

University of Tartu ICPC Team Notebook (2017-2018)

October 30, 2017

Contents

1	gcc ordered set	1
2	Triangle centers	1
3	2D line segment	2
4	Dinic	4
5	Min cost max flow $O(\text{flow} \cdot n^2)$	7
6	Aho Corasick $O(\alpha \sum \text{len})$	8
7	Suffix automaton $O((n + q) \log(\alpha))$	9
8	Templated multi dimensional BIT $O(\log(n)^{\text{dim}})$	10
9	Templated HLD	11
10	Templated Persistent Segment Tree	13
11	FFT $O(n \log(n))$	14
12	MOD int, extended Euclidean	16

1 gcc ordered set

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T>
5 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
6 int main(){
7     ordered_set<int> cur;
8     cur.insert(1);
9     cur.insert(3);
10    cout << cur.order_of_key(2) << endl; // the number of elements in the set less than 2
11    cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in the set(0-based)
12    cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in the set(0-based)
13 }
```

2 Triangle centers

```

1 const double min_delta = 1e-13;
2 const double coord_max = 1e6;
3 typedef complex < double > point;
4 point A, B, C; // vertices of the triangle
5 bool collinear(){
6     double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7     if(min_diff < coord_max * min_delta)
8         return true;
9     point sp = (B - A) / (C - A);
10    double ang = M_PI/2 - abs(arg(sp)) - M_PI/2; //positive angle with the real line
11    return ang < min_delta;
12 }
13 point circum_center(){
14     if(collinear())
15         return point(NAN, NAN);
16     //squared lengths of sides
17     double a2, b2, c2;
18     a2 = norm(B - C);
19     b2 = norm(A - C);
20     c2 = norm(A - B);
```

```

21 //barycentric coordinates of the circumcenter
22 double c_A, c_B, c_C;
23 c_A = a2 * (b2 + c2 - a2); //sin(2 * alpha) may be used as well
24 c_B = b2 * (a2 + c2 - b2);
25 c_C = c2 * (a2 + b2 - c2);
26 double sum = c_A + c_B + c_C;
27 c_A /= sum;
28 c_B /= sum;
29 c_C /= sum;
30 // cartesian coordinates of the circumcenter
31 return c_A * A + c_B * B + c_C * C;
32 }
33 point centroid(){ //center of mass
34     return (A + B + C) / 3.0;
35 }
36 point ortho_center(){ //euler line
37     point O = circum_center();
38     return O + 3.0 * (centroid() - O);
39 };
40 point nine_point_circle_center(){ //euler line
41     point O = circum_center();
42     return O + 1.5 * (centroid() - O);
43 };
44 point in_center(){
45     if(collinear())
46         return point(NAN,NAN);
47     double a, b, c; //side lengths
48     a = abs(B - C);
49     b = abs(A - C);
50     c = abs(A - B);
51     //trilinear coordinates are (1,1,1)
52     //barycentric coordinates
53     double c_A = a, c_B = b, c_C = c;
54     double sum = c_A + c_B + c_C;
55     c_A /= sum;
56     c_B /= sum;
57     c_C /= sum;
58     // cartesian coordinates of the incenter
59     return c_A * A + c_B * B + c_C * C;
60 }

```

3 2D line segment

```

1 const long double PI = acos(-1.0L);
2
3 struct Vec {
4     long double x, y;
5
6     Vec& operator--=(Vec r) {
7         x -= r.x, y -= r.y;
8         return *this;
9     }
10    Vec operator-(Vec r) {return Vec(*this) -= r;}
11
12    Vec& operator+=(Vec r) {
13        x += r.x, y += r.y;
14        return *this;
15    }
16    Vec operator+(Vec r) {return Vec(*this) += r;}
17    Vec operator-() {return {-x, -y};}
18    Vec& operator*=(long double r) {
19        x *= r, y *= r;
20        return *this;
21    }
22    Vec operator*(long double r) {return Vec(*this) *= r;}
23    Vec& operator/=(long double r) {
24        x /= r, y /= r;
25        return *this;
26    }
27    Vec operator/(long double r) {return Vec(*this) /= r;}
28
29    long double operator*(Vec r) {
30        return x * r.x + y * r.y;

```

```

31 }
32 };
33 ostream& operator<<(ostream& l, Vec r) {
34     return l << '(' << r.x << ", " << r.y << ')';
35 }
36 long double len(Vec a) {
37     return hypot(a.x, a.y);
38 }
39 long double cross(Vec l, Vec r) {
40     return l.x * r.y - l.y * r.x;
41 }
42 long double angle(Vec a) {
43     return fmod(atan2(a.y, a.x)+2*PI, 2*PI);
44 }
45 Vec normal(Vec a) {
46     return Vec({-a.y, a.x}) / len(a);
47 }

```

```

1 struct Segment {
2     Vec a, b;
3     Vec d() {
4         return b-a;
5     }
6 };
7 ostream& operator<<(ostream& l, Segment r) {
8     return l << r.a << '-' << r.b;
9 }
10
11 Vec intersection(Segment l, Segment r) {
12     Vec dl = l.d(), dr = r.d();
13     if(cross(dl, dr) == 0)
14         return {nanl(""), nanl("")};
15
16     long double h = cross(dr, l.a-r.a) / len(dr);
17     long double dh = cross(dr, dl) / len(dr);
18
19     return l.a + dl * (h / -dh);
20 }
21
22 //Returns the area bounded by halfplanes
23 long double getArea(vector<Segment> lines) {
24     long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;
25
26     vector<Segment> linesBySide[2];
27     for(auto line : lines) {
28         if(line.b.y == line.a.y) {
29             if(line.a.x < line.b.x)
30                 lowerbound = max(lowerbound, line.a.y);
31             else
32                 upperbound = min(upperbound, line.a.y);
33         }
34         else if(line.a.y < line.b.y)
35             linesBySide[1].push_back(line);
36         else
37             linesBySide[0].push_back({line.b, line.a});
38     }
39
40     sort(linesBySide[0].begin(), linesBySide[0].end(), [](Segment l, Segment r) {
41         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a > normal(r.d())*r.a;
42         return cross(l.d(), r.d()) < 0;
43     });
44     sort(linesBySide[1].begin(), linesBySide[1].end(), [](Segment l, Segment r) {
45         if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a < normal(r.d())*r.a;
46         return cross(l.d(), r.d()) > 0;
47     });
48
49     //Now find the application area of the lines and clean up redundant ones
50     vector<long double> applyStart[2];
51     for(int side = 0; side < 2; side++) {
52         vector<long double> &apply = applyStart[side];
53         vector<Segment> curLines;
54

```

```

55     for(auto line : linesBySide[side]) {
56         while(curLines.size() > 0) {
57             Segment other = curLines.back();
58
59             if(cross(line.d(), other.d()) != 0) {
60                 long double start = intersection(line, other).y;
61                 if(start > apply.back())
62                     break;
63             }
64
65             curLines.pop_back();
66             apply.pop_back();
67         }
68
69         if(curLines.size() == 0)
70             apply.push_back(-HUGE_VALL);
71         else
72             apply.push_back(intersection(line, curLines.back()).y);
73         curLines.push_back(line);
74     }
75
76     linesBySide[side] = curLines;
77 }
78 applyStart[0].push_back(HUGE_VALL);
79 applyStart[1].push_back(HUGE_VALL);
80
81 long double result = 0;
82 {
83     long double lb = -HUGE_VALL, ub;
84     for(int i=0, j=0; i < (int)linesBySide[0].size() && j < (int)linesBySide[1].size(); lb = ub) {
85         ub = min(applyStart[0][i+1], applyStart[1][j+1]);
86
87         long double alb = lb, aub = ub;
88         Segment l0 = linesBySide[0][i], l1 = linesBySide[1][j];
89
90         if(cross(l1.d(), l0.d()) > 0)
91             alb = max(alb, intersection(l0, l1).y);
92         else if(cross(l1.d(), l0.d()) < 0)
93             aub = min(aub, intersection(l0, l1).y);
94         alb = max(alb, lowerbound);
95         aub = min(aub, upperbound);
96         aub = max(aub, alb);
97
98         {
99             long double x1 = l0.a.x + (alb - l0.a.y) / l0.d().y * l0.d().x;
100             long double x2 = l0.a.x + (aub - l0.a.y) / l0.d().y * l0.d().x;
101             result -= (aub - alb) * (x1 + x2) / 2;
102         }
103         {
104             long double x1 = l1.a.x + (alb - l1.a.y) / l1.d().y * l1.d().x;
105             long double x2 = l1.a.x + (aub - l1.a.y) / l1.d().y * l1.d().x;
106             result += (aub - alb) * (x1 + x2) / 2;
107         }
108
109         if(applyStart[0][i+1] < applyStart[1][j+1])
110             i++;
111         else
112             j++;
113     }
114 }
115 return result;
116 }

```

4 Dinic

```

1 struct MaxFlow{
2     typedef long long ll;
3     const ll INF = 1e18;
4     struct Edge{
5         int u,v;
6         ll c,rc;
7         shared_ptr<ll> flow;
8         pair<int,int> id() const {

```

```

9         return make_pair(min(u,v),max(u,v));
10    }
11    Edge(int _u, int _v, ll _c, ll _rc = 0):u(_u),v(_v),c(_c),rc(_rc){
12    }
13    void join(const Edge &t){
14        if(u == t.u){
15            c += t.c;
16            rc += t.rc;
17        }
18        else{
19            c += t.rc;
20            rc += t.c;
21        }
22    }
23 };
24 struct FlowTracker{
25     shared_ptr<ll> flow;
26     ll cap, rcap;
27     bool dir;
28     FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
29     ↪ _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
30     ll rem() const {
31         if(dir == 0){
32             return cap-*flow;
33         }
34         else{
35             return rcap+*flow;
36         }
37     }
38     void add_flow(ll f){
39         if(dir == 0)
40             *flow += f;
41         else
42             *flow -= f;
43         assert(*flow <= cap);
44         assert(-*flow <= rcap);
45     }
46     operator ll() const { return rem(); }
47     void operator--(ll x){ add_flow(x); }
48     void operator+=(ll x){ add_flow(-x); }
49 };
50 int source,sink;
51 vector<vector<int> > adj;
52 vector<vector<FlowTracker> > cap;
53 vector<Edge> edges;
54 MaxFlow(int _source, int _sink):source(_source),sink(_sink){
55     assert(source != sink);
56 }
57 int add_edge(Edge e){
58     edges.push_back(e);
59     return edges.size()-1;
60 }
61 int add_edge(int u, int v, ll c, ll rc = 0){
62     return add_edge(Edge(u,v,c,rc));
63 }
64 void group_edges(){
65     map<pair<int,int>,vector<Edge> > edge_groups;
66     for(auto edge: edges)
67         if(edge.u != edge.v)
68             edge_groups[edge.id()].push_back(edge);
69     vector<Edge> grouped_edges;
70     for(auto group: edge_groups){
71         Edge main_edge = group.second[0];
72         for(int i = 1; i < group.second.size(); ++i)
73             main_edge.join(group.second[i]);
74         grouped_edges.push_back(main_edge);
75     }
76     edges = grouped_edges;
77 }
78 vector<int> now,lvl;
79 void prep(){
80     int max_id = max(source,sink);

```

```

81     for(auto edge : edges)
82         max_id = max(max_id,max(edge.u,edge.v));
83     adj.resize(max_id+1);
84     cap.resize(max_id+1);
85     now.resize(max_id+1);
86     lvl.resize(max_id+1);
87     for(auto &edge : edges){
88         auto flow = make_shared<ll>(0);
89         adj[edge.u].push_back(edge.v);
90         cap[edge.u].push_back(FlowTracker(edge.c,edge.rc,flow,0));
91         adj[edge.v].push_back(edge.u);
92         cap[edge.v].push_back(FlowTracker(edge.c,edge.rc,flow,1));
93         assert(cap[edge.u].back() == edge.c);
94         edge.flow = flow;
95     }
96 }
97 bool dinic_bfs(){
98     fill(now.begin(),now.end(),0);
99     fill(lvl.begin(),lvl.end(),0);
100    lvl[source] = 1;
101    vector<int> bfs(1,source);
102    for(int i = 0; i < bfs.size(); ++i){
103        int u = bfs[i];
104        for(int j = 0; j < adj[u].size(); ++j){
105            int v = adj[u][j];
106            if(cap[u][j] > 0 && lvl[v] == 0){
107                lvl[v] = lvl[u]+1;
108                bfs.push_back(v);
109            }
110        }
111    }
112    return lvl[sink] > 0;
113 }
114 ll dinic_dfs(int u, ll flow){
115     if(u == sink)
116         return flow;
117     while(now[u] < adj[u].size()){
118         int v = adj[u][now[u]];
119         if(lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0){
120             ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
121             if(res > 0){
122                 cap[u][now[u]] -= res;
123                 return res;
124             }
125         }
126         ++now[u];
127     }
128     return 0;
129 }
130 ll calc(){
131     prep();
132     ll ans = 0;
133     while(dinic_bfs()){
134         ll cur = 0;
135         do{
136             cur = dinic_dfs(source,INF);
137             ans += cur;
138         }while(cur > 0);
139     }
140     return ans;
141 }
142 };
143 int main(){
144     int n,m;
145     cin >> n >> m;
146     auto mf = MaxFlow(1,n); // arguments source and sink, memory usage O(largest node index), sink doesn't need
147     ↳ to be last
148     int edge_index;
149     for(int i = 0; i < m; ++i){
150         int a,b,c;
151         cin >> a >> b >> c;
152         //undirected edge is a pair of edges (a,b,c,0) and (a,b,0,c)
153         edge_index = mf.add_edge(a,b,c,c); //store edge index if care about flow value

```

```

153 }
154 mf.group_edges(); // small auxillary to remove multiple edges, only use this if we need to know TOTAL FLOW
    ↪ ONLY
155 cout << mf.calc() << '\n';
156 //cout << *mf.edges[edge_index].flow << '\n'; // ONLY if group_edges() WAS NOT CALLED
157 }

```

5 Min cost max flow $O(\text{flow} \cdot n^2)$

```

1 const int nmax=1055;
2 const ll inf=1e14;
3 int t, n, v; //0 is source, v-1 sink
4 ll rem_flow[nmax][nmax]; //set [x][y] for directed capacity from x to y.
5 ll cost[nmax][nmax]; //set [x][y] for directed cost from x to y. SET TO inf IF NOT USED
6 ll min_dist[nmax];
7 int prev_node[nmax];
8 ll node_flow[nmax];
9 bool visited[nmax];
10 ll tot_cost, tot_flow; //output
11 void min_cost_max_flow(){
12     tot_cost=0;
13     tot_flow=0;
14     ll sink_pot=0;
15     while(true){
16         for(int i=0; i<=v; ++i){
17             min_dist[i]=inf;
18             visited[i]=false;
19         }
20         min_dist[0]=0;
21         node_flow[0]=inf;
22         int min_node;
23         while(true){
24             int min_node=v;
25             for(int i=0; i<v; ++i){
26                 if((!visited[i]) && min_dist[i]<min_dist[min_node]){
27                     min_node=i;
28                 }
29             }
30             if(min_node==v){
31                 break;
32             }
33             visited[min_node]=true;
34             for(int i=0; i<v; ++i){
35                 if((!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){
36                     min_dist[i]=min_dist[min_node]+cost[min_node][i];
37                     prev_node[i]=min_node;
38                     node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
39                 }
40             }
41         }
42         if(min_dist[v-1]==inf){
43             break;
44         }
45         for(int i=0; i<v; ++i){
46             for(int j=0; j<v; ++j){
47                 if(cost[i][j]!=inf){
48                     cost[i][j]+=min_dist[i];
49                     cost[i][j]-=min_dist[j];
50                 }
51             }
52         }
53         sink_pot+=min_dist[v-1];
54         tot_flow+=node_flow[v-1];
55         tot_cost+=sink_pot*node_flow[v-1];
56         int cur=v-1;
57         while(cur!=0){
58             rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
59             rem_flow[cur][prev_node[cur]]+=node_flow[v-1];
60             cost[cur][prev_node[cur]]=0;
61             if(rem_flow[prev_node[cur]][cur]==0){
62                 cost[prev_node[cur]][cur]=inf;
63             }
64             cur=prev_node[cur];

```

```

65     }
66 }
67 }

```

6 Aho Corasick $O(|\alpha| \sum \text{len})$

```

1  const int alpha_size=26;
2  struct node{
3      node *nxt[alpha_size]; //May use other structures to move in trie
4      node *suffix;
5      node(){
6          memset(nxt, 0, alpha_size*sizeof(node *));
7      }
8      int cnt=0;
9  };
10 node *aho_corasick(vector<vector<char> > &dict){
11     node *root= new node;
12     root->suffix = 0;
13     vector<pair<vector<char> *, node *> > cur_state;
14     for(vector<char> &s : dict)
15         cur_state.emplace_back(&s, root);
16     for(int i=0; !cur_state.empty(); ++i){
17         vector<pair<vector<char> *, node *> > nxt_state;
18         for(auto &cur : cur_state){
19             node *nxt=cur.second->nxt[(*(cur.first)[i]);
20             if(nxt){
21                 cur.second=nxt;
22             }else{
23                 nxt = new node;
24                 cur.second->nxt[(*(cur.first)[i]] = nxt;
25                 node *suf = cur.second->suffix;
26                 cur.second = nxt;
27                 nxt->suffix = root; //set correct suffix link
28                 while(suf){
29                     if(suf->nxt[(*(cur.first)[i]]){
30                         nxt->suffix = suf->nxt[(*(cur.first)[i]);
31                         break;
32                     }
33                     suf=suf->suffix;
34                 }
35             }
36             if(cur.first->size() > i+1)
37                 nxt_state.push_back(cur);
38         }
39         cur_state=nxt_state;
40     }
41     return root;
42 }
43 //auxiliary functions for searching and counting
44 node *walk(node *cur, char c){ //longest prefix in dict that is suffix of walked string.
45     while(true){
46         if(cur->nxt[c])
47             return cur->nxt[c];
48         if(!cur->suffix){
49             return cur;
50         }
51         cur = cur->suffix;
52     }
53 }
54 void cnt_matches(node *root, vector<char> &match_in){
55     node *cur = root;
56     for(char c : match_in){
57         cur = walk(cur, c);
58         ++cur->cnt;
59     }
60 }
61 void add_cnt(node *root){ //After counting matches propagete ONCE to suffixes for final counts
62     vector<node *> to_visit = {root};
63     for(int i=0; i<to_visit.size(); ++i){
64         node *cur = to_visit[i];
65         for(int j=0; j<alpha_size; ++j){
66             if(cur->nxt[j]){
67                 to_visit.push_back(cur->nxt[j]);

```



```

68     }
69 }
70 }
71 for(int i=to_visit.size()-1; i>0; --i){
72     to_visit[i]->suffix->cnt += to_visit[i]->cnt;
73 }
74 }

```

7 Suffix automaton $O((n + q) \log(|\alpha|))$

```

1 class AutoNode {
2     private:
3         map< char, AutoNode * > nxt_char;    // Map is faster than hashtable and unsorted arrays
4     public:
5         int len; //Length of longest suffix in equivalence class.
6         AutoNode *suf;
7         bool has_nxt(char c) const {
8             return nxt_char.count(c);
9         }
10        AutoNode *nxt(char c) {
11            if (!has_nxt(c))
12                return NULL;
13            return nxt_char[c];
14        }
15        void set_nxt(char c, AutoNode *node) {
16            nxt_char[c] = node;
17        }
18        AutoNode *split(int new_len, char c) {
19            AutoNode *new_n = new AutoNode;
20            new_n->nxt_char = nxt_char;
21            new_n->len = new_len;
22            new_n->suf = suf;
23            suf = new_n;
24            return new_n;
25        }
26        // Extra functions for matching and counting
27        AutoNode *lower_depth(int depth) { //move to longest suffix of current with a maximum length of depth.
28            if (suf->len >= depth)
29                return suf->lower_depth(depth);
30            return this;
31        }
32        AutoNode *walk(char c, int depth, int &match_len) { //move to longest suffix of walked path that is a
33            ↳ substring
34            match_len = min(match_len, len); //includes depth limit(needed for finding matches)
35            if (has_nxt(c)) { //as suffixes are in classes match_len must be tracte
36                ↳ eternally
37                ++match_len;
38                return nxt(c)->lower_depth(depth);
39            }
40            if (suf)
41                return suf->walk(c, depth, match_len);
42            return this;
43        }
44        int paths_to_end = 0;
45        void set_as_end() { //All suffixes of current node are marked as ending nodes.
46            paths_to_end = 1;
47            if (suf) suf->set_as_end();
48        }
49        bool vis = false;
50        void calc_paths_to_end() { //Call ONCE from ROOT. For each node calculates number of ways to reach an end
51            ↳ node.
52            if (!vis) { //paths_to_end is ocurence count for any strings in current suffix equivalence
53                ↳ class.
54                vis = true;
55                for (auto cur : nxt_char) {
56                    cur.second->calc_paths_to_end();
57                    paths_to_end += cur.second->paths_to_end;
58                }
59            }
60        }
61    };
62 };
63 struct SufAutomaton {
64     AutoNode *last;

```

```

60 AutoNode *root;
61 void extend(char new_c) {
62     AutoNode *new_end = new AutoNode; // The equivalence class containing the whole new string
63     new_end->len = last->len + 1;
64     AutoNode *suf_w_nxt = last; // The whole old string class
65     while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) { // is turned into the longest suffix which
66                                                         // can be turned into a substring of old state
67                                                         // by appending new_c
68         suf_w_nxt->set_nxt(new_c, new_end);
69         suf_w_nxt = suf_w_nxt->suf;
70     }
71     if (!suf_w_nxt) { // The new character isn't part of the old string
72         new_end->suf = root;
73     } else {
74         AutoNode *max_sbstr = suf_w_nxt->nxt(new_c); // Equivalence class containing longest
75                                                         // substring which is a suffix of the new state.
76         if (suf_w_nxt->len + 1 == max_sbstr->len) { // Check whether splitting is needed
77             new_end->suf = max_sbstr;
78         } else {
79             AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
80             new_end->suf = eq_sbstr;
81             // Make suffixes of suf_w_nxt point to eq_sbstr instead of max_sbstr
82             AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
83             while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
84                 w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
85                 w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
86             }
87         }
88     }
89     last = new_end;
90 }
91 SufAutomaton(string to_suffix) {
92     root = new AutoNode;
93     root->len = 0;
94     root->suf = NULL;
95     last = root;
96     for (char c : to_suffix) extend(c);
97 }
98 };

```

8 Templated multi dimensional BIT $O(\log(n)^{\dim})$

```

1 // Fully overloaded any dimensional BIT, use any type for coordinates, elements, return_value.
2 // Includes coordinate compression.
3 template < typename elem_t, typename coord_t, coord_t n_inf, typename ret_t >
4 class BIT {
5     vector< coord_t > positions;
6     vector< elem_t > elems;
7     bool initiated = false;
8
9 public:
10     BIT() {
11         positions.push_back(n_inf);
12     }
13     void initiate() {
14         if (initiated) {
15             for (elem_t &c_elem : elems)
16                 c_elem.initiate();
17         } else {
18             initiated = true;
19             sort(positions.begin(), positions.end());
20             positions.resize(unique(positions.begin(), positions.end()) - positions.begin());
21             elems.resize(positions.size());
22         }
23     }
24     template < typename... loc_form >
25     void update(coord_t cord, loc_form... args) {
26         if (initiated) {
27             int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
28             for (; pos < positions.size(); pos += pos & -pos)
29                 elems[pos].update(args...);
30         } else {
31             positions.push_back(cord);

```

```

32     }
33 }
34 template < typename... loc_form >
35 ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
36     ret_t res = 0;
37     int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
38     for (; pos > 0; pos -= pos & -pos)
39         res += elems[pos].query(args...);
40     return res;
41 }
42 };
43 template < typename internal_type >
44 struct wrapped {
45     internal_type a = 0;
46     void update(internal_type b) {
47         a += b;
48     }
49     internal_type query() {
50         return a;
51     }
52     // Should never be called, needed for compilation
53     void initiate() {
54         cerr << 'i' << endl;
55     }
56     void update() {
57         cerr << 'u' << endl;
58     }
59 };
60 int main() {
61     // return type should be same as type inside wrapped
62     BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll > fenwick;
63     int dim = 2;
64     vector< tuple< int, int, ll > > to_insert;
65     to_insert.emplace_back(1, 1, 1);
66     // set up all positions that are to be used for update
67     for (int i = 0; i < dim; ++i) {
68         for (auto &cur : to_insert)
69             fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include value which won't be used
70         fenwick.initiate();
71     }
72     // actual use
73     for (auto &cur : to_insert)
74         fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
75     cout << fenwick.query(2, 2)<<'\n';
76 }

```

9 Templated HLD

```

1 class dummy {
2 public:
3     dummy () {
4     }
5
6     dummy (int, int) {
7     }
8
9     void set (int, int) {
10    }
11
12    int query (int left, int right) {
13        cout << this << ' ' << left << ' ' << right << endl;
14    }
15 };
16
17 /* T should be the type of the data stored in each vertex;
18  * DS should be the underlying data structure that is used to perform the
19  * group operation. It should have the following methods:
20  * * DS () - empty constructor
21  * * DS (int size, T initial) - constructs the structure with the given size,
22  *   initially filled with initial.
23  * * void set (int index, T value) - set the value at index `index` to `value`
24  * * T query (int left, int right) - return the "sum" of elements between left and right, inclusive.
25  */


```

```

26 template<typename T, class DS>
27 class HLD {
28     int vertexc;
29     vector<int> *adj;
30     vector<int> subtree_size;
31     DS structure;
32     DS aux;
33
34     void build_sizes (int vertex, int parent) {
35         subtree_size[vertex] = 1;
36         for (int child : adj[vertex]) {
37             if (child != parent) {
38                 build_sizes(child, vertex);
39                 subtree_size[vertex] += subtree_size[child];
40             }
41         }
42     }
43
44     int cur;
45     vector<int> ord;
46     vector<int> chain_root;
47     vector<int> par;
48     void build_hld (int vertex, int parent, int chain_source) {
49         cur++;
50         ord[vertex] = cur;
51         chain_root[vertex] = chain_source;
52         par[vertex] = parent;
53
54         if (adj[vertex].size() > 1) {
55             int big_child, big_size = -1;
56             for (int child : adj[vertex]) {
57                 if ((child != parent) &&
58                     (subtree_size[child] > big_size)) {
59                     big_child = child;
60                     big_size = subtree_size[child];
61                 }
62             }
63
64             build_hld(big_child, vertex, chain_source);
65             for (int child : adj[vertex]) {
66                 if ((child != parent) && (child != big_child)) {
67                     build_hld(child, vertex, child);
68                 }
69             }
70         }
71     }
72
73 public:
74     HLD (int _vertexc) {
75         vertexc = _vertexc;
76         adj = new vector<int> [vertexc + 5];
77     }
78
79     void add_edge (int u, int v) {
80         adj[u].push_back(v);
81         adj[v].push_back(u);
82     }
83
84     void build (T initial) {
85         subtree_size = vector<int> (vertexc + 5);
86         ord = vector<int> (vertexc + 5);
87         chain_root = vector<int> (vertexc + 5);
88         par = vector<int> (vertexc + 5);
89         cur = 0;
90         build_sizes(1, -1);
91         build_hld(1, -1, 1);
92         structure = DS (vertexc + 5, initial);
93         aux = DS (50, initial);
94     }
95
96     void set (int vertex, int value) {
97         structure.set(ord[vertex], value);
98     }

```

```

99
100 T query_path (int u, int v) { /* returns the "sum" of the path u->v */
101     int cur_id = 0;
102     while (chain_root[u] != chain_root[v]) {
103         if (ord[u] > ord[v]) {
104             cur_id++;
105             aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
106             u = par[chain_root[u]];
107         } else {
108             cur_id++;
109             aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
110             v = par[chain_root[v]];
111         }
112     }
113
114     cur_id++;
115     aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
116
117     return aux.query(1, cur_id);
118 }
119
120 void print () {
121     for (int i = 1; i <= vertexc; i++) {
122         cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' << par[i] << endl;
123     }
124 }
125 };
126
127 int main () {
128     int vertexc;
129     cin >> vertexc;
130
131     HLD<int, dummy> hld (vertexc);
132     for (int i = 0; i < vertexc - 1; i++) {
133         int u, v;
134         cin >> u >> v;
135
136         hld.add_edge(u, v);
137     }
138     hld.build(0);
139     hld.print();
140
141     int queryc;
142     cin >> queryc;
143     for (int i = 0; i < queryc; i++) {
144         int u, v;
145         cin >> u >> v;
146
147         hld.query_path(u, v);
148         cout << endl;
149     }
150 }

```

10 Templated Persistent Segment Tree

```

1 template<typename T, typename comp>
2 class PersistentST {
3     struct Node {
4         Node *left, *right;
5         int lend, rend;
6         T value;
7
8         Node (int position, T _value) {
9             left = NULL;
10            right = NULL;
11            lend = position;
12            rend = position;
13            value = _value;
14        }
15
16        Node (Node *_left, Node *_right) {
17            left = _left;
18            right = _right;

```

```

19     lend = left->lend;
20     rend = right->rend;
21     value = comp()(left->value, right->value);
22 }
23
24 T query (int qlleft, int qright) {
25     qlleft = max(qlleft, lend);
26     qright = min(qright, rend);
27
28     if (qlleft == lend && qright == rend) {
29         return value;
30     } else if (qlleft > qright) {
31         return comp().identity;
32     } else {
33         return comp()(left->query(qlleft, qright),
34                         right->query(qlleft, qright));
35     }
36 }
37 };
38
39 int size;
40 Node **tree;
41 vector<Node*> roots;
42 public:
43 PersistentST () {
44 }
45
46 PersistentST (int _size, T initial) {
47     for (int i = 0; i < 32; i++) {
48         if ((1 << i) > _size) {
49             size = 1 << i;
50             break;
51         }
52     }
53
54     tree = new Node* [2 * size + 5];
55
56     for (int i = size; i < 2 * size; i++) {
57         tree[i] = new Node (i - size, initial);
58     }
59
60     for (int i = size - 1; i > 0; i--) {
61         tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
62     }
63
64     roots = vector<Node*> (1, tree[1]);
65 }
66
67 void set (int position, T _value) {
68     tree[size + position] = new Node (position, _value);
69     for (int i = (size + position) / 2; i >= 1; i /= 2) {
70         tree[i] = new Node (tree[2 * i], tree[2 * i + 1]);
71     }
72     roots.push_back(tree[1]);
73 }
74
75 int last_revision () {
76     return (int) roots.size() - 1;
77 }
78
79 T query (int qlleft, int qright, int revision) {
80     return roots[revision]->query(qlleft, qright);
81 }
82
83 T query (int qlleft, int qright) {
84     return roots[last_revision()]->query(qlleft, qright);
85 }
86 };

```

11 FFT $O(n \log(n))$

```

1 //Assumes a is a power of two
2 vector<complex<long double> > fastFourierTransform(vector<complex<long double> > a, bool inverse) {

```

```

3  const long double PI = acos(-1.0L);
4  int n = a.size();
5  //Precalculate w
6  vector<complex<long double> > w(n, 0.0L);
7  w[0] = 1;
8  for(int tpow = 1; tpow < n; tpow *= 2)
9      w[tpow] = polar(1.0L, 2*PI * tpow/n * (inverse ? -1 : 1) );
10 for(int i=3, last = 2; i<n; i++) {
11     if(w[i] == 0.0L)
12         w[i] = w[last] * w[i-last];
13     else
14         last = i;
15 }
16
17 //Rearrange a
18 for(int block = n; block > 1; block /= 2) {
19     int half = block/2;
20     vector<complex<long double> > na(n);
21     for(int s=0; s < n; s += block)
22         for(int i=0; i<block; i++)
23             na[s + half*(i%2) + i/2] = a[s+i];
24     a = na;
25 }
26
27 //Now do the calculation
28 for(int block = 2; block <= n; block *= 2) {
29     vector<complex<long double> > na(n);
30     int wb = n/block, half = block/2;
31
32     for(int s=0; s < n; s += block)
33         for(int i=0; i<half; i++) {
34             na[s+i] = a[s+i] + w[wb*i] * a[s+half+i];
35             na[s+half+i] = a[s+i] - w[wb*i] * a[s+half+i];
36         }
37     a = na;
38 }
39
40 return a;
41 }
42
43
44 struct Polynomial {
45     vector<long double> a;
46
47     long double& operator[](int ind) {
48         return a[ind];
49     }
50
51     Polynomial& operator*=(long double r) {
52         for(auto &c : a)
53             c *= r;
54         return *this;
55     }
56     Polynomial operator*(long double r) {return Polynomial(*this) *= r;}
57
58     Polynomial& operator/=(long double r) {
59         for(auto &c : a)
60             c /= r;
61         return *this;
62     }
63     Polynomial operator/(long double r) {return Polynomial(*this) /= r;}
64
65     Polynomial& operator+=(Polynomial r) {
66         if(a.size() < r.a.size())
67             a.resize(r.a.size(), 0.0L);
68         for(int i=0; i<(int)r.a.size(); i++)
69             a[i] += r[i];
70         return *this;
71     }
72     Polynomial operator+(Polynomial r) {return Polynomial(*this) += r;}
73
74     Polynomial& operator-=(Polynomial r) {
75         if(a.size() < r.a.size())

```

```

76     a.resize(r.a.size(), 0.0L);
77     for(int i=0;i<(int)r.a.size();i++)
78         a[i] -= r[i];
79     return *this;
80 }
81 Polynomial operator-(Polynomial r) {return Polynomial(*this) -= r;}
82
83 Polynomial operator*(Polynomial r) {
84     int n = 1;
85     while(n < (int)(a.size() + r.a.size() - 1) )
86         n *= 2;
87
88     vector<complex<long double>> fl(n, 0.0L), fr(n, 0.0L);
89     for(int i=0;i<(int)a.size();i++)
90         fl[i] = a[i];
91     for(int i=0;i<(int)r.a.size();i++)
92         fr[i] = r[i];
93
94     fl = fastFourierTransform(fl, false);
95     fr = fastFourierTransform(fr, false);
96
97     vector<complex<long double>> ret(n);
98     for(int i=0;i<n;i++)
99         ret[i] = fl[i] * fr[i];
100    ret = fastFourierTransform(ret, true);
101
102    Polynomial result;
103    result.a.resize(a.size() + r.a.size() - 1);
104    for(int i=0;i<(int)result.a.size();i++)
105        result[i] = ret[i].real() / n;
106    return result;
107 }
108 };

```

12 MOD int, extended Euclidean

```

1 pair<int, int> extendedEuclideanAlgorithm(int a, int b) {
2     if(b == 0)
3         return make_pair(1, 0);
4     pair<int, int> ret = extendedEuclideanAlgorithm(b, a%b);
5     return {ret.second, ret.first - a/b * ret.second};
6 }
7
8
9 struct Modint {
10     static const int MOD = 1000000007;
11     int val;
12
13     Modint(int nval = 0) {
14         val = nval;
15     }
16
17     Modint& operator+=(Modint r) {
18         val = (val + r.val) % MOD;
19         return *this;
20     }
21     Modint operator+(Modint r) {return Modint(*this) += r;}
22
23     Modint& operator-=(Modint r) {
24         val = (val + MOD - r.val) % MOD;
25         return *this;
26     }
27     Modint operator-(Modint r) {return Modint(*this) -= r;}
28
29     Modint& operator*=(Modint r) {
30         val = 1LL * val * r.val % MOD;
31         return *this;
32     }
33     Modint operator*(Modint r) {return Modint(*this) *= r;}
34
35     Modint inverse() {
36         int ret = extendedEuclideanAlgorithm(val, MOD).first;
37         if(ret < 0)

```

```
38     ret += MOD;
39     return ret;
40 }
41
42 Modint& operator/=(Modint r) {
43     return operator*=(r.inverse() );
44 }
45 Modint operator/(Modint r) {return Modint(*this) /= r;}
46 };
```
