

University of Tartu ICPC Team Notebook

(2018-2019) March 14, 2019

1	crc.sh
2	2D geometry
3	3D geometry
4	gcc ordered set
5	PRNGs and Hash functions
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 $\mathcal{O}(\text{cap} \cdot nm)$
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	Templated HLD $\mathcal{O}(M(n) \log n)$ per query
19	Splay Tree + Link-Cut $\mathcal{O}(N \log N)$
20	Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$ per query
21	Treap $\mathcal{O}(\log n)$ per query
22	Radixsort 50M 64 bit integers as single array in 1 sec

23	FFT 5M length/sec	22
24	Fast mod mult, Rabbin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$	23
25	Symmetric Submodular Functions; Queyrannes's algorithm	24
26	Berlekamp-Massey $\mathcal{O}(\mathcal{L}N)$	25
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		

```

1 alias g++='g++ -g -Wall -Wshadow -DCDEBUG' #.basrc
2 alias a='setxkbmap us -option'
3 alias m='setxkbmap us -option caps:escape'
4 alias ma='setxkbmap us -variant dvp -option caps:escape'
5 #settings
6 gsettings set
7   ↳ org.compiz.core:/org/compiz/profiles/Default/plugins/core/ hsize 4
8 gsettings set org.gnome.desktop.wm.preferences focus-mode 'sloppy'
9 set si cin #.vimrc
10 set ts=4 sw=4 noet
11 set cb=unnamed
12 (global-set-key (kbd "C-x <next>") 'other-window) #.emacs
13 (global-set-key (kbd "C-x <prior>") 'previous-multiframe-window)
14 (global-set-key (kbd "C-M-z") [])ansi-term)
15 (global-linum-mode 1)
16 (column-number-mode 1)
17 (show-paren-mode 1)
18 (setq-default indent-tabs-mode nil)
19 valgrind --vgdb-error=0 ./a <inp & #valgrind
20 gdb a
21 target remote | vgdb

```

```

1 crc.sh
2 #!/bin/envbash
3 for j in `seq 1 1 200`; do
4   sed '/~\s*$/d' $1 | head -$j | tr -d '[:space:]' | cksum | cut -f1
5   ↳ -d ' ' | tail -c 5 #whistespaces don't matter.
6 done #there shouldn't be any COMMENTS.
7 #copy lines being checked to separate file.
8 # $ ./crc.sh tmp.cpp | grep XXXX

```

2 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
        // paral 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; // = |O_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);
```

3 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:

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

4 gcc ordered set

```

1 #define DEBUG(...) cerr << __VA_ARGS__ << endl;
2 #ifndef CDEBUG
3 #undef DEBUG
4 #define DEBUG(...) ((void)0);
5 #define NDEBUG
6 #endif
7 #define ran(i, a, b) for (auto i = (a); i < (b); i++)
8 #include <bits/stdc++.h>
9 typedef long long ll;
10 typedef long double ld;
11 using namespace std;
12 #include <ext/pb_ds/assoc_container.hpp>
13 #include <ext/pb_ds/tree_policy.hpp>
14 using namespace __gnu_pbds;
15 template <typename T>
16 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
17   tree_order_statistics_node_update>;
18 int main() {
19   ordered_set<int> cur;
20   cur.insert(1);
21   cur.insert(3);
22   cout << cur.order_of_key(2)
23     << endl; // the number of elements in the set less than 2
24   cout << *cur.find_by_order(0)
25     << endl; // the 0-th smallest number in the set(0-based)
26   cout << *cur.find_by_order(1)
27     << endl; // the 1-th smallest number in the set(0-based)
28 }

```

#1736

#5119

#3802

#0578

%4198

5 PRNGs and Hash functions

```

1 mt19937 gen;
2 uint64_t rand64() { return gen() ^ ((uint64_t)gen() << 32); }
3 uint64_t rand64() {
4   static uint64_t x = 1; //x != 0
5   x ^= x >> 12;
6   x ^= x << 25;
7   x ^= x >> 27;
8   return x * 0x2545f4914f6cdd1d; // can remove mult
9 }
10 uint64_t mix(uint64_t x){ //can replace constant with variable
11   uint64_t mem[2] = { x, 0xdeadbeeffeebdaedull };
12   asm volatile (
13     "pxor %%xmm0, %%xmm0;"
14     "movdqa (%0), %%xmm1;"
15     "aesenc %%xmm0, %%xmm1;"
16     "movdqa %%xmm1, (%0);"
17     :
18     : "r" (&mem[0])
19     : "memory"
20   );
21   return mem[0]; // use both slots for 128 bit hash

```

#2024

#6956

```

22 }
23 uint64_t mix(uint64_t x) { //x != 0
24   x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
25   x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
26   x = x ^ (x >> 31);
27   return x;
28 }
29 uint64_t unmix(uint64_t x) {
30   x = (x ^ (x >> 31) ^ (x >> 62)) * 0x319642b2d24d8ec3;
31   x = (x ^ (x >> 27) ^ (x >> 54)) * 0x96de1b173f119089;
32   x = x ^ (x >> 30) ^ (x >> 60);
33   return x;
34 }
35 uint64_t combine(uint64_t x, uint64_t y) {
36   if (y < x) swap(x, y); // remove for ord
37   return mix(mix(x) + y);
38 }

```

%9499

#7126

%1575

#4780

%2094

%1466

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; // vertices of the triangle
5 bool collinear() {
6   double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7   if (min_diff < coord_max * min_delta) return true;
8   point sp = (B - A) / (C - A);
9   double ang = M_PI / 2 - abs(abs(arg(sp)) - M_PI / 2);
10  return ang < min_delta; // positive angle with the real line
11 }
12 point circum_center() {
13   if (collinear()) return point(NAN, NAN);
14   // squared lengths of sides
15   double a2 = norm(B - C);
16   double b2 = norm(A - C);
17   double c2 = norm(A - B);
18   // barycentric coordinates of the circumcenter
19   double c_A = a2 * (b2 + c2 - a2); // sin(2 * alpha) works also
20   double c_B = b2 * (a2 + c2 - b2);
21   double c_C = c2 * (a2 + b2 - c2);
22   double sum = c_A + c_B + c_C;
23   c_A /= sum;
24   c_B /= sum;
25   c_C /= sum;
26   return c_A * A + c_B * B + c_C * C; // cartesian
27 }
28 point centroid() { // center of mass
29   return (A + B + C) / 3.0;
30 }
31 point ortho_center() { // euler line
32   point O = circum_center();
33   return O + 3.0 * (centroid() - O);

```

#0823

%8446

#6715

#9407

%6856

#3895

```

34 };
35 point nine_point_circle_center() { // euler line
36     point O = circum_center();
37     return O + 1.5 * (centroid() - O); #8193
38 }; %3031
39 point in_center() {
40     if (collinear()) return point(NAN, NAN);
41     double a = abs(B - C); // side lengths
42     double b = abs(A - C);
43     double c = abs(A - B); #5954
44     // trilinear coordinates are (1,1,1)
45     double sum = a + b + c;
46     a /= sum;
47     b /= sum;
48     c /= sum; // barycentric
49     return a * A + b * B + c * C; // cartesian
50 } #4892 %4892

```

7 Seg-Seg intersection, halfplane intersection area

```

1 struct Seg {
2     Vec a, b;
3     Vec d() { return b - a; }
4 };
5 Vec intersection(Seg l, Seg r) { #6327
6     Vec dl = l.d(), dr = r.d();
7     if (cross(dl, dr) == 0) return {nanl(""), nanl("")};
8     double h = cross(dr, l.a - r.a) / len(dr);
9     double dh = cross(dr, dl) / len(dr);
10    return l.a + dl * (h / -dh); #8893
11 }
12 // Returns the area bounded by halfplanes
13 double calc_area(const vector<Seg>& lines) {
14     double lb = -HUGE_VAL, ub = HUGE_VAL;
15     vector<Seg> slines[2];
16     for (auto line : lines) { #1804
17         if (line.b.y == line.a.y) {
18             if (line.a.x < line.b.x) {
19                 lb = max(lb, line.a.y);
20             } else {
21                 ub = min(ub, line.a.y); #6288
22             }
23         } else if (line.a.y < line.b.y) {
24             slines[1].push_back(line);
25         } else {
26             slines[0].push_back({line.b, line.a}); #3607
27         }
28     }
29     ran(i, 0, 2) {
30         sort(slines[i].begin(), slines[i].end(), [&](Seg l, Seg r) {
31             if (cross(l.d(), r.d()) == 0) #4919
32                 return normal(l.d()) * l.a > normal(r.d()) * r.a;
33             return (1 - 2 * i) * cross(l.d(), r.d()) < 0;

```

```

34     });
35 }
36 // Now find the application area of the lines and clean up redundant
37 // ones
38 vector<double> ap_s[2]; #9949
39 ran(side, 0, 2) {
40     vector<double>& apply = ap_s[side];
41     vector<Seg> clines;
42     for (auto line : slines[side]) {
43         while (clines.size() > 0) { #3099
44             Seg other = clines.back();
45             if (cross(line.d(), other.d()) != 0) {
46                 double start = intersection(line, other).y;
47                 if (start > apply.back()) break;
48             } #7856
49             clines.pop_back();
50             apply.pop_back();
51         }
52         if (clines.size() == 0) {
53             apply.push_back(-HUGE_VAL); #0868
54         } else {
55             apply.push_back(intersection(line, clines.back()).y);
56         }
57         clines.push_back(line);
58     } #8545
59     slines[side] = clines;
60 }
61 ap_s[0].push_back(HUGE_VAL);
62 ap_s[1].push_back(HUGE_VAL);
63 double result = 0; #3234
64 {
65     double lb = -HUGE_VAL, ub;
66     for (int i = 0, j = 0;
67          i < (int)slines[0].size() && j < (int)slines[1].size();
68          lb = ub) { #4531
69         ub = min(ap_s[0][i + 1], ap_s[1][j + 1]);
70         double alb = lb, aub = ub;
71         Seg l[2] = {slines[0][i], slines[1][j]};
72         if (cross(l[1].d(), l[0].d()) > 0) {
73             alb = max(alb, intersection(l[0], l[1]).y);
74         } else if (cross(l[1].d(), l[0].d()) < 0) { #2627
75             aub = min(aub, intersection(l[0], l[1]).y);
76         }
77         alb = max(alb, lb);
78         aub = min(aub, ub);
79         aub = max(aub, alb); #8493
80         ran(k, 0, 2) {
81             double x1 =
82                 l[0].a.x + (alb - l[0].a.y) / l[0].d().y * l[0].d().x;
83             double x2 =
84                 l[0].a.x + (aub - l[0].a.y) / l[0].d().y * l[0].d().x;

```

```

85     result += (-1 + 2 * k) * (aub - alb) * (x1 + x2) / 2;
86 }
87 if (ap_s[0][i + 1] < ap_s[1][j + 1]) {
88     i++;
89 } else {
90     j++;
91 }
92 }
93 }
94 return result;
95 }

```

#9267

#3074

%0513

8 Convex polygon algorithms

```

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

```

#6913

#3216

%8008

#4033

#3588

#6639

#6560

```

40     up_beg = hull.begin() + diff;
41 }
42 bool contains(Vec p) { // O(log(n))
43     if (p < hull.front().F || p > up_beg->F) return false;
44     {
45         auto it_low = lower_bound(
46             hull.begin(), up_beg, MP(MP(p.F, (int)-2e9), MP(0, 0)));
47         if (it_low != hull.begin()) --it_low;
48         Vec a = {it_low->S.F - it_low->F.F, it_low->S.S - it_low->F.S};
49         Vec b = {p.F - it_low->F.F, p.S - it_low->F.S};
50         if (cross(a, b) < 0) // < 0 is inclusive, <= 0 is exclusive
51             return false;
52     }
53     {
54         auto it_up = lower_bound(hull.rbegin(),
55             hull.rbegin() + (hull.end() - up_beg),
56             MP(MP(p.F, (int)2e9), MP(0, 0)));
57         if (it_up - hull.rbegin() == hull.end() - up_beg) --it_up;
58         Vec a = {it_up->F.F - it_up->S.F, it_up->F.S - it_up->S.S};
59         Vec b = {p.F - it_up->S.F, p.S - it_up->S.S};
60         if (cross(a, b) > 0) // > 0 is inclusive, >= 0 is exclusive
61             return false;
62     }
63     return true;
64 }
65 // The function can have only one local min and max
66 // and may be constant only at min and max.
67 template <typename T>
68 SegIt max(function<T(Seg &)> f) { // O(log(n))
69     auto l = hull.begin();
70     auto r = hull.end();
71     SegIt b = hull.end();
72     T b_v;
73     while (r - l > 2) {
74         auto m = l + (r - l) / 2;
75         T l_v = f(*l);
76         T l_n_v = f(*(l + 1));
77         T m_v = f(*m);
78         T m_n_v = f(*(m + 1));
79         if (b == hull.end() || l_v > b_v) {
80             b = l; // If max is at l we may remove it from the range.
81             b_v = l_v;
82         }
83         if (l_n_v > l_v) {
84             if (m_v < l_v) {
85                 r = m;
86             } else {
87                 if (m_n_v > m_v) {
88                     l = m + 1;
89                 } else {
90                     r = m + 1;

```

%0722

#3373

#2197

#7227

%1826

#8566

#3586

#7332

#7279

```

91     }
92 }
93 } else {
94     if (m_v < l_v) {
95         l = m + 1;
96     } else {
97         if (m_n_v > m_v) {
98             l = m + 1;
99         } else {
100             r = m + 1;
101         }
102     }
103 }
104 }
105 T l_v = f(*l);
106 if (b == hull.end() || l_v > b_v) {
107     b = l;
108     b_v = l_v;
109 }
110 if (r - l > 1) {
111     T l_n_v = f(*(l + 1));
112     if (b == hull.end() || l_n_v > b_v) {
113         b = l + 1;
114         b_v = l_n_v;
115     }
116 }
117 return b;
118 }
119 SegIt closest(Vec p) { // p can't be internal(can be on border),
120 // hull must have atleast 3 points
121 Seg &ref_p = hull.front(); // O(log(n))
122 return max(function<double(Seg &)>(
123     [&p, &ref_p](
124         Seg &seg) { // accuracy of used type should be coord^2
125         if (p == seg.F) return 10 - M_PI;
126         Vec v1 = {seg.S.F - seg.F.F, seg.S.S - seg.F.S};
127         Vec v2 = {p.F - seg.F.F, p.S - seg.F.S};
128         ll c_p = cross(v1, v2);
129         if (c_p > 0) { // order the backside by angle
130             Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
131             Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
132             ll d_p = dot(v1, v2);
133             ll c_p = cross(v2, v1);
134             return atan2(c_p, d_p) / 2;
135         }
136         ll d_p = dot(v1, v2);
137         double res = atan2(d_p, c_p);
138         if (d_p <= 0 && res > 0) res = -M_PI;
139         if (res > 0) {
140             res += 20;
141         } else {
#0656
142             res = 10 - res;
143         }
144         return res;
145     }));
146 }
147 template <int DIRECTION> // 1 or -1
148 Vec tan_point(Vec p) { // can't be internal or on border
149 // -1 iff CCW rotation of ray from p to res takes it away from
150 // polygon?
151 Seg &ref_p = hull.front(); // O(log(n))
152 auto best_seg = max(function<double(Seg &)>(
153     [&p, &ref_p](
154         Seg &seg) { // accuracy of used type should be coord^2
155         Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
156         Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
157         ll d_p = dot(v1, v2);
158         ll c_p = DIRECTION * cross(v2, v1);
159         return atan2(c_p, d_p); // order by signed angle
160     }));
161 return best_seg->F;
162 }
163 SegIt max_in_dir(Vec v) { // first is the ans. O(log(n))
164 return max(
165     function<ll(Seg &)>([&v](Seg &seg) { return dot(v, seg.F); }));
166 }
167 pair<SegIt, SegIt> intersections(Seg l) { // O(log(n))
168     int x = l.S.F - l.F.F;
169     int y = l.S.S - l.F.S;
170     Vec dir = {-y, x};
171     auto it_max = max_in_dir(dir);
172     auto it_min = max_in_dir(MP(y, -x));
173     ll opt_val = dot(dir, l.F);
174     if (dot(dir, it_max->F) < opt_val ||
175         dot(dir, it_min->F) > opt_val)
176         return MP(hull.end(), hull.end());
177     SegIt it_r1, it_r2;
178     function<bool(Seg &, Seg &)> inc_c([&dir](Seg &lft, Seg &rgt) {
179         return dot(dir, lft.F) < dot(dir, rgt.F);
180     });
181     function<bool(Seg &, Seg &)> dec_c([&dir](Seg &lft, Seg &rgt) {
182         return dot(dir, lft.F) > dot(dir, rgt.F);
183     });
184     if (it_min <= it_max) {
185         it_r1 = upper_bound(it_min, it_max + 1, l, inc_c) - 1;
186         if (dot(dir, hull.front().F) >= opt_val) {
187             it_r2 = upper_bound(hull.begin(), it_min + 1, l, dec_c) - 1;
188         } else {
189             it_r2 = upper_bound(it_max, hull.end(), l, dec_c) - 1;
190         }
191     } else {
192         it_r1 = upper_bound(it_max, it_min + 1, l, dec_c) - 1;
#7417
193     }
#8283
#5209
#9762
#5037
#9596
#4740
#0276
#0483
#9409

```

```

193     if (dot(dir, hull.front().F) <= opt_val) { #9772
194         it_r2 = upper_bound(hull.begin(), it_max + 1, 1, inc_c) - 1;
195     } else {
196         it_r2 = upper_bound(it_min, hull.end(), 1, inc_c) - 1;
197     }
198 } #9450
199 return MP(it_r1, it_r2);
200 } %1498
201 Seg diameter() { // O(n)
202     Seg res;
203     ll dia_sq = 0;
204     auto it1 = hull.begin();
205     auto it2 = up_beg; #2632
206     Vec v1 = {hull.back().S.F - hull.back().F.F,
207             hull.back().S.S - hull.back().F.S};
208     while (it2 != hull.begin()) {
209         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F,
210                 (it2 - 1)->S.S - (it2 - 1)->F.S}; #5150
211         if (cross(v1, v2) > 0) break;
212         --it2;
213     }
214     while (it2 != hull.end()) { // check all antipodal pairs
215         if (dist_sq(it1->F, it2->F) > dia_sq) { #1246
216             res = {it1->F, it2->F};
217             dia_sq = dist_sq(res.F, res.S);
218         }
219         Vec v1 = {it1->S.F - it1->F.F, it1->S.S - it1->F.S};
220         Vec v2 = {it2->S.F - it2->F.F, it2->S.S - it2->F.S};
221         if (cross(v1, v2) == 0) { #9381
222             if (dist_sq(it1->S, it2->F) > dia_sq) {
223                 res = {it1->S, it2->F};
224                 dia_sq = dist_sq(res.F, res.S);
225             }
226             if (dist_sq(it1->F, it2->S) > dia_sq) { #7011
227                 res = {it1->F, it2->S};
228                 dia_sq = dist_sq(res.F, res.S);
229             } // report cross pairs at parallel lines.
230             ++it1;
231             ++it2; #5626
232         } else if (cross(v1, v2) < 0) {
233             ++it1;
234         } else {
235             ++it2;
236         }
237     } #4406
238     return res;
239 }
240 }; %9383

```

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); }
6     Vec operator-(const Vec &oth) { return {x - oth.x, y - oth.y}; } #2919
7 };
8 ll cross(Vec a, Vec b) { return (ll)a.x * b.y - (ll)a.y * b.x; }
9 ll dot(Vec a, Vec b) { return (ll)a.x * b.x + (ll)a.y * b.y; }
10 struct Edge {
11     Vec tar;
12     Edge *nxt; #8008
13     Edge *inv = NULL;
14     Edge *rep = NULL;
15     bool vis = false;
16 };
17 struct Seg { #7311
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); }
21 };
22 ll orient(Vec a, Vec b, Vec c) { #6432
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 } #6334
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) { #4264
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     } #7305
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]; #1845
40     return res > 0; #6793
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;
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 } #8219
52 Edge *add_point(Vec p, Edge *cur) { // returns outgoing edge
53     Edge *triangle[] = {cur, cur->nxt, cur->nxt->nxt};

```

#8178

```

54 ran(i, 0, 3) {
55     if (orient(triangle[i]->tar, triangle[(i + 1) % 3]->tar, p) < 0)
56         return NULL; #0233
57 }
58 ran(i, 0, 3) {
59     if (triangle[i]->rep) {
60         Edge *res = add_point(p, triangle[i]->rep);
61         if (res) #5636
62             return res; // unless we are on last layer we must exit here
63     }
64 }
65 Edge p_as_e{p};
66 Edge tmp{cur->tar}; #1432
67 tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
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; #8359
72 res->inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
73 res->inv->inv = res;
74 return res;
75 }
76 Edge *del aunay(vector<Vec> &points) { #3029
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};
82     for (Vec &cur : points) { #4575
83         Edge *loc = add_point(cur, res);
84         Edge *out = loc;
85         arr[0] = cur;
86         while (true) {
87             arr[1] = out->tar; #3471
88             arr[2] = out->nxt->tar;
89             Edge *e = out->nxt->inv;
90             if (e && in_c_circle(arr, e->nxt->tar)) {
91                 Edge tmp{cur};
92                 tmp.inv = add_triangle(&tmp, out, e->nxt);
93                 tmp.tar = e->nxt->tar; #9851
94                 tmp.inv->inv = add_triangle(&tmp, e->nxt->nxt, out->nxt->nxt);
95                 out = tmp.inv->nxt;
96                 continue;
97             }
98             out = out->nxt->nxt->inv; #0151
99             if (out->tar == loc->tar) break;
100         }
101     }
102     return res; #6769
103 }
104 void extract_triangles(Edge *cur, vector<vector<Seg> > &res) { %6769
105     if (!cur->vis) {

```

```

106     bool inc = true;
107     Edge *it = cur;
108     do { #3769
109         it->vis = true;
110         if (it->rep) {
111             extract_triangles(it->rep, res);
112             inc = false;
113         } #2104
114         it = it->nxt;
115     } while (it != cur);
116     if (inc) {
117         Edge *triangle[3] = {cur, cur->nxt, cur->nxt->nxt};
118         res.resize(res.size() + 1); #6207
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( #3011
124                     {triangle[i]->tar, triangle[(i + 1) % 3]->tar});
125         }
126         if (tar.empty()) res.pop_back();
127     }
128 } #8602
129 } %5626

```

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 *)); }
6     int cnt = 0; #1006
7 };
8 Node *aho_corasick(vector<vector<char> > &dict) {
9     Node *root = new Node;
10    root->suffix = 0;
11    vector<pair<vector<char> *, Node *> > state; #9056
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) {
16            Node *nxt = cur.second->nxt[(cur.first)[i]];
17            if (nxt) { #1331
18                cur.second = nxt;
19            } else {
20                nxt = new Node;
21                cur.second->nxt[(cur.first)[i]] = nxt;
22                Node *suf = cur.second->suffix; #5283
23                cur.second = nxt;
24                nxt->suffix = root; // set correct suffix link
25                while (suf) {

```

```

26         if (suf->nxt[(*cur.first)[i]]) {
27             nxt->suffix = suf->nxt[(*cur.first)[i]];
28             break;
29         }
30         suf = suf->suffix;
31     }
32 }
33 if (cur.first->size() > i + 1) nstate.push_back(cur);
34 }
35 state = nstate;
36 }
37 return root;
38 }
39 // auxiliary functions for searching and counting
40 Node *walk(Node *cur,
41 char c) { // longest prefix in dict that is suffix of walked string.
42 while (true) {
43     if (cur->nxt[c]) return cur->nxt[c];
44     if (!cur->suffix) return cur;
45     cur = cur->suffix;
46 }
47 }
48 void cnt_matches(Node *root, vector<char> &match_in) {
49     Node *cur = root;
50     for (char c : match_in) {
51         cur = walk(cur, c);
52         ++cur->cnt;
53     }
54 }
55 void add_cnt(Node *root) { // After counting matches propagate ONCE to
56                             // suffixes for final counts
57     vector<Node *> to_visit = {root};
58     ran(i, 0, to_visit.size()) {
59         Node *cur = to_visit[i];
60         ran(j, 0, alpha_size) {
61             if (cur->nxt[j]) to_visit.push_back(cur->nxt[j]);
62         }
63     }
64     for (int i = to_visit.size() - 1; i > 0; --i)
65         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
66 }
67 int main() {
68     int n, len;
69     scanf("%d %d", &len, &n);
70     vector<char> a(len + 1);
71     scanf("%s", a.data());
72     a.pop_back();
73     for (char &c : a) c -= 'a';
74     vector<vector<char>> > dict(n);
75     ran(i, 0, n) {
76         scanf("%d", &len);

```

#3580

#3263

%2882

#5414

%1529

#0015

%8156

#0662

#7950

%0488

```

77     dict[i].resize(len + 1);
78     scanf("%s", dict[i].data());
79     dict[i].pop_back();
80     for (char &c : dict[i]) c -= 'a';
81 }
82 Node *root = aho_corasick(dict);
83 cnt_matches(root, a);
84 add_cnt(root);
85 ran(i, 0, n) {
86     Node *cur = root;
87     for (char c : dict[i]) cur = walk(cur, c);
88     printf("%d\n", cur->cnt);
89 }
90 }

```

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

```

1 struct Node {
2     map<char, Node *> nxt_char;
3     // Map is faster than hashtable and unsorted arrays
4     int len; // Length of longest suffix in equivalence class.
5     Node *suf;
6     bool has_nxt(char c) const { return nxt_char.count(c); }
7     Node *nxt(char c) {
8         if (!has_nxt(c)) return NULL;
9         return nxt_char[c];
10    }
11    void set_nxt(char c, Node *node) { nxt_char[c] = node; }
12    Node *split(int new_len, char c) {
13        Node *new_n = new Node;
14        new_n->nxt_char = nxt_char;
15        new_n->len = new_len;
16        new_n->suf = suf;
17        suf = new_n;
18        return new_n;
19    }
20    // Extra functions for matching and counting
21    Node *lower(int depth) {
22        // move to longest suf of current with a maximum length of depth.
23        if (suf->len >= depth) return suf->lower(depth);
24        return this;
25    }
26    Node *walk(char c, int depth, int &match_len) {
27        // move to longest suffix of walked path that is a substring
28        match_len = min(match_len, len);
29        // includes depth limit (needed for finding matches)
30        if (has_nxt(c)) { // as suffixes are in classes match_len must be
31                            // tracked externally
32            ++match_len;
33            return nxt(c)->lower(depth);
34        }
35        if (suf) return suf->walk(c, depth, match_len);
36        return this;

```

#9664

#8305

#4595

%3114

#2130

#9589

```

37 } %2262
38 int paths_to_end = 0;
39 void set_as_end() { // All suffixes of current node are marked as
40 // ending nodes.
41     paths_to_end += 1;
42     if (suf) suf->set_as_end();
43 } #3041
44 bool vis = false;
45 void calc_paths() {
46     /* Call ONCE from ROOT. For each node calculates number of ways
47     * to reach an end node. paths_to_end is occurrence count for any
48     * strings in current suffix equivalence class. */
49     if (!vis) {
50         vis = true;
51         for (auto cur : nxt_char) { #2404
52             cur.second->calc_paths();
53             paths_to_end += cur.second->paths_to_end;
54         }
55     }
56 } #7906 %7906
57 // Transform into suffix tree of reverse string
58 map<char, Node *> tree_links;
59 int end_dist = 1 << 30;
60 int calc_end_dist() {
61     if (end_dist == 1 << 30) {
62         if (nxt_char.empty()) end_dist = 0; #7524
63         for (auto cur : nxt_char)
64             end_dist = min(end_dist, 1 + cur.second->calc_end_dist());
65     }
66     return end_dist;
67 } #2021
68 bool vis_t = false;
69 void build_suffix_tree(string &s) { // Call ONCE from ROOT.
70     if (!vis_t) {
71         vis_t = true;
72         if (suf) #6270
73             suf->tree_links[s[s.size() - end_dist - suf->len - 1]] = this;
74         for (auto cur : nxt_char) cur.second->build_suffix_tree(s);
75     }
76 }
77 }; #1268 %1268
78 struct SufAuto {
79     Node *last;
80     Node *root;
81     void extend(char new_c) {
82         Node *nlast = new Node; #0936
83         nlast->len = last->len + 1;
84         Node *swn = last;
85         while (swn && !swn->has_nxt(new_c)) {
86             swn->set_nxt(new_c, nlast);
87             swn = swn->suf; #1831

```

```

88 }
89 if (!swn) {
90     nlast->suf = root;
91 } else {
92     Node *max_sbstr = swn->nxt(new_c); #0855
93     if (swn->len + 1 == max_sbstr->len) {
94         nlast->suf = max_sbstr;
95     } else { // remove for minimal DFA that matches suffixes and
96             // crap
97         Node *eq_sbstr = max_sbstr->split(swn->len + 1, new_c);
98         nlast->suf = eq_sbstr; #1749
99         Node *x = swn; // x = with_edge_to_eq_sbstr
100         while (x != 0 && x->nxt(new_c) == max_sbstr) {
101             x->set_nxt(new_c, eq_sbstr);
102             x = x->suf;
103         } #4310
104     }
105 }
106 last = nlast;
107 } #6146
108 SufAuto(string &s) {
109     root = new Node;
110     root->len = 0;
111     root->suf = NULL;
112     last = root; #9604
113     for (char c : s) extend(c);
114     root->calc_end_dist(); // To build suffix tree use reversed string
115     root->build_suffix_tree(s);
116 }
117 }; #6251 %6251

```

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 add_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  return /*ly*/ { /*ry*/ tot_flow /*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      // arguments source and sink, memory usage O(largest node index
148      // +input size)
149      int t;
150      scanf("%d", &t);
151      for (int i = 0; i < t; ++i) {
152          vector<tuple<int, int, ll, ll, ll> > edges;
153          int n;
154          scanf("%d", &n);
155          for (int j = 1; j <= n; ++j) {
156              edges.emplace_back(j, 2 * n + 1, 1, 0, 0);
157          }
158          for (int j = 1; j <= n; ++j) {
159              int card;
160              scanf("%d", &card);
161              edges.emplace_back(0, card, 1, 0, 0);
162          }
163          int ex_c;
164          scanf("%d", &ex_c);
165          for (int j = 0; j < ex_c; ++j) {
166              int a, b;
167              scanf("%d %d", &a, &b);
168              if (b < a) swap(a, b);
169              edges.emplace_back(a, b, nmax, 0, 1);
170              edges.emplace_back(b, n + b, nmax, 0, 0);
171              edges.emplace_back(n + b, a, nmax, 0, 1);
172          }
173          int v = 2 * n + 2;
174          MaxFlow mf(edges);
175          printf("%d\n", (int)mf.calc_flow(0, v - 1).second);

```

```

176      // cout << mf.flow_on_edge(edge_index) << endl;
177  }
178  }

```

13 Min Cost Max Flow with Cycle Cancellation $\mathcal{O}(\text{cap} \cdot nm)$

```

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        int getCap(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    int minCostMaxFlow() {
43        int n = nodes.size();
44        int result = 0;
45        struct State {
46            int p;

```

#2965

#4145

#6369

#8987

#1577

#5057

#5123

#0927

```

47     Edge* used;                                     #7358
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]);                #0078
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 =                      #7871
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));
98                 cur = edge->from(cur);
99             }

```

```

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;
107 }
108 };
109 };

```

#4467

#4029
%2900

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 {
7         Node *from;
8         Node *tar;
9         int w;
10        bool inc;
11    };
12    struct Circle {
13        bool vis = false;
14        vector<Edge*> cont;
15        void clean(int idx);
16    };
17    const static greater<pair<ll, Edge*> > comp;
18    static vector<Circle> to_proc;
19    static bool no_dmst;
20    static Node *root; // Can use inline static since C++17
21    struct Node {
22        Node *par = NULL;
23        vector<pair<int, int> > out_cands; // Circ, edge idx
24        vector<pair<ll, Edge*> > con;
25        bool in_use = false;
26        ll w = 0; // extra to add to edges in con
27        Node *anc() {
28            if (!par) return this;
29            while (par->par) par = par->par;
30            return par;
31        }
32        void clean() {
33            if (!no_dmst) {
34                in_use = false;
35                for (auto &cur : out_cands)
36                    to_proc[cur.first].clean(cur.second);
37            }
38        }

```

#6091

#2186

#4353

#9916

#0564

#0300

#0747

```

39 Node *con_to_root() {
40     if (anc() == root) return root;
41     in_use = true;
42     Node *super = this; // Will become root or the first Node
43                         // encountered in a loop.
44     while (super == this) { #3927
45         while (
46             !con.empty() && con.front().second->tar->anc() == anc()) {
47             pop_heap(con.begin(), con.end(), comp);
48             con.pop_back();
49         } #2561
50         if (con.empty()) {
51             no_dmst = true;
52             return root;
53         }
54         pop_heap(con.begin(), con.end(), comp); #8600
55         auto nxt = con.back();
56         con.pop_back();
57         w = -nxt.first;
58         if (nxt.second->tar
59             ->in_use) { // anc() wouldn't change anything
60             super = nxt.second->tar->anc(); #6612
61             to_proc.resize(to_proc.size() + 1);
62         } else {
63             super = nxt.second->tar->con_to_root();
64         }
65         if (super != root) { #7005
66             to_proc.back().cont.push_back(nxt.second);
67             out_cands.emplace_back(
68                 to_proc.size() - 1, to_proc.back().cont.size() - 1);
69         } else { // Clean circles
70             nxt.second->inc = true; #1096
71             nxt.second->from->clean();
72         }
73     }
74     if (super != root) { // we are some loops non first Node.
75         if (con.size() > super->con.size()) { #2844
76             swap(con,
77                 super->con); // Largest con in loop should not be copied.
78             swap(w, super->w);
79         }
80         for (auto cur : con) { #3498
81             super->con.emplace_back(
82                 cur.first - super->w + w, cur.second);
83             push_heap(super->con.begin(), super->con.end(), comp);
84         }
85     } #6348
86     par = super; // root or anc() of first Node encountered in a
87                 // loop
88     return super;
89 }
90 };

```

```

91 Node *croot; #0309
92 vector<Node> graph;
93 vector<Edge> edges;
94 DMST(int n, vector<EdgeDesc> &desc,
95     int r) { // Self loops and multiple edges are okay. #8100
96     graph.resize(n);
97     croot = &graph[r];
98     for (auto &cur : desc) // Edges are reversed internally
99         edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
100     for (int i = 0; i < desc.size(); ++i)
101         graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]);
102     for (int i = 0; i < n; ++i) #8811
103         make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
104 }
105 bool find() {
106     root = croot;
107     no_dmst = false; #5307
108     for (auto &cur : graph) {
109         cur.con_to_root();
110         to_proc.clear();
111         if (no_dmst) return false;
112     } #6725
113     return true;
114 } #1568
115 ll weight() {
116     ll res = 0;
117     for (auto &cur : edges) {
118         if (cur.inc) res += cur.w;
119     } #6369
120     return res;
121 } #1477
122 };
123 void DMST::Circle::clean(int idx) {
124     if (!vis) {
125         vis = true;
126         for (int i = 0; i < cont.size(); ++i) { #6503
127             if (i != idx) {
128                 cont[i]->inc = true;
129                 cont[i]->from->clean();
130             }
131         }
132     } #8144
133 }
134 const greater<pair<ll, DMST::Edge *> > DMST::comp;
135 vector<DMST::Circle> DMST::to_proc;
136 bool DMST::no_dmst; #2354
137 DMST::Node *DMST::root; #2870

```

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 dfs(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 };
37 bool edge::is_bridge() {
38     return exists &&
39         (dest->seen > rev->dest->val || dest->val < rev->dest->seen);
40 }
41 vert graph[nmax];
42 int main() { // Mechanics Practice BRIDGES
43     int n, m;
44     cin >> n >> m;
45     for (int i = 0; i < m; ++i) {
46         int u, v;
47         scanf("%d %d", &u, &v);
48         graph[u].con.emplace_back(graph + v);
49         graph[v].con.emplace_back(graph + u);
50         graph[u].con.back().rev = &graph[v].con.back();
51         graph[v].con.back().rev = &graph[u].con.back();
52     }
53     graph[1].dfs(1, NULL);

```

#8922

#0116

#1288

#8194

%8624

#7106

#9056

#5223

```

54 int res = 0;
55 for (int i = 1; i <= n; ++i) res += graph[i].cnt_adj_bridges();
56 cout << res / 2 << endl;
57 }

```

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

```

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

```

#0321

#2422

#2081

%7763

#3875

#7568

%5339

#6880

%3565

#9931

#2243

```

47 }
48 };
49 int main() {
50     int n, m;
51     cin >> n >> m;
52     Graph g(2 * m);
53     ran(i, 0, n) {
54         int a, sa, b, sb;
55         cin >> a >> sa >> b >> sb;
56         a--, b--;
57         g.add_edge(2 * a + 1 - sa, 2 * b + sb);
58         g.add_edge(2 * b + 1 - sb, 2 * a + sa);
59     }
60     vector<int> state(2 * m, 0);
61     {
62         vector<int> order = g.topsort();
63         vector<bool> explr(2 * m, false);
64         for (auto u : order) {
65             vector<int> traversed;
66             g.dfs(u, traversed, explr);
67             if (traversed.size() > 0 && !state[traversed[0] ^ 1]) {
68                 for (auto c : traversed) state[c] = 1;
69             }
70         }
71     }
72     ran(i, 0, m) {
73         if (state[2 * i] == state[2 * i + 1]) {
74             cout << "IMPOSSIBLE\n";
75             return 0;
76         }
77     }
78     ran(i, 0, m) cout << state[2 * i + 1] << '\n';
79     return 0;
80 }

```

#0543

%0543

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    void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len) {

```

#7684

#7883

```

17     lhs_lazy = *this;
18     rhs_lazy = *this;
19     init();
20 }
21 void merge(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 %0625 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     void init() {
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             Lazy l_val, r_val;

```

#7654

#0050

#2924

#6280

#6321

#5313

#8874

#2138

#8209

#8104

```

68     val.split(l_val, r_val, R - L);
69     if (l < M) lc = lc->upd(L, M, l, r, l_val, ver);
70     if (M < r) rc = rc->upd(M, R, l, r, r_val, ver);
71     seg.recalc(lc->seg, M - L, rc->seg, R - M);
72 }
73 return this;
74 }
75 void get(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp,
76 bool last_ver) {
77     if (L >= l && R <= r) {
78         tmp->recalc(*lft_res, L - l, seg, R - L);
79         swap(lft_res, tmp);
80     } else {
81         init();
82         int M = (L + R) / 2;
83         if (is_lazy) {
84             Lazy l_val, r_val;
85             lazy.split(l_val, r_val, R - L);
86             lc = lc->upd(L, M, L, M, l_val, ver + last_ver);
87             lc->ver = ver;
88             rc = rc->upd(M, R, M, R, r_val, ver + last_ver);
89             rc->ver = ver;
90             is_lazy = false;
91         }
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 };
115
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) {

```

#8581

#9373

#6654

#2185

#4770

#4873

#1461

#9427

%7542 int main() {

```

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 }
142 }

```

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

```

#9531

%7932

#6178

```

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

```

#2037

#6759

#9593

#0432

#9151

#3027

#8562

#3486

#4566

#2693

```

79     structure = DS(vertexc + 5, initial);
80     aux = DS(50, initial);
81 }
82 void set(int vertex, int value) {
83     structure.set(ord[vertex], value);
84 }
85 T query_path(
86     int u, int v) { /* returns the "sum" of the path u->v */
87     int cur_id = 0;
88     while (chain_root[u] != chain_root[v]) {
89         if (ord[u] > ord[v]) {
90             cur_id++;
91             aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
92             u = par[chain_root[u]];
93         } else {
94             cur_id++;
95             aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
96             v = par[chain_root[v]];
97         }
98     }
99     cur_id++;
100     aux.set(cur_id,
101         structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
102     return aux.query(1, cur_id);
103 }
104 void print() {
105     for (int i = 1; i <= vertexc; i++)
106         cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' '
107             << par[i] << endl;
108 }
109 };
110 int main() {
111     int vertexc;
112     cin >> vertexc;
113     HLD<int, dummy> hld(vertexc);
114     for (int i = 0; i < vertexc - 1; i++) {
115         int u, v;
116         cin >> u >> v;
117         hld.add_edge(u, v);
118     }
119     hld.build(0);
120     hld.print();
121     int queryc;
122     cin >> queryc;
123     for (int i = 0; i < queryc; i++) {
124         int u, v;
125         cin >> u >> v;
126         hld.query_path(u, v);
127         cout << endl;
128     }
129 }

```

#7758

#4754

#4538

#1595

#7150

%1905

19 Splay Tree + Link-Cut $O(N \log N)$

```

1 struct Tree *treev;
2 struct Tree {
3     struct T {
4         int i;
5         constexpr T() : i(-1) {}
6         T(int _i) : i(_i) {}
7         operator int() const { return i; }
8         explicit operator bool() const { return i != -1; }
9         Tree *operator->() { return treev + i; }
10    };
11    T c[2], p;
12    /* insert monoid here */
13    /*lg*/ T link; /*rg*/
14    Tree() {
15        /* init monoid here */
16        /*lg*/ link = -1; /*rg*/
17    }
18 };
19 using T = Tree::T;
20 constexpr T NIL;
21 void update(T t) { /* recalculate the monoid here */
22 }
23 void propagate(T t) {
24     assert(t);
25     /*lg*/
26     for (T c : t->c)
27         if (c) c->link = t->link;
28     /*rg*/
29     /* lazily propagate updates here */
30 }
31 /*lp*/
32 void lazy_reverse(T t) { /* lazily reverse t here */
33 }
34 /*rp*/
35 T splay(T n) {
36     for (;;) {
37         propagate(n);
38         T p = n->p;
39         if (p == NIL) break;
40         propagate(p);
41         ll px = p->c[1] == n;
42         assert(p->c[px] == n);
43         T g = p->p;
44         if (g == NIL) { /* zig */
45             p->c[px] = n->c[px ^ 1];
46             p->c[px]->p = p;
47             n->c[px ^ 1] = p;
48             n->c[px ^ 1]->p = n;
49             n->p = NIL;
50             update(p);

```

#7635

#2337

#0939

#3006

#8514

#3792

#0245

#4750

```

51     update(n);
52     break;
53 }
54 propagate(g);
55 ll gx = g->c[1] == p;
56 assert(g->c[gx] == p);
57 T gg = g->p;
58 ll ggx = gg && gg->c[1] == g;
59 if (gg) assert(gg->c[ggx] == g);
60 if (gx == px) { /* zig zig */
61     g->c[gx] = p->c[gx ^ 1];
62     g->c[gx]->p = g;
63     p->c[gx ^ 1] = g;
64     p->c[gx ^ 1]->p = p;
65     p->c[gx] = n->c[gx ^ 1];
66     p->c[gx]->p = p;
67     n->c[gx ^ 1] = p;
68     n->c[gx ^ 1]->p = n;
69 } else { /* zig zag */
70     g->c[gx] = n->c[gx ^ 1];
71     g->c[gx]->p = g;
72     n->c[gx ^ 1] = g;
73     n->c[gx ^ 1]->p = n;
74     p->c[gx ^ 1] = n->c[gx];
75     p->c[gx ^ 1]->p = p;
76     n->c[gx] = p;
77     n->c[gx]->p = n;
78 }
79 if (gg) gg->c[ggx] = n;
80 n->p = gg;
81 update(g);
82 update(p);
83 update(n);
84 if (gg) update(gg);
85 }
86 return n;
87 }
88 T extreme(T t, int x) {
89     while (t->c[x]) t = t->c[x];
90     return t;
91 }
92 T set_child(T t, int x, T a) {
93     T o = t->c[x];
94     t->c[x] = a;
95     update(t);
96     o->p = NIL;
97     a->p = t;
98     return o;
99 }
100 /****** Link-Cut Tree: *****/
101 T expose(T t) {

```

#8981

#5659

#5608

#1439

#9140

#2928

#9645

#4920

#2821

#4262

```

102 set_child(splay(t), 1, NIL);
103 T leader = splay(extreme(t, 0));
104 if (leader->link == NIL) return t;
105 set_child(splay(leader), 0, expose(leader->link));
106 return splay(t);
107 }
108 void link(T t, T p) {
109     assert(t->link == NIL);
110     t->link = p;
111 }
112 T cut(T t) {
113     T p = t->link;
114     if (p) expose(p);
115     t->link = NIL;
116     return p;
117 }
118 /*lp*/
119 void make_root(T t) {
120     expose(t);
121     lazy_reverse(extreme(splay(t), 0));
122 }
123 /*rp*/

```

20 Templated multi dimensional BIT $\mathcal{O}(\log(n)^{\text{dim}})$ per query

```

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

```

```

29 template <typename... loc_form>
30 R_T query(C_T cx, loc_form... args) { // sum in (-inf, cx)
31     R_T res = 0;
32     int x = lower_bound(pos.begin(), pos.end(), cx) - pos.begin() - 1;
33     for (; x > 0; x -= x & -x) res += elems[x].query(args...);
34     return res;
35 }
36 };
37 template <typename I_T>
38 struct wrapped {
39     I_T a = 0;
40     void update(I_T b) { a += b; }
41     I_T query() { return a; }
42     // Should never be called, needed for compilation
43     void init() { DEBUG('i') }
44     void update() { DEBUG('u') }
45 };
46 // retun type should be same as type inside wrapped
47 BIT<BIT<wrapped<ll>, int, INT_MIN, ll>, int, INT_MIN, ll> fenwick;
48 int dim = 2;
49 vector<tuple<int, int, ll> > to_insert;
50 to_insert.emplace_back(1, 1, 1);
51 // set up all pos that are to be used for update
52 for (int i = 0; i < dim; ++i) {
53     for (auto &cur : to_insert)
54         fenwick.update(get<0>(cur), get<1>(cur));
55     // May include value which won't be used
56     fenwick.init();
57 }
58 // actual use
59 for (auto &cur : to_insert)
60     fenwick.update(get<0>(cur), get<1>(cur), get<2>(cur));
61 cout << fenwick.query(2, 2) << '\n';
62 }

```

21 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;	#4295		
21	if (rch) total += rch->total;			
22	}			
23	};			
24	deque<Node> nodes;			
25	Node* root = 0;	#9633		
26	pair<Node*, Node*> split(int key, Node* cur) {			
27	if (cur == 0) return {0, 0};			
28	pair<Node*, Node*> result;			
29	if (key <= cur->key) {	#5233		
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);	#6988		
35	cur->rch = ret.first;			
36	result = {cur, ret.second};			
37	}			
38	cur->update();			
39	return result;			
40	}	#7230		
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) {	#6282		
46	left->rch = merge(left->rch, right);			
47	top = left;			
48	} else {			
49	right->lch = merge(left, right->lch);	#3510		
50	top = right;			
51	}			
52	top->update();			
53	return top;			
54	}			
55	void insert(int key, int value) {	#8918		
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);	#9760		
61	root = cur;			
62	}			
63	void erase(int key) {			
64	Node *left, *mid, *right;			
65	tie(left, mid) = split(key, root);	#1416		
66	tie(mid, right) = split(key + 1, mid);			
67	root = merge(left, right);			
68	}			
69	long long sum_upto(int key, Node* cur) {	#7634		
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);	#8122		
75	if (cur->lch) result += cur->lch->total;			
76	return result;			
77	}			
78	}			
79	long long get(int l, int r) {			
80	return sum_upto(r + 1, root) - sum_upto(l, root);			
81	}	#0094		
82	};	%4959		
83	// Solution for:			
84	// http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/TREAP			
85	int main() {			
86	ios_base::sync_with_stdio(false);			
87	cin.tie(0);			
88	int m;			
89	Treap treap;			
90	cin >> m;			
91	for (int i = 0; i < m; i++) {			
92	int type;			
93	cin >> type;			
94	if (type == 1) {			
95	int x, y;			
96	cin >> x >> y;			
97	treap.insert(x, y);			
98	} else if (type == 2) {			
99	int x;			
100	cin >> x;			
101	treap.erase(x);			
102	} else {			
103	int l, r;			
104	cin >> l >> r;			
105	cout << treap.get(l, r) << endl;			
106	}			
107	}			
108	return 0;			
109	}			
<hr/>				
22 Radixsort 50M 64 bit integers as single array in 1 sec				
1	template <typename T>			
2	void rsort(T *a, T *b, int size, int d = sizeof(T) - 1) {			
3	int b_s[256]{};			
4	ran(i, 0, size) { ++b_s[(a[i] >> (d * 8)) & 255]; }			
5	// ++b_s[*((uchar *) (a + i) + d)];			
6	T *mem[257];			
7	mem[0] = b;	#5369		

```

8  T **l_b = mem + 1;
9  l_b[0] = b;
10 ran(i, 0, 255) { l_b[i + 1] = l_b[i] + b_s[i]; }
11 for (T *it = a; it != a + size; ++it) {
12     T id = ((*it) >> (d * 8)) & 255;
13     *(l_b[id]++) = *it;
14 }
15 l_b = mem;
16 if (d) {
17     T *l_a[256];
18     l_a[0] = a;
19     ran(i, 0, 255) l_a[i + 1] = l_a[i] + b_s[i];
20     ran(i, 0, 256) {
21         if (l_b[i + 1] - l_b[i] < 100) {
22             sort(l_b[i], l_b[i + 1]);
23             if (d & 1) copy(l_b[i], l_b[i + 1], l_a[i]);
24         } else {
25             rsort(l_b[i], l_a[i], b_s[i], d - 1);
26         }
27     }
28 }
29 }
30 const int nmax = 5e7;
31 ll arr[nmax], tmp[nmax];
32 int main() {
33     for (int i = 0; i < nmax; ++i) arr[i] = ((ll)rand() << 32) | rand();
34     rsort(arr, tmp, nmax);
35     assert(is_sorted(arr, arr + nmax));
36 }

```

23 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) {
4         a /= oth;
5         b /= oth;
6         return *this;
7     }
8 };
9 Complex operator+(const Complex &lft, const Complex &rgt) {
10     return Complex{lft.a + rgt.a, lft.b + rgt.b};
11 }
12 Complex operator-(const Complex &lft, const Complex &rgt) {
13     return Complex{lft.a - rgt.a, lft.b - rgt.b};
14 }
15 Complex operator*(const Complex &lft, const Complex &rgt) {
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) {
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) {
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))),
44                     sin(-idx_p * M_PI / (1 << (ord - 1)))};
45                 if (is_inv) root_pow[idx].b = -root_pow[idx].b;
46             }
47         }
48     }
49     if (invert != is_inv) {
50         is_inv = invert;
51         for (Complex &cur : root_pow) cur.b = -cur.b;
52     }
53     for (int i = 1, j = 0; i < (1 << ord); ++i) {
54         int m = 1 << (ord - 1);
55         bool cont = true;
56         while (cont) {
57             cont = j & m;
58             j ^= m;
59             m >>= 1;
60         }
61         if (i < j) swap(arr[i], arr[j]);
62     }
63     fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
64     if (invert)
65         for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord);
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 } else {
76     int fft_ord = 32 - __builtin_clz(c.size());
77     if (arr[0].size() != 1 << fft_ord)
78         for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord);
79     for (int i = 0; i < 4; ++i)
80         fill(arr[i].begin(), arr[i].end(), Complex{});
81     for (int &cur : a)
82         if (cur < 0) cur += mod;
83     for (int &cur : b)
84         if (cur < 0) cur += mod;
85     const int shift = 15;
86     const int mask = (1 << shift) - 1;
87     for (int i = 0; i < min(a.size(), c.size()); ++i) {
88         arr[0][i].a = a[i] & mask;
89         arr[1][i].a = a[i] >> shift;
90     }
91     for (int i = 0; i < min(b.size(), c.size()); ++i) {
92         arr[0][i].b = b[i] & mask;
93         arr[1][i].b = b[i] >> shift;
94     }
95     for (int i = 0; i < 2; ++i) fft(arr[i], fft_ord, false);
96     for (int i = 0; i < 2; ++i) {
97         for (int j = 0; j < 2; ++j) {
98             int tar = 2 + (i + j) / 2;
99             Complex mult = {0, -0.25};
100             if (i ^ j) mult = {0.25, 0};
101             for (int k = 0; k < (1 << fft_ord); ++k) {
102                 int rev_k = ((1 << fft_ord) - k) % (1 << fft_ord);
103                 Complex ca = arr[i][k] + conj(arr[i][rev_k]);
104                 Complex cb = arr[j][k] - conj(arr[j][rev_k]);
105                 arr[tar][k] = arr[tar][k] + mult * ca * cb;
106             }
107         }
108     }
109     for (int i = 2; i < 4; ++i) {
110         fft(arr[i], fft_ord, true);
111         for (int k = 0; k < (int)c.size(); ++k) {
112             c[k] = (c[k] + (((ll)(arr[i][k].a + 0.5) % mod)
113                         << (shift * 2 * (i - 2)))) %
114                 mod;
115             c[k] = (c[k] + (((ll)(arr[i][k].b + 0.5) % mod)
116                         << (shift * (2 * (i - 2) + 1)))) %
117                 mod;
118         }
119     }
120 }
121 }

```

#8811

#4629

#9591

#2625

#3501

#9971

#4471

#8403

#8289

%1231

24 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.multf(res, a);
19         a = arithm.multf(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.multf(a, a);
37             if (a == 1) return false;
38             if (a == n - 1) break;
39         }
40         if (a != n - 1) return false;
41     }
42     return true;
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;

```

#0237

#0780

%9493

#1758

%2144

#8104

#6402

#0876

#4806

%0975

#2118

```

50 ll v = arithm.sqp1(lv);
51 int idx = 1;
52 int tar = 1;
53 while (true) {
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.multf(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         }
73         break;
74     } else {
75         ll g = __gcd(cur, n);
76         if (g != 1) return g;
77     }
78     v = v_cur;
79     idx += j_stop;
80     if (idx == tar) {
81         lv = v;
82         tar *= 2;
83         v = arithm.sqp1(v);
84         ++idx;
85     }
86 }
87 }
88 } #3542
89 map<ll, int> prime_factor(ll n,
90 map<ll, int> *res = NULL) { // n <= 1<<61, ~1000/s (<500/s on CF)
91     if (!res) {
92         map<ll, int> res_act;
93         for (int p : small_primes) {
94             while (!(n % p)) {
95                 ++res_act[p];
96                 n /= p;
97             }
98         }
99         if (n != 1) prime_factor(n, &res_act);
100         return res_act;
101 }

```

#5290

#4468

#7912

#0906

#7208

#2298

#1174

%3542

#3770

#4612

```

102 if (is_prime(n)) {
103     ++(*res)[n]; #1963
104 } else {
105     ll factor = pollard_rho(n);
106     prime_factor(factor, res);
107     prime_factor(n / factor, res);
108 } #5350
109 return map<ll, int>();
110 } // Usage: fact = prime
    factor(n); %5477

```

25 Symmetric Submodular Functions; Queyrannes'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):

```

