

Contents

1	gcc ordered set	1
2	Triangle centers	1
3	Convex polygon algorithms	2
4	2D line segment	6
5	Dinic	8
6	Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$	10
7	Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$	11
8	Aho Corasick $\mathcal{O}(\alpha \sum \text{len})$	13
9	Suffix automaton $\mathcal{O}((n+q) \log(\alpha))$	14
10	Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$	16
11	Templated HLD $\mathcal{O}(M(n) \log n)$ per query	17
12	Templated Persistent Segment Tree $\mathcal{O}(\log n)$ per query	19
13	FFT $\mathcal{O}(n \log(n))$	20
14	MOD int, extended Euclidean	22
15	Rabbin Miller prime check	22
16	Factsheet	24

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 Convex polygon algorithms

```

1 ll dot(const pair< int, int > &v1, const pair< int, int > &v2) {
2     return (ll)v1.first * v2.first + (ll)v1.second * v2.second;
3 }
4
5 ll cross(const pair< int, int > &v1, const pair< int, int > &v2) {
6     return (ll)v1.first * v2.second - (ll)v2.first * v1.second;
7 }
8
9 ll dist_sq(const pair< int, int > &p1, const pair< int, int > &p2) {
10     return (ll)(p2.first - p1.first) * (p2.first - p1.first) +
11         (ll)(p2.second - p1.second) * (p2.second - p1.second);
12 }
13
14 struct Hull {
15     vector< pair< pair< int, int >, pair< int, int > > > hull;
16     vector< pair< pair< int, int >, pair< int, int > > >::iterator upper_begin;
17     template < typename Iterator >
18     void extend_hull(Iterator begin, Iterator end) { // O(n)
19         vector< pair< int, int > > res;
20         for (auto it = begin; it != end; ++it) {
21             if (res.empty() || *it != res.back()) {

```

```

22     while (res.size() >= 2) {
23         auto v1 = make_pair(res[res.size() - 1].first - res[res.size() - 2].first,
24                             res[res.size() - 1].second - res[res.size() - 2].second);
25         auto v2 = make_pair(it->first - res[res.size() - 2].first,
26                             it->second - res[res.size() - 2].second);
27         if (cross(v1, v2) > 0) {
28             break;
29         }
30         res.pop_back();
31     }
32     res.push_back(*it);
33 }
34 }
35 for (int i = 0; i < res.size() - 1; ++i) {
36     hull.emplace_back(res[i], res[i + 1]);
37 }
38 }
39 Hull(vector< pair< int, int > > &vert) { // atleast 2 distinct points
40     sort(vert.begin(), vert.end()); // O(n log(n))
41     extend_hull(vert.begin(), vert.end());
42     int diff = hull.size();
43     extend_hull(vert.rbegin(), vert.rend());
44     upper_begin = hull.begin() + diff;
45 }
46 bool contains(pair< int, int > p) { // O(log(n))
47     if (p < hull.front().first || p > upper_begin->first) return false;
48     {
49         auto it_low = lower_bound(hull.begin(), upper_begin,
50                                 make_pair(make_pair(p.first, (int)-2e9), make_pair(0, 0)));
51         if (it_low != hull.begin()) {
52             --it_low;
53         }
54         auto v1 = make_pair(it_low->second.first - it_low->first.first,
55                             it_low->second.second - it_low->first.second);
56         auto v2 = make_pair(p.first - it_low->first.first, p.second - it_low->first.second);
57         if (cross(v1, v2) < 0) // < 0 is inclusive, <= 0 is exclusive
58             return false;
59     }
60     {
61         auto it_up = lower_bound(hull.rbegin(), hull.rbegin() + (hull.end() - upper_begin),
62                                 make_pair(make_pair(p.first, (int)2e9), make_pair(0, 0)));
63         if (it_up - hull.rbegin() == hull.end() - upper_begin) {
64             --it_up;
65         }
66         auto v1 = make_pair(it_up->first.first - it_up->second.first,
67                             it_up->first.second - it_up->second.second);
68         auto v2 = make_pair(p.first - it_up->second.first, p.second - it_up->second.second);
69         if (cross(v1, v2) > 0) // > 0 is inclusive, >= 0 is exclusive
70             return false;
71     }
72     return true;
73 }
74 template < typename T > // The function can have only one local min and max and may be constant
75                          // only at min and max.
76 vector< pair< pair< int, int >, pair< int, int > > >::iterator max(
77     function< T(const pair< pair< int, int >, pair< int, int > > &) > f) { // O(log(n))
78     auto l = hull.begin();
79     auto r = hull.end();
80     vector< pair< pair< int, int >, pair< int, int > > >::iterator best = hull.end();
81     T best_val;
82     while (r - l > 2) {
83         auto mid = l + (r - l) / 2;
84         T l_val = f(*l);
85         T l_nxt_val = f(*(l + 1));
86         T mid_val = f(*mid);
87         T mid_nxt_val = f(*(mid + 1));
88         if (best == hull.end() ||
89             l_val > best_val) { // If max is at l we may remove it from the range.
90             best = l;
91             best_val = l_val;
92         }
93         if (l_nxt_val > l_val) {
94             if (mid_val < l_val) {

```

```

95     r = mid;
96   } else {
97     if (mid_nxt_val > mid_val) {
98       l = mid + 1;
99     } else {
100       r = mid + 1;
101     }
102   }
103 } else {
104   if (mid_val < l_val) {
105     l = mid + 1;
106   } else {
107     if (mid_nxt_val > mid_val) {
108       l = mid + 1;
109     } else {
110       r = mid + 1;
111     }
112   }
113 }
114 }
115 T l_val = f(*l);
116 if (best == hull.end() || l_val > best_val) {
117   best = l;
118   best_val = l_val;
119 }
120 if (r - l > 1) {
121   T l_nxt_val = f(*(l + 1));
122   if (best == hull.end() || l_nxt_val > best_val) {
123     best = l + 1;
124     best_val = l_nxt_val;
125   }
126 }
127 return best;
128 }
129 vector< pair< pair< int, int >, pair< int, int > > >::iterator closest(
130   pair< int, int >
131   p) { // p can't be internal(can be on border), hull must have atleast 3 points
132   const pair< pair< int, int >, pair< int, int > > &ref_p = hull.front(); // O(log(n))
133   return max(function< double(const pair< pair< int, int >, pair< int, int > > &) >(
134     [&p, &ref_p](const pair< pair< int, int >, pair< int, int > >
135       &seg) { // accuracy of used type should be coord^-2
136       if (p == seg.first) return 10 - M_PI;
137       auto v1 =
138         make_pair(seg.second.first - seg.first.first, seg.second.second - seg.first.second);
139       auto v2 = make_pair(p.first - seg.first.first, p.second - seg.first.second);
140       ll cross_prod = cross(v1, v2);
141       if (cross_prod > 0) { // order the backside by angle
142         auto v1 = make_pair(ref_p.first.first - p.first, ref_p.first.second - p.second);
143         auto v2 = make_pair(seg.first.first - p.first, seg.first.second - p.second);
144         ll dot_prod = dot(v1, v2);
145         ll cross_prod = cross(v2, v1);
146         return atan2(cross_prod, dot_prod) / 2;
147       }
148       ll dot_prod = dot(v1, v2);
149       double res = atan2(dot_prod, cross_prod);
150       if (dot_prod <= 0 && res > 0) res = -M_PI;
151       if (res > 0) {
152         res += 20;
153       } else {
154         res = 10 - res;
155       }
156       return res;
157     }));
158 }
159 pair< int, int > forw_tan(pair< int, int > p) { // can't be internal or on border
160   const pair< pair< int, int >, pair< int, int > > &ref_p = hull.front(); // O(log(n))
161   auto best_seg = max(function< double(const pair< pair< int, int >, pair< int, int > > &) >(
162     [&p, &ref_p](const pair< pair< int, int >, pair< int, int > >
163       &seg) { // accuracy of used type should be coord^-2
164       auto v1 = make_pair(ref_p.first.first - p.first, ref_p.first.second - p.second);
165       auto v2 = make_pair(seg.first.first - p.first, seg.first.second - p.second);
166       ll dot_prod = dot(v1, v2);
167       ll cross_prod = cross(v2, v1); // cross(v1, v2) for back_tan !!!

```

```

168         return atan2(cross_prod, dot_prod); // order by signed angle
169     }));
170     return best_seg->first;
171 }
172 vector< pair< pair< int, int >, pair< int, int > > >::iterator max_in_dir(
173     pair< int, int > v) { // first is the ans. O(log(n))
174     return max(function< ll(const pair< pair< int, int >, pair< int, int > > &) >(
175         [&v](const pair< pair< int, int >, pair< int, int > > &seg) { return dot(v, seg.first); }));
176     );
177 pair< vector< pair< pair< int, int >, pair< int, int > > >::iterator,
178     vector< pair< pair< int, int >, pair< int, int > > >::iterator >
179 intersections(pair< pair< int, int >, pair< int, int > > line) { // O(log(n))
180     int x = line.second.first - line.first.first;
181     int y = line.second.second - line.first.second;
182     auto dir = make_pair(-y, x);
183     auto it_max = max_in_dir(dir);
184     auto it_min = max_in_dir(make_pair(y, -x));
185     ll opt_val = dot(dir, line.first);
186     if (dot(dir, it_max->first) < opt_val || dot(dir, it_min->first) > opt_val) {
187         return make_pair(hull.end(), hull.end());
188     }
189     vector< pair< pair< int, int >, pair< int, int > > >::iterator it_r1, it_r2;
190     function< bool(const pair< pair< int, int >, pair< int, int > > &,
191         const pair< pair< int, int >, pair< int, int > > &) >
192         inc_comp([&dir](const pair< pair< int, int >, pair< int, int > > &lft,
193             const pair< pair< int, int >, pair< int, int > > &rgt) {
194             return dot(dir, lft.first) < dot(dir, rgt.first);
195         });
196     function< bool(const pair< pair< int, int >, pair< int, int > > &,
197         const pair< pair< int, int >, pair< int, int > > &) >
198         dec_comp([&dir](const pair< pair< int, int >, pair< int, int > > &lft,
199             const pair< pair< int, int >, pair< int, int > > &rgt) {
200             return dot(dir, lft.first) > dot(dir, rgt.first);
201         });
202     if (it_min <= it_max) {
203         it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
204         if (dot(dir, hull.front().first) >= opt_val) {
205             it_r2 = upper_bound(hull.begin(), it_min + 1, line, dec_comp) - 1;
206         } else {
207             it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
208         }
209     } else {
210         it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
211         if (dot(dir, hull.front().first) <= opt_val) {
212             it_r2 = upper_bound(hull.begin(), it_max + 1, line, inc_comp) - 1;
213         } else {
214             it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
215         }
216     }
217     return make_pair(it_r1, it_r2);
218 }
219 pair< pair< int, int >, pair< int, int > > diameter() { // O(n)
220     pair< pair< int, int >, pair< int, int > > res;
221     ll dia_sq = 0;
222     auto it1 = hull.begin();
223     auto it2 = upper_begin;
224     auto v1 = make_pair(hull.back().second.first - hull.back().first.first,
225         hull.back().second.second - hull.back().first.second);
226     while (it2 != hull.begin()) {
227         auto v2 = make_pair((it2 - 1)->second.first - (it2 - 1)->first.first,
228             (it2 - 1)->second.second - (it2 - 1)->first.second);
229         ll decider = cross(v1, v2);
230         if (decider > 0) break;
231         --it2;
232     }
233     while (it2 != hull.end()) { // check all antipodal pairs
234         if (dist_sq(it1->first, it2->first) > dia_sq) {
235             res = make_pair(it1->first, it2->first);
236             dia_sq = dist_sq(res.first, res.second);
237         }
238         auto v1 =
239             make_pair(it1->second.first - it1->first.first, it1->second.second - it1->first.second);
240         auto v2 =

```

```

241     make_pair(it2->second.first - it2->first.first, it2->second.second - it2->first.second);
242     ll decider = cross(v1, v2);
243     if (decider == 0) { // report cross pairs at parallel lines.
244         if (dist_sq(it1->second, it2->first) > dia_sq) {
245             res = make_pair(it1->second, it2->first);
246             dia_sq = dist_sq(res.first, res.second);
247         }
248         if (dist_sq(it1->first, it2->second) > dia_sq) {
249             res = make_pair(it1->first, it2->second);
250             dia_sq = dist_sq(res.first, res.second);
251         }
252         ++it1;
253         ++it2;
254     } else if (decider < 0) {
255         ++it1;
256     } else {
257         ++it2;
258     }
259 }
260 return res;
261 }
262 };

```

4 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 }

```

5 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
    ↪ _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
29     ll rem() const {
30         if(dir == 0){
31             return cap-*flow;
32         }
33         else{
34             return rcap-*flow;
35         }
36     }
37     void add_flow(ll f){
38         if(dir == 0)
39             *flow += f;
40         else
41             *flow -= f;
42         assert(*flow <= cap);
43         assert(-*flow <= rcap);
44     }
45     operator ll() const { return rem(); }
46     void operator--=(ll x){ add_flow(x); }
47     void operator+=(ll x){ add_flow(-x); }
48 };
49 int source,sink;
50 vector<vector<int> > adj;
51 vector<vector<FlowTracker> > cap;
52 vector<Edge> edges;
53 MaxFlow(int _source, int _sink):source(_source),sink(_sink){
54     assert(source != sink);
55 }
56 int add_edge(Edge e){
57     edges.push_back(e);
58     return edges.size()-1;
59 }
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,lv1;
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     lv1.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(lv1.begin(),lv1.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
147     // need 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
154     }
155     mf.group_edges(); // small auxillary to remove multiple edges, only use this if we need to know TOTAL
156     // FLOW ONLY
157     cout << mf.calc() << '\n';
158     //cout << *mf.edges[edge_index].flow << '\n'; // ONLY if group_edges() WAS NOT CALLED
159 }

```

6 Min Cost Max Flow with successive dijkstra $\mathcal{O}(\text{flow} \cdot n^2)$

```

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(){ //incase of negative edges have to add Bellman-Ford that is run once.
12     tot_cost=0; //Does not work with negative cycles.
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){ //Use Dijkstra to calculate potentials
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){ //Apply potentials to edge costs.
46             for(int j=0; j<v; ++j){ //Found path from source to sink becomes 0 cost.
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){ //Backtrack along found path that now has 0 cost.
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 }

```

7 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$

```

1 struct Network {
2     struct Node;
3
4     struct Edge {
5         Node *u, *v;
6         int f, c, cost;
7
8         Node* from(Node* pos) {
9             if(pos == u)
10                 return v;
11             return u;
12         }
13         int getCap(Node* pos) {

```

```

14     if(pos == u)
15         return c-f;
16     return f;
17 }
18 int addFlow(Node* pos, int toAdd) {
19     if(pos == u) {
20         f += toAdd;
21         return toAdd * cost;
22     }
23     else {
24         f -= toAdd;
25         return -toAdd * cost;
26     }
27 }
28
29 };
30 struct Node {
31     vector<Edge*> conn;
32     int index;
33 };
34
35 deque<Node> nodes;
36 deque<Edge> edges;
37
38 Node* addNode() {
39     nodes.push_back(Node());
40     nodes.back().index = nodes.size()-1;
41     return &nodes.back();
42 }
43 Edge* addEdge(Node* u, Node* v, int f, int c, int cost) {
44     edges.push_back({u, v, f, c, cost});
45     u->conn.push_back(&edges.back());
46     v->conn.push_back(&edges.back());
47     return &edges.back();
48 }
49
50
51 //Assumes all needed flow has already been added
52 int minCostMaxFlow() {
53     int n = nodes.size();
54     int result = 0;
55
56     struct State {
57         int p;
58         Edge* used;
59     };
60
61     while(1) {
62         vector<vector<State> > state(1, vector<State>(n, {0, 0}));
63
64         for(int lev = 0; lev < n; lev++) {
65             state.push_back(state[lev]);
66             for(int i=0;i<n;i++)
67                 if(lev == 0 || state[lev][i].p < state[lev-1][i].p) {
68
69                     for(Edge* edge : nodes[i].conn) if(edge->getCap(&nodes[i]) > 0) {
70                         int np = state[lev][i].p + (edge->u == &nodes[i] ? edge->cost : -edge->cost);
71                         int ni = edge->from(&nodes[i])->index;
72
73                         if(np < state[lev+1][ni].p) {
74                             state[lev+1][ni].p = np;
75                             state[lev+1][ni].used = edge;
76                         }
77                     }
78                 }
79             }
80
81     //Now look at the last level
82     bool valid = false;
83
84     for(int i=0;i<n;i++)
85         if(state[n-1][i].p > state[n][i].p) {
86             valid = true;

```

```

87     vector<Edge*> path;
88
89     int cap = 1000000000;
90     Node* cur = &nodes[i];
91     int clev = n;
92
93     vector<bool> explr(n, false);
94
95     while(!explr[cur->index]) {
96         explr[cur->index] = true;
97
98         State cstate = state[clev][cur->index];
99         cur = cstate.used->from(cur);
100
101         path.push_back(cstate.used);
102     }
103
104     reverse(path.begin(), path.end() );
105
106     {
107         int i=0;
108         Node* cur2 = cur;
109
110         do {
111             cur2 = path[i]->from(cur2);
112             i++;
113         }while(cur2 != cur);
114
115         path.resize(i);
116     }
117
118     for(auto edge : path) {
119         cap = min(cap, edge->getCap(cur));
120         cur = edge->from(cur);
121     }
122
123     for(auto edge : path) {
124         result += edge->addFlow(cur, cap);
125         cur = edge->from(cur);
126     }
127 }
128
129 if(!valid) break;
130 }
131
132 return result;
133 }
134
135 }
136 };

```

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

```

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 propagate 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 }

```

9 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
    ↪ substring
33     match_len = min(match_len, len); //includes depth limit(needed for finding matches)
34     if (has_nxt(c)) { //as suffixes are in classes match_len must be
    ↪ tracte eternally
35         ++match_len;
36         return nxt(c)->lower_depth(depth);
37     }
38     if (suf)
39         return suf->walk(c, depth, match_len);
40     return this;
41 }
42 int paths_to_end = 0;
43 void set_as_end() { //All suffixes of current node are marked as ending nodes.
44     paths_to_end = 1;
45     if (suf) suf->set_as_end();
46 }
47 bool vis = false;
48 void calc_paths_to_end() { //Call ONCE from ROOT. For each node calculates number of ways to reach an
    ↪ end node.
49     if (!vis) { //paths_to_end is ocurence count for any strings in current suffix
    ↪ equivalence class.
50         vis = true;
51         for (auto cur : nxt_char) {
52             cur.second->calc_paths_to_end();
53             paths_to_end += cur.second->paths_to_end;
54         }
55     }
56 }
57 };
58 struct SufAutomaton {
59     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
        // can be turned into a substring of old state
        // by appending new_c
66             suf_w_nxt->set_nxt(new_c, new_end);
67             suf_w_nxt = suf_w_nxt->suf;
68         }
69         if (!suf_w_nxt) { // The new character isn't part of the old string
70             new_end->suf = root;
71         } else {
72             AutoNode *max_sbstr = suf_w_nxt->nxt(new_c); // Equivalence class containing longest
73             // substring which is a suffix of the new state.
74             if (suf_w_nxt->len + 1 == max_sbstr->len) { // Check whether splitting is needed
75                 new_end->suf = max_sbstr;
76             } else {
77                 AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
78                 new_end->suf = eq_sbstr;
79                 // Make suffixes of suf_w_nxt point to eq_sbstr instead of max_sbstr
80                 AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
81                 while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
82                     w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
83                     w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
84                 }
85             }
86         }
87     }
88 };

```

```

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 };

```

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

```

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 }

```

11 Templated HLD $\mathcal{O}(M(n) \log n)$ per query

```

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 }

```

12 Templated Persistent Segment Tree $\mathcal{O}(\log n)$ per query

```

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 PersistentST (int _size, T initial) {
46     for (int i = 0; i < 32; i++) {
47         if ((1 << i) > _size) {
48             size = 1 << i;
49             break;
50         }
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 qlft, int qright, int revision) {
80     return roots[revision]->query(qlft, qright);
81 }
82
83 T query (int qlft, int qright) {
84     return roots[last_revision()]->query(qlft, qright);
85 }
86 };

```

13 FFT $\mathcal{O}(n \log(n))$

```

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 };

```

14 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 };

```

15 Rabbin Miller prime check

```

1 __int128 pow_mod(__int128 a, ll n, __int128 mod) {
2     __int128 res = 1;
3     for (ll i = 0; i < 64; ++i) {
4         if (n & (1LL << i)) {
5             res = (res * a) % mod;
6         }
7         a = (a * a) % mod;
8     }
9     return res;
10 }
11
12 bool is_prime(ll n) { //guaranteed for 64 bit numbers

```

```
13  if (n == 2 || n == 3) return true;
14  if (!(n & 1) || n == 1) return false;
15  static vector< char > witnesses = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
16  ll s = __builtin_ctz(n - 1);
17  ll d = (n - 1) >> s;
18  __int128 mod = n;
19  for (__int128 a : witnesses) {
20      if (a >= mod) break;
21      a = pow_mod(a, d, mod);
22      if (a == 1 || a == mod - 1) continue;
23      for (ll r = 1; r < s; ++r) {
24          a = a * a % mod;
25          if (a == 1) return false;
26          if (a == mod - 1) break;
27      }
28      if (a != mod - 1) return false;
29  }
30  return true;
31 }
```

16 Factsheet

Combinatorics Cheat Sheet

Useful formulas

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ — number of ways to choose k objects out of n

$\binom{n+k-1}{k-1}$ — number of ways to choose k objects out of n with repetitions

$[n]$ — Stirling numbers of the first kind; number of permutations of n elements with k cycles

$$[n+1] = n[n] + [n-1]$$

$$(x)_n = x(x-1)\dots x-n+1 = \sum_{k=0}^n (-1)^{n-k} [n]_k x^k$$

$\{n\}_k$ — Stirling numbers of the second kind; number of partitions of set $1, \dots, n$ into k disjoint subsets.

$$\{n+1\}_k = k\{n\}_k + \{n\}_{k-1}$$

$$\sum_{k=0}^n \{n\}_k (x)_k = x^n$$

$$C_n = \frac{1}{n+1} \binom{2n}{n} \text{ — Catalan numbers}$$

$$C(x) = \frac{1-\sqrt{1-4x}}{2x}$$

Binomial transform

If $a_n = \sum_{k=0}^n \binom{n}{k} b_k$, then $b_n = \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} a_k$

$$\bullet a = (1, x, x^2, \dots), b = (1, (x+1), (x+1)^2, \dots)$$

$$\bullet a_i = i^k, b_i = \{n\}_i i!$$

Burnside's lemma

Let G be a group of *action* on set X (Ex.: cyclic shifts of array, rotations and symmetries of $n \times n$ matrix, ...)

Call two objects x and y *equivalent* if there is an action f that transforms x to y : $f(x) = y$.

The number of equivalence classes then can be calculated as follows: $C = \frac{1}{|G|} \sum_{f \in G} |X^f|$, where X^f

is the set of *fixed points* of f : $X^f = \{x | f(x) = x\}$

Generating functions

Ordinary generating function (o.g.f.) for sequence $a_0, a_1, \dots, a_n, \dots$ is $A(x) = \sum_{n=0}^{\infty} a_n x^n$

Exponential generating function (e.g.f.) for sequence $a_0, a_1, \dots, a_n, \dots$ is $A(x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}$

$$B(x) = A'(x), b_{n-1} = n \cdot a_n$$

$$c_n = \sum_{k=0}^n a_k b_{n-k} \text{ (o.g.f. convolution)}$$

$$c_n = \sum_{k=0}^n \binom{n}{k} a_k b_{n-k} \text{ (e.g.f. convolution, compute with FFT using } \widetilde{a}_n = \frac{a_n}{n!})$$

General linear recurrences

If $a_n = \sum_{k=1}^n b_k a_{n-k}$, then $A(x) = \frac{a_0}{1-B(x)}$. We also can compute all a_n with Divide-and-Conquer algorithm in $O(n \log^2 n)$.

Inverse polynomial modulo x^l

Given $A(x)$, find $B(x)$ such that $A(x)B(x) = 1 + x^l \cdot Q(x)$ for some $Q(x)$

$$1. \text{ Start with } B_0(x) = \frac{1}{a_0}$$

$$2. \text{ Double the length of } B(x): B_{k+1}(x) = (-B_k(x)^2 A(x) + 2B_k(x)) \bmod x^{2^{k+1}}$$

Fast subset convolution

Given array a_i of size 2^k , calculate $b_i = \sum_{j \& i = i} b_j$

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      a[i + (1 << b)] += a[i]
```

Hadamard transform

Treat array a of size 2^k as k -dimensional array of size $2 \times 2 \times \dots \times 2$, calculate FFT of that array:

```
for b = 0..k-1
  for i = 0..2^k-1
    if (i & (1 << b)) != 0:
      u = a[i], v = a[i + (1 << b)]
      a[i] = u + v
      a[i + (1 << b)] = u - v
```

- **Fermat's little theorem.** Let p be prime. Then, for each integer a :

$$a^{p-1} \equiv 1 \pmod{p}.$$

Thus:

$$a^k \equiv a^{k \bmod (p-1)} \pmod{p}.$$

Also:

$$a^{p-2} \equiv a^{-1} \pmod{p}.$$

- **Iterating over subsets.** Let `mask` be the binary representation of a set. Then `for (int i = mask; i != 0; i = (i - 1) & mask)` will iterate over all the nonempty subsets of `mask`.
- **Chinese remainder theorem.** We know that:

$$x \equiv a_1 \pmod{n_1}$$

$$x \equiv a_2 \pmod{n_2}$$

where n_1 and n_2 are (co)prime. We want to find $a_{1,2}$ so that:

$$x \equiv a_{1,2} \pmod{n_1 \cdot n_2}.$$

A solution is given by:

$$a_{1,2} = a_1 m_2 n_2 + a_2 m_1 n_1,$$

where m_1 and m_2 are integers so that $m_1 n_1 + m_2 n_2 = 1$. Those values can be found using the Extended Euclidean algorithm.

- **Sum of harmonic series.**

$$\frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n} \in \mathcal{O}(\log n)$$

- **Number of primes below...**

10^2	25
10^3	168
10^4	1229
10^5	9592
10^6	78498
10^7	664579