

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 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query
- 19 Splay Tree + Link-Cut $\mathcal{O}(N \log N)$
- 20 Templatized 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	University of Tartu
24 Fast mod mult, Rabin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$	23	
25 Symmetric Submodular Functions; Queyranne's algorithm	24	
26 Berlekamp-Massey $\mathcal{O}(\mathcal{L}N)$	25	
4	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 7 gsettings set org.gnome.desktop.wm.preferences focus-mode 'slippy' 8 set si cin #.vimrc 9 set ts=4 sw=4 noet 10 set cb=unnamed 11 (global-set-key (kbd "C-x <next>") 'other-window) #.emacs 12 (global-set-key (kbd "C-x <prior>") 'previous-multiframe-window) 13 (global-set-key (kbd "C-M-z") 'ansi-term) 14 (global-linum-mode 1) 15 (column-number-mode 1) 16 (show-paren-mode 1) 17 (setq-default indent-tabs-mode nil) 18 valgrind --vgdb-error=0 ./a <inp & #valgrind 19 gdb a 20 target remote vgdb	
16	1 crc.sh	
17	1#!/bin/env bash 2 for j in `seq 1 1 200` ; do 3 sed '/^\$\s*\$/d' \$1 head -\$j tr -d '[:space:]' cksum cut -f1 → -d ' ' tail -c 5 #whitespace don't matter. 4 done #there shouldn't be any COMMENTS. 5 #copy lines being checked to separate file. 6 # \$./crc.sh tmp.cpp grep XXXX	
21		

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 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);
```

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); } %4798
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;" #2024
14         "movdqa (%0), %%xmm1;" #6087
15         "aesenc %%xmm0, %%xmm1;" #9038
16         "movdqa %%xmm1, (%0);"
17         :
18         : "r" (&mem[0])
19         : "memory"
20     );
21     return mem[0]; // use both slots for 128 bit hash

```

#6956

```

22 }
23 uint64_t mix(uint64_t x) { //x != 0
24     x = (x ^ (x >> 30)) * 0xbff8476d1ce4e5b9;
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 }

```

#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; // vertixes of the triangle
5 bool collinear() { #0823
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(arg(sp)) - M_PI / 2;
10    return ang < min_delta; // positive angle with the real line
11 }
11 } #8446
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); #6715
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; #9407
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); #3895
33 }

```

%8446 #6856 #3895

```

34 };
35 point nine_point_circle_center() { // euler line
36   point O = circum_center();
37   return O + 1.5 * (centroid() - O);
38 };
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);
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) {
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);
11 }
#6327
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) {
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);
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});
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   }
#8193 %3031
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) {
44         Seg other = clines.back(); #3099
45         if (cross(line.d(), other.d()) != 0) {
46           double start = intersection(line, other).y;
47           if (start > apply.back()) break;
48         }
49         clines.pop_back();
50         apply.pop_back();
51       }
52       if (clines.size() == 0) { #7856
53         apply.push_back(-HUGE_VAL);
54       } else { #0868
55         apply.push_back(intersection(line, clines.back()).y);
56       }
57       clines.push_back(line);
58     }
59     slines[side] = clines; #8545
60   }
61   ap_s[0].push_back(HUGE_VALL);
62   ap_s[1].push_back(HUGE_VALL);
63   double result = 0; #3234
64   {
65     double lb = -HUGE_VALL, ub;
66     for (int i = 0, j = 0; #4531
67       i < (int)slines[0].size() && j < (int)slines[1].size();
68       lb = ub) {
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) { #2627
73         alb = max(alb, intersection(l[0], l[1]).y);
74       } else if (cross(l[1].d(), l[0].d()) < 0) {
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       }
86     }
87   }
88 }

```

```

85     result += (-1 + 2 * k) * (aub - alb) * (x1 + x2) / 2;          #9267
86 }
87 if (ap_s[0][i + 1] < ap_s[1][j + 1]) {
88     i++;
89 } else {
90     j++;
91 }
92 }
93 return result;
94 }
95 } %0513

```

8 Convex polygon algorithms

```

1 typedef pair<int, int> Vec;
2 typedef pair<Vec, Vec> Seg;
3 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; } #6913
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);          #3216 %8008
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)          #4033
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;          #3588
28                     r.pop_back();
29                 }
30                 r.push_back(*it);
31             }
32         } #6639
33         ran(i, 0, (int)r.size() - 1) hull.emplace_back(r[i], r[i + 1]);
34     }
35     Hull(vector<Vec> &vert) { // atleast 2 distinct points
36         sort(vert.begin(), vert.end()); // O(n log(n))
37         extend(vert.begin(), vert.end());          #6560
38         int diff = hull.size();
39         extend(vert.rbegin(), vert.rend());

```

```

40     up_beg = hull.begin() + diff;          %0722
41 }
42 bool contains(Vec p) { // O(log(n))          #3373
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;          #2197
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;          #7227
61     }
62     return true;          %1826
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))          #8566
68     auto l = hull.begin();
69     auto r = hull.end();
70     SegIt b = hull.end();          #3586
71     T b_v;
72     while (r - l > 2) {
73         auto m = l + (r - l) / 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. #7332
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             }
92         }
93     }
94 }
95 } #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) { #731
107     b = l;
108     b_v = l_v;
109 }
110 if (r - l > 1) { #446
111     T l_n_v = f(*(l + 1));
112     if (b == hull.end() || l_n_v > b_v) { #597
113         b = l + 1;
114         b_v = l_n_v;
115     }
116 }
117 return b; #908
118 }
119 SegIt closest(Vec p) { // p can't be internal(can be on border), #950
120     // hull must have atleast 3 points
121     Seg &ref_p = hull.front(); // O(log(n))
122     return max(function<double>( #013
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 #506
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); #046
138                 if (d_p <= 0 && res > 0) res = -M_PI;
139                 if (res > 0) {
140                     res += 20;
141                 } else {

```

```

#0656             res = 10 - res;
142         }
143     }
144     return res;
145 });
146 }
#7311 template <int DIRECTION> // 1 or -1
147 Vec tan_point(Vec p) { // can't be internal or on border
148     // -1 iff CCW rotation of ray from p to res takes it away from
149     // polygon?
150     Seg &ref_p = hull.front(); // O(log(n))
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 SegIt max_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 }
#9864 pair<SegIt, SegIt> intersections(Seg l) { // O(log(n))
166     int x = l.S.F - l.F.F;
167     int y = l.S.S - l.F.S;
168     Vec dir = {-y, x};
169     auto it_max = max_in_dir(dir);
170     auto it_min = max_in_dir(MP(y, -x));
171     ll opt_val = dot(dir, l.F);
172     if (dot(dir, it_max->F) < opt_val ||
173         dot(dir, it_min->F) > opt_val)
174         return MP(hull.end(), hull.end());
175     return SegIt(it_r1, it_r2);
176 }
#0134 function<bool>(Seg &, Seg &) > inc_c([&dir](Seg &lft, Seg &rgt) {
177     return dot(dir, lft.F) < dot(dir, rgt.F);
178 });
179 function<bool>(Seg &, Seg &) > dec_c([&dir](Seg &lft, Seg &rgt) {
180     return dot(dir, lft.F) > dot(dir, rgt.F);
181 });
182 if (it_min <= it_max) {
183     it_r1 = upper_bound(it_min, it_max + 1, l, inc_c) - 1;
184     if (dot(dir, hull.front().F) >= opt_val) {
185         it_r2 = upper_bound(hull.begin(), it_min + 1, l, dec_c) - 1;
186     } else {
187         it_r2 = upper_bound(it_max, hull.end(), l, dec_c) - 1;
188     }
189 } else {
190     it_r1 = upper_bound(it_max, it_min + 1, l, dec_c) - 1;
191 }
#0469

```

```

193     if (dot(dir, hull.front().F) <= opt_val) { #9772
194         it_r2 = upper_bound(hull.begin(), it_max + 1, l, inc_c) - 1;
195     } else {
196         it_r2 = upper_bound(it_min, hull.end(), l, inc_c) - 1;
197     }
198 } #9450
199 return MP(it_r1, it_r2);
200 }
201 Seg diameter() { // O(n)
202     Seg res;
203     ll dia_sq = 0;
204     auto it1 = hull.begin();
205     auto it2 = up_beg;
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};
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;
232         } else if (cross(v1, v2) < 0) { #5626
233             ++it1;
234         } else {
235             ++it2;
236         }
237     }
238     return res;
239 }
240 };

```

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}; }
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;
13     Edge *inv = NULL;
14     Edge *rep = NULL;
15     bool vis = false;
16 };
17 struct Seg { #8008
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) { #7311
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) { #6334
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 }
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; #1845
41 }
42 Edge *add_triangle(Edge *a, Edge *b, Edge *c) { #8219
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 }
52 Edge *add_point(Vec p, Edge *cur) { // returns outgoing edge #8178
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;
57     #0233
58 }
59 ran(i, 0, 3) {
60     if (triangle[i]->rep) {
61         Edge *res = add_point(p, triangle[i]->rep);
62         if (res)
63             return res; // unless we are on last layer we must exit here
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;
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) { #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) { #3471
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)) {
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;
99             if (out->tar == loc->tar) break;
100        }
101    }
102    return res;
103 } #6769 #6769
104 void extract_triangles(Edge *cur, vector<vector<Seg> &res) {
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             }
114             it = it->nxt;
115         } while (it != cur); #2104
116         if (inc) { #2104
117             Edge *triangle[3] = {cur, cur->nxt, cur->nxt->nxt}; #6207
118             res.resize(res.size() + 1);
119             vector<Seg> &tar = res.back();
120             ran(i, 0, 3) { #3011
121                 if ((abs(triangle[i]->tar.x) < max_co &&
122                     abs(triangle[(i + 1) % 3]->tar.x) < max_co))
123                     tar.push_back(
124                         {triangle[i]->tar, triangle[(i + 1) % 3]->tar}); #3011
125             }
126             if (tar.empty()) res.pop_back(); #3011
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]]; #1331
17            if (nxt) {
18                cur.second = nxt;
19            } else { #1331
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) { #5283

```

```

26     if (suf->nxt[(*cur.first)[i]]) {
27         nxt->suffix = suf->nxt[(*cur.first)[i]];
28         break;
29     }
30     suf = suf->suffix;
31 }
32 if (cur.first->size() > i + 1) nstate.push_back(cur);
33 state = nstate;
34 }
35 return root;
36 }
37 // auxiliary functions for searching and counting
38 Node *walk(Node *cur,
39 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;
43     cur = cur->suffix;
44 }
45 }
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);
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) { #9664
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) { #8305
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; #4595
18        return new_n; #3114
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) { #2130
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;

```

```

37 }
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 }
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) {
52             cur.second->calc_paths(); #2404
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)
73             suf->tree_links[s.size() - end_dist - suf->len - 1] = this; #6270
74         for (auto cur : nxt_char) cur.second->build_suffix_tree(s);
75     }
76 } #1268 %1268
77 struct SufAuto {
78     Node *last;
79     Node *root;
80     void extend(char new_c) {
81         Node *nlast = new Node; #0936
82         nlast->len = last->len + 1;
83         Node *swn = last;
84         while (swn && !swn->has_nxt(new_c)) {
85             swn->set_nxt(new_c, nlast);
86             swn = swn->suf;
87             #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); #1749
98             nlast->suf = eq_sbstr;
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            }
104        }
105    }
106    last = nlast;
107 }
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 /*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 }
22 }
23 void init() {
24 for (int i = 0; i < start.size(); i++) {
25 start[i] = i;
26 now[i] = 0;
27 lvl[i] = -1;
28 adj[i].clear();
29 rcap[i] = cap[i] = orig_cap[i];
30 cap_loc[i] = -1;
31 }
32 }
33 void update() {
34 queue<int> q;
35 q.push(source);
36 visited[source] = true;
37 while (!q.empty()) {
38 int cur = q.front();
39 q.pop();
40 for (int i : adj[cur]) {
41 if (cap[i] > 0 && !visited[i]) {
42 visited[i] = true;
43 now[i] = now[cur] + 1;
44 lvl[i] = lvl[cur] + 1;
45 q.push(i);
46 }
47 }
48 }
49 }
50 ll flow() {
51 ll ans = 0;
52 while (true) {
53 update();
54 if (now[sink] < 0) break;
55 int cur = sink;
56 ll f = INF;
57 while (cur != source) {
58 f = min(f, cap[adj[cur][now[cur]]]);
59 cur = adj[cur][now[cur]];
60 }
61 for (int i : adj[source]) {
62 if (cap[i] < f) continue;
63 cap[i] -= f;
64 cap[adj[source][now[source]]] += f;
65 }
66 ans += f;
67 }
68 return ans;
69 }
70 };

```


```

```

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 }
110 /*lp*/ bool recalc_dist_bellman_ford() /*ry*/ { // 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 }
117 /*rp*/ /*ly*/ pair<ll, /*ly*/ ll /*ly*/ /*ry*/ /*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     return /*ly*/ {/*ry*/ tot_flow /*ly*/, tot_cost} /*ry*/;
138 }
139 ll flow_on_edge(int idx) {
140     assert(idx < cap.size());
141     return orig_cap[idx] - cap[cap_loc[idx]];
142 }
143 };
144 const int nmax = 1055;
145 int main() {
146     // arguments source and sink, memory usage O(largest node index
147     // + input size)
148     int t;
149     scanf("%d", &t);
150     for (int i = 0; i < t; ++i) {
151         vector<tuple<int, int, ll, ll, ll>> edges;
152         int n;
153         scanf("%d", &n);
154         for (int j = 1; j <= n; ++j) {
155             edges.emplace_back(j, 2 * n + 1, 1, 0, 0);
156         }
157         for (int j = 1; j <= n; ++j) {
158             int card;
159             scanf("%d", &card);
160             edges.emplace_back(0, card, 1, 0, 0);
161         }
162         int ex_c;
163         scanf("%d", &ex_c);
164         for (int j = 0; j < ex_c; ++j) {
165             int a, b;
166             scanf("%d %d", &a, &b);
167             if (b < a) swap(a, b);
168             edges.emplace_back(a, b, nmax, 0, 1);
169             edges.emplace_back(b, n + b, nmax, 0, 0);
170             edges.emplace_back(n + b, a, nmax, 0, 1);
171         }
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 Cancelling $\mathcal{O}(cap \cdot nm)$

```

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;
```

```

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;          #3940
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) {        #5398
74             valid = true;
75             vector<Edge*> path;
76             int cap = 1000000000;
77             Node* cur = &nodes[i];
78             int clev = n;
79             vector<bool> expr(n, false);
80             while (!expr[cur->index]) {                  #6663
81                 expr[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);                           #9784
95             }
96             for (auto edge : path) {
97                 cap = min(cap, edge->getCap(cur));      #9838
98                 cur = edge->from(cur);
99             }

```

```

100         }
101         for (auto edge : path) {
102             result += edge->addFlow(cur, cap);
103             cur = edge->from(cur);                   #4467
104         }
105         if (!valid) break;
106     }
107     return result;                                #4029
108 }                                                 %2900
109 };



---



## 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); #2186
16 };
17 const static greater<pair<ll, Edge *>> comp;
18 static vector<Circle> to_proc; #4353
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() { #9916
28 if (!par) return this;
29 while (par->par) par = par->par;
30 return par;
31 }
32 void clean() { #0564
33 if (!no_dmst) {
34 in_use = false;
35 for (auto &cur : out_cands)
36 to_proc[cur.first].clean(cur.second);
37 }
38 }

```


```

```

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             }
50         if (con.empty()) { #2561
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     }
86     par = super; // root or anc() of first Node encountered in a
87         // loop
88     return super;
89 }

```

```

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        }
113        return true; #6725
114    }
115    ll weight() { #1568
116        ll res = 0;
117        for (auto &cur : edges) {
118            if (cur.inc) res += cur.w;
119        }
120        return res;
121    }
122    };
123 void DMST::Circle::clean(int idx) { #6369
124     if (!vis) { #1477
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     }
133 }
134 const greater<pair<ll, DMST::Edge *>> &DMST::comp; #2354
135 vector<DMST::Circle> DMST::to_proc; #2870
136 bool DMST::no_dmst;
137 DMST::Node *DMST::root;

```

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

```

1 struct vert;
2 struct edge {

```

<pre> 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 } #7106 31 int cnt_adj_bridges() { 32 int res = 0; 33 for (edge &nxt : con) res += nxt.is_bridge(); 34 return res; 35 } #9056 36 }; 37 bool edge::is_bridge() { 38 return exists && 39 (dest->seen > rev->dest->val dest->val < rev->dest->seen); 40 } #5223 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); </pre>	#8922	#0116	#1288	#8194	%8624	%7106	%9056
<pre> 54 int res = 0; 55 for (int i = 1; i <= n; ++i) res += graph[i].cnt_adj_bridges(); 56 cout << res / 2 << endl; 57 } <hr/> 16 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$ </pre>	#0321	#2422	#2081	#3875	#7568	#6880	#3565
<pre> 1 struct Graph { 2 int n; 3 vector<vector<int> > con; 4 Graph(int nsiz) { 5 n = nsiz; 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; </pre>	#0321	#2422	#2081	#3875	#7568	#6880	#3565
	#7763	#2422	#2081	#3875	#7568	#6880	#3565
	#7763	#2422	#2081	#3875	#7568	#6880	#3565
	#2422	#2081	#3875	#7568	#6880	#3565	#3565
	#2422	#2081	#3875	#7568	#6880	#3565	#3565
	#2422	#2081	#3875	#7568	#6880	#3565	#3565
	#2422	#2081	#3875	#7568	#6880	#3565	#3565
	#2422	#2081	#3875	#7568	#6880	#3565	#3565
	#2422	#2081	#3875	#7568	#6880	#3565	#3565

```

47 }
48 }; #0543
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--;
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> expr(2 * m, false);
64         for (auto u : order) {
65             vector<int> traversed;
66             g.dfs(u, traversed, expr);
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 }

```

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; #7684
6     }
7 } __attribute__((packed));
8 struct Lazy {
9     ll add;
10    ll assign_val; // LLONG_MIN if no assign;
11    void init() { #7883
12        add = 0;
13        assign_val = LLONG_MIN;
14    }
15    Lazy() { init(); }
16    void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len) {

```

```

17     lhs_lazy = *this;
18     rhs_lazy = *this;
19     init();
20 }
21 void merge(Lazy &oth, int len) { #0050
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 { #2924
29     if (assign_val != LLONG_MIN) {
30         cur.sum = len * assign_val;
31     }
32     cur.sum += len * add; #6280
33 }
34 } __attribute__((packed)); #0625 struct Node { // Following code should not need to be modified
35     int ver;
36     bool is_lazy = false;
37     Seg seg;
38     Lazy lazy; #6321
39     Node *lc = NULL, *rc = NULL;
40     void init() { #5313
41         if (!lc) {
42             lc = new Node{ver};
43             rc = new Node{ver};
44         }
45     }
46     Node *upd(int L, int R, int l, int r, Lazy &val, int tar_ver) { #8874
47         if (ver != tar_ver) {
48             Node *rep = new Node(*this); #2138
49             rep->ver = tar_ver;
50             return rep->upd(L, R, l, r, val, tar_ver);
51         }
52         if (L >= l && R <= r) {
53             val.apply_to_seg(seg, R - L);
54             lazy.merge(val, R - L);
55             is_lazy = true;
56         } else {
57             init(); #8209
58             int M = (L + R) / 2;
59             if (is_lazy) {
60                 Lazy l_val, r_val;
61                 lazy.split(l_val, r_val, R - L);
62                 lc = lc->upd(L, M, l_val, ver);
63                 rc = rc->upd(M, R, r_val, ver); #8104
64                 is_lazy = false;
65             }
66             Lazy l_val, r_val;
67         }

```

```

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);           #8581
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) {                                         #9373
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();                                              #6654
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;                                     #2185
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     }                                                       #4770
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;                                                 #4873
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;                                #1461
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;                      #9427
111        roots[ver]->get(0, len, l, r, pres, ptmp, roots.size() - 1);
112        return *pres;
113    }
114 };                                                       %7542
115 int n, m; // solves Mechanics Practice LAZY
116 cin >> n >> m;
117 SegTree seg_tree(1 << 17);
118 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 }
142 }                                                       #6178

```

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) {}                                         #9531
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;                                               #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     int cur;
37     vector<int> ord;
38     vector<int> chain_root;
39     vector<int> par;
40     void build_hld(int vertex, int parent, int chain_source) {
41         cur++;
42         ord[vertex] = cur;
43         chain_root[vertex] = chain_source;
44         par[vertex] = parent;
45         if (adj[vertex].size() > 1 ||
46             (vertex == 1 && adj[vertex].size() == 1)) {
47             int big_child, big_size = -1;
48             for (int child : adj[vertex]) {
49                 if ((child != parent) && (subtree_size[child] > big_size)) {
50                     big_child = child;
51                     big_size = subtree_size[child];
52                 }
53             }
54             build_hld(big_child, vertex, chain_source);
55             for (int child : adj[vertex]) {
56                 if ((child != parent) && (child != big_child))
57                     build_hld(child, vertex, child);
58             }
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
#7758
#4754
#4538
#1595
#7150
%1905
#18
}
structure = DS(vertexc + 5, initial);
aux = DS(50, initial);
}
void set(int vertex, int value) {
structure.set(ord[vertex], value);
}
T query_path(
int u, int v) { /* returns the "sum" of the path u->v */
int cur_id = 0;
while (chain_root[u] != chain_root[v]) {
if (ord[u] > ord[v]) {
cur_id++;
aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
u = par[chain_root[u]];
} else {
cur_id++;
aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
v = par[chain_root[v]];
}
}
cur_id++;
aux.set(cur_id,
structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
return aux.query(1, cur_id);
}
void print() {
for (int i = 1; i <= vertexc; i++) {
cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' '
<< par[i] << endl;
}
}
int main() {
int vertexc;
cin >> vertexc;
HLD<int, dummy> hld(vertexc);
for (int i = 0; i < vertexc - 1; i++) {
int u, v;
cin >> u >> v;
hld.add_edge(u, v);
}
hld.build();
hld.print();
int queryc;
cin >> queryc;
for (int i = 0; i < queryc; i++) {
int u, v;
cin >> u >> v;
hld.query_path(u, v);
cout << endl;
}
}
}

```

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         if (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 ^ 1] = p;
47             n->c[px ^ 1] = p;
48             n->c[px ^ 1] = n;
49             n->p = NIL;
50             update(p);

```

```

51         update(n);
52         break;
53     }
54     propagate(g);
55     if (gx = g->c[1] == p;
56         assert(g->c[gx] == p);
57     T gg = g->p;
58     if (gg && gg->c[1] == g;
59         assert(gg->c[ggx] == g);
60         if (gx == px) { /* zig zig */
61             g->c[gx] = p->c[gx ^ 1];
62             g->c[gx ^ 1] = p;
63             p->c[gx ^ 1] = g;
64             p->c[gx ^ 1] = p;
65             p->c[gx] = n->c[gx ^ 1];
66             p->c[gx ^ 1] = p;
67             n->c[gx ^ 1] = p;
68             n->c[gx ^ 1] = p;
69         } else { /* zig zag */
70             g->c[gx] = n->c[gx ^ 1];
71             g->c[gx ^ 1] = p;
72             n->c[gx ^ 1] = g;
73             n->c[gx ^ 1] = p;
74             p->c[gx ^ 1] = n->c[gx];
75             p->c[gx ^ 1] = p;
76             n->c[gx] = p;
77             n->c[gx ^ 1] = p;
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) {

```

```

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)); #4240
122 }
123 /*rp*/ %8430

```

20 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 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; #3273
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 { #2594
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(); #7303
23         for (; x < (int)pos.size(); x += x & -x)
24             elems[x].update(args...);
25     } else {
26         pos.push_back(cx);
27     }
28 } #8505

```

```

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; #2526
35 }
36 };
37 template <typename I_T>
38 struct wrapped {
39     I_T a = 0; #6509
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 }; #2858
46 // return 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 }
63 }


```

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

```

1 mt19937 randgen; #5615
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) { #5698
11            key = new_key;
12            value = new_value;
13            priority = randgen();
14            total = new_value;
15            lch = 0;
16        }
17    };
18    Node root;
19    void update(int key, int value, unsigned int priority) {
20        Node* cur = &root;
21        while (true) {
22            if (key < cur->key) {
23                if (cur->lch == 0) {
24                    cur->lch = new Node(key, value, priority);
25                    break;
26                } else {
27                    cur = cur->lch;
28                }
29            } else {
30                if (cur->rch == 0) {
31                    cur->rch = new Node(key, value, priority);
32                    break;
33                } else {
34                    cur = cur->rch;
35                }
36            }
37        }
38        update(cur->key, cur->value, cur->priority);
39    }
40    void print() {
41        print(root);
42    }
43    void print(Node* cur) {
44        if (cur == 0) return;
45        print(cur->lch);
46        cout << cur->key << " " << cur->value << endl;
47        print(cur->rch);
48    }
49    void print() {
50        print(root);
51    }
52    void print() {
53        print(root);
54    }
55    void print() {
56        print(root);
57    }
58    void print() {
59        print(root);
60    }
61    void print() {
62        print(root);
63    }
64    void print() {
65        print(root);
66    }
67    void print() {
68        print(root);
69    }
70    void print() {
71        print(root);
72    }
73    void print() {
74        print(root);
75    }
76    void print() {
77        print(root);
78    }
79    void print() {
80        print(root);
81    }
82    void print() {
83        print(root);
84    }
85    void print() {
86        print(root);
87    }
88    void print() {
89        print(root);
90    }
91    void print() {
92        print(root);
93    }
94    void print() {
95        print(root);
96    }
97    void print() {
98        print(root);
99    }
100   void print() {
101      print(root);
102  }
103  void print() {
104     print(root);
105  }
106  void print() {
107     print(root);
108  }
109  void print() {
110     print(root);
111  }
112  void print() {
113     print(root);
114  }
115  void print() {
116     print(root);
117  }
118  void print() {
119     print(root);
120  }
121  void print() {
122     print(root);
123  }
124  void print() {
125     print(root);
126  }
127  void print() {
128     print(root);
129  }
130  void print() {
131     print(root);
132  }
133  void print() {
134     print(root);
135  }
136  void print() {
137     print(root);
138  }
139  void print() {
140     print(root);
141  }
142  void print() {
143     print(root);
144  }
145  void print() {
146     print(root);
147  }
148  void print() {
149     print(root);
150  }
151  void print() {
152     print(root);
153  }
154  void print() {
155     print(root);
156  }
157  void print() {
158     print(root);
159  }
160  void print() {
161     print(root);
162  }
163  void print() {
164     print(root);
165  }
166  void print() {
167     print(root);
168  }
169  void print() {
170     print(root);
171  }
172  void print() {
173     print(root);
174  }
175  void print() {
176     print(root);
177  }
178  void print() {
179     print(root);
180  }
181  void print() {
182     print(root);
183  }
184  void print() {
185     print(root);
186  }
187  void print() {
188     print(root);
189  }
190  void print() {
191     print(root);
192  }
193  void print() {
194     print(root);
195  }
196  void print() {
197     print(root);
198  }
199  void print() {
200     print(root);
201  }
202  void print() {
203     print(root);
204  }
205  void print() {
206     print(root);
207  }
208  void print() {
209     print(root);
210  }
211  void print() {
212     print(root);
213  }
214  void print() {
215     print(root);
216  }
217  void print() {
218     print(root);
219  }
220  void print() {
221     print(root);
222  }
223  void print() {
224     print(root);
225  }
226  void print() {
227     print(root);
228  }
229  void print() {
230     print(root);
231  }
232  void print() {
233     print(root);
234  }
235  void print() {
236     print(root);
237  }
238  void print() {
239     print(root);
240  }
241  void print() {
242     print(root);
243  }
244  void print() {
245     print(root);
246  }
247  void print() {
248     print(root);
249  }
250  void print() {
251     print(root);
252  }
253  void print() {
254     print(root);
255  }
256  void print() {
257     print(root);
258  }
259  void print() {
260     print(root);
261  }
262  void print() {
263     print(root);
264  }
265  void print() {
266     print(root);
267  }
268  void print() {
269     print(root);
270  }
271  void print() {
272     print(root);
273  }
274  void print() {
275     print(root);
276  }
277  void print() {
278     print(root);
279  }
280  void print() {
281     print(root);
282  }
283  void print() {
284     print(root);
285  }
286  void print() {
287     print(root);
288  }
289  void print() {
290     print(root);
291  }
292  void print() {
293     print(root);
294  }
295  void print() {
296     print(root);
297  }
298  void print() {
299     print(root);
300  }
301  void print() {
302     print(root);
303  }
304  void print() {
305     print(root);
306  }
307  void print() {
308     print(root);
309  }
310  void print() {
311     print(root);
312  }
313  void print() {
314     print(root);
315  }
316  void print() {
317     print(root);
318  }
319  void print() {
320     print(root);
321  }
322  void print() {
323     print(root);
324  }
325  void print() {
326     print(root);
327  }
328  void print() {
329     print(root);
330  }
331  void print() {
332     print(root);
333  }
334  void print() {
335     print(root);
336  }
337  void print() {
338     print(root);
339  }
340  void print() {
341     print(root);
342  }
343  void print() {
344     print(root);
345  }
346  void print() {
347     print(root);
348  }
349  void print() {
350     print(root);
351  }
352  void print() {
353     print(root);
354  }
355  void print() {
356     print(root);
357  }
358  void print() {
359     print(root);
360  }
361  void print() {
362     print(root);
363  }
364  void print() {
365     print(root);
366  }
367  void print() {
368     print(root);
369  }
370  void print() {
371     print(root);
372  }
373  void print() {
374     print(root);
375  }
376  void print() {
377     print(root);
378  }
379  void print() {
380     print(root);
381  }
382  void print() {
383     print(root);
384  }
385  void print() {
386     print(root);
387  }
388  void print() {
389     print(root);
390  }
391  void print() {
392     print(root);
393  }
394  void print() {
395     print(root);
396  }
397  void print() {
398     print(root);
399  }
400  void print() {
401     print(root);
402  }
403  void print() {
404     print(root);
405  }
406  void print() {
407     print(root);
408  }
409  void print() {
410     print(root);
411  }
412  void print() {
413     print(root);
414  }
415  void print() {
416     print(root);
417  }
418  void print() {
419     print(root);
420  }
421  void print() {
422     print(root);
423  }
424  void print() {
425     print(root);
426  }
427  void print() {
428     print(root);
429  }
430  void print() {
431     print(root);
432  }
433  void print() {
434     print(root);
435  }
436  void print() {
437     print(root);
438  }
439  void print() {
440     print(root);
441  }
442  void print() {
443     print(root);
444  }
445  void print() {
446     print(root);
447  }
448  void print() {
449     print(root);
450  }
451  void print() {
452     print(root);
453  }
454  void print() {
455     print(root);
456  }
457  void print() {
458     print(root);
459  }
460  void print() {
461     print(root);
462  }
463  void print() {
464     print(root);
465  }
466  void print() {
467     print(root);
468  }
469  void print() {
470     print(root);
471  }
472  void print() {
473     print(root);
474  }
475  void print() {
476     print(root);
477  }
478  void print() {
479     print(root);
480  }
481  void print() {
482     print(root);
483  }
484  void print() {
485     print(root);
486  }
487  void print() {
488     print(root);
489  }
490  void print() {
491     print(root);
492  }
493  void print() {
494     print(root);
495  }
496  void print() {
497     print(root);
498  }
499  void print() {
500     print(root);
501  }
502  void print() {
503     print(root);
504  }
505  void print() {
506     print(root);
507  }
508  void print() {
509     print(root);
510  }
511  void print() {
512     print(root);
513  }
514  void print() {
515     print(root);
516  }
517  void print() {
518     print(root);
519  }
520  void print() {
521     print(root);
522  }
523  void print() {
524     print(root);
525  }
526  void print() {
527     print(root);
528  }
529  void print() {
530     print(root);
531  }
532  void print() {
533     print(root);
534  }
535  void print() {
536     print(root);
537  }
538  void print() {
539     print(root);
540  }
541  void print() {
542     print(root);
543  }
544  void print() {
545     print(root);
546  }
547  void print() {
548     print(root);
549  }
550  void print() {
551     print(root);
552  }
553  void print() {
554     print(root);
555  }
556  void print() {
557     print(root);
558  }
559  void print() {
560     print(root);
561  }
561  void print() {
562     print(root);
563  }
563  void print() {
564     print(root);
565  }
565  void print() {
566     print(root);
567  }
567  void print() {
568     print(root);
569  }
569  void print() {
570     print(root);
571  }
571  void print() {
572     print(root);
573  }
573  void print() {
574     print(root);
575  }
575  void print() {
576     print(root);
577  }
577  void print() {
578     print(root);
579  }
579  void print() {
580     print(root);
581  }
581  void print() {
582     print(root);
583  }
583  void print() {
584     print(root);
585  }
585  void print() {
586     print(root);
587  }
587  void print() {
588     print(root);
589  }
589  void print() {
590     print(root);
591  }
591  void print() {
592     print(root);
593  }
593  void print() {
594     print(root);
595  }
595  void print() {
596     print(root);
597  }
597  void print() {
598     print(root);
599  }
599  void print() {
600     print(root);
601  }
601  void print() {
602     print(root);
603  }
603  void print() {
604     print(root);
605  }
605  void print() {
606     print(root);
607  }
607  void print() {
608     print(root);
609  }
609  void print() {
610     print(root);
611  }
611  void print() {
612     print(root);
613  }
613  void print() {
614     print(root);
615  }
615  void print() {
616     print(root);
617  }
617  void print() {
618     print(root);
619  }
619  void print() {
620     print(root);
621  }
621  void print() {
622     print(root);
623  }
623  void print() {
624     print(root);
625  }
625  void print() {
626     print(root);
627  }
627  void print() {
628     print(root);
629  }
629  void print() {
630     print(root);
631  }
631  void print() {
632     print(root);
633  }
633  void print() {
634     print(root);
635  }
635  void print() {
636     print(root);
637  }
637  void print() {
638     print(root);
639  }
639  void print() {
640     print(root);
641  }
641  void print() {
642     print(root);
643  }
643  void print() {
644     print(root);
645  }
645  void print() {
646     print(root);
647  }
647  void print() {
648     print(root);
649  }
649  void print() {
650     print(root);
651  }
651  void print() {
652     print(root);
653  }
653  void print() {
654     print(root);
655  }
655  void print() {
656     print(root);
657  }
657  void print() {
658     print(root);
659  }
659  void print() {
660     print(root);
661  }
661  void print() {
662     print(root);
663  }
663  void print() {
664     print(root);
665  }
665  void print() {
666     print(root);
667  }
667  void print() {
668     print(root);
669  }
669  void print() {
670     print(root);
671  }
671  void print() {
672     print(root);
673  }
673  void print() {
674     print(root);
675  }
675  void print() {
676     print(root);
677  }
677  void print() {
678     print(root);
679  }
679  void print() {
680     print(root);
681  }
681  void print() {
682     print(root);
683  }
683  void print() {
684     print(root);
685  }
685  void print() {
686     print(root);
687  }
687  void print() {
688     print(root);
689  }
689  void print() {
690     print(root);
691  }
691  void print() {
692     print(root);
693  }
693  void print() {
694     print(root);
695  }
695  void print() {
696     print(root);
697  }
697  void print() {
698     print(root);
699  }
699  void print() {
700     print(root);
701  }
701  void print() {
702     print(root);
703  }
703  void print() {
704     print(root);
705  }
705  void print() {
706     print(root);
707  }
707  void print() {
708     print(root);
709  }
709  void print() {
710     print(root);
711  }
711  void print() {
712     print(root);
713  }
713  void print() {
714     print(root);
715  }
715  void print() {
716     print(root);
717  }
717  void print() {
718     print(root);
719  }
719  void print() {
720     print(root);
721  }
721  void print() {
722     print(root);
723  }
723  void print() {
724     print(root);
725  }
725  void print() {
726     print(root);
727  }
727  void print() {
728     print(root);
729  }
729  void print() {
730     print(root);
731  }
731  void print() {
732     print(root);
733  }
733  void print() {
734     print(root);
735  }
735  void print() {
736     print(root);
737  }
737  void print() {
738     print(root);
739  }
739  void print() {
740     print(root);
741  }
741  void print() {
742     print(root);
743  }
743  void print() {
744     print(root);
745  }
745  void print() {
746     print(root);
747  }
747  void print() {
748     print(root);
749  }
749  void print() {
750     print(root);
751  }
751  void print() {
752     print(root);
753  }
753  void print() {
754     print(root);
755  }
755  void print() {
756     print(root);
757  }
757  void print() {
758     print(root);
759  }
759  void print() {
760     print(root);
761  }
761  void print() {
762     print(root);
763  }
763  void print() {
764     print(root);
765  }
765  void print() {
766     print(root);
767  }
767  void print() {
768     print(root);
769  }
769  void print() {
770     print(root);
771  }
771  void print() {
772     print(root);
773  }
773  void print() {
774     print(root);
775  }
775  void print() {
776     print(root);
777  }
777  void print() {
778     print(root);
779  }
779  void print() {
780     print(root);
781  }
781  void print() {
782     print(root);
783  }
783  void print() {
784     print(root);
785  }
785  void print() {
786     print(root);
787  }
787  void print() {
788     print(root);
789  }
789  void print() {
790     print(root);
791  }
791  void print() {
792     print(root);
793  }
793  void print() {
794     print(root);
795  }
795  void print() {
796     print(root);
797  }
797  void print() {
798     print(root);
799  }
799  void print() {
800     print(root);
801  }
801  void print() {
802     print(root);
803  }
803  void print() {
804     print(root);
805  }
805  void print() {
806     print(root);
807  }
807  void print() {
808     print(root);
809  }
809  void print() {
810     print(root);
811  }
811  void print() {
812     print(root);
813  }
813  void print() {
814     print(root);
815  }
815  void print() {
816     print(root);
817  }
817  void print() {
818     print(root);
819  }
819  void print() {
820     print(root);
821  }
821  void print() {
822     print(root);
823  }
823  void print() {
824     print(root);
825  }
825  void print() {
826     print(root);
827  }
827  void print() {
828     print(root);
829  }
829  void print() {
830     print(root);
831  }
831  void print() {
832     print(root);
833  }
833  void print() {
834     print(root);
835  }
835  void print() {
836     print(root);
837  }
837  void print() {
838     print(root);
839  }
839  void print() {
840     print(root);
841  }
841  void print() {
842     print(root);
843  }
843  void print() {
844     print(root);
845  }
845  void print() {
846     print(root);
847  }
847  void print() {
848     print(root);
849  }
849  void print() {
850     print(root);
851  }
851  void print() {
852     print(root);
853  }
853  void print() {
854     print(root);
855  }
855  void print() {
856     print(root);
857  }
857  void print() {
858     print(root);
859  }
859  void print() {
860     print(root);
861  }
861  void print() {
862     print(root);
863  }
863  void print() {
864     print(root);
865  }
865  void print() {
866     print(root);
867  }
867  void print() {
868     print(root);
869  }
869  void print() {
870     print(root);
871  }
871  void print() {
872     print(root);
873  }
873  void print() {
874     print(root);
875  }
875  void print() {
876     print(root);
877  }
877  void print() {
878     print(root);
879  }
879  void print() {
880     print(root);
881  }
881  void print() {
882     print(root);
883  }
883  void print() {
884     print(root);
885  }
885  void print() {
886     print(root);
887  }
887  void print() {
888     print(root);
889  }
889  void print() {
890     print(root);
891  }
891  void print() {
892     print(root);
893  }
893  void print() {
894     print(root);
895  }
895  void print() {
896     print(root);
897  }
897  void print() {
898     print(root);
899  }
899  void print() {
900     print(root);
901  }
901  void print() {
902     print(root);
903  }
903  void print() {
904     print(root);
905  }
905  void print() {
906     print(root);
907  }
907  void print() {
908     print(root);
909  }
909  void print() {
910     print(root);
911  }
911  void print() {
912     print(root);
913  }
913  void print() {
914     print(root);
915  }
915  void print() {
916     print(root);
917  }
917  void print() {
918     print(root);
919  }
919  void print() {
920     print(root);
921  }
921  void print() {
922     print(root);
923  }
923  void print() {
924     print(root);
925  }
925  void print() {
926     print(root);
927  }
927  void print() {
928     print(root);
929  }
929  void print() {
930     print(root);
931  }
931  void print() {
932     print(root);
933  }
933  void print() {
934     print(root);
935  }
935  void print() {
936     print(root);
937  }
937  void print() {
938     print(root);
939  }
939  void print() {
940     print(root);
941  }
941  void print() {
942     print(root);
943  }
943  void print() {
944     print(root);
945  }
945  void print() {
946     print(root);
947  }
947  void print() {
948     print(root);
949  }
949  void print() {
950     print(root);
951  }
951  void print() {
952     print(root);
953  }
953  void print() {
954     print(root);
955  }
955  void print() {
956     print(root);
957  }
957  void print() {
958     print(root);
959  }
959  void print() {
960     print(root);
961  }
961  void print() {
962     print(root);
963  }
963  void print() {
964     print(root);
965  }
965  void print() {
966     print(root);
967  }
967  void print() {
968     print(root);
969  }
969  void print() {
970     print(root);
971  }
971  void print() {
972     print(root);
973  }
973  void print() {
974     print(root);
975  }
975  void print() {
976     print(root);
977  }
977  void print() {
978     print(root);
979  }
979  void print() {
980     print(root);
981  }
981  void print() {
982     print(root);
983  }
983  void print() {
984     print(root);
985  }
985  void print() {
986     print(root);
987  }
987  void print() {
988     print(root);
989  }
989  void print() {
990     print(root);
991  }
991  void print() {
992     print(root);
993  }
993  void print() {
994     print(root);
995  }
995  void print() {
996     print(root);
997  }
997  void print() {
998     print(root);
999  }
999  void print() {
1000    print(root);
1001  }
1001  void print() {
1002    print(root);
1003  }
1003  void print() {
1004    print(root);
1005  }
1005  void print() {
1006    print(root);
1007  }
1007  void print() {
1008    print(root);
1009  }
1009  void print() {
1010    print(root);
1011  }
1011  void print() {
1012    print(root);
1013  }
1013  void print() {
1014    print(root);
1015  }
1015  void print() {
1016    print(root);
1017  }
1017  void print() {
1018    print(root);
1019  }
1019  void print() {
1020    print(root);
1021  }
1021  void print() {
1022    print(root);
1023  }
1023  void print() {
1024    print(root);
1025  }
1025  void print() {
1026    print(root);
1027  }
1027  void print() {
1028    print(root);
1029  }
1029  void print() {
1030    print(root);
1031  }
1031  void print() {
1032    print(root);
1033  }
1033  void print() {
1034    print(root);
1035  }
1035  void print() {
1036    print(root);
1037  }
1037  void print() {
1038    print(root);
1039  }
1039  void print() {
1040    print(root);
1041  }
1041  void print() {
1042    print(root);
1043  }
1043  void print() {
1044    print(root);
1045  }
1045  void print() {
1046    print(root);
1047  }
1047  void print() {
1048    print(root);
1049  }
1049  void print() {
1050    print(root);
1051  }
1051  void print() {
1052    print(root);
1053  }
1053  void print() {
1054    print(root);
1055  }
1055  void print() {
1056    print(root);
1057  }
1057  void print() {
1058    print(root);
1059  }
1059  void print() {
1060    print(root);
1061  }
1061  void print() {
1062    print(root);
1063  }
1063  void print() {
1064    print(root);
1065  }
1065  void print() {
1066    print(root);
1067  }
1067  void print() {
1068    print(root);
1069  }
1069  void print() {
1070    print(root);
1071  }
1071  void print() {
1072    print(root);
1073  }
1073  void print() {
1074    print(root);
1075  }
1075  void print() {
1076    print(root);
1077  }
1077  void print() {
1078    print(root);
1079  }
1079  void print() {
1080    print(root);
1081  }
1081  void print() {
1082    print(root);
1083  }
1083  void print() {
1084    print(root);
1085  }
1085  void print() {
1086    print(root);
1087  }
1087  void print() {
1088    print(root);
1089  }
1089  void print() {
1090    print(root);
1091  }
1091  void print() {
1092    print(root);
1093  }
1093  void print() {
1094    print(root);
1095  }
1095  void print() {
1096    print(root);
1097  }
1097  void print() {
1098    print(root);
1099  }
1099  void print() {
1100    print(root);
1101  }
1101  void print() {
1102    print(root);
1103  }
1103  void print() {
1104    print(root);
1105  }
1105  void print() {
1106    print(root);
1107  }
1107  void print() {
1108    print(root);
1109  }
1109  void print() {
1110    print(root);
1111  }
1111  void print() {
1112    print(root);
1113  }
1113  void print() {
1114    print(root);
1115  }
1115  void print() {
1116    print(root);
1117  }
1117  void print() {
1118    print(root);
1119  }
1119  void print() {
1120    print(root);
1121  }
1121  void print() {
1122    print(root);
1123  }
1123  void print() {
1124    print(root);
1125  }
1125  void print() {
1126    print(root);
1127  }
1127  void print() {
1128    print(root);
1129  }
1129  void print() {
1130    print(root);
1131  }
1131  void print() {
1132    print(root);
1133  }
1133  void print() {
1134    print(root);
1135  }
1135  void print() {
1136    print(root);
1137  }
1137  void print() {
1138    print(root);
1139  }
1139  void print() {
1140    print(root);
1141  }
1141  void print() {
1142    print(root);
1143  }
1143  void print() {
1144    print(root);
1145  }
1145  void print()
```

```

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};
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 }
79 long long get(int l, int r) {
80     return sum_upto(r + 1, root) - sum_upto(l, root);
81 }
82 };
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 }

#4295 #9633 #5233 #6988 #7230 #6282 #3510 #8918 #9760 #1416 #7634 #8122 #0094 #4959 #5369

```

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;

```

```

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};           #8384
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{                                #5371
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) {          #7637
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) {          #0670
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;                                     #0102
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);          #3349
42                 idx_p += 1 << (ord - inc_pow), ++idx) {
43                 root_pow[idx] = Complex{cos(-idx_p * M_PI / (1 << (ord - 1))), sin(-idx_p * M_PI / (1 << (ord - 1)))};
44                 if (is_inv) root_pow[idx].b = -root_pow[idx].b;
45             }
46         }
47     }
48     if (invert != is_inv) {
49         is_inv = invert;
50         for (Complex &cur : root_pow) cur.b = -cur.b;           #7526
51     }
52     for (int i = 1, j = 0; i < (1 << ord); ++i) {
53         int m = 1 << (ord - 1);
54         bool cont = true;                                         #0510
55         while (cont) {
56             cont = j & m;
57             j ^= m;
58             m >>= 1;
59         }
60         if (i < j) swap(arr[i], arr[j]);                         #0506
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); #4380
65     mult_poly_mod(                                            %4380
66         vector<int> &a, vector<int> &b, vector<int> &c) { // c += a*b
67         static vector<Complex>                                // correct upto 0.5-2M elements(mod ~ 1e9)
68         arr[4]; // correct upto 0.5-2M elements(mod ~ 1e9)
69     }
70 }

```

```

71 if (c.size() < 400) { #8811
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()); #4629
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) #9591
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; #2625
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) { #3501
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) { #9971
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) { #4471
110        fft(arr[i], fft_ord, true);
111        for (int k = 0; k < (int)c.size(); ++k) { #8403
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 } #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; #0237
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; #0780
11        return res; // in [0, n-1)
12    }
13    ull sqp1(ull a) { return multf(a, a) + 1; } %9493
14 };
15 ull pow_mod(ull a, ull n, ModArithm &arithm) {
16     ull res = 1;
17     for (ull i = 1; i <= n; i <= 1) { #1758
18         if (n & i) res = arithm.multf(res, a);
19         a = arithm.multf(a, a);
20     }
21     return res; %2144
22 }
23 vector<char> small_primes = { #8104
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; #6402
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; #0876
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; #4806
41         }
42         return true; %0975
43     }
44 ll pollard_rho(ll n) { #2118
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) { #5290
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)); #4468
59         v_cur = arithm.sqp1(v_cur);
60         ++idx;
61     }
62     if (!cur) {
63         for (int j = 0; j < cum_cnt; ++j) { #7912
64             ll g = __gcd(abs(v - lv), n);
65             if (g == 1) {
66                 v = arithm.sqp1(v);
67             } else if (g == n) {
68                 break; #0906
69             } else {
70                 return g;
71             }
72         }
73         break; #7208
74     } else {
75         ll g = __gcd(cur, n);
76         if (g != 1) return g; #2298
77     }
78     v = v_cur;
79     idx += j_stop;
80     if (idx == tar) { #1174
81         lv = v;
82         tar *= 2;
83         v = arithm.sqp1(v);
84         ++idx;
85     }
86 }
87 } #3542 %3542
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) { #3770
93         while (!(n % p)) {
94             ++res_act[p];
95             n /= p;
96         }
97     }
98     if (n != 1) prime_factor(n, &res_act); #4612
99     return res_act;
100 }

```

```

102 if (is_prime(n)) {
103     ++(*res)[n];
104 } else {
105     ll factor = pollard_rho(n);
106     prime_factor(factor, res);
107     prime_factor(n / factor, res);
108 }
109 return map<ll, int>();
110 } // Usage: fact = prime
#5290   #5350
#4468   #5477



---



## 25 Symmetric Submodular Functions; Queyranne's algorithm



#7912 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



#0906



```

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]
#2298
#1174
%3542
CF)
#3770
#4612

```


```

```

35     yield tu
36     s = merge(tu, s)


---


26 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());
5     cs[0] = K::unity;                                     #0349
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++) {           #4390
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) {
15            m++;                                         #8445
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++)
20                cs[i + m] -= w * bs[i];                  #9661
21            l = k + 1 - l;
22            swap(bs, ts);
23            b = d;
24            m = 1;
25        } else {                                         #2815
26            K w = d / b;
27            for (int i = 0; i < (int)cs.size() - m; i++)
28                cs[i + m] -= w * bs[i];
29            m++;
30        }
31    }
32    cs.resize(l + 1);
33    while (cs.back() == K::zero) cs.pop_back();
34    return cs;
35 }                                                 #6267          %6267

```
