        }
86        int mx = (lx+rx)/2;
87        if (i < mx) set(i, v, 2*x+1, lx, mx);
88        else set(i, v, 2*x+2, mx, rx);
89        nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
90
91    void set(int i, int v){
92        set(i, v, 0, 0, size);
93    }
94
95    void rangeUpdate(int l, int r, int v){
96        rangeUpdate(l,r,v,0,0,size);
97    }
98
99    void rangeUpdate(int l, int r, int v, int x, int lx
100 , int rx){
101        unlazy(x, lx, rx);
102        if (rx-lx < 1 || rx <= l || lx >= r) return;
103        if (l <= lx && rx <= r) propagate(x, lx,
104                                         rx, v);
105        int mx = (lx+rx)/2;
106        rangeUpdate(l,r,v,2*x+1, lx, mx);
107        rangeUpdate(l,r,v,2*x+2, mx, rx);
108        nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
109
110    node query(int l, int r){
111        return query(l,r,0,0,size);
112    }
113
114    node query(int l, int r, int x, int lx, int rx){
115        unlazy(x, lx, rx);
116        if (rx-lx < 1 || rx <= l || lx >= r) return
117            NEUTRAL;
118        if (l <= lx && rx <= r) return nodes[x];
119        int mx = (lx+rx)/2;
120        node left = query(l,r,2*x+1, lx, mx);
121        node right = query(l,r,2*x+2, mx, rx);
122    }

```

```

120     return merge(left,right);
121 }
122 void unlazy(int x, int lx, int rx){
123     if (hasLazy[x]){
124         propagate(x,lx,rx,lazy[x]);
125         hasLazy[x] = false;
126     }
127 }
128
129 void propagate(int x, int lx, int rx, int v){
130     nodes[x].sum = (rx-lx)*v;
131     nodes[x].seg = max((rx-lx)*v,0ll);
132     nodes[x].pre = max((rx-lx)*v,0ll);
133     nodes[x].suf = max((rx-lx)*v,0ll);
134     if (rx-lx > 1){
135         lazy[2*x+1] = v;
136         lazy[2*x+2] = v;
137         hasLazy[2*x+1] = true;
138         hasLazy[2*x+2] = true;
139     }
140 }
141
142 };

```

2 Graphs

2.1 BFS 0-1

Time: $\mathcal{O}(n + m)$

```

1 vi bfs01(int s){
2     vi d(n, INF);
3     d[s] = 0;
4     deque<int> q;
5     q.push_front(s);
6     while(!q.empty()){
7         int u = q.front(); q.pop_front();
8         for (auto [w,v] : adj[u]){
9             if (d[u]+w < d[v]){
10                 d[v] = d[u] + w;
11                 if (w == 1) q.push_back(v);
12                 else q.push_front(v);
13             }
14         }
15     }
16     return d;
17 }

```

2.2 Dijkstra

Time: $\mathcal{O}(m \log n)$

```

1 void dijkstra(int s){
2     int d, u, v;
3     dist = vi(n, INF);
4     dist[s] = 0;
5     priority_queue<ii, vii, greater<ii>> pq;
6     pq.emplace(0,s);
7     while(!pq.empty()){
8         auto [d,u] = pq.top(); pq.pop();
9         if (d > dist[u]) continue;
10        for (auto &[w,v] : adj[u]){
11            if (dist[v] > dist[u] + w){

```

```

13                 dist[v] = dist[u] + w;
14                 pq.emplace(dist[v], v);
15             }
16         }
17     }
18 }

```

2.3 Dinic - Flow/matchings

- General Network: $\mathcal{O}(VE \log U)$.
- Unit Capacity Network: $\mathcal{O}(\min(V^{2/3}, E^{1/2}) \cdot E)$. Often considered $\mathcal{O}(E\sqrt{V})$.
- Bipartite Matching: $\mathcal{O}(\min(V^{2/3}, E^{1/2}) \cdot E)$. Often considered $\mathcal{O}(E\sqrt{V})$.

```

1 struct Dinic {
2     struct Edge {
3         int u, v;
4         ll cap, flow = 0;
5         Edge(int u, int v, ll cap) : u(u), v(v), cap(cap)
6             cap) {}
7     };
8     const ll flow_inf = 1e18;
9     vector<Edge> edges;
10    vvi adj;
11    int n, m = 0;
12    int s, t;
13    vi level, ptr;
14    queue<int> q;
15
16    Dinic(int n): n(n) {
17        adj.resize(n);
18        level.resize(n);
19        ptr.resize(n);
20    }
21
22    void add_edge(int u, int v, ll cap) {
23        edges.emplace_back(u,v,cap);
24        edges.emplace_back(v,u,0);
25        adj[u].push_back(m++);
26        adj[v].push_back(m++);
27    }
28
29    bool bfs(ll delta){
30        queue<int> q;
31        q.push(s);
32        while(!q.empty()){
33            int u = q.front(); q.pop();
34            for (int id : adj[u]){
35                auto &e = edges[id];
36                if (e.cap - e.flow < delta) continue;
37                if (level[e.v] != -1) continue;
38                level[e.v] = level[u]+1;
39                q.push(e.v);
40            }
41            if (level[t] != -1) return true;
42        }
43    }
44
45    ll dfs(int u, ll pushed) {
46        if (pushed == 0) return 0;
47        if (u == t) return pushed;
48
49        for (int &cid = ptr[u]; cid < (int)adj[u].size
50            (); cid++){
51            int id = adj[u][cid];
52            auto &e = edges[id];
53            if (level[u]+1 != level[e.v]) continue;
54            ll tr = dfs(e.v,min(pushed, e.cap - e.flow)
55                );
56            if (tr == 0) continue;
57            e.flow += tr;
58            edges[id^1].flow -= tr;
59        }
60        return 0;
61    }
62
63    ll maxflow(int s, int t){
64        this->s = s; this->t = t;
65        ll max_c = 0;
66        for (auto &e : edges) max_c = max(max_c, e.cap)
67            ;
68
69        ll delta = 1;
70        while(delta <= max_c) delta <= 1;
71        delta >= 1;
72
73        ll f = 0;
74        for (;delta > 0; delta >= 1){
75            while(true){
76                fill(level.begin(), level.end(), -1);
77                level[s] = 0;
78                if (!bfs(delta)) break;
79                fill(ptr.begin(), ptr.end(), 0);
80                while(ll pushed = dfs(s,flow_inf)) f += pushed;
81            }
82        }
83        // call constructor with (n1+n2+2) beforehand (dont
84        // add edges manually)
85        // assumes pairs are 1-indexed
86        vii maxmatchings(int n1, int n2, const vii& pairs){
87            for (int i = 1; i <= n1; i++)
88                add_edge(0,i,1);
89
90            for (int i = 1; i <= n2; i++)
91                add_edge(i+n1,n-1,1);
92
93            for (auto &[u,v] : pairs)
94                add_edge(u,v+n1,1);
95
96            maxflow(0,n-1);
97
98            vii matchings;
99            for (auto &e : edges){
100                if (e.u >= 1 && e.u <= n1 && e.flow == 1 &&
101                    e.v > n1){
102                    matchings.emplace_back(e.u,e.v-n1);
103                }
104            }
105            return matchings;
106        }
107
108        vii mincut(int s, int t){
109            maxflow(s,t);
110            queue<int> q; q.push(s);
111            vector<bool> reachable(n);
112            reachable[s] = true;

```

```

112     while(!q.empty()){
113         int u = q.front(); q.pop();
114         for (auto &id : adj[u]){
115             int v = edges[id].v;
116             if (edges[id].cap - edges[id].flow > 0
117                 && !reachable[v]) {
118                 reachable[v] = true;
119                 q.push(v);
120             }
121         }
122     }
123     vvi minCutEdges;
124
125     for (int i = 0; i < m; i += 2) {
126         const Edge& edge = edges[i];
127         if (reachable[edge.u] && !reachable[edge.v])
128             ]]) {
129             minCutEdges.emplace_back(edge.u, edge.v);
130         }
131     }
132     return minCutEdges;
133 }

```

2.4 Floyd-Warshall

Time: $\mathcal{O}(n^3)$

```

1 vvi dist(MAX, vi(MAX, INF));
2
3 void floyd_marshall(){
4     for (int k = 1; k <= n; k++)
5         for (int i = 1; i <= n; i++)
6             for (int j = 1; j <= n; j++)
7                 dist[i][j] = min(dist[i][j], dist[i][k]+dist[k]
8                     ][j]);
9 }

```

2.5 Hopcroft-Karp - Bipartite Matching

Bipartite matching such as Kuhn but faster. BFS until first layer missing match, DFS for the BFS graph to find pairings. Time: $\mathcal{O}(E\sqrt{V})W$

```

1 int n, m, k;
2 vvi adj;
3 vi p, dist; /*p is in matching for [0, n[ and parent
4     for [n, n+m[*/
5
6 int bfs(){
7     queue<int> q;
8     dist = vi(n+m, inf);
9     for(int i = 0; i < n; i++){
10        if(p[i] == -1) q.push(i), dist[i] = 0;
11    }
12    int min_dist_match = inf;
13    while(!q.empty()){
14        int u = q.front(); q.pop();
15        if(dist[u] > min_dist_match) continue;
16        for(auto v: adj[u]){
17
18            if(p[v] == -1) min_dist_match = dist[u];
19            else if(dist[p[v]] == inf){
20                dist[p[v]] = dist[u] + 1;
21                q.push(p[v]);
22            }
23        }
24    }
25
26    int dfs(int u){
27        for(auto v: adj[u]){
28            if(p[v] == -1 || (dist[u]+1 == dist[p[v]] &&
29                dfs(p[v]))){
30                p[v] = u;
31                p[u] = 1;
32                return true;
33            }
34        }
35        dist[u] = inf;
36        return false;
37    }
38
39    int hopkarp(){
40        p = vi(n+m, -1);
41        int matchings = 0;
42        while(bfs()){
43            for(int i = 0; i < n; i++){
44                if(p[i] == -1 && dfs(i)) matchings++;
45            }
46        }
47        return matchings;
48    }
49
50    void create(){
51        adj = vvi(n+m);
52        for(int i = 0; i < k; i++){
53            int u, v;
54            cin >> u >> v; u--; v--;
55            v += n;
56            adj[u].push_back(v);
57        }
58    }

```

2.6 Hungarian

Solves minimum cost assignment for n workers and m jobs.

Time: $\mathcal{O}((n+m)^3)$

```

1 // cost should be (cost[worker][job])
2 pair<int, vvi> hungarian(int n, int m, const vvi &cost)
3 {
4     if (n == 0) return {0, {}};
5     int N = max(n, m);
6
7     vi u(N+1), v(N+1), p(N+1), way(N+1);
8
9     const int INF = 1e9;
10    for (int i = 1; i <= n; ++i) {
11        p[0] = i;
12        int j0 = 0;
13        vi minv(N + 1, INF);
14        vector<bool> used(N + 1, false);
15
16        do {
17            used[j0] = true;
18
19            if(p[v] == -1) min_dist_match = dist[u];
20            else if(dist[p[v]] == inf){
21                dist[p[v]] = dist[u] + 1;
22                q.push(p[v]);
23            }
24        }
25
26    int i0 = p[j0], delta = INF, j1;
27
28    for (int j = 1; j <= N; ++j) {
29        if (!used[j]) {
30            int cur = cost[i0 - 1][j - 1] - u[i0] - v[j];
31            if (cur < minv[j]) {
32                minv[j] = cur;
33                way[j] = j0;
34            }
35            if (minv[j] < delta) {
36                delta = minv[j];
37                j1 = j;
38            }
39        }
40
41        for (int j = 0; j <= N; ++j) {
42            if (used[j]) {
43                u[p[j]] += delta;
44                v[j] -= delta;
45            } else {
46                minv[j] -= delta;
47            }
48        }
49        j0 = j1;
50    }
51
52    int total_cost = 0;
53    for (int j = 1; j <= m; ++j) {
54        if (p[j] != 0) {
55            total_cost += cost[p[j] - 1][j - 1];
56        }
57    }
58
59 // {worker, job}[] 0-indexed
60 vii matchings;
61 for (int j = 1; j <= m; ++j) {
62     if (p[j] != 0) {
63         matchings.push_back({p[j] - 1, j - 1});
64     }
65 }
66
67 return {total_cost, matchings};

```

2.7 Kosaraju - SCCs

Computes the strongly connected components of a graph. Also computes the reverse topological order (if it exists). Time: $\mathcal{O}(n + m)$

```

1 void dfs1(int u){
2     vis[u] = 1;
3     for (auto v : adj[u]){
4         if (!vis[v]) dfs1(v);
5     }
6
7 ts.push_back(u);
8 }

```

```

10 void dfs2(int u, int c){
11     scc[u] = c;
12     for (auto v : adjT[u])
13         if (!scc[v]) dfs2(v, c);
14 }
15
16 // usage
17 for (int i = 0; i < n; i++)
18     if (!vis[i]) dfs1(i);
19
20 reverse(ts.begin(), ts.end());
21
22 int c = 1;
23 for (auto u : ts)
24     if (!scc[u]) dfs2(u, c++);

```

2.8 MST - Kruskal

Time: $\mathcal{O}(m \log m)$

```

1 vector<pair<int,ii>> edges;
2 int kruskal(){
3     int cost = 0;
4     for (int i = 0; i < n; i++) {_p[i]=i; _rank[i]=0;}
5     sort(edges.begin(), edges.end());
6     for (auto &[w,uv] : edges){
7         auto [u,v] = uv;
8         if (_find(u) != _find(v)){
9             cost += w;
10            _union(u, v);
11        }
12    }

```

2.9 Kuhn - Bipartite Matching

Bipartite matching. Time: $\mathcal{O}(VE)$

```

1 int matchings;
2 vi p, vis;
3 vii match;
4
5 int dfs(int u){
6     if(vis[u]) return 0;
7     vis[u] = 1;
8     for(auto v: adj[u]){
9         if(p[v] == -1 || dfs(p[v])){
10            p[v] = u;
11            return 1;
12        }
13    }
14    return 0;
15 }
16
17 void kuhn(){
18     matchings = 0;
19     p = vi(n+m, -1);
20     for(int i = 0; i < n; i++){
21         vis = vi(n, 0);
22         matchings += dfs(i);
23     }
24     for(int i = n; i < n+m; i++){
25         if(p[i] != -1) match.push_back(ii(p[i], i));
26     }
27 }

```

```

28 void create(){
29     adj = vvi(n+m);
30     for(int i = 0; i < k; i++){
31         int u, v;
32         cin >> u >> v; u--; v--;
33         adj[u].push_back(v+n);
34     }
35 }
36
47
48     cur = t;
49     while (cur != s) {
50         Edge &e = adj[pv[cur]][pe[cur]];
51         e.capacity -= f;
52         adj[e.to][e.rev].capacity += f;
53         cur = pv[cur];
54     }
55
56     return {total_cost, flow};
57 }
58
59
60 // needed only if negative costs exists
61 void spfa(int s, vll &dist, vi &pv, vi &pe) {
62     dist.assign(n, INF);
63     pv.assign(n, -1);
64     pe.assign(n, -1);
65     vector<bool> inq(n, false);
66     queue<int> q;
67
68     dist[s] = 0;
69     q.push(s);
70     inq[s] = true;
71
72     while (!q.empty()) {
73         int u = q.front(); q.pop();
74         inq[u] = false;
75         for (int i = 0; i < adj[u].size(); i++) {
76             Edge &e = adj[u][i];
77             int v = e.to;
78             if (e.capacity > 0 && dist[v] > dist[u]
79                 + e.cost) {
80                 dist[v] = dist[u] + e.cost;
81                 pv[v] = u;
82                 pe[v] = i;
83                 if (!inq[v]) {
84                     inq[v] = true;
85                     q.push(v);
86                 }
87             }
88         }
89     }
90
91     void dijkstra(int s, vll &h, vll &dist, vi &pv, vi
92     &pe) {
93         dist.assign(n, INF);
94         pv.assign(n, -1);
95         pe.assign(n, -1);
96         dist[s] = 0;
97
98         priority_queue<lli, vector<lli>, greater<lli>>
99         pq;
100        pq.emplace(0, s);
101
102        while (!pq.empty()) {
103            auto [d, u] = pq.top(); pq.pop();
104            if (d > dist[u]) continue;
105
106            for (int i = 0; i < adj[u].size(); i++) {
107                Edge &e = adj[u][i];
108                if (e.capacity <= 0) continue;
109                int v = e.to;
110
111                int reduced_cost = e.cost + h[u] - h[v];
112                if (dist[u] != INF && dist[v] > dist[u]
113                    + reduced_cost) {
114                    dist[v] = dist[u] + reduced_cost;
115                    pv[v] = u;
116                }
117            }
118        }
119    }

```

```

113         pe[v] = i;
114     }
115     pq.push({dist[v], v});
116   }
117 }
118 }
119 };
120
121 // usage
122 int nodes = 302; // amount of nodes in the network
123 MinCostFlow mcf(nodes);
124
125 for (int i = 0; i < 150; i++){
126     mcf.add_edge(0, i+1, 1, 0); // source to node
127     mcf.add_edge(i+151, nodes-1, 1, 0); // node to sink
128 }
129
130 for (int i = 0; i < n; i++){
131     int a, b, c; cin >> a >> b >> c;
132     mcf.add_edge(a, b+150, 1, -c); // edges in between
133     // (-c to maximize the cost)
134 }
135 // final max cost is -cost
136 auto [cost, flow] = mcf.min_cost_flow(0, nodes-1, 150);

```

2.11 MST - Prim

Time: $\mathcal{O}(m \log n)$

```

1 vvii adj, mst;
2 vi taken;
3
4 int prim(){
5     priority_queue<iii, vector<iii>, greater<iii>> pq;
6     taken[0] = 1;
7     for (auto [w,v] : adj[0]){
8         if (!taken[v]) pq.push({w, {0,v}});
9     }
10
11    int cost = 0;
12    while (!pq.empty()){
13        auto [w,pw] = pq.top(); pq.pop();
14        auto [p,u] = pw;
15        if (!taken[u]) {
16            cost += w;
17            mst[p].emplace_back(w,u);
18            mst[u].emplace_back(w,p);
19            taken[u] = 1;
20            for (auto [w,v] : adj[u]){
21                if (!taken[v]) {
22                    pq.push({w,{u,v}});
23                }
24            }
25        }
26    }
27    return cost;
28 }

```

3 DP

3.1 Bin Packing

Time: $\mathcal{O}(n \cdot 2^n)$ Space: $\mathcal{O}(2^n)$

```

1 vi w(n);
2
3 vector<ii> dp(1<<n, ii(INF,0));
4 // dp[i] = for the subset i(bitmask) (A,B) is the pair
5 // where
5 // A - the min number of knapsacks to store this subset
5 // B - the min size of a used knapsack
6
7 dp[0] = ii(0,INF);
8 for (int subset = 1; subset < (1<<n); subset++){
9     for (int item = 0; item < n; item++){
10         if (!(subset >> item)&1)) continue;
11         int prevsubset = subset - (1<<item);
12         ii prev = dp[prevsubset];
13
14         if (prev.second + w[item] <= x) {
15             // can fill the knapsack, fill it
16             dp[subset] = min(dp[subset], ii(prev.first,
17                               prev.second+w[item]));
18         } else {
19             // cant fill the knapsack, create a new one
20             dp[subset] = min(dp[subset], ii(prev.first
21                                         +1, w[item]));
22         }
23     }
24 }
25 cout << dp[(1<<n)-1].first << endl;

```

3.2 Broken Profile DP

Solves the problem of counting how many ways to fill an $n \times m$ grid using 1×2 tiles. This technique can be used whenever the state dependence is only on the previous state (column). Time: $\mathcal{O}(mn2^n)$ Space: $\mathcal{O}(mn2^n)$

```

1
2 int dp[1002][12][1024];
3 dp[0][0][0] = 1;
4
5 for (int i = 0; i < m; i++){
6     for (int j = 0; j < n; j++){
7         for (int mask = 0; mask < (1<<n); mask++){
8             if (mask & (1<<j)){
9                 int nxt_mask = mask - (1<<j);
10                dp[i][j+1][nxt_mask] += dp[i][j][mask];
11                dp[i][j+1][nxt_mask] %= M;
12            } else {
13                int q = mask + (1 << j);
14                dp[i][j+1][q] += dp[i][j][mask];
15                dp[i][j+1][q] %= M;
16                if (j < n-1 && (mask & (1<<(j+1)))==0){
17                    q = mask + (1 << (j+1));
18                    dp[i][j+1][q] += dp[i][j][mask];
19                    dp[i][j+1][q] %= M;
20                }
21            }
22        }
23    }
24
25    for (int p = 0; p < (1<<n); p++){
26        dp[i+1][0][p] = dp[i][n][p];
27    }
28 }

```

3.3 Convex Hull Trick (CHT)

• Recurrence form:

TODO formulas

- **Slope monotonicity:** If coefficients a_j (slopes) are inserted in strictly decreasing (or increasing) order as j grows, and
- **Query monotonicity:** Values x_i for query come in non-decreasing (min) or increasing (max) order consistent with slope order,
- **Complexity:**
 - Insertion + amortized query in $\mathcal{O}(1)$ per operation (pointer walk) under monotonicity.
 - Non-monotonic case, generic CHT via binary search: $\mathcal{O}(\log n)$ per query.
 - General alternative: Li Chao Tree for insertions/queries in arbitrary order, $\mathcal{O}(\log M)$ per operation (where M is the domain of x).

• Constraints:

- If it cannot be written in linear form, CHT does not apply.
- If there is no monotonicity of slopes or queries, consider Li Chao Tree or CHT variant with binary search.

The example below solves the dp where the recurrence is:

TODO formulas

```

1 struct CHT {
2     struct Line { // y = mx + c
3         int m, c;
4
5         Line(int m, int c) : m(m), c(c) {}
6
7         int val(int x){
8             return m*x + c;
9         }
10
11         int floorDiv(int num, int den) {
12             if (den < 0) num = -num, den = -den;
13             if (num >= 0) return num / den;
14             else return -((-num + den - 1) / den );
15         }
16
17         int ceilDiv(int num, int den) {
18             if (den < 0) num = -num, den = -den;
19             if (num >= 0) return (num + den - 1) / den;
20             else return -((-num) / den );
21         }
22
23     }
24
25     int intersect(Line l){
26
27
28 }

```

```

25     //  $m_1x + c_1 = m_2x + c_2$ 
26     //  $x = (c_2 - c_1)/(m_1 - m_2)$ 
27     // if slopes are increasing, use floor div
28     return ceilDiv(l.c - c, m - l.m);
29 }
30 }
31 deque<pair<Line, int>> dq;
32 void insert(int m, int c){
33     Line newLine(m, c);
34
35     if (!dq.empty() && newLine.m == dq.back().first
36         .m) {
37         // If slopes increasing, change to <=
38         if (newLine.c >= dq.back().first.c) return;
39         else dq.pop_back();
40     }
41
42     // if slopes increasing, change to <=
43     while (dq.size() > 1 && dq.back().second >= dq.
44         back().first.intersect(newLine)){
45         dq.pop_back();
46     }
47
48     if (dq.empty()){
49         // assuming queries are positive numbers,
50         // may change to -INF or +INF if needed
51         dq.emplace_back(newLine, 0);
52     }
53
54     dq.emplace_back(newLine, dq.back().first.
55         intersect(newLine));
56 }
57
58 // dont use query and queryNonMonotonicValues in
59 // the same problem
60 int query(int x){
61     while (dq.size() > 1){
62         // if slopes increasing, change to >=
63         if (dq[1].second <= x) dq.pop_front();
64         else break;
65     }
66     return dq[0].first.val(x);
67 }
68
69 int queryNonMonotonicValues(int x){
70     int l=0, r=dq.size()-1, ans=0;
71     while (l <= r) {
72         int mid = (l+r)>>1;
73         if (dq[mid].second <= x) {
74             ans = mid;
75             l = mid + 1;
76         } else {
77             r = mid - 1;
78         }
79     }
80     return dq[ans].first.val(x);
81 }
82
83 void solve(){
84     int n, c; cin >> n >> c;
85     vi h(n);
86     for (auto &x : h) cin >> x;
87
88 }
89

```

```

90     vi dp(n);
91     dp[0] = 0;
92     CHT cht;
93     cht.insert(-2*h[0], h[0]*h[0]);
94     for (int i = 1; i < n; i++){
95         dp[i] = cht.query(h[i]) + c + h[i]*h[i];
96         cht.insert(-2*h[i], h[i]*h[i] + dp[i]);
97     }
98     cout << dp[n-1] << endl;
99 }

```

3.4 Edit Distance (Levenshtein)

Very similar to LCS, in the sense that it considers prefixes already computed. Time: $\mathcal{O}(mn)$ Space: $\mathcal{O}(mn)$

```

1  vvi dp(n+1, vi(m+1));
2  for (int i = 0; i <= n; i++) dp[i][0] = i;
3  for (int i = 0; i <= m; i++) dp[0][i] = i;
4  for (int i = 1; i <= n; i++){
5      for (int j = 1; j <= m; j++){
6          dp[i][j] = min(
7              min(dp[i][j-1]+1, dp[i-1][j]+1),
8              dp[i-1][j-1]+(s[i-1] != t[j-1])
9          );
10     }
11 }

```

3.5 Knapsack - 1D

The spirit here is the same as the 2D version, but here it iterates on the knapsack capacity backwards, to ensure that the value of $dp[j-w[i]]$ is not considering the item i . Time: $\mathcal{O}(nW)$ Space: $\mathcal{O}(W)$

```

1  vi dp(W+1);
2  for (int i = 0; i < n; i++){
3      for (int j = W; j >= w[i]; j--){
4          dp[j] = max(dp[j], v[i] + dp[j-w[i]]);
5      }
6  }

```

3.6 Knapsack - 2D

Time: $\mathcal{O}(nW)$ Space: $\mathcal{O}(nW)$

```

1  vvi dp(n+1, vi(W+1));
2  for (int c = 1; c <= W; c++){
3      for (int i = 1; i <= n; i++){
4          dp[i][c] = dp[i-1][w];
5          if (c-w[i-1] >= 0) {
6              dp[i][c] = max(dp[i][c], dp[i-1][c-w[i-1]]
7                  + v[i-1]);
8          }
9      }
}

```

3.7 LCS - Longest Common Subsequence

Subsequence generation included here. Time: $\mathcal{O}(mn)$ Space: $\mathcal{O}(mn)$

```

1  vvi dp(n+1, vi(m+1));
2  vvi p(n+1, vii(m+1));
3
4  for (int i = 1; i <= n; i++){
5      for (int j = 1; j <= m; j++){
6          if (a[i-1] == b[j-1]){
7              dp[i][j] = dp[i-1][j-1]+1;
8              p[i][j] = {i-1, j-1};
9          } else if (dp[i][j-1] > dp[i-1][j]){
10              dp[i][j] = dp[i][j-1];
11              p[i][j] = {i, j-1};
12          } else {
13              dp[i][j] = dp[i-1][j];
14              p[i][j] = {i-1, j};
15          }
16      }
17  }
18
19  ii pos = ii(n,m);
20  stack<int> st;
21  while(pos != ii(0,0)){
22      auto [i,j] = pos;
23      if (p[i][j] == ii(i-1, j-1)) st.push(a[i-1]);
24      pos = p[i][j];
25  }
26  cout << st.size() << endl;
27  while (!st.empty()){
28      cout << st.top() << ' ';
29      st.pop();
30  }
31  cout << endl;

```

3.8 LiChao Tree

Generalization of CHT for linear functions that do not need to be sorted. Inspired by segtree. Queries and insertions are all $\mathcal{O}(\log M)$. Where M is the size of the query interval the tree receives.

```

1 // Li Chao tree for minimum (or maximum) over domain [L
, R].
2 // T should support +, *, comparisons.
3 // For integer x use eps = 0 and discrete mid+1
// splitting;
4 // For floating use eps > 0 and continuous splitting
// without +1.
5 template<typename T>
6 struct lichao_tree {
7     // if max lichao, change to ::min()
8     static const T identity = numeric_limits<T>::max();
9
10    struct Line {
11        T m, c;
12
13        Line() {
14            m = 0;
15            c = identity;
16        }
17
18        Line(T m, T c) : m(m), c(c) {}
19
}

```

```

20     T val(T x) { return m * x + c; }
21 };
22
23 struct Node {
24     Line line;
25     Node *lc, *rc;
26
27     Node() : lc(0), rc(0) {}
28 };
29
30 T L, R, eps;
31 deque<Node> buffer;
32 Node* root;
33
34 Node* new_node() {
35     buffer.emplace_back();
36     return &buffer.back();
37 }
38
39 lichao_tree() {}
40
41 lichao_tree(T _L, T _R, T _eps) {
42     init(_L, _R, _eps);
43 }
44
45 void clear() {
46     buffer.clear();
47     root = nullptr;
48 }
49
50 void init(T _L, T _R, T _eps) {
51     clear();
52     L = _L;
53     R = _R;
54     eps = _eps;
55
56     root = new_node();
57 }
58
59 void insert(Node* &cur, T l, T r, Line line) {
60     if (!cur) {
61         cur = new_node();
62         cur->line = line;
63         return;
64     }
65
66     T mid = l + (r - 1) / 2;
67     if (r - 1 <= eps) return;
68
69     // if max lichao, change to >
70     if (line.val(mid) < cur->line.val(mid))
71         swap(line, cur->line);
72
73     // if max lichao, change to >
74     if (line.val(l) < cur->line.val(l)) insert(cur
75         ->lc, l, mid, line);
76     else insert(cur->rc, mid + 1, r, line);
77 }
78
79 T query(Node* &cur, T l, T r, T x) {
80     if (!cur) return identity;
81
82     T mid = l + (r - 1) / 2;
83     T res = cur->line.val(x);
84     if (r - 1 <= eps) return res;
85
86     // if max lichao, change min to max
87     if (x <= mid) return min(res, query(cur->lc, l,
88         mid, x));
89     else return min(res, query(cur->rc, mid + 1, r,
90         x));
91
92     void insert(T m, T c) { insert(root, L, R, Line(m,
93         c)); }
94
95     T query(T x) { return query(root, L, R, x); }
96 }
97

```

3.9 LIS - Longest Increasing Subsequence

Time: $\mathcal{O}(n \log n)$

```

1 int lis(vi &a){
2     int n = a.size();
3     vi len(n+1, INF);
4     len[0] = -INF;
5     for (int i = 0; i < n; i++){
6         int l = upper_bound(len.begin(), len.end(), a[i]
7             ) - len.begin();
8         if (len[l-1] < a[i] && a[i] < len[l]) len[l] = a[i];
9     }
10    int ans = 0;
11    for (int i = 0; i <= n; i++){
12        if (len[i] < INF) ans = i;
13    }
14    return ans;
15 }

```

3.10 SOSDP

```

1 int k; // amount of bits
2 vi a(1<<k);
3 // sosdp
4 for (int bit = 0; bit < k; bit++){
5     for (int mask = 0; mask < (1<<k); mask++){
6         if ((1<<bit) & mask) {
7             a[mask] += a[mask ^ (1<<bit)];
8         }
9     }
10 }
11 // do stuff (such as multiplication for OR convolution)
12 // sosdp inverse
13 for (int bit = 0; bit < k; bit++){
14     for (int mask = 0; mask < (1<<k); mask++){
15         if ((1<<bit) & mask) {
16             a[mask] -= a[mask ^ (1<<bit)];
17         }
18     }
19 }
20 }
21 }

```

3.11 Subset Sum

Almost identical to Knapsack, this code contains the subset reconstruction. Time: $\mathcal{O}(nS)$ Space: $\mathcal{O}(nS)$

```

1 vvi dp(n+1, vi(sum+1));
2 vvii p(n+1, vii(sum+1));
3

```

```

4 dp[0][0] = 1;
5
6 for (int i = 1; i <= n; i++){
7     for (int s = 1; s <= sum; s++){
8         if (s-a[i-1] >= 0 && dp[i-1][s-a[i-1]]){
9             // sum is possible taking item i
10            p[i][s] = {i-1, s-a[i-1]};
11            dp[i][s] = 1;
12        } else if (dp[i-1][s]){
13            // sum not possible taking item i
14            // but still possible with other items (<i)
15            p[i][s] = {i-1, s};
16            dp[i][s] = 1;
17        }
18    }
19 }
20
21 if (!dp[n][target]){
22     cout << -1 << endl;
23     return;
24 }
25
26 vi subset;
27 ii pos = {n, target};
28 while(pos != ii(0, 0)){
29     auto [i, s] = pos;
30     if (p[i][s].second != s) subset.push_back(a[i-1]);
31     pos = p[i][s];
32 }

```

4 Trees

4.1 Sum of distances

Given a tree,

$$f(u, v) :=$$

distance from

$$u$$

to

$$v$$

in the tree, compute

$$\sum_{u,v} f(u, v)$$

Time: $\mathcal{O}(n)$

```

1 vvi adj;
2 vi sum_going_down, sum_going_up, sz;
3
4 void dfs(int u, int p){
5     for (auto v : adj[u]){
6         if (v == p) continue;
7         dfs(v, u);
8         sz[u] += sz[v];
9         sum_going_down[u] += sum_going_down[v];
10    }
11    sum_going_down[u] += sz[u];

```

```

12 }
13
14 void dfs2(int u, int p, int par_ans){
15     int up_amount = sz[0] - sz[u];
16     sum_going_up[u] += par_ans + up_amount;
17     int sum = sum_going_down[u];
18     for (auto v : adj[u]){
19         if (v == p) continue;
20         int par_amount = sz[0] - sz[v];
21         dfs2(v,u, par_ans + par_amount + sum - (sum_going_down[v]+sz[v]));
22     }
23 }
24
25 void solve(){
26     int n; cin >> n;
27     adj = vvi(n);
28     sum_going_down = sum_going_up = vi(n);
29     sz = vi(n,1);
30
31     for (int i = 1; i < n; i++){
32         int a, b; cin >> a >> b;
33         a--;
34         adj[a].push_back(b);
35         adj[b].push_back(a);
36     }
37
38     dfs(0,0);
39     dfs2(0,0,0);
40
41     for (int i = 0; i < n; i++) {
42         cout << sum_going_down[i]+sum_going_up[i] << ' ';
43     }
44     cout << endl;
45 }

```

4.2 Edge HLD

Sometimes the value is on the edges, for this few things need to change, but here is a template. Pre-computation: $\mathcal{O}(n)$ Queries: $\mathcal{O}(\log^2 n)$

```

1 struct EdgeHLD {
2     int n, timer = 0;
3     vvi adj;
4     vi parent, depth, size, heavy, head, value;
5     vi tin, tout;
6
7     segtree seg;
8
9     void init(int _n, vvi& _adj) {
10        n = _n;
11        adj = _adj;
12        value.assign(n, 0);
13        parent.assign(n, -1);
14        depth.assign(n, 0);
15        size.assign(n, 0);
16        heavy.assign(n, -1);
17        head.assign(n, 0);
18        tin.assign(n, 0);
19        tout.assign(n, 0);
20        timer = 0;
21
22        // edgeWeight[v] = weight of edge (parent[v], v)
23        // for v>0
24    }
25
26    void dfs1(int u, int p, int w) {
27        size[u] = 1;
28        parent[u] = p;
29        value[u] = w;
30        int max_sz = 0;
31        for (auto [v,w] : adj[u]) {
32            if (v == p) continue;
33            depth[v] = depth[u] + 1;
34            int sz = dfs1(v, u, w);
35            size[u] += sz;
36            if (sz > max_sz) {
37                max_sz = sz;
38                heavy[u] = v;
39            }
40        }
41        return size[u];
42    }
43
44    void dfs2(int u, int h) {
45        tin[u] = timer++;
46        head[u] = h;
47        if (heavy[u] != -1)
48            dfs2(heavy[u], h);
49        for (auto [v,w] : adj[u]) {
50            if (v != parent[u] && v != heavy[u])
51                dfs2(v, v);
52        }
53        tout[u] = timer;
54    }
55
56    // u deve ser o filho
57    void update_edge(int u, int val) {
58        seg.set(tin[u], val);
59    }
60
61    void rangeUpdate(int u, int v, int x) {
62        while (head[u] != head[v]) {
63            if (depth[head[u]] < depth[head[v]]) swap(u, v);
64            seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
65            u = parent[head[u]];
66        }
67        if (depth[u] > depth[v]) swap(u, v);
68        seg.rangeUpdate(tin[u] + 1, tin[v] + 1, x); // +1 to skip LCA's edge
69    }
70
71    void update_subtree(int u, int x) {
72        // updates all edges in subtree of u (skip
73        // incoming edge to u)
74        seg.rangeUpdate(tin[u] + 1, tout[u], x);
75    }
76
77    segtree::node query(int u, int v) {
78        segtree::node res = seg.NEUTRAL;
79
80        while (head[u] != head[v]) {
81            if (depth[head[u]] < depth[head[v]]) swap(u, v);
82            res = seg.merge(res, seg.query(tin[head[u]], tin[u] + 1));
83            u = parent[head[u]];
84        }
85
86        if (depth[u] > depth[v]) swap(u, v);
87        res = seg.merge(res, seg.query(tin[u] + 1, tin[v] + 1)); // skip LCA's edge
88
89        return res;
90    }
91
92    segtree::node query_subtree(int u) {
93        // query all edges in subtree of u
94        return seg.query(tin[u] + 1, tout[u]);
95    }
96
97
98    segtree::node query(int u, int v) {
99        // query all edges in path from u to v
100       return seg.query(tin[u], tout[v]);
101   }
102 }

```

```

24    // root (0) has no parent, so its value is
25    // dummy (0)
26    dfs1(0,0,0);
27    dfs2(0, 0);
28
29    vi linear(n);
30    for (int u = 0; u < n; u++)
31        linear[tin[u]] = value[u]; // position
32        stores edge weight
33
34    seg.init(linear);
35
36    int dfs1(int u, int p, int w) {
37        size[u] = 1;
38        parent[u] = p;
39        value[u] = w;
40        int max_sz = 0;
41        for (auto [v,w] : adj[u]) {
42            if (v == p) continue;
43            depth[v] = depth[u] + 1;
44            int sz = dfs1(v, u, w);
45            size[u] += sz;
46            if (sz > max_sz) {
47                max_sz = sz;
48                heavy[u] = v;
49            }
50        }
51        return size[u];
52    }
53
54    void dfs2(int u, int h) {
55        tin[u] = timer++;
56        head[u] = h;
57        if (heavy[u] != -1)
58            dfs2(heavy[u], h);
59        for (auto [v,w] : adj[u]) {
60            if (v != parent[u] && v != heavy[u])
61                dfs2(v, v);
62        }
63        tout[u] = timer;
64    }
65
66    // u deve ser o filho
67    void update_edge(int u, int val) {
68        seg.set(tin[u], val);
69    }
70
71    void rangeUpdate(int u, int v, int x) {
72        while (head[u] != head[v]) {
73            if (depth[head[u]] < depth[head[v]]) swap(u, v);
74            seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
75            u = parent[head[u]];
76        }
77        if (depth[u] > depth[v]) swap(u, v);
78        seg.rangeUpdate(tin[u] + 1, tin[v] + 1, x); // +1 to skip LCA's edge
79    }
80
81    void update_subtree(int u, int x) {
82        // updates all edges in subtree of u (skip
83        // incoming edge to u)
84        seg.rangeUpdate(tin[u] + 1, tout[u], x);
85    }
86
87    segtree::node query(int u, int v) {
88        segtree::node res = seg.NEUTRAL;
89
90        while (head[u] != head[v]) {
91            if (depth[head[u]] < depth[head[v]]) swap(u, v);
92            res = seg.merge(res, seg.query(tin[head[u]], tin[u] + 1));
93            u = parent[head[u]];
94        }
95
96        if (depth[u] > depth[v]) swap(u, v);
97        res = seg.merge(res, seg.query(tin[u] + 1, tin[v] + 1)); // skip LCA's edge
98
99        return res;
100    }
101
102    segtree::node query_subtree(int u) {
103        // query all edges in subtree of u
104        return seg.query(tin[u] + 1, tout[u]);
105    }
106
107
108    segtree::node query(int u, int v) {
109        // query all edges in path from u to v
110        return seg.query(tin[u], tout[v]);
111    }
112 }

```

4.3 HLD - Heavy light decomposition

If you need to compute a function on a path in a tree and need to support value updates on nodes, HLD is the way. Pre-computation: $\mathcal{O}(n)$ Queries: $\mathcal{O}(\log^2 n)$

OBS: this implementation uses the same segtree as this notebook, with 0-indexing and open-closed interval convention. Ideally, just change the segtree to change the computed function, the HLD struct remains the same. OBS2: this template also supports mass updates (path/subtree) and subtree queries.

```

1 struct HLD {
2     int n, timer = 0;
3     vvi adj;
4     vi parent, depth, size, heavy, head, value;
5     vi tin, tout;
6
7     segtree seg;
8
9     void init(int _n, vi& _value, vvi& _adj) {
10        n = _n;
11        adj = _adj;
12        value = _value;
13        parent.assign(n, -1);
14        depth.assign(n, 0);
15        size.assign(n, 0);
16        heavy.assign(n, -1);
17        head.assign(n, 0);
18        tin.assign(n, 0);
19        tout.assign(n, 0);
20        timer = 0;
21
22        dfs1(0);
23        dfs2(0, 0);
24
25        vi linear(n);
26        for (int u = 0; u < n; u++)
27            linear[tin[u]] = value[u];
28
29        seg.init(linear);
30
31        int dfs1(int u) {
32
33        }
34
35        int dfs2(int u, int h) {
36
37        }
38
39        void rangeUpdate(int u, int v, int x) {
40
41        }
42
43        void update_subtree(int u, int x) {
44
45        }
46
47        segtree::node query(int u, int v) {
48
49        }
50
51        segtree::node query_subtree(int u) {
52
53        }
54
55        segtree::node query(int u, int v) {
56
57        }
58
59        segtree::node query(int u, int v) {
60
61        }
62
63        segtree::node query(int u, int v) {
64
65        }
66
67        segtree::node query(int u, int v) {
68
69        }
70
71        segtree::node query(int u, int v) {
72
73        }
74
75        segtree::node query(int u, int v) {
76
77        }
78
79        segtree::node query(int u, int v) {
80
81        }
82
83        segtree::node query(int u, int v) {
84
85        }
86
87        segtree::node query(int u, int v) {
88
89        }
90
91        segtree::node query(int u, int v) {
92
93        }
94
95        segtree::node query(int u, int v) {
96
97        }
98
99        segtree::node query(int u, int v) {
100
101        }
102
103        segtree::node query(int u, int v) {
104
105        }
106
107        segtree::node query(int u, int v) {
108
109        }
110
111        segtree::node query(int u, int v) {
112
113        }
114
115        segtree::node query(int u, int v) {
116
117        }
118
119        segtree::node query(int u, int v) {
120
121        }
122
123        segtree::node query(int u, int v) {
124
125        }
126
127        segtree::node query(int u, int v) {
128
129        }
130
131        segtree::node query(int u, int v) {
132
133        }
134
135        segtree::node query(int u, int v) {
136
137        }
138
139        segtree::node query(int u, int v) {
140
141        }
142
143        segtree::node query(int u, int v) {
144
145        }
146
147        segtree::node query(int u, int v) {
148
149        }
150
151        segtree::node query(int u, int v) {
152
153        }
154
155        segtree::node query(int u, int v) {
156
157        }
158
159        segtree::node query(int u, int v) {
160
161        }
162
163        segtree::node query(int u, int v) {
164
165        }
166
167        segtree::node query(int u, int v) {
168
169        }
170
171        segtree::node query(int u, int v) {
172
173        }
174
175        segtree::node query(int u, int v) {
176
177        }
178
179        segtree::node query(int u, int v) {
180
181        }
182
183        segtree::node query(int u, int v) {
184
185        }
186
187        segtree::node query(int u, int v) {
188
189        }
190
191        segtree::node query(int u, int v) {
192
193        }
194
195        segtree::node query(int u, int v) {
196
197        }
198
199        segtree::node query(int u, int v) {
200
201        }
202
203        segtree::node query(int u, int v) {
204
205        }
206
207        segtree::node query(int u, int v) {
208
209        }
210
211        segtree::node query(int u, int v) {
212
213        }
214
215        segtree::node query(int u, int v) {
216
217        }
218
219        segtree::node query(int u, int v) {
220
221        }
222
223        segtree::node query(int u, int v) {
224
225        }
226
227        segtree::node query(int u, int v) {
228
229        }
230
231        segtree::node query(int u, int v) {
232
233        }
234
235        segtree::node query(int u, int v) {
236
237        }
238
239        segtree::node query(int u, int v) {
240
241        }
242
243        segtree::node query(int u, int v) {
244
245        }
246
247        segtree::node query(int u, int v) {
248
249        }
250
251        segtree::node query(int u, int v) {
252
253        }
254
255        segtree::node query(int u, int v) {
256
257        }
258
259        segtree::node query(int u, int v) {
260
261        }
262
263        segtree::node query(int u, int v) {
264
265        }
266
267        segtree::node query(int u, int v) {
268
269        }
270
271        segtree::node query(int u, int v) {
272
273        }
274
275        segtree::node query(int u, int v) {
276
277        }
278
279        segtree::node query(int u, int v) {
280
281        }
282
283        segtree::node query(int u, int v) {
284
285        }
286
287        segtree::node query(int u, int v) {
288
289        }
290
291        segtree::node query(int u, int v) {
292
293        }
294
295        segtree::node query(int u, int v) {
296
297        }
298
299        segtree::node query(int u, int v) {
300
301        }
302
303        segtree::node query(int u, int v) {
304
305        }
306
307        segtree::node query(int u, int v) {
308
309        }
310
311        segtree::node query(int u, int v) {
312
313        }
314
315        segtree::node query(int u, int v) {
316
317        }
318
319        segtree::node query(int u, int v) {
320
321        }
322
323        segtree::node query(int u, int v) {
324
325        }
326
327        segtree::node query(int u, int v) {
328
329        }
330
331        segtree::node query(int u, int v) {
332
333        }
334
335        segtree::node query(int u, int v) {
336
337        }
338
339        segtree::node query(int u, int v) {
340
341        }
342
343        segtree::node query(int u, int v) {
344
345        }
346
347        segtree::node query(int u, int v) {
348
349        }
350
351        segtree::node query(int u, int v) {
352
353        }
354
355        segtree::node query(int u, int v) {
356
357        }
358
359        segtree::node query(int u, int v) {
360
361        }
362
363        segtree::node query(int u, int v) {
364
365        }
366
367        segtree::node query(int u, int v) {
368
369        }
370
371        segtree::node query(int u, int v) {
372
373        }
374
375        segtree::node query(int u, int v) {
376
377        }
378
379        segtree::node query(int u, int v) {
380
381        }
382
383        segtree::node query(int u, int v) {
384
385        }
386
387        segtree::node query(int u, int v) {
388
389        }
390
391        segtree::node query(int u, int v) {
392
393        }
394
395        segtree::node query(int u, int v) {
396
397        }
398
399        segtree::node query(int u, int v) {
400
401        }
402
403        segtree::node query(int u, int v) {
404
405        }
406
407        segtree::node query(int u, int v) {
408
409        }
410
411        segtree::node query(int u, int v) {
412
413        }
414
415        segtree::node query(int u, int v) {
416
417        }
418
419        segtree::node query(int u, int v) {
420
421        }
422
423        segtree::node query(int u, int v) {
424
425        }
426
427        segtree::node query(int u, int v) {
428
429        }
430
431        segtree::node query(int u, int v) {
432
433        }
434
435        segtree::node query(int u, int v) {
436
437        }
438
439        segtree::node query(int u, int v) {
440
441        }
442
443        segtree::node query(int u, int v) {
444
445        }
446
447        segtree::node query(int u, int v) {
448
449        }
450
451        segtree::node query(int u, int v) {
452
453        }
454
455        segtree::node query(int u, int v) {
456
457        }
458
459        segtree::node query(int u, int v) {
460
461        }
462
463        segtree::node query(int u, int v) {
464
465        }
466
467        segtree::node query(int u, int v) {
468
469        }
470
471        segtree::node query(int u, int v) {
472
473        }
474
475        segtree::node query(int u, int v) {
476
477        }
478
479        segtree::node query(int u, int v) {
480
481        }
482
483        segtree::node query(int u, int v) {
484
485        }
486
487        segtree::node query(int u, int v) {
488
489        }
490
491        segtree::node query(int u, int v) {
492
493        }
494
495        segtree::node query(int u, int v) {
496
497        }
498
499        segtree::node query(int u, int v) {
500
501        }
502
503        segtree::node query(int u, int v) {
504
505        }
506
507        segtree::node query(int u, int v) {
508
509        }
510
511        segtree::node query(int u, int v) {
512
513        }
514
515        segtree::node query(int u, int v) {
516
517        }
518
519        segtree::node query(int u, int v) {
520
521        }
522
523        segtree::node query(int u, int v) {
524
525        }
526
527        segtree::node query(int u, int v) {
528
529        }
530
531        segtree::node query(int u, int v) {
532
533        }
534
535        segtree::node query(int u, int v) {
536
537        }
538
539        segtree::node query(int u, int v) {
540
541        }
542
543        segtree::node query(int u, int v) {
544
545        }
546
547        segtree::node query(int u, int v) {
548
549        }
550
551        segtree::node query(int u, int v) {
552
553        }
554
555        segtree::node query(int u, int v) {
556
557        }
558
559        segtree::node query(int u, int v) {
560
561        }
562
563        segtree::node query(int u, int v) {
564
565        }
566
567        segtree::node query(int u, int v) {
568
569        }
570
571        segtree::node query(int u, int v) {
572
573        }
574
575        segtree::node query(int u, int v) {
576
577        }
578
579        segtree::node query(int u, int v) {
580
581        }
582
583        segtree::node query(int u, int v) {
584
585        }
586
587        segtree::node query(int u, int v) {
588
589        }
590
591        segtree::node query(int u, int v) {
592
593        }
594
595        segtree::node query(int u, int v) {
596
597        }
598
599        segtree::node query(int u, int v) {
600
601        }
602
603        segtree::node query(int u, int v) {
604
605        }
606
607        segtree::node query(int u, int v) {
608
609        }
610
611        segtree::node query(int u, int v) {
612
613        }
614
615        segtree::node query(int u, int v) {
616
617        }
618
619        segtree::node query(int u, int v) {
620
621        }
622
623        segtree::node query(int u, int v) {
624
625        }
626
627        segtree::node query(int u, int v) {
628
629        }
630
631        segtree::node query(int u, int v) {
632
633        }
634
635        segtree::node query(int u, int v) {
636
637        }
638
639        segtree::node query(int u, int v) {
640
641        }
642
643        segtree::node query(int u, int v) {
644
645        }
646
647        segtree::node query(int u, int v) {
648
649        }
650
651        segtree::node query(int u, int v) {
652
653        }
654
655        segtree::node query(int u, int v) {
656
657        }
658
659        segtree::node query(int u, int v) {
660
661        }
662
663        segtree::node query(int u, int v) {
664
665        }
666
667        segtree::node query(int u, int v) {
668
669        }
670
671        segtree::node query(int u, int v) {
672
673        }
674
675        segtree::node query(int u, int v) {
676
677        }
678
679        segtree::node query(int u, int v) {
680
681        }
682
683        segtree::node query(int u, int v) {
684
685        }
686
687        segtree::node query(int u, int v) {
688
689        }
690
691        segtree::node query(int u, int v) {
692
693        }
694
695        segtree::node query(int u, int v) {
696
697        }
698
699        segtree::node query(int u, int v) {
700
701        }
702
703        segtree::node query(int u, int v) {
704
705        }
706
707        segtree::node query(int u, int v) {
708
709        }
710
711        segtree::node query(int u, int v) {
712
713        }
714
715        segtree::node query(int u, int v) {
716
717        }
718
719        segtree::node query(int u, int v) {
720
721        }
722
723        segtree::node query(int u, int v) {
724
725        }
726
727        segtree::node query(int u, int v) {
728
729        }
730
731        segtree::node query(int u, int v) {
732
733        }
734
735        segtree::node query(int u, int v) {
736
737        }
738
739        segtree::node query(int u, int v) {
740
741        }
742
743        segtree::node query(int u, int v) {
744
745        }
746
747        segtree::node query(int u, int v) {
748
749        }
750
751        segtree::node query(int u, int v) {
752
753        }
754
755        segtree::node query(int u, int v) {
756
757        }
758
759        segtree::node query(int u, int v) {
760
761        }
762
763        segtree::node query(int u, int v) {
764
765        }
766
767        segtree::node query(int u, int v) {
768
769        }
770
771        segtree::node query(int u, int v) {
772
773        }
774
775        segtree::node query(int u, int v) {
776
777        }
778
779        segtree::node query(int u, int v) {
780
781        }
782
783        segtree::node query(int u, int v) {
784
785        }
786
787        segtree::node query(int u, int v) {
788
789        }
790
791        segtree::node query(int u, int v) {
792
793        }
794
795        segtree::node query(int u, int v) {
796
797        }
798
799        segtree::node query(int u, int v) {
800
801        }
802
803        segtree::node query(int u, int v) {
804
805        }
806
807        segtree::node query(int u, int v) {
808
809        }
810
811        segtree::node query(int u, int v) {
812
813        }
814
815        segtree::node query(int u, int v) {
816
817        }
818
819        segtree::node query(int u, int v) {
820
821        }
822
823        segtree::node query(int u, int v) {
824
825        }
826
827        segtree::node query(int u, int v) {
828
829        }
830
831        segtree::node query(int u, int v) {
832
833        }
834
835        segtree::node query(int u, int v) {
836
837        }
838
839        segtree::node query(int u, int v) {
840
841        }
842
843        segtree::node query(int u, int v) {
844
845        }
846
847        segtree::node query(int u, int v) {
848
849        }
850
851        segtree::node query(int u, int v) {
852
853        }
854
855        segtree::node query(int u, int v) {
856
857        }
858
859        segtree::node query(int u, int v) {
860
861        }
862
863        segtree::node query(int u, int v) {
864
865        }
866
867        segtree::node query(int u, int v) {
868
869        }
870
871        segtree::node query(int u, int v) {
872
873        }
874
875        segtree::node query(int u, int v) {
876
877        }
878
879        segtree::node query(int u, int v) {
880
881        }
882
883        segtree::node query(int u, int v) {
884
885        }
886
887        segtree::node query(int u, int v) {
888
889        }
890
891        segtree::node query(int u, int v) {
892
893        }
894
895        segtree::node query(int u, int v) {
896
897        }
898
899        segtree::node query(int u, int v) {
900
901        }
902
903        segtree::node query(int u, int v) {
904
905        }
906
907        segtree::node query(int u, int v) {
908
909        }
910
911        segtree::node query(int u, int v) {
912
913        }
914
915        segtree::node query(int u, int v) {
916
917        }
918
919        segtree::node query(int u, int v) {
920
921        }
922
923        segtree::node query(int u, int v) {
924
925        }
926
927        segtree::node query(int u, int v) {
928
929        }
930
931        segtree::node query(int u, int v) {
932
933        }
934
935        segtree::node query(int u, int v) {
936
937        }
938
939        segtree::node query(int u, int v) {
940
941        }
942
943        segtree::node query(int u, int v) {
944
945        }
946
947        segtree::node query(int u, int v) {
948
949        }
950
951        segtree::node query(int u, int v) {
952
953        }
954
955        segtree::node query(int u, int v) {
956
957        }
958
959        segtree::node query(int u, int v) {
960
961        }
962
963        segtree::node query(int u, int v) {
964
965        }
966
967        segtree::node query(int u, int v) {
968
969        }
970
971        segtree::node query(int u, int v) {
972
973        }
974
975        segtree::node query(int u, int v) {
976
977        }
978
979        segtree::node query(int u, int v) {
980
981        }
982
983        segtree::node query(int u, int v) {
984
985        }
986
987        segtree::node query(int u, int v) {
988
989        }
990
991        segtree::node query(int u, int v) {
992
993        }
994
995        segtree::node query(int u, int v) {
996
997        }
998
999        segtree::node query(int u, int v) {
1000
1001        }
1002
1003        segtree::node query(int u, int v) {
1004
1005        }
1006
1007        segtree::node query(int u, int v) {
1008
1009        }
1010
1011        segtree::node query(int u, int v) {
1012
1013        }
1014
1015        segtree::node query(int u, int v) {
1016
1017        }
1018
1019        segtree::node query(int u, int v) {
1020
1021        }
1022
1023        segtree::node query(int u, int v) {
1024
1025        }
1026
1027        segtree::node query(int u, int v) {
1028
1029        }
1030
1031        segtree::node query(int u, int v) {
1032
1033        }
1034
1035        segtree::node query(int u, int v) {
1036
1037        }
1038
1039        segtree::node query(int u, int v) {
1040
1041        }
1042
1043        segtree::node query(int u, int v) {
1044
1045        }
1046
1047        segtree::node query(int u, int v) {
1048
1049        }
1050
1051        segtree::node query(int u, int v) {
1052
1053        }
1054
1055        segtree::node query(int u, int v) {
1056
1057        }
1058
1059        segtree::node query(int u, int v) {
1060
1061        }
1062
1063        segtree::node query(int u, int v) {
1064
1065        }
1066
1067        segtree::node query(int u, int v) {
1068
1069        }
1070
1071        segtree::node query(int u, int v) {
1072
1073        }
1074
1075        segtree::node query(int u, int v) {
1076
1077        }
1078
1079        segtree::node query(int u, int v) {
1080
1081        }
1082
1083        segtree::node query(int u, int v) {
1084
1085        }
1086
1087        segtree::node query(int u, int v) {
1088
1089        }
1090
1091        segtree::node query(int u, int v) {
1092
1093        }
1094
1095        segtree::node query(int u, int v) {
1096
1097        }
1098
1099        segtree::node query(int u, int v) {
1100
1101        }
1102
1103        segtree::node query(int u, int v) {
1104
1105        }
1106
1107        segtree::node query(int u, int v) {
1108
1109        }
1110
1111        segtree::node query(int u, int v) {
1112
1113        }
1114
1115        segtree::node query(int u, int v) {
1116
1117        }
1118
1119        segtree::node query(int u, int v) {
1120
1121        }
1122
1123        segtree::node query(int u, int v) {
1124
1125        }
1126
1127        segtree::node query(int u, int v) {
1128
1129        }
1130

```

```

33     size[u] = 1;
34     int max_sz = 0;
35     for (int v : adj[u]) {
36         if (v == parent[u]) continue;
37         parent[v] = u;
38         depth[v] = depth[u] + 1;
39         int sz = dfs1(v);
40         size[u] += sz;
41         if (sz > max_sz) {
42             max_sz = sz;
43             heavy[u] = v;
44         }
45     }
46     return size[u];
47 }
48
49 void dfs2(int u, int h) {
50     tin[u] = timer++;
51     head[u] = h;
52     if (heavy[u] != -1)
53         dfs2(heavy[u], h);
54     for (int v : adj[u]) {
55         if (v != parent[u] && v != heavy[u])
56             dfs2(v, v);
57     }
58     tout[u] = timer;
59 }
60
61 void update(int u, int val) {
62     seg.set(tin[u], val);
63 }
64
65 void rangeUpdate(int u, int v, int x) {
66     while (head[u] != head[v]) {
67         if (depth[head[u]] < depth[head[v]]) swap(u, v);
68         seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
69     }
70     u = parent[head[u]];
71     if (depth[u] > depth[v]) swap(u, v);
72     seg.rangeUpdate(tin[u], tin[v] + 1, x);
73 }
74
75 void update_subtree(int u, int x) {
76     seg.rangeUpdate(tin[u], tout[u], x);
77 }
78
79 segtree::node query(int u, int v) {
80     segtree::node res = seg.NEUTRAL;
81     while (head[u] != head[v]) {
82         if (depth[head[u]] < depth[head[v]])
83             swap(u, v);
84         res = seg.merge(res, seg.query(tin[head[v]], tin[v] + 1));
85         u = parent[head[u]];
86     }
87     if (depth[u] > depth[v]) swap(u, v);
88     res = seg.merge(res, seg.query(tin[u], tin[v] + 1));
89     return res;
90 }
91
92 segtree::node query_subtree(int u) {
93     return seg.query(tin[u], tout[u]);
94 }
95 };

```

4.4 LCA - RMQ

Pre-computation: $\mathcal{O}(n \log n)$ Queries: $\mathcal{O}(1)$ OBS: call first `dfs(root)` and then `buildSparseTable()` before making queries. Also remember to call `eulertournodes.reserve(2 · n)` and `eulertourdepths.reserve(2 · n)` to optimize memory allocation time of `pushback`.

```

1 int n, timer = 0;
2 vi tin, dep, euler_tour_nodes, euler_tour_depths;
3 vvi ch;
4 vvii sparse_table;
5
6 void dfs(int u) {
7     euler_tour_nodes.push_back(u);
8     euler_tour_depths.push_back(dep[u]);
9     tin[u] = timer++;
10
11    for (int v : ch[u]) {
12        dep[v] = dep[u] + 1;
13        dfs(v);
14        euler_tour_nodes.push_back(u);
15        euler_tour_depths.push_back(dep[u]);
16    }
17
18    timer++;
19}
20
21 void buildSparseTable() {
22     int m = euler_tour_depths.size();
23     sparse_table.assign(LOGN, vii(m));
24
25     for (int i = 0; i < m; i++) {
26         sparse_table[0][i] = {euler_tour_depths[i], i};
27     }
28
29     for (int i = 1; (1 << i) <= m; i++) {
30         int len = 1<<i;
31         for (int time = 0; time+len <= m; time++) {
32             ii ans1 = sparse_table[i-1][time];
33             ii ans2 = sparse_table[i-1][time + len/2];
34             sparse_table[i][time] = min(ans1, ans2);
35         }
36     }
37 }
38
39 int lca(int u, int v) {
40     int tu = tin[u];
41     int tv = tin[v];
42     if (tu > tv) swap(tu, tv);
43
44     int k = __bit_width((tv - tu + 1)) - 1;
45
46     ii ans1 = sparse_table[k][tu];
47     ii ans2 = sparse_table[k][tv - (1 << k) + 1];
48
49     if (ans1.first <= ans2.first) {
50         return euler_tour_nodes[ans1.second];
51     }
52     return euler_tour_nodes[ans2.second];
53 }

```

4.5 LCA - binary lifting

Pre-computation: $\mathcal{O}(n \log n)$ Queries: $\mathcal{O}(\log n)$ OBS: just call `dfs(root)` before starting queries.

```

1 vvi adj, up;
2 vi tin, tout;
3 int timer = 0;
4
5 void dfs(int u, int p){
6     tin[u] = timer++;
7     for (auto v : adj[u]){
8         if (v == p) continue;
9         up[v][0] = u;
10        for (int dist = 1; dist < LOGN; dist++){
11            up[v][dist] = up[up[v][dist-1]][dist-1];
12        }
13        dfs(v);
14    }
15    tout[u] = timer++;
16 }
17
18 int isAncestor(int u, int v){
19     return tin[u] <= tin[v] && tout[v] <= tout[u];
20 }
21
22 int lca(int u, int v){
23     if (isAncestor(u,v)) return u;
24     if (isAncestor(v,u)) return v;
25     for (int dist = LOGN-1; dist >= 0; dist--){
26         if(!isAncestor(up[u][dist],v)) u = up[u][dist];
27     }
28     return up[u][0];
29 }

```

5 Problemas clássicos

5.1 2sat

```

1 struct TwoSatSolver {
2     int n;
3     vvi adj, adjT;
4     vector<bool> vis, assignment;
5     vi topo, scc;
6
7     void build(int _n){
8         n = 2*_n;
9         adj.assign(n,vi());
10        adjT.assign(n,vi());
11    }
12
13    int get(int u){
14        if (u < 0) return 2*(~u)+1;
15        else return 2*u;
16    }
17
18    // u -> v
19    void add_impl(int u, int v){
20        u = get(u), v = get(v);
21        adj[u].push_back(v);
22        adjT[v].push_back(u);
23        adj[v^1].push_back(u^1);
24        adjT[u^1].push_back(v^1);
25    }
26
27    // u || v
28    void add_or(int u, int v){
29        add_impl(~u, v);
30    }
31
32    // u @@ v
33    void add_and(int u, int v){
34

```

```

34     add_or(u,u); add_or(v,v);
35 }
36 // u ^ v (equiv of x != v)
37 void add_xor(int u, int v){
38     add_impl(u, ~v);
39     add_impl(~u, v);
40 }
41
42 // u == v
43 void add_equals(int u, int v){
44     add_impl(u, v);
45     add_impl(v, u);
46 }
47
48 void toposort(int u){
49     vis[u] = true;
50     for (int v : adj[u])
51         if (!vis[v]) toposort(v);
52     topo.push_back(u);
53 }
54
55 void dfs(int u, int c){
56     scc[u] = c;
57     for (int v : adjT[u])
58         if (!scc[v]) dfs(v, c);
59 }
60
61 pair<bool, vector<bool>> solve(){
62     topo.clear();
63     vis.assign(n, false);
64
65     for (int i = 0; i < n; i++)
66         if (!vis[i]) toposort(i);
67
68     reverse(topo.begin(), topo.end());
69
70     scc.assign(n, 0);
71     int c = 0;
72     for (int u : topo)
73         if (!scc[u]) dfs(u, ++c);
74
75     assignment.assign(n/2, false);
76     for (int i = 0; i < n; i += 2){
77         if (scc[i] == scc[i+1]) return {false, {}};
78         assignment[i/2] = scc[i] > scc[i+1];
79     }
80
81     return {true, assignment};
82 }
83 }
```

5.2 Next Greater Element

One of the classic stack applications. Easy to translate to lower, leq or geq, just change the comparator of the while.

```

1 vi next_greater_elem(n, n);
2
3 stack<ii> st;
4 for (int i = 0; i < n; i++){
5     while (!st.empty() && st.top().first < h[i]){
6         next_greater_elem[st.top().second] = i;
7         st.pop();
8     }
9     st.emplace(h[i], i);
10 }
```

6 Strings

6.1 Hashing

Creation time: $\mathcal{O}(n)$ Access time: $\mathcal{O}(1)$ Space: $\mathcal{O}(n)$

```

1 class Hashing{
2     const int mod0 = 1e9+7;
3     vi pmod0;
4     vull pmod1;
5
6     public:
7     void CalcP(int mn, int n){
8         random_device rd;
9         uniform_int_distribution<int> dist(mn+2, mod0
10             -1);
11         int p = dist(rd);
12         if(p % 2 == 0) p--;
13         pmod0 = vi(n);
14         pmod1 = vull(n);
15         pmod0[0] = pmod1[0] = 1;
16         for(int i = 1; i < n; i++){
17             pmod0[i] = (pmod0[i-1] * p) % mod0;
18             pmod1[i] = (pmod1[i-1] * p);
19         }
20
21     viull DistinctSubstrHashes(string base, int
22         offsetVal){
23         int n = base.size();
24         viull ans;
25         for(int i = 0; i < n; i++){
26             int h0 = 0;
27             ull h1 = 0;
28             for(int j = i; j < n; j++){
29                 h0 = (h0 + (base[j]-offsetVal)*pmod0[j-i]) %
30                     mod0;
31                 h1 = (h1 + (base[j]-offsetVal)*pmod1[j-i]);
32                 ans.push_back(iull(h0, h1));
33             }
34         }
35         sort(ans.begin(), ans.end());
36         auto last = unique(ans.begin(), ans.end());
37         ans.erase(last, ans.end());
38         return ans;
39     };
40     viull WindowHash(string data, int offsetVal, int
41         lenWindow){
42         int n = data.size();
43         int h0 = 0;
44         ull h1 = 0;
45         viull ans;
46         for(int i = 0; i < lenWindow; i++){
47             h0 = (h0 + (data[i]+offsetVal)*pmod0[i]) %
48                 mod0;
49             h1 = (h1 + (data[i]+offsetVal)*pmod1[i]);
50         }
51         ans.push_back(iull((h0*pmod0[n-1])%mod0, h1*
52             pmod1[n-1]));
53         for(int i = lenWindow; i < n; i++){
54             h0 = (h0 - (data[i-lenWindow]+offsetVal)*
55                 pmod0[i-lenWindow]) % mod0;
56             h0 = (h0 + mod0) % mod0;
57             h0 = (h0 + (data[i]+offsetVal)*pmod0[i]) %
58                 mod0;
59             h1 = (h1 - (data[i-lenWindow]+offsetVal)*
60                 pmod1[i-lenWindow]));
61         }
62     }
63 }
```

```

54         h1 = (h1 + (data[i]+offsetVal)*pmod1[i]);
55         ans.push_back(iull((h0*pmod0[n-1-(i-
56             lenWindow+1)])%mod0, h1*pmod1[n-1-(i-
57             lenWindow+1)]));
58     };
59 }
```

6.2 KMP

```

1 vi compute_lps(const string &pat){
2     int m = pat.length();
3     vi lps(m);
4     int len = 0;
5     for (int i = 1; i < m; i++){
6         while(len > 0 && pat[i] != pat[len])
7             len = lps[len-1];
8         if (pat[i] == pat[len]) len++;
9         lps[i] = len;
10    }
11    return lps;
12 }
13
14 // find all occurrences
15 vi kmp_search(const string &txt, const string &pat){
16     int n = txt.length();
17     int m = pat.length();
18     if (m == 0) return {};
19     vi lps = compute_lps(pat);
20     vi occurrences;
21     int j = 0;
22     for (int i = 0; i < n; i++){
23         while (j > 0 && txt[i] != pat[j])
24             j = lps[j-1];
25         if (txt[i] == pat[j]) j++;
26         if (j == m) {
27             occurrences.push_back(i-m+1);
28             j = lps[j-1];
29         }
30     }
31     return occurrences;
32 }
33
34 // find all occurrences (simpler version)
35 vi kmp_search(const string &txt, const string &pat){
36     int n = txt.length(), m = pat.length();
37     vi lps = compute_lps(pat + '#' + txt);
38     vi occurrences;
39     for (int i = 0; i < n+m+1; i++){
40         if (lps[i] == pat.length())
41             occurrences.push_back(i-m*2);
42     }
43     return occurrences;
44 }
45
46 // borda sao os prefixos que tambem sao sufixos
47 vi find_borders(const string &s){
48     vi lps = compute_lps(s);
49     int i = s.length()-1;
50
51     vi ans;
52     while (lps[i] > 0){
53         ans.push_back(lps[i]);
54         i = lps[i]-1;
55     }
56     reverse(ans.begin(), ans.end());
57 }
```

```
58     return ans;
59 }
```

6.3 Suffix Array

Time: $\mathcal{O}(n \log n)$ Space: $\mathcal{O}(n)$

```
1 struct SuffixArray {
2     int sz;
3     vi suff_ind, lcp;
4     viii suffs;
5
6     void radix_sort() {
7         if (sz <= 1) return;
8         viii suffs_new(sz);
9         vii cnt(sz + 1, 0); /*rever esse tamanho*/
10
11        for (auto& item : suffs) cnt[item.first.second] += 1;
12        for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
13        for (int i = sz - 1; i >= 0; --i) suffs_new[--cnt[suffs[i].first.second]] = suffs[i];
14
15        cnt.assign(sz + 1, 0);
16        for (auto& item : suffs_new) cnt[item.first.first]++;
17        for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
18        for (int i = sz - 1; i >= 0; --i) suffs_new[--cnt[suffs_new[i].first.first]] = suffs_new[i];
19    }
20
21    void build_lcp(vi& a) {
22        lcp.assign(sz, 0);
23        vi rank(sz);
24        for (int i = 0; i < sz; ++i) rank[suff_ind[i]] = i;
25
26        int h = 0;
27        for (int i = 0; i < sz; ++i) {
28            if (rank[i] == sz - 1) { h = 0; continue; }
29            if (h > 0) h--;
30            int j = suff_ind[rank[i] + 1];
31            while (i + h < sz && j + h < sz && a[i + h] == a[j + h]) h++;
32            lcp[rank[i] + 1] = h;
33        }
34
35        void build(vi& a) {
36            a.push_back(0);
37            sz = a.size();
38            suffs.resize(sz);
39            suffs_ind.resize(sz);
40            vi equiv(sz);
41
42            for (int i = 0; i < sz; ++i) suffs[i] = iii(ii(a[i], a[i]), i);
43            radix_sort();
44            for (int i = 1; i < sz; ++i) {
45                auto [c, ci] = suffs[i];
46                auto [p, pi] = suffs[i-1];
47                equiv[ci] = equiv[pi] + (c > p);
48            }
49
50            for (int suflen = 1; suflen < sz; suflen *= 2)
51            {
52
```

```
53                for (int i = 0; i < sz; ++i) {
54                    suffs[i] = {{equiv[i], equiv[(i + suflen) % sz]}, i};
55                }
56                radix_sort();
57                for (int i = 1; i < sz; ++i) {
58                    auto [c, ci] = suffs[i];
59                    auto [p, pi] = suffs[i-1];
60                    equiv[ci] = equiv[pi] + (c > p);
61                }
62
63                for (int i = 0; i < sz; ++i) suffs_ind[i] = suffs[i].second;
64                build_lcp(a);
65
66                a.pop_back();
67                sz--;
68                suffs_ind.erase(suffs_ind.begin());
69                lcp.erase(lcp.begin());
70            }
71        }
72    }
73 }
```

6.4 Z

$z[i] := \max(k) | s[0..k-1] = s[i..i+k-1]$

Time: $\mathcal{O}(n + m)$ Space: $\mathcal{O}(n + m)$

```
1 vi compute_z(const string &s) {
2     int n = s.length();
3     vi z(n);
4     int l = 0, r = 0;
5
6     for (int i = 1; i < n; i++) {
7         if (i <= r)
8             z[i] = min(r - i + 1, z[i-1]);
9
10        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
11            z[i]++;
12        if (i + z[i] - 1 > r) {
13            l = i;
14            r = i + z[i] - 1;
15        }
16    }
17
18    return z;
19}
20
21 vi find_occurrences(const string &txt, const string &pat){
22     vi occurrences;
23     vi z = compute_z(pat + '#' + txt);
24     int n = txt.length(), m = pat.length();
25     for (int i = 0; i < n+m+1; i++){
26         if (z[i] == m) occurrences.push_back(i-m-1);
27     }
28     return occurrences;
29}
```

7 Math

7.1 Combinatorics (Pascal's Triangle)

Computes " n choose k ". Requires factorials to be pre-computed. Time: $\mathcal{O}(\log ZAP)$

7.1.1 Combinatorial Analysis

Fundamental Counting Principles

- **Permutations:** The number of ways to arrange k items from a set of n distinct items.

$$P(n, k) = \frac{n!}{(n-k)!}$$

- **Combinations (Binomial Coefficient):** The number of ways to choose k items from a set of n distinct items, regardless of order.

$$\binom{n}{k} = C(n, k) = \frac{n!}{k!(n-k)!}$$

- **Combinations with Repetition (Stars and Bars):** The number of ways to choose k items of n types, allowing repetitions. Equivalently, the number of ways to distribute k identical balls into n distinct urns.

$$\binom{k+n-1}{n-1} = \binom{k+n-1}{k}$$

Binomial Coefficient Properties and Pascal's Triangle

• Pascal's Triangle

$$[n = 0 : \binom{0}{0} \quad n = 1 : \binom{1}{0} \quad n = 2 : \binom{2}{0} \quad \binom{2}{1}]$$

- **Stifel's Relation:** Each element in Pascal's Triangle is the sum of the two elements immediately above it.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- Symmetry:** Elements of a row are symmetric with respect to the center. Choosing k elements is the same as choosing the $n - k$ elements to be left behind.

$$\binom{n}{k} = \binom{n}{n-k}$$

- Row Sum:** The sum of all elements in row n of Pascal's Triangle (where the first row is $n = 0$) is equal to 2^n .

$$\sum_{k=0}^n \binom{n}{k} = 2^n$$

- Hockey Stick Identity:** The sum of elements in a diagonal, starting at

$$\binom{r}{r}$$

and ending at

$$\binom{n}{r}$$

, is equal to the element in the next row and next column,

$$\binom{n+1}{r+1}$$

$$\sum_{i=r}^n \binom{i}{r} = \binom{n+1}{r+1}$$

- Binomial Theorem:**

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$$

- Vandermonde's Identity:**

$$\sum_{j=0}^k \binom{m}{j} \binom{n}{k-j} = \binom{m+n}{k}$$

The easiest way to understand the identity is through a counting problem. Imagine you have a committee with m men and n women. How many ways can you form a subcommittee of k people?

Way 1 Direct Counting

You have a total of $m+n$ people and need to choose k of them. The number of ways to do this is simply:

$$\binom{m+n}{k}$$

Way 2 Counting by Cases

We can divide the problem into cases, based on how many men (j) are chosen for the subcommittee. Case 0: Choose 0 men and k women. The number of ways is

$$\binom{m}{0} \binom{n}{k}$$

Case 1: Choose 1 man and $k-1$ women. The number of ways is $\binom{m}{1} \binom{n}{k-1}$.
Case j : Choose j men and $k-j$ women. The number of ways is $\binom{m}{j} \binom{n}{k-j}$.

Other Important Concepts

- Catalan Numbers:** A sequence of natural numbers that occurs in various counting problems (e.g., number of binary trees, balanced parenthesis expressions).

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

- Stirling Numbers of the Second Kind:** The number of ways to partition a set of n labeled objects into k non-empty unlabeled subsets. Denoted by $S(n, k)$ or

$$\{n \ k\}$$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

- Pigeonhole Principle:** If n items are put into m boxes, with $n > m$, then at least one box must contain more than one item.

```
1 // n escolhe k
2 // linha n, coluna k no triangulo (indexadas em 0)
3 int pascal(int n, int k){
4     int num = fat[n];
5     int den = (fat[k]*fat[n-k])%ZAP;
6     return (num*expbin(den, ZAP-2))%ZAP;
7 }
```

7.2 Convolutions

7.2.1 AND convolution

$$c[k] = \sum_{i+j=k} a[i] \cdot b[j]$$

```
1 vector<mint<MOD>> and_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = a.size(); // must be pow of 2
4     for (int j = 1; j < n; j <= 1) {
5         for (int i = 0; i < n; i++) {
6             if (i&j) {
7                 a[i^j] += a[i];
8                 b[i^j] += b[i];
9             }
10        }
11    }
12    for (int j = 1; j < n; j <= 1) {
13        for (int i = 0; i < n; i++) {
14            if (i&j) a[i^j] -= a[i];
15        }
16    }
17 }
18 }
19 return a;
20 }
```

7.2.2 GCD convolution

$$c[k] = \sum_{\gcd(i,j)=k} a[i] \cdot b[j]$$

```
1 vector<mint<MOD>> gcd_conv(vi a, vi b){
2     int n = (int)max(a.size(), b.size());
3     a.resize(n);
4     b.resize(n);
5     vector<mint<MOD>> c(n);
6     for (int i = 1; i < n; i++) {
7         mint<MOD> x = 0;
8         mint<MOD> y = 0;
9         for (int j = i; j < n; j += i) {
10            x += a[j];
11            y += b[j];
12        }
13        c[i] = x*y;
14    }
15    for (int i = n-1; i >= 1; i--) {
16        for (int j = 2 * i; j < n; j += i)
17            c[i] -= c[j];
18    }
19 }
20 }
```

7.2.3 LCM convolution

$$c[k] = \sum_{\text{lcm}(i,j)=k} a[i] \cdot b[j]$$

```

1 vector<mint<MOD>> lcm_conv(vi a, vi b){
2     int n = (int)max(a.size(), b.size());
3     a.resize(n);
4     b.resize(n);
5     vector<mint<MOD>> c(n), x(n), y(n);
6     for (int i = 1; i < n; i++) {
7         for (int j = i; j < n; j += i) {
8             x[j] += a[i];
9             y[j] += b[i];
10        }
11        c[i] = x[i]*y[i];
12    }
13    for (int i = 1; i < n; i++)
14        for (int j = 2 * i; j < n; j += i)
15            c[j] -= c[i];
16
17    return c;
18 }
```

7.2.4 OR convolution

$$c[k] = \sum_{i|j=k} a[i] \cdot b[j]$$

```

1 vector<mint<MOD>> or_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = a.size(); // must be pow of 2
4     for (int j = 1; j < n; j <= 1) {
5         for (int i = 0; i < n; i++) {
6             if (i&j) {
7                 a[i] += a[i^j];
8                 b[i] += b[i^j];
9             }
10        }
11
12        for (int i = 0; i < n; i++) a[i] *= b[i];
13
14        for (int j = 1; j < n; j <= 1) {
15            for (int i = 0; i < n; i++) {
16                if (i&j) a[i] -= a[i^j];
17            }
18        }
19
20    return a;
21 }
```

7.2.5 XOR convolution

$$c[k] = \sum_{i \oplus j=k} a[i] \cdot b[j]$$

```

1 void fwht(vector<mint<MOD>> &a, bool inv){
2     int n = a.size(); // must be pow of 2
3     for (int step = 1; step < n; step <= 1) {
4         for (int i = 0; i < n; i += 2*step) {
5             for (int j = i; j < i+step; j++) {
```

```

6                 auto u = a[j];
7                 auto v = a[j+step];
8                 a[j] = u+v;
9                 a[j+step] = u-v;
10            }
11        }
12    }
13    if (inv) for (auto &x : a) x /= n;
14
15    vector<mint<MOD>> xor_conv(vector<mint<MOD>> a, vector<
16        mint<MOD>> b){
17        int n = a.size();
18        fwht(a, 0), fwht(b, 0);
19        for (int i = 0; i < n; i++) a[i] *= b[i];
20        fwht(a, 1);
21        return a;
22    }
23
24
25
26        if (i == 2) i--;
27    }
28    if (n > 1) ans *= 2;
29    return ans;
30 }
31
32 // O(sqrt(n))
33 // Calcula a soma de todos os divisores de um numero n.
34 int somaDivisores(int n) {
35     int ans = 1;
36     for (int i = 2; i * i <= n; i += 2) {
37         if (n % i == 0) {
38             int exp = 0;
39             while (n % i == 0) {
40                 n /= i; exp++;
41             }
42             int aux = expbin(i, exp + 1);
43             ans *= ((aux - 1) / (i - 1));
44         }
45         if (i == 2) i--;
46     }
47
48     if (n > 1) ans *= (n + 1);
49
50 } 
```

7.3 Extended Euclid

Time: $\mathcal{O}(\log n)$.

```

1 int extended_gcd(int a, int b, int &x, int &y) {
2     x = 1, y = 0;
3     int x1 = 0, y1 = 1;
4     while (b) {
5         int q = a / b;
6         tie(x, x1) = make_tuple(x1, x - q * x1);
7         tie(y, y1) = make_tuple(y1, y - q * y1);
8         tie(a, b) = make_tuple(b, a - q * b);
9     }
10    return a;
11 }
```

7.4 Factorization

Time: $\mathcal{O}(\sqrt{n})$

```

1 // OBS: tem outras variantes mais rapidas no caderno da
2 // UDESC
3 // O(sqrt(n)) fatores repetidos
4 vi fatora(int n) {
5     vi factors;
6     for (int x = 2; x * x <= n; x++) {
7         while (n % x == 0) {
8             factors.push_back(x);
9             n /= x;
10        }
11    }
12    if (n > 1) factors.push_back(n);
13    return factors;
14 }
15
16 // O(sqrt(n))
17 // Calcula a quantidade de divisores de um numero n.
18 int qtdDivisores(int n) {
19     int ans = 1;
20     for (int i = 2; i * i <= n; i += 2) {
21         int exp = 0;
22         while (n % i == 0) {
23             n /= i; exp++;
24         }
25         if (exp > 0) ans *= (exp + 1);
```

7.5 FFT - Fast Fourier Transform

Divide and conquer algorithm used for convolutions and polynomial multiplication. Vector size a is a power of 2.
Time: $\mathcal{O}(n \log n)$ Space: $\mathcal{O}(n)$

```

1 void fft(vector<cd> &a, bool invert){
2     int len = a.size();
3     for(int i = 1, j = 0; i < len; i++){
4         int bit = len >> 1;
5         while(bit & j){
6             j ^= bit;
7             bit >>= 1;
8         }
9         j ^= bit;
10        if(i < j) swap(a[i], a[j]);
11    }
12    for(int l = 2; l <= len; l <= 1) {
13        double ang = 2*PI/l * (invert ? -1: 1);
14        cd wd(cos(ang), sin(ang));
15        for(int i = 0; i < len; i += l){
16            cd w(1);
17            for(int j = 0; j < l/2; j++){
18                cd u = a[i+j], v = a[i+j+l/2];
19                a[i+j] = u+w*v;
20                a[i+j+l/2] = u-w*v;
21                w *= wd;
22            }
23        }
24    }
25    if(invert){
26        for(int i = 0; i < len; i++){
27            a[i] /= len;
28        }
29    }
30 }
```

7.6 Inclusion-Exclusion Principle

TODO: rewrite math statement

```

1 // Exemplo:
2 // Contar numeros de 1 a n divisiveis por uma lista de
3 // primos.
4 int n;
5 vi primes;
6 int factors = primes.size();
7 int total_divisible = 0;
8
9 for (int i = 1; i < (1 << factors); i++) {
10    int current_lcm = 1;
11    int subset_size = 0;
12
13    // calcula lcm do subconjunto
14    for (int j = 0; j < factors; j++) {
15        if (i & (1<<j)) {
16            subset_size++;
17            current_lcm = lcm(current_lcm, primes[j]);
18            if (current_lcm > n) break;
19        }
20    }
21
22    if (current_lcm > n) {
23        continue;
24    }
25
26    int count = n / current_lcm;
27
28    // Aplica o Princípio da Inclusão-Exclusão:
29    // Se o tamanho do subconjunto eh ímpar, adiciona.
30    // Se o tamanho do subconjunto eh par, subtrai.
31    if (subset_size & 1) {
32        total_divisible += count;
33    } else {
34        total_divisible -= count;
35    }
36 }
```

7.7 Mint

```

1 template<ll MOD>
2 struct mint {
3     ll val;
4     mint(ll v = 0) {
5         if (v < 0) v = v % MOD + MOD;
6         if (v >= MOD) v %= MOD;
7         val = v;
8     }
9     mint& operator+=(const mint& other) {
10        val += other.val;
11        if (val >= MOD) val -= MOD;
12        return *this;
13    }
14    mint& operator-=(const mint& other) {
15        val -= other.val;
16        if (val < 0) val += MOD;
17        return *this;
18    }
19    mint& operator*=(const mint& other) {
20        val = (val * other.val) % MOD;
21        return *this;
22    }
23    mint& operator/=(const mint& other) {
24        val = (val * inv(other).val) % MOD;
```

```

25        return *this;
26    }
27    friend mint operator+(mint a, const mint& b) {
28        return a += b;
29    }
30    friend mint operator-(mint a, const mint& b) {
31        return a -= b;
32    }
33    friend mint operator*(mint a, const mint& b) {
34        return a *= b;
35    }
36    friend mint operator/(mint a, const mint& b) {
37        return a /= b;
38    }
39    static mint power(mint b, ll e) {
40        mint ans = 1;
41        while (e > 0) {
42            if (e & 1) ans *= b;
43            b *= b;
44            e /= 2;
45        }
46        return ans;
47    }
48    static mint inv(mint n) { return power(n, MOD - 2);
49    }
50 }
```

7.8 Modular Inverse

If m is prime, can use binary exponentiation to compute a^{p-2} (Fermat's Little Theorem).

This code works for non-prime m , as long as it is coprime to a .

Time: $\mathcal{O}(\log m)$

```

1 int modInverse(int a, int m) {
2     int x, y;
3     int g = extendedGcd(a, m, x, y);
4     if (g != 1) return -1;
5     return (x % m + m) % m;
6 }
```

7.9 Euler's Totient

Returns the amount of numbers smaller than n that are coprime to n . Time: $\mathcal{O}(\sqrt{n})$

```

1 int phi(int n){
2     int ans = n;
3     for (int i = 2; i*i <= n; i++){
4         if (n%i == 0){
5             while(n%i == 0) n/=i;
6             ans -= ans/i;
7         }
8     }
9     if (n>1) ans -= ans/n;
10    return ans;
11 }
```

8 Geometry

8.1 Convex hull - Graham Scan

Time: $\mathcal{O}(n \log n)$

```

1 #define CLOCKWISE -1
2 #define COUNTERCLOCKWISE 1
3 #define INCLUDE_COLLINEAR 0 // pode mudar
4
5 struct Point {
6     ll x, y;
7     bool operator==(Point const& t) const {
8         return x == t.x && y == t.y;
9     }
10 }
11
12 struct Vec {
13     int x, y, z;
14 };
15
16 Vec cross(Vec v1, Vec v2){
17     int x = v1.y*v2.z - v1.z*v2.y;
18     int y = -v1.x*v2.z + v1.z*v2.x;
19     int z = v1.x*v2.y - v1.y*v2.x;
20     return {x,y,z};
21 }
22
23 ll dist2(Point p1, Point p2){
24     int dx = p1.x-p2.x;
25     int dy = p1.y-p2.y;
26     return dx*dx+dy*dy;
27 }
28
29 ll orientation(Point pivot, Point a, Point b){
30     Vec va = {a.x-pivot.x, a.y-pivot.y, 0};
31     Vec vb = {b.x-pivot.x, b.y-pivot.y, 0};
32     Vec v = cross(va,vb);
33     if (v.z < 0) return CLOCKWISE;
34     if (v.z > 0) return COUNTERCLOCKWISE;
35     return 0;
36 }
37
38 bool clock_wise(Point pivot, Point a, Point b) {
39     int o = orientation(pivot, a, b);
40     return o < 0 || (INCLUDE_COLLINEAR && o == 0);
41 }
42
43 bool collinear(Point a, Point b, Point c) { return
44     orientation(a, b, c) == 0; }
45
46 vector<Point> convex_hull(vector<Point> &points, bool
47     counterClockwise) {
48     int n = points.size();
49     Point pivot = *min_element(points.begin(), points.
50         end(), []()(Point a, Point b) {
51         return ii(a.y, a.x) < ii(b.y, b.x);
52     });
53     sort(points.begin(), points.end(), [&](Point a,
54         Point b) {
55         int o = orientation(pivot, a, b);
56         if (o == 0) return dist2(pivot, a) < dist2(
57             pivot, b);
58         return o == CLOCKWISE;
59     });
60
61     if (INCLUDE_COLLINEAR) {
62         int i = n-1;
```

```

59     while (i >= 0 && collinear(pivot, points[i],
60         points.back())) i--;
61     reverse(points.begin() + i + 1, points.end());
62 }
63
64 vector<Point> hull;
65 for (auto p : points) {
66     while (hull.size() > 1 && !clock_wise(hull[hull.size() - 2], hull.back(), p))
67         hull.pop_back();
68     hull.push_back(p);
69 }
70 if (!INCLUDE_COLLINEAR && hull.size() == 2 && hull[0] == hull[1])
71     hull.pop_back();
72 if (counterClockwise && hull.size() > 1) {
73     vector<Point> reversed_hull = hull;
74     reverse(reversed_hull.begin() + 1,
75             reversed_hull.end());
76     return reversed_hull;
77 }
78 }
```

8.2 Basic elements - geometry lib

- Basic elements for using the geometry lib, contains points, vector operations and distances between points, distance between point and segment, distance between segments, segment intersection check, orientation check (ccw).
- Always use long double for floating point. Only use floating point if indispensable.
- For $a == b$, use $|a - b| < \text{eps}!!!!$

Time: $\mathcal{O}(1)$

8.2.1 Polygon Area

- Heron's Formula for triangle area:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

, where a, b, and c are the triangle sides and $s = (a+b+c)/2$

TODO Shoelace

- Pick's Theorem for polygon area with integer coordinates:

$$A = a + b/2 - 1$$

, where a is the number of integer coordinates inside the polygon and b is the number of integer coordinates on the polygon boundary. b can be calculated for each edge as

$$b = \gcd(xi + 1 - xi, yi + 1 - yi) + 1$$

Polygon Area Time: $\mathcal{O}(n)$

8.2.2 Point in polygon

Sum of edge angles relative to the point must sum to 2π
Time: $\mathcal{O}(n \log n)$

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 typedef long double ld;
4 #define eps 1e-9
5 #define pi 3.141592653589
6 #define int long long int
7
8 struct pt {
9     int x, y;
10    int operator==(pt b) {
11        return x == b.x && y == b.y;
12    }
13    int operator<(pt b) {
14        if(x == b.x) return y < b.y;
15        return x < b.x;
16    }
17    pt operator-(pt b) {
18        return {x - b.x, y - b.y};
19    }
20    pt operator+(pt b) {
21        return {x+b.x, y + b.y};
22    }
23 };
24
25 int cross(pt u, pt v) {
26     return u.x * v.y - u.y * v.x;
27 }
28 int dot(pt u, pt v) {
29     return u.x * v.x + u.y * v.y;
30 }
31 ld norm(pt u) {
32     return sqrt(dot(u, u));
33 }
34 ld dist(pt u, pt v) {
35     return norm(u - v);
36 }
37 int ccw(pt u, pt v) { // cuidado com colineares!!!!
38     return (cross(u, v) > eps)?1:(fabs(cross(u, v)) <
39         eps)?0:-1;
40 }
41 int pointInSegment(pt a, pt u, pt v) { // checks if a
42     lies in uv
43     if(ccw(v - u, a - u)) return 0;
```

```

44     vector<pt> pts = {a, u, v};
45     sort(pts.begin(), pts.end());
46     return pts[1] == a;
47 }
48 ld angle(pt u, pt v) { // angle between two vectors
49     ld c = cross(u, v);
50     ld d = dot(u,v);
51     return atan2l(c, d);
52 }
53 int intersect(pt sa, pt sb, pt ra, pt rb) { // not sure
54     if it works when one of the segments is a point
55     pt s = sb - sa, r = rb - ra;
56     if(pointInSegment(sa, ra, rb) || pointInSegment(sb,
57         ra, rb) || pointInSegment(ra, sa, sb) ||
58         pointInSegment(rb, sa, sb)) return 1;
59     return !(ccw(s, ra - sa) == ccw(s, rb - sa) || ccw(
60         r, sa - ra) == ccw(r, sb - ra));
61 }
62
63 ld polygonArea(vector<pt>& p) { // not signed (for
64     signed area remove the absolute value at the end)
65     ld area = 0;
66     int n = p.size() - 1; // p[n] = p[0]
67     for(int i = 0; i < n; i++) {
68         area += cross(p[i], p[i + 1]);
69     }
70     return fabs(area)/2;
71 }
72
73 int pointInPolygon(pt a, vector<pt>& p) { // returns 0
74     for point in BOUNDARY, 1 for point in polygon and
75     -1 for outside
76     ld total = 0;
77     int n = p.size() - 1;
78     for(int i = 0; i < n; i++) {
79         pt u = p[i] - a;
80         pt v = p[i + 1] - a;
81         if(fabs(dist(p[i], a) + dist(p[i + 1], a) -
82             dist(p[i], p[i + 1])) < eps) {
83             return 0;
84         }
85         total += angle(u, v);
86     }
87
88     return (fabs(fabs(total) - 2 * pi) < eps)?1:-1;
89 }
90
91
92
93
94
95 }
```