

University of Tartu ICPC Team Notebook

(2018-2019) March 11, 2019

- 1 Setup
- 2 crc.sh
- 3 gcc ordered set
- 4 2D geometry
- 5 3D geometry
- 6 Triangle centers
- 7 Seg-Seg intersection, halfplane intersection area
- 8 Convex polygon algorithms
- 9 Delaunay triangulation $\mathcal{O}(n \log n)$
- 10 Aho Corasick $\mathcal{O}(|\alpha| \sum \text{len})$
- 11 Suffix automaton and tree $\mathcal{O}((n + q) \log(|\alpha|))$
- 12 Dinic
- 13 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$
- 14 DMST $\mathcal{O}(E \log V)$
- 15 Bridges $\mathcal{O}(n)$
- 16 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$
- 17 Generic persistent compressed lazy segment tree
- 18 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query
- 19 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ per query
- 20 Treap $\mathcal{O}(\log n)$ per query
- 21 Radixsort 50M 64 bit integers as single array in 1 sec
- 22 FFT 5M length/sec

<pre> 23 Fast mod mult, Rabbin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$ 24 Symmetric Submodular Functions; Queyrannes's algorithm 24 25 Berlekamp-Massey $\mathcal{O}(\mathcal{L}N)$ </pre> <hr/> <pre> 1 1 Setup 1 1 2 crc.sh 1 1 3 gcc ordered set 1 1 4 2D geometry 1 1 5 3D geometry 1 1 6 Triangle centers 1 1 7 Seg-Seg intersection, halfplane intersection area 1 1 8 Convex polygon algorithms 1 1 9 Delaunay triangulation $\mathcal{O}(n \log n)$ 1 1 10 Aho Corasick $\mathcal{O}(\alpha \sum \text{len})$ 1 1 11 Suffix automaton and tree $\mathcal{O}((n + q) \log(\alpha))$ 1 1 12 Dinic 1 1 13 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$ 1 1 14 DMST $\mathcal{O}(E \log V)$ 1 1 15 Bridges $\mathcal{O}(n)$ 1 1 16 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$ 1 1 17 Generic persistent compressed lazy segment tree 1 1 18 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query 1 1 19 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ per query 1 1 20 Treap $\mathcal{O}(\log n)$ per query 1 1 21 Radixsort 50M 64 bit integers as single array in 1 sec 1 1 22 FFT 5M length/sec </pre>	<p>University of Tartu</p> <p>22</p> <p>24</p> <p>24</p> <hr/> <p>1 1 Setup</p> <p>1 2 crc.sh</p> <p>1 3 gcc ordered set</p> <p>1 4 2D geometry</p> <p>1 5 3D geometry</p> <p>1 6 Triangle centers</p> <p>1 7 Seg-Seg intersection, halfplane intersection area</p> <p>1 8 Convex polygon algorithms</p> <p>1 9 Delaunay triangulation $\mathcal{O}(n \log n)$</p> <p>1 10 Aho Corasick $\mathcal{O}(\alpha \sum \text{len})$</p> <p>1 11 Suffix automaton and tree $\mathcal{O}((n + q) \log(\alpha))$</p> <p>1 12 Dinic</p> <p>1 13 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{flow} \cdot nm)$</p> <p>1 14 DMST $\mathcal{O}(E \log V)$</p> <p>1 15 Bridges $\mathcal{O}(n)$</p> <p>1 16 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$</p> <p>1 17 Generic persistent compressed lazy segment tree</p> <p>1 18 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query</p> <p>1 19 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ per query</p> <p>1 20 Treap $\mathcal{O}(\log n)$ per query</p> <p>1 21 Radixsort 50M 64 bit integers as single array in 1 sec</p> <p>1 22 FFT 5M length/sec</p>
--	--

```
25   cout << *cur.find_by_order(1)
26   << endl;// the 1-th smallest number in the set(0-based)
```

4 2D geometry

Define $\text{orient}(A, B, C) = \overline{AB} \times \overline{AC}$. CCW iff > 0 . Define $\text{perp}((a, b)) = (-b, a)$. The vectors are orthogonal.

For line $ax + by = c$ def $\bar{v} = (-b, a)$.

Line through P and Q has $\bar{v} = \overline{PQ}$ and $c = \bar{v} \times P$. $\text{side}_l(P) = \bar{v}_l \times P - c_l$ sign determines which side P is on from l .

$\text{dist}_l(P) = \text{side}_l(P)/\|\bar{v}_l\|$ squared is integer.

Sorting points along a line: comparator is $\bar{v} \cdot A < \bar{v} \cdot B$.

Translating line by \bar{t} : new line has $c' = c + \bar{v} \times \bar{t}$.

Line intersection: is $(c_l \bar{v}_m - c_m \bar{v}_l)/(\bar{v}_l \times \bar{v}_m)$.

Project P onto l : is $P - \text{perp}(v) \text{side}_l(P)/\|v\|^2$.

Angle bisectors: $\bar{v} = \bar{v}_l/\|\bar{v}_l\| + \bar{v}_m/\|\bar{v}_m\|$

$c = c_l/\|\bar{v}_l\| + c_m/\|\bar{v}_m\|$.

P is on segment AB iff $\text{orient}(A, B, P) = 0$ and $\overline{PA} \cdot \overline{PB} \leq 0$.

Proper intersection of AB and CD exists iff $\text{orient}(C, D, A)$ and $\text{orient}(C, D, B)$ have opp. signs and $\text{orient}(A, B, C)$ and $\text{orient}(A, B, D)$ have opp. signs. Coordinates:

$$\frac{A \text{orient}(C, D, B) - B \text{orient}(C, D, A)}{\text{orient}(C, D, B) - \text{orient}(C, D, A)}.$$

Circumcircle center:

```
pt circumCenter(pt a, pt b, pt c) {
    b = b-a, c = c-a; // consider coordinates relative to A
    assert(cross(b,c) != 0); // no circumcircle if A,B,C aligned
    return a + perp(b*sq(c) - c*sq(b))/cross(b,c)/2;
```

Circle-line intersect:

```
int circleLine(pt o, double r, line l, pair<pt, pt> &out) {
    double h2 = r*r - l.sqDist(o);
    if (h2 >= 0) { // the line touches the circle
        pt p = l.proj(o); // point P
        pt h = l.v*sqrt(h2)/abs(l.v); // vector parallel to l, of len h
        out = {p-h, p+h};
    }
    return 1 + sgn(h2);
```

Circle-circle intersect:

```
int circleCircle(pt o1, double r1, pt o2, double r2, pair<pt,pt> &out) {
    pt d=o2-o1; double d2=sq(d);
```

```
if (d2 == 0) {assert(r1 != r2); return 0;} // concentric circles
double pd = (d2 + r1*r1 - r2*r2)/2; // = |0_1P| * d
double h2 = r1*r1 - pd*pd/d2; // = h^2
if (h2 >= 0) {
    pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
    ;
    out = {p-h, p+h};}
return 1 + sgn(h2);
```

Tangent lines:

```
int tangents(pt o1, double r1, pt o2, double r2,
    bool inner, vector<pair<pt,pt>> &out) {
    if (inner) r2 = -r2;
    pt d = o2-o1;
    double dr = r1-r2, d2 = sq(d), h2 = d2-dr*dr;
    if (d2 == 0 || h2 < 0) {assert(h2 != 0);
        return 0;}
    for (double sign : {-1,1}) {
        pt v = (d*dr + perp(d)*sqrt(h2)*sign)/d2;
        out.push_back({o1 + v*r1, o2 + v*r2});}
    return 1 + (h2 > 0);
```

5 3D geometry

$\text{orient}(P, Q, R, S) = (\overline{PQ} \times \overline{PR}) \cdot \overline{PS}$.

S above PQR iff > 0 .

For plane $ax + by + cz = d$ def $\bar{n} = (a, b, c)$.

Line with normal \bar{n} through point P has $d = \bar{n} \cdot P$.

$\text{side}_\Pi(P) = \bar{n} \cdot P - d$ sign determines side from Π .

$\text{dist}_\Pi(P) = \text{side}_\Pi(P)/\|\bar{n}\|$.

Translating plane by \bar{t} makes $d' = d + \bar{n} \cdot \bar{t}$.

Plane-plane intersection of has direction $\bar{n}_1 \times \bar{n}_2$ and goes through $((d_1 \bar{n}_2 - d_2 \bar{n}_1) \times \bar{d})/\|\bar{d}\|^2$.

Line-line distance:

```
double dist(line3d l1, line3d l2) {
    p3 n = l1.d*l2.d;
    if (n == zero) // parallel
        return l1.dist(l2.o);
    return abs((l2.o-l1.o)|n)/abs(n);
```

Spherical to Cartesian:

$(r \cos \varphi \cos \lambda, r \cos \varphi \sin \lambda, r \sin \varphi)$.

Sphere-line intersection:

```
int sphereLine(p3 o, double r, line3d l, pair<p3, p3> &out) {
    double h2 = r*r - l.sqDist(o);
    if (h2 < 0) return 0; // the line doesn't touch the sphere
    p3 p = l.proj(o); // point P
    p3 h = l.d*sqrt(h2)/abs(l.d); // vector parallel to l, of length h
    out = {p-h, p+h};
```

```
return 1 + (h2 > 0);
```

Great-circle distance between points A and B is $r\angle AOB$.

Spherical segment intersection:

```
bool properInter(p3 a, p3 b, p3 c, p3 d, p3 &out)
    ) {
    p3 ab = a*b, cd = c*d; // normals of planes OAB and OCD
    int oa = sgn(cd|a),
        ob = sgn(cd|b),
        oc = sgn(ab|c),
        od = sgn(ab|d);
    out = ab*cd*od; // four multiplications => careful with overflow !
    return (oa != ob && oc != od && oa != oc);
}
bool onSphSegment(p3 a, p3 b, p3 p) {
    p3 n = a*b;
    if (n == zero)
        return a*p == zero && (a|p) > 0;
    return (n|p) == 0 && (n|a*p) >= 0 && (n|b*p) <= 0;
}
```

```
struct directionSet : vector<p3> {
    using vector::vector; // import constructors
    void insert(p3 p) {
        for (p3 q : *this) if (p*q == zero) return;
        push_back(p);
    }
};
```

```
directionSet intersSph(p3 a, p3 b, p3 c, p3 d) {
    assert(validSegment(a, b) && validSegment(c, d));
    p3 out;
    if (properInter(a, b, c, d, out)) return {out};
    directionSet s;
    if (onSphSegment(c, d, a)) s.insert(a);
    if (onSphSegment(c, d, b)) s.insert(b);
    if (onSphSegment(a, b, c)) s.insert(c);
    if (onSphSegment(a, b, d)) s.insert(d);
    return s;
}
```

Angle between spherical segments AB and AC is angle between $A \times B$ and $A \times C$.

Oriented angle: subtract from 2π if mixed product is negative.

Area of a spherical polygon:

$$r^2[\text{sum of interior angles} - (n-2)\pi].$$

6 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;// vertexes of the triangle
5 bool collinear() #823
6     double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7     if (min_diff < coord_max * min_delta) return true;
8     point sp = (B - A) / (C - A);
9     double ang = #638
10        M_PI / 2
11        abs(arg(sp)) - M_PI / 2;// positive angle with the real line
12     return ang < min_delta; %0446
13
14 point circum_center(){
15     if (collinear()) return point(NAN, NAN);
16     // squared lengths of sides
17     double a2 = norm(B - C);
18     double b2 = norm(A - C);
19     double c2 = norm(A - B) #715
20     // barycentric coordinates of the circumcenter
21     double c_A = a2 * (b2 + c2 - a2); // sin(2 * alpha) works also
22     double c_B = b2 * (a2 + c2 - b2);
23     double c_C = c2 * (a2 + b2 - c2);
24     double sum = c_A + c_B + c_C;
25     c_A /= sum
26     c_B /= sum; #407
27     c_C /= sum;
28     return c_A * A + c_B * B + c_C * C;// cartesian
29
30 point centroid(){// center of mass
31     return (A + B + C) / 3.0;
32 }
33 point ortho_center() {// euler line
34     point O = circum_center()
35     return O + 3.0 * (centroid() - O); #895
36 };
37 point nine_point_circle_center{// euler line
38     point O = circum_center();
39     return O + 1.5 * (centroid() - O)
40 }
41 point in_center(){
42     if (collinear()) return point(NAN, NAN);
43     double a = abs(B - C); // side lengths
44     double b = abs(A - C);
45     double c = abs(A - B)
46     // trilinear coordinates are (1,1,1) #193
47     double sum = a + b + c;
48     a /= sum;
49     b /= sum; %031
50     c /= sum; // barycentric

```

```

51     return a * A + b * B + c * C;// cartesian #596
52
53     7 Seg-Seg intersection, halfplane intersection area
54     struct Seg {
55         Vec a, b;
56         Vecd({ return b - a; })
57     };
58     Vec intersection(Seg l, Seg r) #327
59         Vec dl = l.d(), dr = r.d();
60         if (cross(dl, dr) == 0) return {nanl "", nanl ""};
61         double h = cross(dr, l.a - r.a) / len(dr);
62         double dh = cross(dr, dl) / len(dr);
63         return l.a + dl * (h / -dh) #893
64     }// Returns the area bounded by halfplanes
65     double calc_area(vector<Seg> lines{
66         double lb = -HUGE_VAL, ub = HUGE_VAL; #454
67         vector<Seg> linesBySide[2];
68         for (auto line : lines)
69             if (line.b.y == line.a.y) {
70                 if (line.a.x < line.b.x) {
71                     lb = max(lb, line.a.y); #029
72                 } else {
73                     ub = min(ub, line.a.y)
74                 }
75             } else if (line.a.y < line.b.y) {
76                 linesBySide[1].push_back(line);
77             } else {
78                 linesBySide[0].push_back({line.b, line.a}) #613
79             }
80         }
81         sort(
82             linesBySide[0].begin(), linesBySide[0].end(), [] (Seg l, Seg r) {
83                 if (cross(l.d(), r.d()) == 0) #123
84                     return normal(l.d()) * l.a > normal(r.d()) * r.; #115
85                 return cross(l.d(), r.d()) < ;
86             });
87         sort(
88             linesBySide[1].begin(), linesBySide[1].end(), [] (Seg l, Seg r) {
89                 if (cross(l.d(), r.d()) == 0)
90                     return normal(l.d()) * l.a < normal(r.d()) * r.a;
91                 return cross(l.d(), r.d()) > ;
92             });
93         // Now find the application area of the lines and clean up redundant
94         // ones
95         vector<double> applyStart[2] #597
96         for (int side = 0; side < 2; side++) {
97             vector<double> &apply = applyStart[side];
98             vector<Seg> curLines;
99             for (auto line : linesBySide[side]) {
100                 while (curLines.size() > 0) #412
101                     Seg other = curLines.back();
102                     if (cross(other.d(), line.d()) < 0)
103                         curLines.pop_back();
104                     else
105                         apply.push_back(cross(other.d(), line.d()));
106             }
107         }
108     }

```

```

49     if (cross(line.d(), other.d()) != 0) {
50         double start = intersection(line, other).y;
51         if (start > apply.back()) break;
52
53         curLines.pop_back();
54         apply.pop_back();
55     }
56     if (curLines.size() == 0) {
57         apply.push_back(-HUGE_VAL)
58     } else {
59         apply.push_back(intersection(line, curLines.back()).y);
60     }
61     curLines.push_back(line);
62
63     linesBySide[side] = curLines;
64 }
65 applyStart[0].push_back(HUGE_VALL);
66 applyStart[1].push_back(HUGE_VALL);
67 double result = 0
68 {
69     double lb = -HUGE_VALL, ub;
70     for (int i = 0, j = 0; i < (int)linesBySide[0].size() &&
71             j < (int)linesBySide[1].size();
72         lb = ub)
73         ub = min(applyStart[0][i + 1], applyStart[1][j + 1]);
74         double alb = lb, aub = ub;
75         Seg 10 = linesBySide[0][i], 11 = linesBySide[1][j];
76         if (cross(11.d(), 10.d()) > 0) {
77             alb = max(alb, intersection(10, 11).y)
78         } else if (cross(11.d(), 10.d()) < 0) {
79             aub = min(aub, intersection(10, 11).y);
80         }
81         alb = max(alb, lb);
82         aub = min(aub, ub)
83         aub = max(aub, alb);
84     {
85         double x1 = 10.a.x + (alb - 10.a.y) / 10.d().y * 10.d().x;
86         double x2 = 10.a.x + (aub - 10.a.y) / 10.d().y * 10.d().x;
87         result -= (aub - alb) * (x1 + x2) / 2
88     }
89     {
90         double x1 = 11.a.x + (alb - 11.a.y) / 11.d().y * 11.d().x;
91         double x2 = 11.a.x + (aub - 11.a.y) / 11.d().y * 11.d().x;
92         result += (aub - alb) * (x1 + x2) / 2
93     }
94     if (applyStart[0][i + 1] < applyStart[1][j + 1]) {
95         i++;
96     } else {
97         j++
98     }
99 }
100 
```

#503

#321

#047

#908

#251

#743

#839

#075

#717

#446

```

101     return result;

```

8 Convex polygon algorithms

```

1 typedef pair<int, int> Vec;
2 typedef pair<Vec, Vec> Seg;
3 typedef vector<Seg>::iterator SegIt; #define F first
4 #define S second
5 #define MP(x, y) make_pair(x, y)
6 lldot(Vec &v1, Vec &v2){ return (ll)v1.F * v2.F + (ll)v1.S * v2.S; }
7 llcross(Vec &v1, Vec &v2){
8     return (ll)v1.F * v2.S - (ll)v2.F * v1.S;
9 }
10 lldist_sq(Vec &p1, Vec &p2{
11     return (ll)(p2.F - p1.F) * (p2.F - p1.F) +
12         (ll)(p2.S - p1.S) * (p2.S - p1.S);
13 }
14 struct Hull {
15     vector<Seg> hull;
16     SegIt up_beg;
17     template <typename It>
18     void extend(It beg, It end) { // O(n)
19         vector<Vec> r;
20         for (auto it = beg; it != end; ++it) {
21             if (r.empty() || *it != r.back()) {
22                 while (r.size() >= 2) {
23                     int n = r.size()
24                     Vec v1 = {r[n - 1].F - r[n - 2].F, r[n - 1].S - r[n - 2].S};
25                     Vec v2 = {it->F - r[n - 2].F, it->S - r[n - 2].S};
26                     if (cross(v1, v2) > 0) break;
27                     r.pop_back();
28                 }
29                 r.push_back(*it);
30             }
31         }
32         ran(i, 0, (int)r.size() - 1) hull.emplace_back(r[i], r[i + 1]);
33     }
34 Hull(vector<Vec> &vert) { // atleast 2 distinct points
35     sort(vert.begin(), vert.end()); // O(n log(n))
36     extend(vert.begin(), vert.end());
37     int diff = hull.size();
38     extend(vert.rbegin(), vert.rend())
39     up_beg = hull.begin() + diff;
40 }
41 bool contains(Vec p{// O(log(n))
42     if (p < hull.front().F || p > up_beg->F) return false;
43     {
44         auto it_low = lower_bound(
45             hull.begin(), up_beg, MP(MP(p.F, (int)-2e9), MP(0, 0))) #542
46         if (it_low != hull.begin()) --it_low;
47         Vec a = {it_low->S.F - it_low->F.F, it_low->S.S - it_low->F.S};
48         Vec b = {p.F - it_low->F.F, p.S - it_low->F.S};

```

@#608

#541

%008

#096

#442

#605

#572

#964

%722

#542

```

49     if (cross(a, b) < 0) // < 0 is inclusive, <=0 is exclusive
50         return false
51     }
52     {
53         auto it_up = lower_bound(hull.rbegin(),
54             hull.rbegin() + (hull.end() - up_beg),
55             MP(MP(p.F, (int)2e9), MP(0, 0)))
56         if (it_up - hull.rbegin() == hull.end() - up_beg) --it_up;
57         Vec a = {it_up->F.F - it_up->S.F, it_up->F.S - it_up->S.S};
58         Vec b = {p.F - it_up->S.F, p.S - it_up->S.S};
59         if (cross(a, b) > 0) // > 0 is inclusive, >=0 is exclusive
60             return false
61     }
62     return true;
63
64 // The function can have only one local min and max
65 // and may be constant only at min and max.
66 template <typename T>
67 SegIt max(function<T(Seg &)> f) { // O(log(n))
68     auto l = hull.begin();
69     auto r = hull.end();
70     SegIt b = hull.end()
71     T b_v;
72     while (r - l > 2) {
73         auto m = l + (r - 1) / 2;
74         T l_v = f(*l);
75         T l_n_v = f(*(l + 1))
76         T m_v = f(*m);
77         T m_n_v = f(*(m + 1));
78         if (b == hull.end() || l_v > b_v) {
79             b = l; // If max is at l we may remove it from the range.
80             b_v = l_v
81         }
82         if (l_n_v > l_v) {
83             if (m_v < l_v) {
84                 r = m;
85             } else
86                 if (m_n_v > m_v) {
87                     l = m + 1;
88                 } else {
89                     r = m + 1;
90                 }
91         } else {
92             if (m_v < l_v) {
93                 l = m + 1;
94             } else
95                 if (m_n_v > m_v) {
96                     l = m + 1;
97                 } else {
98                     r = m + 1;
99                 }
100    }
101
102    }
103
104    T l_v = f(*l);
105    if (b == hull.end() || l_v > b_v) #864
106        b = l;
107        b_v = l_v;
108    }
109    if (r - l > 1) {
110        T l_n_v = f(*(l + 1)) #972
111        if (b == hull.end() || l_n_v > b_v) {
112            b = l + 1;
113            b_v = l_n_v;
114        }
115    }
116
117    return b;
118
119
120 SegIt closest(Vec p{ // p can't be internal(can be on border), #504
121     // hull must have atleast 3 points
122     Seg &ref_p = hull.front(); // O(log(n))
123     return max(function<double>(Seg &)( #086
124         [&p, &ref_p]()
125             Seg &seg){ // accuracy of used type should be coord^-2 #071
126                 if (p == seg.F) return 10 - M_PI;
127                 Vec v1 = {seg.S.F - seg.F.F, seg.S.S - seg.F.S};
128                 Vec v2 = {p.F - seg.F.F, p.S - seg.F.S};
129                 ll c_p = cross(v1, v2);
130                 if (c_p > 0) // order the backside by angle #868
131                     Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
132                     Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
133                     ll d_p = dot(v1, v2);
134                     ll c_p = cross(v2, v1);
135                     return atan2(c_p, d_p) / #384
136             }
137             ll d_p = dot(v1, v2);
138             double res = atan2(d_p, c_p);
139             if (d_p <= 0 && res > 0) res = -M_PI;
140             if (res > 0)
141                 res += 20;
142             } else {
143                 res = 10 - res;
144             }
145             return res
146         }));
147
148
149 template <int DIRECTION> // 1 or -1 #631
150 Vectan_point(Vec p{ // can't be internal or on border #283
151     // -1 iff CCW rotation of ray from p to res takes it away from
152     // polygon?
153     Seg &ref_p = hull.front(); // O(log(n))
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
617
618
619
619
620
621
622
623
624
625
625
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
158
```

```

151 auto best_seg = max(function<double>(Seg &)>
152   [&p, &ref_p]
153   Seg &seg) { // accuracy of used type should be coord-2
154   Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
155   Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
156   ll d_p = dot(v1, v2);
157   ll c_p = DIRECTION * cross(v2, v1)
158   return atan2(c_p, d_p); // order by signed angle
159 });
160 return best_seg->F;
161
162 SegItmax_in_dir(Vec v{ // first is the ans. O(log(n))
163   return max(
164     function<ll>(&Seg &>)([&v](Seg &seg){ return dot(v, seg.F); }));
165   } %596
166 pair<SegIt, SegIt> intersections(Seg l) { // O(log(n))
167   int x = l.S.F - l.F.F;
168   int y = l.S.S - l.F.S;
169   Vec dir = {-y, x};
170   auto it_max = max_in_dir(dir)
171   auto it_min = max_in_dir(MP(y, -x));
172   ll opt_val = dot(dir, l.F);
173   if (dot(dir, it_max->F) < opt_val ||
174       dot(dir, it_min->F) > opt_val)
175     return MP(hull.end(), hull.end()); #276
176 SegIt it_r1, it_r2;
177 function<bool>(Seg &, Seg &) > inc_c([&dir](Seg &lft, Seg &rgt) {
178   return dot(dir, lft.F) < dot(dir, rgt.F);
179 });
180 function<bool>(Seg &, Seg &) > dec_c([&dir](Seg &lft, Seg &rgt)
181   ↵ #431
182   return dot(dir, lft.F) > dot(dir, rgt.F);
183 );
183 if (it_min <= it_max) {
184   it_r1 = upper_bound(it_min, it_max + 1, l, inc_c) - 1;
185   if (dot(dir, hull.front().F) >= opt_val) #689
186     it_r2 = upper_bound(hull.begin(), it_min + 1, l, dec_c) - 1;
187   } else {
188     it_r2 = upper_bound(it_max, hull.end(), l, dec_c) - 1;
189   }
190 } else #552
191   it_r1 = upper_bound(it_max, it_min + 1, l, dec_c) - 1;
192   if (dot(dir, hull.front().F) <= opt_val) {
193     it_r2 = upper_bound(hull.begin(), it_max + 1, l, inc_c) - 1;
194   } else {
195     it_r2 = upper_bound(it_min, hull.end(), l, inc_c) - 1 #220
196   }
197 return MP(it_r1, it_r2);
198
199 Segdiameter({ // O(n)
200
#209
#762
%037
#762
#150
#632
#246
#529
#406
#362
#936
#732
%
```

```

201 Seg res;
202 ll dia_sq = 0;
203 auto it1 = hull.begin();
204 auto it2 = up_beg
205 Vec v1 = {hull.back().S.F - hull.back().F.F,
206   hull.back().S.S - hull.back().F.S};
207 while (it2 != hull.begin()) {
208   Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F,
209     (it2 - 1)->S.S - (it2 - 1)->F.S}
210   if (cross(v1, v2) > 0) break;
211   --it2;
212 }
213 while (it2 != hull.end()) { // check all antipodal pairs
214   if (dist_sq(it1->F, it2->F) > dia_sq) #246
215     res = {it1->F, it2->F};
216   dia_sq = dist_sq(res.F, res.S);
217 }
218 Vec v1 = {it1->S.F - it1->F.F, it1->S.S - it1->F.S};
219 Vec v2 = {it2->S.F - it2->F.F, it2->S.S - it2->F.S} #529
220 if (cross(v1, v2) == 0) {
221   if (dist_sq(it1->S, it2->F) > dia_sq) {
222     res = {it1->S, it2->F};
223     dia_sq = dist_sq(res.F, res.S);
224   }
225   if (dist_sq(it1->F, it2->S) > dia_sq) {
226     res = {it1->F, it2->S};
227     dia_sq = dist_sq(res.F, res.S);
228 } // report cross pairs at parallel lines. #362
229 ++it1
230 ++it2;
231 } else if (cross(v1, v2) < 0) {
232   ++it1;
233 } else {
234   ++it2
235 }
236 }
237 return res;
238 }

#936
#732
%
```

9 Delaunay triangulation $\mathcal{O}(n \log n)$

```

1 const int max_co = (1 << 28) - 5;
2 struct Vec {
3   int x, y;
4   bool operator==(const Vec &oth) { return x == oth.x && y == oth.y; }
5   bool operator!=(const Vec &oth) { return !operator==(oth); } #679
6   Vec operator-(const Vec &oth) { return {x - oth.x, y - oth.y}; }
7 };
8 llcross(Vec a, Vec b{ return (ll)a.x * b.y - (ll)a.y * b.x; }
9 lldot(Vec a, Vec b{ return (ll)a.x * b.x + (ll)a.y * b.y; }
10 struct Edge
11   Vec tar;
12   Edge *nxt;

```

```

13 Edge *inv = NULL;
14 Edge *rep = NULL;
15 bool vis = false
16 };
17 struct Seg {
18 Vec a, b;
19 bool operator==(const Seg &oth) { return a == oth.a && b == oth.b; }
20 bool operator!=(const Seg &oth) { return !operator==(oth); } #245
21 };
22 llorient(Vec a, Vec b, Vec c{
23 return (ll)a.x * (b.y - c.y) + (ll)b.x * (c.y - a.y) +
24 (ll)c.x * (a.y - b.y);
25
26 bool in_c_circle(Vec *arr, Vec d{
27 if (cross(arr[1] - arr[0], arr[2] - arr[0]) == 0)
28 return true; // degenerate
29 ll m[3][3];
30 ran(i, 0, 3) #264
31 m[i][0] = arr[i].x - d.x;
32 m[i][1] = arr[i].y - d.y;
33 m[i][2] = m[i][0] * m[i][0];
34 m[i][2] += m[i][1] * m[i][1];
35
36 __int128 res = 0;
37 res += (__int128)(m[0][0] * m[1][1] - m[0][1] * m[1][0]) * m[2][2];
38 res += (__int128)(m[1][0] * m[2][1] - m[1][1] * m[2][0]) * m[0][2];
39 res -= (__int128)(m[0][0] * m[2][1] - m[0][1] * m[2][0]) * m[1][2];
40 return res > 0 #845
41
42 Edge add_triangle(Edge *a, Edge *b, Edge *c{
43 Edge *old[] = {a, b, c};
44 Edge *tmp = new Edge[3];
45 ran(i, 0, 3) {
46 old[i]->rep = tmp + i #219
47 tmp[i] = {old[i]->tar, tmp + (i + 1) % 3, old[i]->inv};
48 if (tmp[i].inv) tmp[i].inv->inv = tmp + i;
49 }
50 return tmp;
51
52 Edge add_point(Vec p, Edge *cur{ // returns outgoing edge
53 Edge *triangle[] = {cur, cur->nxt, cur->nxt->nxt};
54 ran(i, 0, 3) {
55 if (orient(triangle[i]->tar, triangle[(i + 1) % 3]->tar, p) < 0)
56 return NULL #233
57 }
58 ran(i, 0, 3) {
59 if (triangle[i]->rep) {
60 Edge *res = add_point(p, triangle[i]->rep);
61 if (res
62 return res; // unless we are on last layer we must exit here #636
63 }

64 }
65 Edge p_as_e[p];
66 Edge tmp{cur->tar}
67 tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt); #432
68 Edge *res = tmp.inv->nxt;
69 tmp.tar = cur->tar;
70 tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
71 tmp.tar = cur->tar #359
72 res->inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
73 res->inv->inv = res;
74 return res;
75 }
76 Edge *delaunay(vector<Vec> &points) #029
77 random_shuffle(points.begin(), points.end());
78 Vec arr[] = {{4 * max_co, 4 * max_co}, {-4 * max_co, max_co},
79 {max_co, -4 * max_co}};
80 Edge *res = new Edge[3];
81 ran(i, 0, 3) res[i] = {arr[i], res + (i + 1) % 3} #480
82 for (Vec &cur : points) {
83 Edge *loc = add_point(cur, res);
84 Edge *out = loc;
85 arr[0] = cur;
86 while (true) #601
87 arr[1] = out->tar;
88 arr[2] = out->nxt->tar;
89 Edge *e = out->nxt->inv;
90 if (e && in_c_circle(arr, e->nxt->tar)) { #056
91 Edge tmp{cur}
92 tmp.inv = add_triangle(&tmp, out, e->nxt);
93 tmp.tar = e->nxt->tar;
94 tmp.inv->inv = add_triangle(&tmp, e->nxt->nxt, out->nxt->nxt);
95 out = tmp.inv->nxt;
96 continue #173
97 }
98 out = out->nxt->nxt->inv;
99 if (out->tar == loc->tar) break;
100 }
101
102 return res; #032
103
104 void extract_triangles(Edge *cur, vector<vector<Seg> > &res{ #769
105 if (!cur->vis) {
106 bool inc = true;
107 Edge *it = cur;
108 do
109 it->vis = true;
110 if (it->rep) {
111 extract_triangles(it->rep, res);
112 inc = false;
113
114 it = it->nxt; #104
}

```

```

115 } while (it != cur);
116 if (inc) {
117     Edge *triangle[3] = {cur, cur->nxt, cur->nxt->nxt};
118     res.resize(res.size() + 1) #207
119     vector<Seg> &tar = res.back();
120     ran(i, 0, 3) {
121         if ((abs(triangle[i]->tar.x) < max_co &&
122             abs(triangle[(i + 1) % 3]->tar.x) < max_co))
123             tar.push_back #011
124             {triangle[i]->tar, triangle[(i + 1) % 3]->tar};
125     }
126     if (tar.empty()) res.pop_back();
127 }
128 #602

```

10 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() { memset(nxt, 0, alpha_size * sizeof(Node *)); #248
6         int cnt = 0;
7     };
8     Node aho_corasick(vector<vector<char> > &dict{
9         Node *root = new Node;
10        root->suffix = 0 #292
11        vector<pair<vector<char> *, Node *> > state;
12        for (vector<char> &s : dict) state.emplace_back(&s, root);
13        for (int i = 0; !state.empty(); ++i) {
14            vector<pair<vector<char> *, Node *> > nstate;
15            for (auto &cur : state) #306
16                Node *nxt = cur.second->nxt[(*cur.first)[i]];
17                if (nxt) {
18                    cur.second = nxt;
19                } else {
20                    nxt = new Node #266
21                    cur.second->nxt[(*cur.first)[i]] = nxt;
22                    Node *suf = cur.second->suffix;
23                    cur.second = nxt;
24                    nxt->suffix = root; // set correct suffix link
25                    while (suf) #249
26                        if (suf->nxt[(*cur.first)[i]]) {
27                            nxt->suffix = suf->nxt[(*cur.first)[i]];
28                            break;
29                        }
30                        suf = suf->suffix #562
31                    }
32                if (cur.first->size() > i + 1) nstate.push_back(cur);
33            }
34            state = nstate #417
35        }
36    }

```

```

37     return root; #882 // auxilary functions for searching and counting
38
39 Node walk(Node *cur, char c{// longest prefix in dict that is suffix of walked string.
40     while (true) {
41         if (cur->nxt[c]) return cur->nxt[c];
42         if (!cur->suffix) return cur #414
43         cur = cur->suffix;
44     }
45 }
46
47 void cnt_matches(Node *root, vector<char> &match_in{ #529
48     Node *cur = root;
49     for (char c : match_in) {
50         cur = walk(cur, c);
51         ++cur->cnt
52     }
53 }
54 void add_cnt(Node *root{// After counting matches propagate ONCE to #156
55     // suffixes for final counts
56     vector<Node *> to_visit = {root};
57     ran(i, 0, to_visit.size()) {
58         Node *cur = to_visit[i];
59         ran(j, 0, alpha_size);
60         if (cur->nxt[j]) to_visit.push_back(cur->nxt[j]); #662
61     }
62 }
63 for (int i = to_visit.size() - 1; i > 0; --i)
64     to_visit[i]->suffix->cnt += to_visit[i]->cnt #950
65
66 int main(){ #488
67     int n, len;
68     scanf "%d %d", &len, &n);
69     vector<char> a(len + 1);
70     scanf "%s", a.data());
71     a.pop_back();
72     for (char &c : a) c -= 'a';
73     vector<vector<char> > dict(n);
74     ran(i, 0, n) {
75         scanf "%d", &len);
76         dict[i].resize(len + 1);
77         scanf "%s", dict[i].data());
78         dict[i].pop_back();
79         for (char &c : dict[i]) c -= 'a';
80     }
81     Node *root = aho_corasick(dict);
82     cnt_matches(root, a);
83     add_cnt(root);
84     ran(i, 0, n) {
85         Node *cur = root;
86         for (char c : dict[i]) cur = walk(cur, c);
87         printf "%d\n", cur->cnt);

```

```

88 }


---


11 Suffix automaton and tree  $\mathcal{O}((n+q)\log(|\alpha|))$ 


---


1 class Node {
2     private:
3         map<char, Node *>
4             nxt_char; // Map is faster than hashtable and unsorted arrays
5     public:
6         int len; // Length of longest suffix in equivalence class. #994
7         Node *suf;
8         bool has_nxt(char c) const { return nxt_char.count(c); }
9         Node *nxt(char c){ #788
10             if (!has_nxt(c)) return NULL;
11             return nxt_char[c];
12         }
13         void set_nxt(char c, Node *node){ nxt_char[c] = node; }
14         Node *split(int new_len, char c{ #449
15             Node *new_n = new Node;
16             new_n->nxt_char = nxt_char;
17             new_n->len = new_len;
18             new_n->suf = suf;
19             suf = new_n;
20             return new_n
21         } #130
22         %044
23         // Extra functions for matching and counting
24         Node *lower_depth(int depth{ // move to longest suffix of current #736
25             // with a maximum length of depth.
26             if (suf->len >= depth) return suf->lower_depth(depth);
27             return this;
28         }
29         Node *walk(char c, int depth #736
30             int &match_len) { // move to longest suffix of walked path that is
31             // a substring
32             match_len = min(match_len,
33                 len); // includes depth limit(needed for finding matches)
34             if (has_nxt(c)) { // as suffixes are in classes match_len must be
35                 // tracked externally
36                 ++match_len
37                 return nxt(c->lower_depth(depth);
38             }
39             if (suf) return suf->walk(c, depth, match_len);
40             return this;
41         } #153
42         %969
43         int paths_to_end = 0;
44         void set_as_end({ // All suffixes of current node are marked as
45             // ending nodes.
46             paths_to_end += 1;
47             if (suf) suf->set_as_end();
48         } #041
49         bool vis = false;
50         void calc_paths_to_end({ // Call ONCE from ROOT. For each node
51             // calculates number of ways to reach an

```

```

52             // end node.
53             if (!vis) { // paths_to_end is occurrence count for any strings in
54                 // current suffix equivalence class. #035
55                 vis = true;
56                 for (auto cur : nxt_char)
57                     cur.second->calc_paths_to_end();
58                 paths_to_end += cur.second->paths_to_end;
59             }
59             // Transform into suffix tree of reverse string
60             map<char, Node *> tree_links;
61             int end_dist = 1 << 30;
62             int calc_end_dist({ #996
63                 if (end_dist == 1 << 30) {
64                     if (nxt_char.empty()) end_dist = 0
65                     for (auto cur : nxt_char)
66                         end_dist = min(end_dist, 1 + cur.second->calc_end_dist());
67                 }
68                 return end_dist;
69             } #021
70             bool vis_t = false;
71             void build_suffix_tree(string &s{ // Call ONCE from ROOT.
72                 if (!vis_t) {
73                     vis_t = true;
74                     if (suf
75                         suf->tree_links[s.size() - end_dist - suf->len - 1]] = this;
76                         for (auto cur : nxt_char) cur.second->build_suffix_tree(s);
77                     }
78                 }
79             } #268
80         struct SufAuto {
81             Node *last;
82             Node *root;
83             void extend(char new_c{ #340
84                 Node *new_end = new Node
85                 new_end->len = last->len + 1;
86                 Node *suf_w_nxt = last;
87                 while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {
88                     suf_w_nxt->set_nxt(new_c, new_end);
89                     suf_w_nxt = suf_w_nxt->suf
90                 }
91                 if (!suf_w_nxt) {
92                     new_end->suf = root;
93                 } else {
94                     Node *max_sbstr = suf_w_nxt->nxt(new_c)
95                     if (suf_w_nxt->len + 1 == max_sbstr->len) {
96                         new_end->suf = max_sbstr;
97                     } else {
98                         Node *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
99                         new_end->suf = eq_sbstr
100 } #217
100 } #618
100 } #295

```

```

101     Node *w_edge_to_eq_sbstr = suf_w_nxt;
102     while (w_edge_to_eq_sbstr != 0 &&
103             w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
104         w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
105         w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf
106     }
107 }
108 last = new_end;
109
110 SufAuto(string &s) {
111     root = new Node;
112     root->len = 0;
113     root->suf = NULL;
114     last = root
115     for (char c : s) extend(c);
116     root->calc_end_dist(); // To build suffix tree use reversed string
117     root->build_suffix_tree(s);
118 }

```

12 Dinic

```

1 struct MaxFlow {
2     const static ll INF = 1e18;
3     int source, sink;
4     ll sink_pot = 0;
5     vector<int> start, now, lvl, adj, rcap, cap_loc, bfs;
6     vector<bool> visited;
7     vector<ll> cap, orig_cap/*lg*/, cost;
8     priority_queue<pair<ll, int>, vector<pair<ll, int> >,
9         greater<pair<ll, int> > >
10    dist_que/*rg*/;
11    void addd_flow(int idx, ll flow, bool cont = true{
12        cap[idx] -= flow;
13        if (cont) add_flow(rcap[idx], -flow, false);
14    }
15    MaxFlow(
16        const vector<tuple<int, int, ll, ll/*ly*/, ll/*ry*/> > &edges) {
17        for (auto &cur : edges) { // from, to, cap, rcap/*ly*/, cost/*ry*/
18            start.resize(
19                max(max(get<0>(cur), get<1>(cur)) + 2, (int)start.size()));
20            ++start[get<0>(cur) + 1];
21            ++start[get<1>(cur) + 1];
22        }
23        for (int i = 1; i < start.size(); ++i) start[i] += start[i - 1];
24        now = start;
25        adj.resize(start.back());
26        cap.resize(start.back());
27        rcap.resize(start.back());
28        /*ly*/ cost.resize(start.back()); /*ry*/
29        for (auto &cur : edges) {
30            int u, v;
31            ll c, rc/*ly*/, c_cost/*ry*/;

```

```

32             tie(u, v, c, rc/*ly*/, c_cost/*ry*/) = cur;
33             assert(u != v);
34             adj[now[u]] = v;
35             adj[now[v]] = u;
36             rcap[now[u]] = now[v];
37             rcap[now[v]] = now[u];
38             cap_loc.push_back(now[u]);
39             /*ly*/ cost[now[u]] = c_cost;
40             cost[now[v]] = -c_cost; /*ry*/
41             cap[now[u]++] = c;
42             cap[now[v]++] = rc;
43             orig_cap.push_back(c);
44         }
45     }
46     bool dinic_bfs() {
47         lvl.clear();
48         lvl.resize(start.size());
49         bfs.clear();
50         bfs.resize(1, source);
51         now = start;
52         lvl[source] = 1;
53         for (int i = 0; i < bfs.size(); ++i) {
54             int u = bfs[i];
55             while (now[u] < start[u + 1]) {
56                 int v = adj[now[u]];
57                 if /*ly*/ cost[now[u]] == 0 && /*ry*/ cap[now[u]] > 0 &&
58                     lvl[v] == 0) {
59                     lvl[v] = lvl[u] + 1;
60                     bfs.push_back(v);
61                 }
62                 ++now[u];
63             }
64         }
65         return lvl[sink];
66     }
67     ll dinic_dfs(int u, ll flow) {
68         if (u == sink) return flow;
69         while (now[u] < start[u + 1]) {
70             int v = adj[now[u]];
71             if (lvl[v] == lvl[u] + 1/*ly*/ && cost[now[u]] == 0/*ry*/ &&
72                 cap[now[u]] != 0) {
73                 ll res = dinic_dfs(v, min(flow, cap[now[u]]));
74                 if (res) {
75                     add_flow(now[u], res);
76                     return res;
77                 }
78             }
79             ++now[u];
80         }
81         return 0;
82     }

```

```

83 /*ly*/ bool recalc_dist(bool check_imp = false{
84     now = start;
85     visited.clear();
86     visited.resize(start.size());
87     dist_que.emplace(0, source);
88     bool imp = false;
89     while (!dist_que.empty()) {
90         int u;
91         ll dist;
92         tie(dist, u) = dist_que.top();
93         dist_que.pop();
94         if (!visited[u]) {
95             visited[u] = true;
96             if (check_imp && dist != 0) imp = true;
97             if (u == sink) sink_pot += dist;
98             while (now[u] < start[u + 1]) {
99                 int v = adj[now[u]];
100                if (!visited[v] && cap[now[u]])
101                    dist_que.emplace(dist + cost[now[u]], v);
102                cost[now[u]] += dist;
103                cost[rcap[now[u]++]] -= dist;
104            }
105        }
106    }
107    if (check_imp) return imp;
108    return visited[sink];
109 }                                         /*ry*/
110 /*lp*/ bool recalc_dist_bellman_ford() { // return whether there is
111                                         // a negative cycle
112     int i = 0;
113     for (; i < (int)start.size() - 1 && recalc_dist(true); ++i) {
114     }
115     return i == (int)start.size() - 1;
116 } /*rp*/
117 /*ly*/ pair<ll,/*ry*/ ll/*ly*/>//*ry*/ calc_flow(
118     int _source, int _sink) {
119     source = _source;
120     sink = _sink;
121     assert(max(source, sink) < start.size() - 1);
122     ll tot_flow = 0;
123     ll tot_cost = 0;
124     /*lp*/ if (recalc_dist_bellman_ford()) {
125         assert(false);
126     } else {                                         /*rp*/
127     /*ly*/ while (recalc_dist()) { /*ry*/
128         ll flow = 0;
129         while (dinic_bfs()) {
130             now = start;
131             ll cur;
132             while (cur = dinic_dfs(source, INF)) flow += cur;
133         }
134         tot_flow += flow;
135         /*ly*/ tot_cost += sink_pot * flow; /*ry*/
136     }
137 }
138 }                                         /*ly*/ /*ry*/ {/*ry*/ tot_flo/*ly*/ , tot_cost} /*ry*/ ;
139 */
140 ll flow_on_edge(int idx) {
141     assert(idx < cap.size());
142     return orig_cap[idx] - cap[cap_loc[idx]];
143 }
144 };
145 const int nmax = 1055;
146 int main(){
147     int t;
148     scanf("%d", &t);
149     for (int i = 0; i < t; ++i) {
150         vector<tuple<int, int, ll, ll, ll> > edges;
151         int n;
152         scanf("%d", &n);
153         for (int j = 1; j <= n; ++j) {
154             edges.emplace_back(j, 2 * n + 1, 1, 0, 0);
155         }
156         for (int j = 1; j <= n; ++j) {
157             int card;
158             scanf("%d", &card);
159             edges.emplace_back(0, card, 1, 0, 0);
160         }
161         int ex_c;
162         scanf("%d", &ex_c);
163         for (int j = 0; j < ex_c; ++j) {
164             int a, b;
165             scanf("%d %d", &a, &b);
166             if (b < a) swap(a, b);
167             edges.emplace_back(a, b, nmax, 0, 1);
168             edges.emplace_back(b, n + b, nmax, 0, 0);
169             edges.emplace_back(n + b, a, nmax, 0, 1);
170         }
171         int v = 2 * n + 2;
172         MaxFlowmf(edges;
173         printf("%d\n", (int)mf.calc_flow(0, v - 1).second);
174     }
175 /*
176     int n,m;
177     cin >> n >> m;
178     // arguments source and sink, memory usage O(largest node index +
179     // input size), sink doesn't need to be last index
180     vector<tuple<int, int, ll, ll> > edges; for(int i = 0; i < m; ++i) int a,b; ll c;
181     scanf("%d %d %lld", &a, &b, &c);
182     if(a != b)
183         edges.emplace_back(a,b,c,c); // (a,b,c,0) for directed

```

```

181 MaxFlow mf(edges);
182 cout << mf.calcflow(1, n) << '\n';
183 //cout << mf.flow_on_edge(edge_index) << endl; // return flow on
184 this edge
185 */



---



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



---


1 struct Network {
2     struct Node;
3     struct Edge {
4         Node *u, *v;
5         int f, c, cost
6         Node*from(Node* pos{
7             if (pos == u) return v;
8             return u;
9         }
10        intgetCap(Node* pos
11            if (pos == u) return c - f;
12            return f;
13        }
14        int addFlow(Node* pos, int toAdd) {
15            if (pos == u)
16                f += toAdd;
17                return toAdd * cost;
18            } else {
19                f -= toAdd;
20                return -toAdd * cost
21            }
22        }
23    };
24    struct Node {
25        vector<Edge*> conn
26        int index;
27    };
28    deque<Node> nodes;
29    deque<Edge> edges;
30    Node*addNode(
31        nodes.push_back(Node());
32        nodes.back().index = nodes.size() - 1;
33        return &nodes.back();
34    }
35    Edge*addEdge(Node* u, Node* v, int f, int c, int cost
36        edges.push_back({u, v, f, c, cost});
37        u->conn.push_back(&edges.back());
38        v->conn.push_back(&edges.back());
39        return &edges.back();
40    }
41    // Assumes all needed flow has already been added
42    intminCostMaxFlow({
43        int n = nodes.size();
44        int result = 0;
45        struct State {

```

```

46        int p
47        Edge* used;
48    };
49    while (1) {
50        vector<vector<State>> state(1, vector<State>(n, {0, 0}));
51        for (int lev = 0; lev < n; lev++)
52            state.push_back(state[lev]);
53        for (int i = 0; i < n; i++) {
54            if (lev == 0 || state[lev][i].p < state[lev - 1][i].p) {
55                for (Edge* edge : nodes[i].conn) {
56                    if (edge->getCap(&nodes[i]) > 0)
57                        int np =
58                            state[lev][i].p +
59                            (edge->u == &nodes[i] ? edge->cost : -edge->cost);
60                    int ni = edge->from(&nodes[i])->index;
61                    if (np < state[lev + 1][ni].p)
62                        state[lev + 1][ni].p = np;
63                        state[lev + 1][ni].used = edge;
64                }
65            }
66        }
67    }
68    }
69 }
70 // Now look at the last level
71 bool valid = false;
72 for (int i = 0; i < n; i++)
73    if (state[n - 1][i].p > state[n][i].p) {
74        valid = true;
75        vector<Edge*> path;
76        int cap = 1000000000;
77        Node* cur = &nodes[i]
78        int clev = n;
79        vector<bool> explr(n, false);
80        while (!explr[cur->index]) {
81            explr[cur->index] = true;
82            State cstate = state[clev][cur->index]
83            cur = cstate.used->from(cur);
84            path.push_back(cstate.used);
85        }
86        reverse(path.begin(), path.end());
87
88        int i = 0;
89        Node* cur2 = cur;
90        do {
91            cur2 = path[i]->from(cur2);
92            i++;
93        } while (cur2 != cur);
94        path.resize(i);
95    }
96    for (auto edge : path) {

```

```

97     cap = min(cap, edge->getCap(cur))          #297
98     cur = edge->from(cur);
99   }
100  for (auto edge : path) {
101    result += edge->addFlow(cur, cap);
102    cur = edge->from(cur)
103  }
104  if (!valid) break;
105 }
106 return result                                #550
107 }



---


14 DMST  $\mathcal{O}(E \log V)$ 

1 struct EdgeDesc {
2     int from, to, w;
3 };
4 struct DMST {
5     struct Node
6     struct Edge {                                #091
7         Node *from;
8         Node *tar;
9         int w;
10        bool inc
11    };
12    struct Circle {
13        bool vis = false;
14        vector<Edge *> contents;
15        void clean(int idx)                      #186
16    };
17    const static greater<pair<ll, Edge *>>
18    comp; // Can use inline static since C++17
19    static vector<Circle> to_process;
20    static bool no_dmst                         #478
21    static Node *root;
22    struct Node {
23        Node *par = NULL;
24        vector<pair<int, int>> out_cands; // Circ, edge idx
25        vector<pair<ll, Edge *>> con
26        bool in_use = false;
27        ll w = 0; // extra to add to edges in con
28        Node anc() {
29            if (!par return thi;
30            while (par->par) par = par->par
31            return par;
32        }
33        void clean() {
34            if (!no_dmst) {
35                in_use = false
36                for (auto &cur : out_cands)
37                    to_process[cur.first].clean(cur.second);
38            }
39        }
40    }
41    Node con_to_root(
42        if (anc() == root) return root;
43        in_use = true;
44        Node *super = this; // Will become root or the first Node
45                                         // encountered in a loop.
46    while (super == this) {                      #363
47        !con.empty() && con.front().second->tar->anc() == anc() {
48            pop_heap(con.begin(), con.end(), comp);
49            con.pop_back();
50        }
51        if (con.empty())                           #506
52            no_dmst = true;
53        return root;
54    }
55    pop_heap(con.begin(), con.end(), comp);
56    auto nxt = con.back()                      #541
57    con.pop_back();
58    w = -nxt.first;
59    if (nxt.second->tar
60        ->in_use) { // anc() wouldn't change anything
61        super = nxt.second->tar->anc()           #174
62        to_process.resize(to_process.size() + 1);
63    } else {
64        super = nxt.second->tar->con_to_root();
65    }
66    if (super != root)                         #595
67        to_process.back().contents.push_back(nxt.second);
68        out_cands.emplace_back(to_process.size() - 1,
69                               to_process.back().contents.size() - 1);
70    } else { // Clean circles
71        nxt.second->inc = true                 #848
72        nxt.second->from->clean();
73    }
74 }
75 if (super != root) { // we are some loops non first Node.      #860
76    if (con.size() > super->con.size())
77        swap(con,
78               super->con); // Largest con in loop should not be copied.
79        swap(w, super->w);
80    }
81    for (auto cur : con)                      #064
82        super->con.emplace_back(
83            cur.first - super->w + w, cur.second);
84        push_heap(super->con.begin(), super->con.end(), comp);
85    }
86
87    par = super; // root or anc() of first Node encountered in a      #295
88                                         // loop
89    return super;

```

```

90     }
91 };
92 Node *cur_root
93 vector<Node> graph;
94 vector<Edge> edges;
95 DMST(int n, vector<EdgeDesc> &desc,
96       int r) { // Self loops and multiple edges are okay.
97     graph.resize(n)
98     cur_root = &graph[r];
99     for (auto &cur : desc) // Edges are reversed internally
100       edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
101     for (int i = 0; i < desc.size(); ++i)
102       graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]) #895
103     for (int i = 0; i < n; ++i)
104       make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
105   }
106 bool find() {
107   root = cur_root
108   no_dmst = false;
109   for (auto &cur : graph) {
110     cur.con_to_root();
111     to_process.clear();
112     if (no_dmst) return false
113   }
114   return true;
115 }
116 ll weight(){
117   ll res = 0;
118   for (auto &cur : edges) {
119     if (cur.inc) res += cur.w;
120   }
121   return res;
122 };
123 void DMST::Circle::clean(int idx) {
124   if (!vis) {
125     vis = true;
126     for (int i = 0; i < contents.size(); ++i)
127       if (i != idx) {
128         contents[i]->inc = true;
129         contents[i]->from->clean();
130       }
131   }
132 }
133 }
134 }
135 const greater<pair<ll, DMST::Edge *>> > DMST::comp;
136 vector<DMST::Circle> DMST::to_process;
137 bool DMST::no_dmst
#417

```

15 Bridges $\mathcal{O}(n)$

```

1 struct vert;
2 struct edge {

```

```

3   bool exists = true;
4   vert *dest;
5   edge *rev
6   edge(vert *_dest) : dest(_dest) { rev = NULL; }
7   vert &operator*() { return *dest; }
8   vert *operator->() { return dest; }
9   bool is_bridge();
10 }
11 struct vert {
12   deque<edge> con;
13   int val = 0;
14   int seen;
15   int ddfs(int upd, edge *ban){ // handles multiple edges
16     if (!val) {
17       val = upd;
18       seen = val;
19       for (edge &nxt : con) {
20         if (nxt.exists && (&nxt) != ban
21             seen = min(seen, nxt->dfs(upd + 1, nxt.rev));
22       }
23     }
24     return seen;
25   }
26   void remove_adj_bridges(){
27     for (edge &nxt : con) {
28       if (nxt.is_bridge()) nxt.exists = false;
29     }
30   }
31   int cnt_adj_bridges(){
32     int res = 0;
33     for (edge &nxt : con) res += nxt.is_bridge();
34     return res;
35   };
36   bool edge::is_bridge() {
37     return exists &&
38           (dest->seen > rev->dest->val || dest->val < rev->dest->seen);
39   }
40   vert graph[nmax];
41   int main(){ // Mechanics Practice BRIDGES
42     int n, m;
43     cin >> n >> m;
44     for (int i = 0; i < m; ++i) {
45       int u, v;
46       scanf("%d %d", &u, &v);
47       graph[u].con.emplace_back(graph + v);
48       graph[v].con.emplace_back(graph + u);
49       graph[u].con.back().rev = &graph[v].con.back();
50       graph[v].con.back().rev = &graph[u].con.back();
51     }
52     graph[1].dfs(1, NULL);
53   }

```

```

54 int res = 0;
55 for (int i = 1; i <= n; ++i) res += graph[i].cnt_adj_bridges();
56 cout << res / 2 << endl;


---


16 2-Sat  $\mathcal{O}(n)$  and SCC  $\mathcal{O}(n)$ 


---


1 struct Graph {
2     int n;
3     vector<vector<int>> conn;
4     Graph(int nsize) {
5         n = nsize
6         conn.resize(n);
7     }
8     void add_edge(int u, int v) { conn[u].push_back(v); }
9     void _topsort_dfs(int pos, vector<int> &result, vector<bool> &explr,
10                      vector<vector<int>> &revconn)
11     if (explr[pos]) return;
12     explr[pos] = true;
13     for (auto next : revconn[pos])
14         _topsort_dfs(next, result, explr, revconn);
15     result.push_back(pos)
16 }
17 vector<int> topsort() {
18     vector<vector<int>> revconn(n);
19     for (int u = 0; u < n; u++) {
20         for (auto v : conn[u]) revconn[v].push_back(u)
21     }
22     vector<int> result;
23     vector<bool> explr(n, false);
24     for (int i = 0; i < n; i++)
25         _topsort_dfs(i, result, explr, revconn)
26     reverse(result.begin(), result.end());
27     return result;
28 }
29 void dfs(int pos, vector<int> &result, vector<bool> &explr) {
30     if (explr[pos]) return
31     explr[pos] = true;
32     for (auto next : conn[pos]) dfs(next, result, explr);
33     result.push_back(pos);
34 }
35 vector<vector<int>> scc() {
36     vector<int> order = topsort();
37     reverse(order.begin(), order.end());
38     vector<bool> explr(n, false);
39     vector<vector<int>> results
40     for (auto it = order.rbegin(); it != order.rend(); ++it) {
41         vector<int> component;
42         _topsort_dfs(*it, component, explr, conn);
43         sort(component.begin(), component.end());
44         results.push_back(component)
45     }
46     sort(results.begin(), results.end());
47     return results;

```

```

48     }
49 }
%983 // Solution for:
50 // http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
51 intmain(){
52     int n, m;
53     cin >> n >> m;
54     Graphg(2 * m);
55     for (int i = 0; i < n; i++) {
56         int a, sa, b, sb;
57         cin >> a >> sa >> b >> sb;
58         a--, b--;
59         g.add_edge(2 * a + 1 - sa, 2 * b + sb);
60         g.add_edge(2 * b + 1 - sb, 2 * a + sa);
61     }
62     vector<int> state(2 * m, 0);
63     {
64         vector<int> order = g.topsort();
65         vector<bool> explr(2 * m, false);
66         for (auto u : order) {
67             vector<int> traversed;
68             g.dfs(u, traversed, explr);
69             if (traversed.size() > 0 && !state[traversed[0] ^ 1]) {
70                 for (auto c : traversed) state[c] = 1;
71             }
72         }
73     }
74     for (int i = 0; i < m; i++) {
75         if (state[2 * i] == state[2 * i + 1]) {
76             cout << "IMPOSSIBLE\n";
77             return 0;
78         }
79     }
80     for (int i = 0; i < m; i++) {
81         cout << state[2 * i + 1] << '\n';
82     }
83     return 0;

```

17 Generic persistent compressed lazy segment tree

```

1 struct Seg {
2     ll sum = 0;
3     void recalc(const Seg &lhs_seg, int lhs_len, const Seg &rhs_seg,
4                 int rhs_len{
5                     sum = lhs_seg.sum + rhs_seg.sum
6                 }
7 } __attribute__((packed));
8 struct Lazy {
9     ll add;
10    ll assign_val; // LLONG_MIN if no assign;
11    void init(){
12        add = 0;
13        assign_val = LLONG_MIN;
14    }

```

```

15 Lazy() { init();
16 voidsplit(Lazy &lhs_lazy, Lazy &rhs_lazy, int len{
17     lhs_lazy = *this;
18     rhs_lazy = *this;
19     init();
20 }
21 voidmerge(Lazy &oth, int len{
22     if (oth.assign_val != LLONG_MIN) {
23         add = 0;
24         assign_val = oth.assign_val;
25     }
26     add += oth.add;
27 }
28 void apply_to_seg(Seg &cur, int len) const {
29     if (assign_val != LLONG_MIN) {
30         cur.sum = len * assign_val
31     }
32     cur.sum += len * add;
33 }
34 } __attribute__((packed))
35 struct Node { // Following code should not need to be modified
36     int ver;
37     bool is_lazy = false;
38     Seg seg;
39     Lazy lazy;
40     Node *lc = NULL, *rc = NULL;
41     voidinit(){
42         if (!lc) {
43             lc = new Node{ver};
44             rc = new Node{ver}
45         }
46     }
47     Node upd(int L, int R, int l, int r, Lazy &val, int tar_ver{
48         if (ver != tar_ver) {
49             Node *rep = new Node(*this)
50             rep->ver = tar_ver;
51             return rep->upd(L, R, l, r, val, tar_ver);
52         }
53         if (L >= l && R <= r) {
54             val.apply_to_seg(seg, R - L)
55             lazy.merge(val, R - L);
56             is_lazy = true;
57         } else {
58             init();
59             int M = (L + R) / 2
60             if (is_lazy) {
61                 Lazy l_val, r_val;
62                 lazy.split(l_val, r_val, R - L);
63                 lc = lc->upd(L, M, L, M, l_val, ver);
64                 rc = rc->upd(M, R, M, R, r_val, ver)
65                 is_lazy = false;
66             }
67         }
68     }
69     val.split(l_val, r_val, R - L);
70     if (l < M) lc = lc->upd(L, M, l, r, l_val, ver)
71     if (M < r) rc = rc->upd(M, R, l, r, r_val, ver);
72     seg.recalc(lc->seg, M - L, rc->seg, R - M);
73 }
74     return this;
75 }
76 voididget(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp,
77           bool last_ver{
78     if (L >= l && R <= r) {
79         tmp->recalc(*lft_res, L - l, seg, R - L);
80         swap(lft_res, tmp)
81     } else {
82         init();
83         int M = (L + R) / 2;
84         if (is_lazy) {
85             Lazy l_val, r_val
86             lazy.split(l_val, r_val, R - L);
87             lc = lc->upd(L, M, L, M, l_val, ver + last_ver);
88             lc->ver = ver;
89             rc = rc->upd(M, R, M, R, r_val, ver + last_ver);
90             rc->ver = ver
91             is_lazy = false;
92             if (l < M) lc->get(L, M, l, r, lft_res, tmp, last_ver);
93             if (M < r) rc->get(M, R, l, r, lft_res, tmp, last_ver);
94         }
95     }
96 } __attribute__((packed));
97 struct SegTree { // indexes start from 0, ranges are [beg, end)
98     vector<Node *> roots; // versions start from 0
99     int len
100     SegTree(int _len) : len(_len) { roots.push_back(new Node{0}); }
101     int upd(int l, int r, Lazy &val, bool new_ver = false) {
102         Node *cur_root =
103             roots.back()->upd(0, len, l, r, val, roots.size() - !new_ver);
104         if (cur_root != roots.back()) roots.push_back(cur_root)
105         return roots.size() - 1;
106     }
107     Seg get(int l, int r, int ver = -1) {
108         if (ver == -1) ver = roots.size() - 1;
109         Seg seg1, seg2
110         Seg *pres = &seg1, *ptmp = &seg2;
111         roots[ver]->get(0, len, l, r, pres, ptmp, roots.size() - 1);
112         return *pres;
113     }
114 } #542 %542
115 intmain(){
116     int n, m; // solves Mechanics Practice LAZY
117     cin >> n >> m;

```

```

118 SegTree seg_tree(1 << 17;
119 for (int i = 0; i < n; ++i) {
120     Lazy tmp;
121     scanf("%lld", &tmp.assign_val);
122     seg_tree.upd(i, i + 1, tmp);
123 }
124 for (int i = 0; i < m; ++i) {
125     int o;
126     int l, r;
127     scanf("%d %d %d", &o, &l, &r);
128     --l;
129     if (o == 1) {
130         Lazy tmp;
131         scanf("%lld", &tmp.add);
132         seg_tree.upd(l, r, tmp);
133     } else if (o == 2) {
134         Lazy tmp;
135         scanf("%lld", &tmp.assign_val);
136         seg_tree.upd(l, r, tmp);
137     } else {
138         Seg res = seg_tree.get(l, r);
139         printf("%lld\n", res.sum);
140     }
141 }
```

18 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query

```

1 class dummy {
2 public:
3     dummy() {}
4     dummy(int, int) {}
5     void set(int, int) { #531
6         intquery(int left, int right{
7             cout << this << ' ' << left << ' ' << right << endl;
8         }
9     } /*932 /* T should be the type of the data stored in each vertex;
10    * DS should be the underlying data structure that is used to perform
11    * the group operation. It should have the following methods:
12    * * DS () - empty constructor
13    * * DS (int size, T initial) - constructs the structure with the
14    * given size, initially filled with initial.
15    * * void set (int index, T value) - set the value at index `index` to
16    * `value`.
17    * * T query (int left, int right) - return the "sum" of elements
18    * between left and right, inclusive.
19 */
20 template <typename T, class DS>
21 class HLD {
22     int vertexc;
23     vector<int> *adj;
24     vector<int> subtree_size #178
25     DS structure;
26     DS aux;
```

```

27     void build_sizes(int vertex, int parent{ #037
28         subtree_size[vertex] = 1;
29         for (int child : adj[vertex])
30             if (child != parent) {
31                 build_sizes(child, vertex);
32                 subtree_size[vertex] += subtree_size[child];
33             }
34     }
35     int cur;
36     vector<int> ord;
37     vector<int> chain_root;
38     vector<int> par #593
39     void build_hld(int vertex, int parent, int chain_source{ #432
40         cur++;
41         ord[vertex] = cur;
42         chain_root[vertex] = chain_source;
43         par[vertex] = parent
44         if (adj[vertex].size() > 1 ||
45             (vertex == 1 && adj[vertex].size() == 1)) {
46             int big_child, big_size = -1;
47             for (int child : adj[vertex]) {
48                 if ((child != parent) && (subtree_size[child] > big_size))
49                     #042
50                     big_child = child;
51                     big_size = subtree_size[child];
52             }
53         }
54         build_hld(big_child, vertex, chain_source) #254
55         for (int child : adj[vertex]) {
56             if ((child != parent) && (child != big_child))
57                 build_hld(child, vertex, child);
58         }
59     }
60     public:
61     HLD(int _vertexc) { #461
62         vertexc = _vertexc;
63         adj = new vector<int>[vertexc + 5] #800
64     }
65     void add_edge(int u, int v) {
66         adj[u].push_back(v);
67         adj[v].push_back(u);
68     }
69     void build(T initial{ #587
70         subtree_size = vector<int>(vertexc + 5);
71         ord = vector<int>(vertexc + 5);
72         chain_root = vector<int>(vertexc + 5);
73         par = vector<int>(vertexc + 5);
74         cur = 0;
75         build_sizes(1, -1); #976
76     }
```

```

77     build_hld(1, -1, 1);
78     structure = DS(vertexc + 5, initial);
79     aux = DS(50, initial)                                #638
80 }
81 void set(int vertex, int value) {
82     structure.set(ord[vertex], value);
83 }
84 T query_path
85     int u, int v {/* returns the "sum" of the path u->v */}
86     int cur_id = 0;
87     while (chain_root[u] != chain_root[v]) {
88         if (ord[u] > ord[v]) {
89             cur_id++                                         #052
90             aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
91             u = par[chain_root[u]];
92         } else {
93             cur_id++;                                         #485
94             aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]))
95             v = par[chain_root[v]];
96         }
97     }
98     cur_id++;
99     aux.set(cur_id)                                       #041
100    structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
101    return aux.query(1, cur_id);                           %905
102
103 void print(){
104     for (int i = 1; i <= vertexc; i++)
105         cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' '
106         << par[i] << endl;
107 }
108 };
109 int main(){
110     int vertexc;
111     cin >> vertexc;
112     HLD<int, dummy> hld(vertexc);
113     for (int i = 0; i < vertexc - 1; i++) {
114         int u, v;
115         cin >> u >> v;
116         hld.add_edge(u, v);
117     }
118     hld.build(0);
119     hld.print();
120     int queryc;
121     cin >> queryc;
122     for (int i = 0; i < queryc; i++) {
123         int u, v;
124         cin >> u >> v;
125         hld.query_path(u, v);
126         cout << endl;
127     }

```

19 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ per query

```

1 // Fully overloaded any dimensional BIT, use any type for coordinates,
2 // elements, return_value. Includes coordinate compression.
3 template <typename elem_t, typename coord_t, coord_t n_inf,
4           typename ret_t>
5 class BIT {
6     vector<coord_t> positions;
7     vector<elem_t> elems;
8     bool initiated = false;                                #324
9 public:
10    BIT() { positions.push_back(n_inf); }
11    void initiate() {
12        if (initiated)
13            for (elem_t &c_elem : elems) c_elem.initiate();   #330
14        else {
15            initiated = true;
16            sort(positions.begin(), positions.end());
17            positions.resize(unique(positions.begin(), positions.end()) - #822
18                               positions.begin());
19            elems.resize(positions.size());
20        }
21    }
22    template <typename... loc_form>                      #620
23    void update(coord_t cord, loc_form... args{
24        if (initiated) {
25            int pos =
26                lower_bound(positions.begin(), positions.end(), cord) - #346
27                positions.begin()
28            for (; pos < positions.size(); pos += pos & -pos)
29                elems[pos].update(args...);
30        } else {
31            positions.push_back(cord);                         #542
32        }
33    }
34    template <typename... loc_form>
35    ret_t query(coord_t cord,
36                loc_form... args) { // sum in open interval (-inf, cord)   #326
37        ret_t res = 0
38        int pos = (lower_bound(positions.begin(), positions.end(), cord) - #549
39                    positions.begin()) - 1;
40        for (; pos > 0; pos -= pos & -pos)
41            res += elems[pos].query(args...);
42        return res;
43    }
44 };
45 template <typename internal_type>
46 struct wrapped
47     internal_type a = 0;                                #616

```

```

49 voidupdate(internal_type b{ a += b; }
50 internal_typequery({ return a; }
51 // Should never be called, needed for compilation
52 voidinitiate({ cerr <<'i' << endl; }
53 voidupdate({ cerr <<'u' << endl;
54 }
55 intmain(){
56 // return type should be same as type inside wrapped
57 BIT<BIT<wrapped<ll>, int, INT_MIN, ll>, int, INT_MIN, ll> fenwick;
58 int dim = 2;
59 vector<tuple<int, int, ll> > to_insert;
60 to_insert.emplace_back(1, 1, 1);
61 // set up all positions that are to be used for update
62 for (int i = 0; i < dim; ++i) {
63     for (auto &cur : to_insert)
64         fenwick.update(get<0>(cur),
65             get<1>(cur));// May include value which won't be used
66     fenwick.initiate();
67 }
68 // actual use
69 for (auto &cur : to_insert)
70     fenwick.update(get<0>(cur), get<1>(cur), get<2>(cur));
71 cout << fenwick.query(2, 2) << '\n';

```

20 Treap $\mathcal{O}(\log n)$ per query

```

1 mt19937 randgen;
2 struct Treap {
3     struct Node {
4         int key;
5         int value;
6         unsigned int priority;
7         long long total;
8         Node* lch;
9         Node* rch;
10        Node(int new_key, int new_value)
11            key = new_key;
12            value = new_value;
13            priority = randgen();
14            total = new_value;
15            lch = 0
16            rch = 0;
17    }
18    void update() {
19        total = value;
20        if (lch) total += lch->total
21        if (rch) total += rch->total;
22    }
23};
24 deque<Node> nodes;
25 Node* root = 0
26 pair<Node*, Node*> split(int key, Node* cur) {
27     if (cur == 0) return {0, 0};

```

#636
#714

#295

#633

```

28     pair<Node*, Node*> result;
29     if (key <= cur->key) {
30         auto ret = split(key, cur->lch)
31         cur->lch = ret.second;
32         result = {ret.first, cur};
33     } else {
34         auto ret = split(key, cur->rch);
35         cur->rch = ret.first
36         result = {cur, ret.second};
37     }
38     cur->update();
39     return result;
40 }
41 Node*merge(Node* left, Node* right{
42     if (left == 0) return right;
43     if (right == 0) return left;
44     Node* top;
45     if (left->priority < right->priority)
46         left->rch = merge(left->rch, right);
47         top = left;
48     } else {
49         right->lch = merge(left, right->lch);
50         top = right
51     }
52     top->update();
53     return top;
54 }
55 void insert(int key, int value)
56     nodes.push_back(Node(key, value));
57     Node* cur = &nodes.back();
58     pair<Node*, Node*> ret = split(key, root);
59     cur = merge(ret.first, cur);
60     cur = merge(cur, ret.second)
61     root = cur;
62 }
63 void erase(int key) {
64     Node *left, *mid, *right;
65     tie(left, mid) = split(key, root)
66     tie(mid, right) = split(key + 1, mid);
67     root = merge(left, right);
68 }
69 long long sum_upto(int key, Node* cur) {
70     if (cur == 0) return 0
71     if (key <= cur->key) {
72         return sum_upto(key, cur->lch);
73     } else {
74         long long result = cur->value + sum_upto(key, cur->rch);
75         if (cur->lch) result += cur->lch->total
76         return result;
77     }
78 }

```

#233
#988
#230
#282
#510
#918
#760
#416
#634
#122

#232
#295
#633

```

79 long long get(int l, int r) {
80     return sum_upto(r + 1, root) - sum_upto(l, root)          #509
81 }
82 } %959 // Solution for:
83 // http://codeforces.com/group/U01GDa2Gub/contest/219104/problem/TREAP
84 intmain{
85     ios_base::sync_with_stdio(false);
86     cin.tie(0);
87     int m;
88     Treap treap;
89     cin >> m;
90     for (int i = 0; i < m; i++) {
91         int type;
92         cin >> type;
93         if (type == 1) {
94             int x, y;
95             cin >> x >> y;
96             treap.insert(x, y);
97         } else if (type == 2) {
98             int x;
99             cin >> x;
100            treap.erase(x);
101        } else {
102            int l, r;
103            cin >> l >> r;
104            cout << treap.get(l, r) << endl;
105        }
106    }
107    return 0;

```

21 Radixsort 50M 64 bit integers as single array in 1 sec

```

1 typedef unsigned char uchar;
2 template <typename T>
3 void msd_radixsort(
4     T *start, T *sec_start, int arr_size, int d = sizeof(T) - 1) {      #866
5     const int msd_radix_lim = 100
6     const T mask = 255;
7     int bucket_sizes[256]{};
8     for (T *it = start; it != start + arr_size; ++it) {
9         ++bucket_sizes[((*it) >> (d * 8)) & mask];
10        //++bucket_sizes[*((uchar*)it + d)];                                #772
11
12     T *locs_mem[257];
13     locs_mem[0] = sec_start;
14     T **locs = locs_mem + 1;
15     locs[0] = sec_start;
16     for (int j = 0; j < 255; ++j)                                         #818
17         locs[j + 1] = locs[j] + bucket_sizes[j];
18
19     for (T *it = start; it != start + arr_size; ++it) {                  #361
20         uchar bucket_id = ((*it) >> (d * 8)) & mask;
21         *(locs[bucket_id]++) = *it

```

```

22     }
23     locs = locs_mem;
24     if (d) {
25         T *locs_old[256];
26         locs_old[0] = start                                         #153
27         for (int j = 0; j < 255; ++j) {
28             locs_old[j + 1] = locs_old[j] + bucket_sizes[j];
29         }
30         for (int j = 0; j < 256; ++j) {                                #867
31             if (locs[j + 1] - locs[j] < msd_radix_lim)
32                 std::sort(locs[j], locs[j + 1]);
33             if (d & 1) {
34                 copy(locs[j], locs[j + 1], locs_old[j]);
35             }
36         } else
37             msd_radixsort(locs[j], locs_old[j], bucket_sizes[j], d - 1); #946
38     }
39 }
40 } %225
41 const int nmax = 5e7;
42 ll arr[nmax], tmp[nmax];
43 intmain{
44     for (int i = 0; i < nmax; ++i) arr[i] = ((ll)rand() << 32) | rand();
45     msd_radixsort(arr, tmp, nmax);
46     assert(is_sorted(arr, arr + nmax));

```

22 FFT 5M length/sec

integer $c = a * b$ is accurate if $c_i < 2^{49}$

```

1 struct Complex {
2     double a = 0, b = 0;
3     Complex &operator/=(const int &oth) {                               #139
4         a /= oth;
5         b /= oth
6         return *this;
7     }
8 }
9 Complex operator+(const Complex &lft, const Complex &rgt) {           #384
10    return Complex{lft.a + rgt.a, lft.b + rgt.b}
11 }
12 Complex operator-(const Complex &lft, const Complex &rgt) {           #385
13    return Complex{lft.a - rgt.a, lft.b - rgt.b};
14 }
15 Complex operator*(const Complex &lft, const Complex &rgt)           #560
16    return Complex{
17        lft.a * rgt.a - lft.b * rgt.b, lft.a * rgt.b + lft.b * rgt.a};
18 }
19 Complex conj(const Complex &cur) { return Complex{cur.a, -cur.b}; }
20 void fft_rec(Complex *arr, Complex *root_pow, int len)                  #385
21     if (len != 1) {
22         fft_rec(arr, root_pow, len >> 1);

```

```

23     fft_rec(arr + len, root_pow, len >> 1);
24 }
25 root_pow += len
26 for (int i = 0; i < len; ++i) {
27     Complex tmp = arr[i] + root_pow[i] * arr[i + len];
28     arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
29     arr[i] = tmp;
30 }
31 }
32 void fft(vector<Complex> &arr, int ord, bool invert) {
33     assert(arr.size() == 1 << ord);
34     static vector<Complex> root_pow(1);
35     static int inc_pow = 1
36     static bool is_inv = false;
37     if (inc_pow <= ord) {
38         int idx = root_pow.size();
39         root_pow.resize(1 << ord);
40         for (; inc_pow <= ord; ++inc_pow) #517
41             for (int idx_p = 0; idx_p < 1 << (ord - 1);
42                 idx_p += 1 << (ord - inc_pow), ++idx) {
43                 root_pow[idx] = Complex{cos(-idx_p * M_PI / (1 << (ord - 1))), #105
44                     sin(-idx_p * M_PI / (1 << (ord - 1)))};
45                 if (is_inv) root_pow[idx].b = -root_pow[idx].b
46             }
47     }
48     if (invert != is_inv) { #750
49         is_inv = invert
50         for (Complex &cur : root_pow) cur.b = -cur.b;
51     }
52     for (int i = 1, j = 0; i < (1 << ord); ++i) {
53         int m = 1 << (ord - 1);
54         bool cont = true
55         while (cont) { #122
56             cont = j & m;
57             j ^= m;
58             m >>= 1;
59         }
60         if (i < j) swap(arr[i], arr[j]);
61     }
62     fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
63     if (invert)
64         for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord) #343
65     %380
66 }
67 void mult_poly_mod(
68     vector<int> &a, vector<int> &b, vector<int> &c // c += a*b
69     static vector<Complex>
70     arr[4]; // correct upto 0.5-2M elements(mod ~ 1e9)
71     if (c.size() < 400)
72         for (int i = 0; i < a.size(); ++i)
73             for (int j = 0; j < b.size() && i + j < c.size(); ++j)
74                 c[i + j] = ((ll)a[i] * b[j] + c[i + j]) % mod;
75 }
76 } else {
77     int fft_ord = 32 - __builtin_clz(c.size()) #629
78     if (arr[0].size() != 1 << fft_ord)
79         for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord);
80     for (int i = 0; i < 4; ++i)
81         fill(arr[i].begin(), arr[i].end(), Complex{});
82     for (int &cur : a #591
83         if (cur < 0) cur += mod;
84     for (int &cur : b)
85         if (cur < 0) cur += mod;
86     const int shift = 15;
87     const int mask = (1 << shift) - 1
88     for (int i = 0; i < min(a.size(), c.size()); ++i) #625
89         arr[0][i].a = a[i] & mask;
90         arr[1][i].a = a[i] >> shift;
91     for (int i = 0; i < min(b.size(), c.size()); ++i) #528
92         arr[0][i].b = b[i] & mask;
93         arr[1][i].b = b[i] >> shift;
94     for (int i = 0; i < 2; ++i) fft(arr[i], fft_ord, false);
95     for (int i = 0; i < 2; ++i) #644
96         for (int j = 0; j < 2; ++j) {
97             int tar = 2 + (i + j) / 2;
98             Complex mult = {0, -0.25};
99             if (i ^ j) mult = {0.25, 0};
100            for (int k = 0; k < (1 << fft_ord); ++k) #983
101                int rev_k = ((1 << fft_ord) - k) % (1 << fft_ord);
102                Complex ca = arr[i][k] + conj(arr[i][rev_k]);
103                Complex cb = arr[j][k] - conj(arr[j][rev_k]);
104                arr[tar][k] = arr[tar][k] + mult * ca * cb;
105            }
106        }
107    }
108    for (int i = 2; i < 4; ++i) { #471
109        fft(arr[i], fft_ord, true);
110        for (int k = 0; k < (int)c.size(); ++k) #403
111            c[k] = (c[k] + (((ll)(arr[i][k].a + 0.5) % mod)
112                            << (shift * 2 * (i - 2)))) %
113                            mod;
114            c[k] = (c[k] + (((ll)(arr[i][k].b + 0.5) % mod)
115                            << (shift * (2 * (i - 2) + 1)))) #108
116                            mod;
117        }
118    }
119 }
120 }



---



### 23 Fast mod mult, Rabin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$

1 struct ModArithm {
2     ull n;
```

```

3 ld rec;
4 ModArithm(ull _n) : n(_n) { // n in [2, 1<<63)
5     rec = 1.0L / n
6 }
7 ull multf(ull a, ull b) { // a, b in [0, min(2*n, 1<<63))
8     ull mult = (ld)a * b * rec + 0.5L;
9     ll res = a * b - mult * n;
10    if (res < 0) res += n
11    return res; // in [0, n-1)
12 }
13 ull sqp1(ull a) { return multf(a, a) + 1; }
14 }
15 ull pow_mod(ull a, ull n, ModArithm &arithm{
16     ull res = 1;
17     for (ull i = 1; i <= n; i <= 1) {
18         if (n & i) res = arithm.mulf(res, a);
19         a = arithm.mulf(a, a)
20     }
21     return res;
22 }
23 vector<char> small_primes = {
24     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
25 bool is_prime(ull n){ // n <= 1<<63, 1M rand/s
26     ModArithm arithm(n);
27     if (n == 2 || n == 3) return true
28     if (!(n & 1) || n == 1) return false;
29     ull s = __builtin_ctz(n - 1);
30     ull d = (n - 1) >> s;
31     for (ull a : small_primes) {
32         if (a >= n) break
33         a = pow_mod(a, d, arithm);
34         if (a == 1 || a == n - 1) continue;
35         for (ull r = 1; r < s; ++r) {
36             a = arithm.mulf(a, a);
37             if (a == 1) return false
38             if (a == n - 1) break;
39         }
34         if (a != n - 1) return false;
40     }
41     return true
42 }
43
44 ll pollard_rho(ll n{
45     ModArithm arithm(n);
46     int cum_cnt = 64 - __builtin_clz(n);
47     cum_cnt *= cum_cnt / 5 + 1;
48     while (true)
49         ll lv = rand() % n;
50         ll v = arithm.sqp1(lv);
51         int idx = 1;
52         int tar = 1;
53         while (true)
#237
#780
#493
#758
%144
#104
#402
#876
#806
%975
#118
#290
54         ll cur = 1;
55         ll v_cur = v;
56         int j_stop = min(cum_cnt, tar - idx);
57         for (int j = 0; j < j_stop; ++j) {
58             cur = arithm.mulf(cur, abs(v_cur - lv));
59             v_cur = arithm.sqp1(v_cur);
60             ++idx;
61         }
62         if (!cur) {
63             for (int j = 0; j < cum_cnt; ++j)
64                 ll g = __gcd(abs(v - lv), n);
65                 if (g == 1) {
66                     v = arithm.sqp1(v);
67                 } else if (g == n) {
68                     break;
69                 } else {
70                     return g;
71                 }
72             break;
73         } else {
74             ll g = __gcd(cur, n);
75             if (g != 1) return g;
76         }
77         v = v_cur;
78         idx += j_stop;
79         if (idx == tar) {
80             lv = v;
81             tar *= 2;
82             v = arithm.sqp1(v);
83             ++idx;
84         }
85     }
86 }
87
88 map<ll, int> prime_factor(ll n,
89     map<ll, int> *res = NULL) { // n <= 1<<61, ~1000/s (<500/s on CF)
90     if (!res) {
91         map<ll, int> res_act;
92         for (int p : small_primes)
#770
#612
#963
93             while (!(n % p)) {
94                 ++res_act[p];
95                 n /= p;
96             }
97         }
98         if (n != 1) prime_factor(n, &res_act);
99         return res_act;
100     }
101     if (is_prime(n)) {
102         ++(*res)[n]
103     } else {
104
#468
#912
#906
#208
#298
#174
%542
#963

```

```

105    ll factor = pollard_rho(n);
106    prime_factor(factor, res);
107    prime_factor(n / factor, res);
108
109    return map<ll, int>();

```

24 Symmetric Submodular Functions; Queyranne's algorithm

SSF: such function $f : V \rightarrow R$ that satisfies $f(A) = f(V/A)$ and for all $x \in V, X \subseteq Y \subseteq V$ it holds that $f(X+x) - f(X) \leq f(Y+x) - f(Y)$. **Hereditary family:** such set $I \subseteq 2^V$ so that $X \subset Y \wedge Y \in I \Rightarrow X \in I$. **Loop:** such $v \in V$ so that $v \notin I$. breaklines

```

1 def minimize():
2     s = merge_all_loops()
3     while size >= 3:
4         t, u = find_pp()
5         {u} is a possible minimizer
6         tu = merge(t, u)
7         if tu not in I:
8             s = merge(tu, s)
9     for x in V:
10        {x} is a possible minimizer
11 def find_pp():
12     W = {s} # s as in minimizer()
13     todo = V/W
14     ord = []
15     while len(todo) > 0:
16         x = min(todo, key=lambda x: f(W+{x}) - f({x}))
17         W += {x}
18         todo -= {x}
19         ord.append(x)
20     return ord[-1], ord[-2]
21 def enum_all_minimal_minimizers(X):
22     # X is a inclusionwise minimal minimizer
23     s = merge(s, X)
24     yield X
25     for {v} in I:
26         if f({v}) == f(X):
27             yield X
28             s = merge(v, s)
29     while size(V) >= 3:
30         t, u = find_pp()
31         tu = merge(t, u)
32         if tu not in I:
33             s = merge(tu, s)
34         elif f({tu}) == f(X):
35             yield tu
36             s = merge(tu, s)

```

25 Berlekamp-Massey $O(\mathcal{LN})$

```

1 template <typename K>
2 static vector<K> berlekamp_massey(vector<K> ss) {
3     vector<K> ts(ss.size());

```

```

4     vector<K> cs(ss.size()); #349
5     cs[0] = K::unity
6     fill(cs.begin() + 1, cs.end(), K::zero);
7     vector<K> bs = cs;
8     int l = 0, m = 1;
9     K b = K::unity;
10    for (int k = 0; k < (int)ss.size(); k++) #390
11        K d = ss[k];
12        assert(l <= k);
13        for (int i = 1; i <= l; i++) d += cs[i] * ss[k - i];
14        if (d == K::zero) { #445
15            m++;
16        } else if (2 * l <= k) {
17            K w = d / b;
18            ts = cs;
19            for (int i = 0; i < (int)cs.size() - m; i++) #661
20                cs[i + m] -= w * bs[i];
21            l = k + 1 - l;
22            swap(bs, ts);
23            b = d;
24            m = 1;
25        } else {
26            K w = d / b;
27            for (int i = 0; i < (int)cs.size() - m; i++) #815
28                cs[i + m] -= w * bs[i];
29            m++;
30        }
31    cs.resize(l + 1); #888
32    while (cs.back() == K::zero) cs.pop_back();
33    return cs;
34

```