

University of Tartu ICPC Team Notebook (2018-2019)

March 25, 2019

Maxflow Complexity
Min Rotation of string
Series
Symmetric Submodular Functions; Queyranne's algorithm
Subset convolution $\mathcal{O}(N \log N)$
Pythagorean Triples
Primes
Estimates
Möbius inversion
Derangements
Burnside's lemma
Partition function
Stirling numbers of the first kind
Eulerian numbers
Stirling numbers of the second kind
Bell numbers
Catalan numbers
2D geometry
3D geometry
Triangle centers
Seg-Seg intersection, halfplane intersection area
Convex polygon algorithms
Delaunay triangulation $O(n \log n)$
Contest setup
crc.sh
gcc ordered set, hashtable
PRNGs and Hash functions
Memorypool
Radixsort 50M 64 bit integers as single array in 1 sec
FFT 10-15M length/sec
Fast mod mult, Rabin Miller prime check, Pollard rho factorization $O(\sqrt{p})$
Berlekamp-Massey $O(LN)$
Linear algebra
Polynomial roots and $O(n^2)$ interpolation
Simplex algorithm
Dinic
Min Cost Max Flow with Cycle Cancelling $O(Cnm)$

Global Min Cut $O(V^3)$	15
Aho Corasick $O(\alpha \sum \text{len})$	15
Suffix automaton and tree $O((n+q)\log(\alpha))$	
10+M length/s	16
Palindromic tree $O(n)$	17
DMST $O(E \log V)$	17
Dominator tree $O(N \log N)$	18
Bridges $O(n)$	18
2-Sat $O(n)$ and SCC $O(n)$	19
Templated multi dimensional BIT $O(\log(n)^d)$ per query	19
Treap $O(\log(n))$ per query	20
Generic persistent compressed lazy segment tree	20
Templated HLD $O(M(n) \log n)$ per query	21
Splay Tree + Link-Cut $O(N \log N)$	22
Maxflow Complexity	
$\mathcal{O}(V^2 E)$ – Dinic	
$\Theta(VE \log U)$ – Capacity scaling	
$\Theta(\text{flow}E)$ – Small flow	
$\Theta(\min\{V^{\frac{2}{3}}, E^{\frac{1}{2}}\}E)$ – Unitary capacities	
$\Theta(\sqrt{V}E)$ – Each vertex other than S,T has only a single incoming unitary edge or outgoing one (bipartite matching)	
$\Theta(\text{flow}E \log V)$ – Min-cost-max flow	
<p>L-R-flow</p>	
Min Rotation of string	
<pre>int a=0, N=s.size(); s += s; ran(b,0,N){ ran(i,0,N) { if (a+i == b s[a+i] < s[b+i]) { b += max(0, i-1); break; } if (s[a+i] > s[b+i]) { a = b; break; } } }</pre>	

```

15     }
15 }
16 return a;
17 Series
18
19     
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, (-\infty < x < \infty)$$

20
21     
$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, (-1 < x \leq 1)$$

22
23     
$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, (-1 \leq x \leq 1)$$

24
25     
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, (-\infty < x < \infty)$$

26
27     
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, (-\infty < x < \infty)$$

28

29 Symmetric Submodular Functions; Queyranne's
30 algorithm
31 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  $f(X+x) - f(X) \leq f(Y+x) - f(Y)$ .
32 Hereditary family: such set  $I \subseteq 2^V$  so that  $X \subset Y \wedge Y \in I \Rightarrow X \in I$ .
33 Loop: such  $v \in V$  so that  $v \notin I$ .
34
35 def minimize():
36     s = merge_all_loops()
37     while size >= 3:
38         t, u = find_pp()
39         {u} is a possible minimizer
40         tu = merge(t, u)
41         if tu not in I:
42             s = merge(tu, s)
43         for x in V:
44             {x} is a possible minimizer
45 def find_pp():
46     W = {s} # s as in minimizer()
47     todo = V/W
48     ord = []
49     while len(todo) > 0:
50         x = min(todo, key=lambda x: f(W+{x}) -
51                 f({x}))
52         W += {x}
53         todo -= {x}
54         ord.append(x)
55     return ord[-1], ord[-2]

```

```

def enum_all_minimal_minimizers(X):
    # X is a inclusionwise minimal minimizer
    s = merge(s, X)
    yield X
    for {v} in I:
        if f({v}) == f(X):
            yield X
            s = merge(v, s)
    while size(V) >= 3:
        t, u = find_pp()
        tu = merge(t, u)
        if tu not in I:
            s = merge(tu, s)
        elif f({tu}) == f(X):
            yield tu
            s = merge(tu, s)
Subset convolution  $\mathcal{O}(N \log N)$ 
c[z] =  $\sum_{z=x \oplus y} a[x] \cdot b[y]$ , where  $\oplus$  is one of AND, OR, XOR. The size of  $a$  must be a power of two.
void FST(vi& a, bool inv) { //size(a) == 2^x
    for (int n = (int)a.size(), step = 1; step <
        ~n; step *= 2) {
        for (int i = 0; i < n; i += 2 * step)
            ran(j, i, i + step) {
                int &u = a[j], &v = a[j + step];
                tie(u, v) =
                    inv ? pii(v - u, u) : pii(v, u + v);
                    // AND
                    inv ? pii(v, u - v) : pii(u + v, u);
                    // OR
                    pii(u + v, u - v);
                    // XOR
            }
        if (inv) trav(x, a) x /= a.size(); // XOR
        ~only
    }
    vi conv(vi a, vi b) {
        FST(a, 0);
        FST(b, 0);
        ran(i, 0, (int)a.size()) a[i] *= b[i];
        FST(a, 1);
        return a;
    }
}

```

Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \quad b = k \cdot (2mn), \quad c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

Primitive roots exist modulo any prime power p^a , except for $p = 2, a > 2$, and there are $\varphi(\varphi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^\times$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

Estimates

$$\sum_{d|n} d = O(n \log \log n).$$

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

Möbius inversion

$$\forall n : g(n) = \sum_{d|n} f(d) \iff \forall n : f(n) = \sum_{d|n} \mu(d)g\left(\frac{n}{d}\right)$$

Derangements

Permutations of a set such that none of the elements appear in their original position.

$$\begin{aligned} D(n) &= (n-1)(D(n-1) + D(n-2)) = \\ &nD(n-1) + (-1)^n = \left\lfloor \frac{n!}{e} \right\rfloor \end{aligned}$$

Burnside's lemma

Given a group G of symmetries and a set X , the number of elements of X up to symmetry equals

$$\frac{1}{|G|} \sum_{g \in G} |X^g|,$$

where X^g are the elements fixed by g ($g \cdot x = x$).

If $f(n)$ counts “configurations” (of some sort) of length n , we can ignore rotational symmetry using $G = \mathbb{Z}_n$ to get

$$g(n) = \frac{1}{n} \sum_{k=0}^{n-1} f(\gcd(n, k)) = \frac{1}{n} \sum_{k|n} f(k)\varphi(n/k).$$

Partition function

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k - 1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	$\sim 2e5$	$\sim 2e8$

Stirling numbers of the first kind

Number of permutations on n items with k cycles.

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k), \quad c(0, 0) = 1$$

$$\sum_{k=0}^n c(n, k)x^k = x(x+1) \dots (x+n-1)$$

$$c(8, k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$$

$$c(n, 2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$$

Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j :s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j :s s.t. $\pi(j) \geq j$, k j :s s.t. $\pi(j) > j$.

$$E(n, k) = (n-k)E(n-1, k-1) + (k+1)E(n-1, k)$$

$$E(n, 0) = E(n, n-1) = 1$$

$$E(n, k) = \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n$$

Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

$$S(n, k) = S(n-1, k-1) + kS(n-1, k)$$

$$S(n, 1) = S(n, n) = 1$$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

Bell numbers

Total number of partitions of n distinct elements. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$. For p prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

Catalan numbers

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1, C_{n+1} = \frac{2(2n+1)}{n+2} C_n, C_{n+1} = \sum C_i C_{n-i}$$

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots$$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

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; // = |O_1P| * d
    double h2 = r1*r1 - pd*pd/d2; // = h^2
    if (h2 >= 0) {
        pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
        out = {p-h, p+h};}
    return 1 + sgn(h2);
```

Tangent lines:

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

3D geometry

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

S above $PQRS$ 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:

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

```

Triangle centers
const double min_delta = 1e-13;
const double coord_max = 1e6;
typedef complex<double> point;
point A, B, C; // vertexes of the triangle
bool collinear() {
    double min_diff =
        min(abs(A - B), min(abs(A - C), abs(B - C)));
    if (min_diff < coord_max * min_delta) return true;
    point sp = (B - A) / (C - A);
    double ang = M_PI / 2 - abs(abs(arg(sp)) - M_PI / 2);
    return ang < min_delta;
    // positive angle with the real line
}
point circum_center() {
    if (collinear()) return point(NAN, NAN);
    // squared lengths of sides
    double a2 = norm(B - C);
    double b2 = norm(A - C);
    double c2 = norm(A - B);
    // barycentric coordinates of the circumcenter
    // sin(2 * alpha) works also
    double c_A = a2 * (b2 + c2 - a2);
    double c_B = b2 * (a2 + c2 - b2);
    double c_C = c2 * (a2 + b2 - c2);
    double sum = c_A + c_B + c_C;
    c_A /= sum;
    c_B /= sum;
    c_C /= sum;
    return c_A * A + c_B * B + c_C * C; // cartesian
}
point centroid() { // center of mass
    return (A + B + C) / 3.0;
}
point ortho_center() { // euler line
    point O = circum_center();
    return O + 3.0 * (centroid() - O);
};
point nine_point_circle_center() { // euler line
    point O = circum_center();
    return O + 1.5 * (centroid() - O);
};
point in_center() {
    if (collinear()) return point(NAN, NAN);
    double a = abs(B - C); // side lengths
    double b = abs(A - C);
    double c = abs(A - B);
    // trilinear coordinates are (1,1,1)
    double sum = a + b + c;
    a /= sum;
    b /= sum;
    c /= sum;
    return a * A + b * B + c * C; // cartesian
}

```

		9596		
}				
Seg-Seg intersection, halfplane intersection area				
struct Seg {				
Vec a, b;				
Vec d() { return b - a; }				
};				
Vec intersection(Seg l, Seg r) {				
Vec dl = l.d(), dr = r.d();				
if (cross(dl, dr) == 0) return {nanl(""), nanl("")};				
double h = cross(dr, l.a - r.a) / len(dr);				
double dh = cross(dr, dl) / len(dr);	3280			
return l.a + dl * (h / -dh);				
}				
// Returns the area bounded by halfplanes				
double calc_area(const vector<Seg>& lines) {				
double lb = -HUGE_VAL, ub = HUGE_VAL;				
vector<Seg> slines[2];				
for (auto line : lines) {				
if (line.b.y == line.a.y) {				
if (line.a.x < line.b.x) {				
lb = max(lb, line.a.y);				
} else {				
ub = min(ub, line.a.y);				
}				
} else if (line.a.y < line.b.y) {				
slines[1].push_back(line);				
} else {				
slines[0].push_back({line.b, line.a});				
}				
}				
ran (i, 0, 2) {				
sort (slines[i].begin(), slines[i].end(),				
λ &(Seg l, Seg r) {				
if (cross(l.d(), r.d()) == 0)				
return normal(l.d()) * l.a >				
normal(r.d()) * r.a;				
return (1 - 2 * i) * cross(l.d(), r.d()) < 0;				
});				
}				
// Now find the application area of the lines and clean				
// up redundant ones				
vector<double> ap_s[2];				
ran(side, 0, 2) {				
vector<double>& apply = ap_s[side];				
vector<Seg> clines;				
for (auto line : slines[side]) {				
while (clines.size() > 0) {				
Seg other = clines.back();				
if (cross(line.d(), other.d()) != 0) {				
double start = intersection(line, other).y;				
if (start > apply.back()) break ;				
}				
clines.pop_back();				
apply.pop_back();				
}				
if (clines.size() == 0) {				
apply.push_back(-HUGE_VAL);				
}				
}				
else {				
apply.push_back(
intersection(line, clines.back().y));				
}				
clines.push_back(line);				
slines[side] = clines;				
ap_s[0].push_back(HUGE_VALL);	8994			
ap_s[1].push_back(HUGE_VALL);				
double result = 0;				
{				
double lb = -HUGE_VALL, ub;				
for (int i = 0, j = 0; i < (int)slines[0].size() &&				
j < (int)slines[1].size();				
lb = ub) {	4531			
ub = min(ap_s[0][i + 1], ap_s[1][j + 1]);				
double alb = lb, aub = ub;				
Seg l[2] = {slines[0][i], slines[1][j]};				
if (cross(l[1].d(), l[0].d()) > 0) {	5904			
alb = max(alb, intersection(l[0], l[1]).y);				
} else if (cross(l[1].d(), l[0].d()) < 0) {				
aub = min(aub, intersection(l[0], l[1]).y);				
}				
alb = max(alb, lb);				
aub = min(aub, ub);				
aub = max(aub, alb);				
ran(k, 0, 2) {				
double x1 = l[0].a.x + (alb - l[0].a.y) /				
l[0].d().y * l[0].d().x;				
double x2 = l[0].a.x + (aub - l[0].a.y) /				
l[0].d().y * l[0].d().x;	8864			
result +=				
(-1 + 2 * k) * (aub - alb) * (x1 + x2) / 2;				
}				
if (ap_s[0][i + 1] < ap_s[1][j + 1]) {				
i++;				
} else {				
j++;				
}				
}				
}				
return result;			3672	
			Convex polygon algorithms	
			typedef pair<int, int> Vec;	
			typedef pair<Vec, Vec> Seg;	
			typedef vector<Seg>::iterator SegIt;	
			#define F first	
			#define S second	
			#define MP(x, y) make_pair(x, y)	
			Vec sub(const Vec &v1, const Vec &v2) {	2608
			return MP(v1.F - v2.F, v1.S - v2.S);	
			}	
			11 dot(const Vec &v1, const Vec &v2) {	0940
			return (11)v1.F * v2.F + (11)v1.S * v2.S;	

```

11 cross(const Vec &v1, const Vec &v2) {
    return (ll)v1.F * v2.S - (ll)v2.F * v1.S;
}
11 dist_sq(const Vec &p1, const Vec &p2) {
    return (ll)(p2.F - p1.F) * (p2.F - p1.F) +
        (ll)(p2.S - p1.S) * (p2.S - p1.S);
}
struct Point;
multiset<Point>::iterator end_node;
struct Point {
    Vec p;
    typename multiset<Point>::iterator get_it() const {
        // gcc rb_tree dependent
        tuple<void *> tmp = {(void *)this - 32};
        return *(multiset<Point>::iterator *)(&tmp);
    }
    bool operator<(const Point &rhs) const {
        return (p.F < rhs.p.F); // sort by x
    }
    bool operator<(const Vec &q) const {
        auto nxt = next(get_it()); // convex hull trick
        if (nxt == end_node) return 0; // nxt == end()
        return q.S * dot(p, {q.F, 1}) <
            q.S * dot(nxt->p, {q.F, 1});
    }
};
template <int part> // 1 = upper, -1 = lower
struct HullDynamic : public multiset<Point, less<> > {
    bool bad(iterator y) {
        if (y == begin()) return 0;
        auto x = prev(y);
        auto z = next(y);
        if (z == end())
            return y->p.F == x->p.F && y->p.S <= x->p.S;
        return part *
            cross(sub(y->p, x->p), sub(y->p, z->p)) <=
            0;
    }
    void insert_point(int m, int b) { // O(log(N))
        auto y = insert({{m, b}});
        if (bad(y)) {
            erase(y);
            return;
        }
        while (next(y) != end() && bad(next(y)))
            erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll eval(
        int x) { // O(log(N)) upper maximize dot({x, 1}, v)
        end_node =
            end(); // lower minimize dot({x, 1}, v)
        auto it = lower_bound((Vec){x, part});
        return (ll)it->p.F * x + it->p.S;
    }
};
struct Hull {

```

```

vector<Seg> hull;
SegIt up_beg;
template <typename It>
void extend(It beg, It end) { // O(n)
    vector<Vec> r;
    for (auto it = beg; it != end; ++it) {
        if (r.empty() || *it != r.back()) {
            while (r.size() >= 2) {
                int n = r.size();
                Vec v1 = {r[n - 1].F - r[n - 2].F,
                           r[n - 1].S - r[n - 2].S}; 1365
                Vec v2 = {
                    it->F - r[n - 2].F, it->S - r[n - 2].S};
                if (cross(v1, v2) > 0) break;
                r.pop_back();
            }
            r.push_back(*it);
        }
    }
    ran(i, 0, (int)r.size() - 1) 8095
    hull.emplace_back(r[i], r[i + 1]);
}
Hull(vector<Vec> &vert) { // atleast 2 distinct points
    sort(vert.begin(), vert.end()); // O(n log(n))
    extend(vert.begin(), vert.end());
    int diff = hull.size(); 6560
    extend(vert.rbegin(), vert.rend());
    up_beg = hull.begin() + diff;
}
bool contains(Vec p) { // O(log(n))
    if (p < hull.front().F || p > up_beg->F)
        return false;
{
    auto it_low = lower_bound(hull.begin(), up_beg,
                               MP(MP(p.F, (int)-2e9), MP(0, 0))); 1542
    if (it_low != hull.begin()) --it_low;
    Vec a = {it_low->S.F - it_low->F.F,
             it_low->S.S - it_low->F.S};
    Vec b = {p.F - it_low->F.F, p.S - it_low->F.S}; 1144
    if (cross(a, b) < 0) // < 0 is inclusive, <= 0 is exclusive
        return false;
}
{
    auto it_up = lower_bound(hull.rbegin(),
                             hull.rbegin() + (hull.end() - up_beg),
                             MP(MP(p.F, (int)2e9), MP(0, 0))); 9423
    if (it_up - hull.rbegin() == hull.end() - up_beg)
        --it_up;
    Vec a = {it_up->F.F - it_up->S.F,
             it_up->F.S - it_up->S.S}; 0193
    Vec b = {p.F - it_up->S.F, p.S - it_up->S.S};
    if (cross(a, b) > 0) // > 0 is inclusive, >= 0 is exclusive
        return false;
}
return true;
}

```

```

} } %3267
// The function can have only one local min and max
// and may be constant only at min and max.
template <typename T>
SegIt max(function<T(Seg &)> f) { // O(log(n))
    auto l = hull.begin();
    auto r = hull.end();
    SegIt b = hull.end(); 8566
    T b_v;
    while (r - l > 2) {
        auto m = l + (r - l) / 2;
        T l_v = f(*l);
        T l_n_v = f(*(l + 1));
        T m_v = f(*m);
        T m_n_v = f(*(m + 1));
        if (b == hull.end() || l_v > b_v) { 3580
            b = l; // If max is at l we may remove it from
                    // the range.
            b_v = l_v;
        }
        if (l_n_v > l_v) {
            if (m_v < l_v) {
                r = m;
            } else {
                if (m_n_v > m_v) {
                    l = m + 1;
                } else {
                    r = m + 1;
                }
            }
        } else {
            if (m_v < l_v) {
                l = m + 1;
            } else {
                if (m_n_v > m_v) {
                    l = m + 1;
                } else {
                    r = m + 1;
                }
            }
        }
    }
    T l_v = f(*l);
    if (b == hull.end() || l_v > b_v) { 7715
        b = l;
        b_v = l_v;
    }
    if (r - l > 1) { 2147
        T l_n_v = f(*(l + 1));
        if (b == hull.end() || l_n_v > b_v) {
            b = l + 1;
            b_v = l_n_v;
        }
    }
    return b;
}
SegIt closest( %5939

```

```

Vec p) { // p can't be internal(can be on border),
    // hull must have atleast 3 points
Seg &ref_p = hull.front(); // O(log(n))
return max(function<double>(Seg &){(
    [&p, &ref_p](Seg &seg) { // accuracy of used type
        // should be coord^-2
        if (p == seg.F) return 10 - M_PI; 0134
        Vec v1 = {seg.S.F - seg.F.F, seg.S.S - seg.F.S};
        Vec v2 = {p.F - seg.F.F, p.S - seg.F.S};
        ll c_p = cross(v1, v2);
        if (c_p > 0) { // order the backside by angle
            Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
            Vec v2 = {seg.F.F - p.F, seg.F.S - p.S}; 5063
            ll d_p = dot(v1, v2);
            ll c_p = cross(v2, v1);
            return atan2(c_p, d_p) / 2;
        }
        ll d_p = dot(v1, v2);
        double res = atan2(d_p, c_p);
        if (d_p <= 0 && res > 0) res = -M_PI;
        if (res > 0) { 5050
            res += 20;
        } else {
            res = 10 - res;
        }
        return res;
    }));
} %5632
template <int DIRECTION> // 1 or -1
Vec tan_point(
    Vec p) { // can't be internal or on border
    // -1 iff CCW rotation of ray from p to res takes it
    // away from
    // polygon?
Seg &ref_p = hull.front(); // O(log(n))
auto best_seg = max(function<double>(Seg &){(
    [&p, &ref_p](Seg &seg) { // accuracy of used type
        // should be coord^-2
        Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
        Vec v2 = {seg.F.F - p.F, seg.F.S - p.S}; 0212
        ll d_p = dot(v1, v2);
        ll c_p = DIRECTION * cross(v2, v1);
        return atan2(c_p, d_p); // order by signed angle
    });
    return best_seg->F;
}) %5890
SegIt max_in_dir(
    Vec v) { // first is the ans. O(log(n))
return max(function<ll>(Seg &){(
    [&v](Seg &seg) { return dot(v, seg.F); }})); 5805
} pair<SegIt, SegIt> intersections(Seg l) { // O(log(n))
    int x = l.S.F - l.F.F;
    int y = l.S.S - l.F.S;
    Vec dir = {-y, x};
    auto it_max = max_in_dir(dir); 4740
    auto it_min = max_in_dir(MP(y, -x));
    ll opt_val = dot(dir, l.F);
    if (dot(dir, it_max->F) < opt_val ||
        dot(dir, it_min->F) > opt_val)
        return MP(hull.end(), hull.end()); 8921
    SegIt it_r1, it_r2;
    function<bool>(const Seg &, const Seg &) inc_c(
        [&dir](const Seg &lft, const Seg &rgt) {
            return dot(dir, lft.F) < dot(dir, rgt.F);
        });
    function<bool>(const Seg &, const Seg &) dec_c(
        [&dir](const Seg &lft, const Seg &rgt) {
            return dot(dir, lft.F) > dot(dir, rgt.F);
        });
    if (it_min <= it_max) { 1765
        it_r1 =
            upper_bound(it_min, it_max + 1, l, inc_c) - 1;
        if (dot(dir, hull.front().F) >= opt_val) {
            it_r2 = upper_bound( 8531
                hull.begin(), it_min + 1, l, dec_c) -
                1;
        } else {
            it_r2 =
                upper_bound(it_max, hull.end(), l, dec_c) - 1;
        }
    } else {
        it_r1 =
            upper_bound(it_max, it_min + 1, l, dec_c) - 1;
        if (dot(dir, hull.front().F) <= opt_val) { 1538
            it_r2 = upper_bound( 1538
                hull.begin(), it_max + 1, l, inc_c) -
                1;
        } else {
            it_r2 =
                upper_bound(it_min, hull.end(), l, inc_c) - 1;
        }
    }
    return MP(it_r1, it_r2); 2632
} %2632
Seg diameter() { // O(n)
    Seg res;
    ll dia_sq = 0;
    auto it1 = hull.begin();
    auto it2 = up_beg;
    Vec v1 = {hull.back().S.F - hull.back().F.F,
              hull.back().S.S - hull.back().F.S}; 2168
    while (it2 != hull.begin()) {
        Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F,
                  (it2 - 1)->S.S - (it2 - 1)->F.S};
        if (cross(v1, v2) > 0) break; 4807
        --it2;
    }
    while ( 7128
        it2 != hull.end()) { // check all antipodal pairs
        if (dist_sq(it1->F, it2->F) > dia_sq) {
            res = {it1->F, it2->F};
            dia_sq = dist_sq(res.F, res.S);
        }
    }
    Vec v1 = {
        it1->S.F - it1->F.F, it1->S.S - it1->F.S};
    Vec v2 = {
        it2->S.F - it2->F.F, it2->S.S - it2->F.S}; 9381
    if (cross(v1, v2) == 0) {
        if (dist_sq(it1->S, it2->F) > dia_sq) {
            res = {it1->S, it2->F};
            dia_sq = dist_sq(res.F, res.S);
        }
        if (dist_sq(it1->F, it2->S) > dia_sq) { 8171
            res = {it1->F, it2->S};
            dia_sq = dist_sq(res.F, res.S);
        }
    } // report cross pairs at parallel lines.
    ++it1;
    ++it2;
} else if (cross(v1, v2) < 0) {
    ++it1;
} else {
    ++it2;
}
return res;
} %1111
Delaunay triangulation O(nlogn)
const int max_co = (1 << 28) - 5;
struct Vec {
    int x, y;
    bool operator==(const Vec &oth) {
        return x == oth.x && y == oth.y;
    }
    bool operator!=(const Vec &oth) { 2500
        return !operator==(oth);
    }
    Vec operator-(const Vec &oth) {
        return {x - oth.x, y - oth.y};
    }
}; 11
cross(Vec a, Vec b) {
    return (ll)a.x * b.y - (ll)a.y * b.x; 8725
}
dot(Vec a, Vec b) {
    return (ll)a.x * b.x + (ll)a.y * b.y;
}
struct Edge {
    Vec tar;
    Edge *nxxt;
    Edge *inv = NULL;
    Edge *rep = NULL;
    bool vis = false;
};
struct Seg {
    Vec a, b;
    bool operator==(const Seg &oth) {
        return a == oth.a && b == oth.b;
    }
    bool operator!=(const Seg &oth) { 8977
        return !operator==(oth);
    }
}

```

```

    return !operator==(oth);
}

lli orient(Vec a, Vec b, Vec c) {
    return (ll)a.x * (b.y - c.y) + (ll)b.x * (c.y - a.y) +
        (ll)c.x * (a.y - b.y); 3775
} %3775
bool in_c_circle(Vec *arr, Vec d) {
    if (cross(arr[1] - arr[0], arr[2] - arr[0]) == 0)
        return true; // degenerate
    l1 m[3][3];
    ran(i, 0, 3) { 4264
        m[i][0] = arr[i].x - d.x;
        m[i][1] = arr[i].y - d.y;
        m[i][2] = m[i][0] * m[i][0];
        m[i][2] += m[i][1] * m[i][1];
    }
    __int128 res = 0; //double seems to work as well 5099
    res += (__int128)(m[0][0] * m[1][1] - m[0][1] * m[1][0]) *
        m[2][2];
    res += 6577
    (__int128)(m[1][0] * m[2][1] - m[1][1] * m[2][0]) *
        m[0][2];
    res -=
    (__int128)(m[0][0] * m[2][1] - m[0][1] * m[2][0]) *
        m[1][2];
    return res > 0; 1845
}
Edge *add_triangle(Edge *a, Edge *b, Edge *c) { %1845
    Edge *old[] = {a, b, c};
    Edge *tmp = new Edge[3];
    ran(i, 0, 3) {
        old[i]->rep = tmp + i; 8219
        tmp[i] = {
            old[i]->tar, tmp + (i + 1) % 3, old[i]->inv};
            if (tmp[i].inv) tmp[i].inv->inv = tmp + i;
        }
        return tmp;
    }
Edge *add_point( 0087
    Vec p, Edge *cur) { // returns outgoing edge
    Edge *triangle[] = {cur, cur->nxt, cur->nxt->nxt};
    ran(i, 0, 3) {
        if (orient(triangle[i]->tar,
            triangle[(i + 1) % 3]->tar, p) < 0)
            return NULL;
    }
    ran(i, 0, 3) {
        if (triangle[i]->rep) {
            Edge *res = add_point(p, triangle[i]->rep);
            if (res)
                return res; // unless we are on last layer we
                // must exit here
        }
    }
    Edge p_as_e{p}; 5490
}

Edge tmp{cur->tar};
tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
Edge *res = tmp.inv->nxt;
tmp.tar = cur->tar; 4005
tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
tmp.tar = cur->tar;
res->inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
res->inv->inv = res; 3259
return res;
}

Edge *delaunay(vector<Vec> &points) {
    random_shuffle(points.begin(), points.end());
    Vec arr[] = {{4 * max_co, 4 * max_co}, 5591
        {-4 * max_co, max_co}, {max_co, -4 * max_co}};
    Edge *res = new Edge[3];
    ran(i, 0, 3) res[i] = {arr[i], res + (i + 1) % 3};
    for (Vec &cur : points) {
        Edge *loc = add_point(cur, res);
        Edge *out = loc;
        arr[0] = cur;
        while (true) {
            arr[1] = out->tar;
            arr[2] = out->nxt->tar;
            Edge *e = out->nxt->inv;
            if (e && in_c_circle(arr, e->nxt->tar)) {
                Edge tmp{cur};
                tmp.inv = add_triangle(&tmp, out, e->nxt);
                tmp.tar = e->nxt->tar; 9851
                tmp.inv->inv =
                    add_triangle(&tmp, e->nxt->nxt, out->nxt->nxt);
                out = tmp.inv->nxt;
                continue;
            }
            out = out->nxt->nxt->inv;
            if (out->tar == loc->tar) break;
        }
        return res;
    }

    void extract_triangles(
        Edge *cur, vector<vector<Seg>> &res) {
        if (!cur->vis) {
            bool inc = true;
            Edge *it = cur;
            do {
                it->vis = true;
                if (it->rep) {
                    extract_triangles(it->rep, res);
                    inc = false;
                }
                it = it->nxt;
            } while (it != cur);
            if (inc) {
                Edge *triangle[3] = {cur, cur->nxt, cur->nxt->nxt};
                res.resize(res.size() + 1);
                vector<Seg> &tar = res.back();
                ran(i, 0, 3) { 1771
                    if ((abs(triangle[i]->tar.x) < max_co &&
                        abs(triangle[(i + 1) % 3]->tar.x) <
                        max_co))
                        tar.push_back({triangle[i]->tar,
                            triangle[(i + 1) % 3]->tar});
                }
                if (tar.empty()) res.pop_back();
            }
        }
    } %8602
}

Contest setup
alias g++='g++ -g -Wall -Wshadow -Wconversion \ #.bashrc
-fsanitize=undefined,address -DCDEBUG' #.bashrc
alias a='setxkbmap us -option' #.bashrc
alias m='setxkbmap us -option caps:escape' #.bashrc
alias ma='setxkbmap us -variant dvp \ #.bashrc
-option caps:escape' #.bashrc
gsettings set org.compiz.core: \ #settings
/org/compiz/profiles/Default/plugins/core/ hsize 4
gsettings set org.gnome.desktop.wm.preferences \ #settings
focus-mode 'sloppy'
gvim template.cpp
cd samps #copy everything
for d in *; do cd $d; for f in *; do \
    cp $f "../${d,,}${f,,}"; done; \
    cd ..; cp "../template.cpp" "../${d,,}.cpp"; done
cd ..
set si cin #.vimrc
set ts=4 sw=4 noet #.vimrc
set cb=unnamedplus #.vimrc
(global-set-key (kbd "C-x <next>") 'other-window) #.emacs
(global-set-key (kbd "C-x <prior>") \ #.emacs
    'previous-multiframe-window) #.emacs
(global-set-key (kbd "C-M-z") 'ansi-term) #.emacs
(global-linum-mode 1) #.emacs
(column-number-mode 1) #.emacs
(show-paren-mode 1) #.emacs
(setq-default indent-tabs-mode nil) #.emacs
valgrind --vgdb-error=0 ./a <inp & #valgrind
gdb a #valgrind
target remote | vgdb
crc.sh
#!/bin/env bash
for j in `seq $2 1 $3`; do #whitespace don't matter.
    sed '/^\s*$/{ $1 | head -$j | tr -d '[:space:]' \ #.emacs
        | cksum | cut -f1 -d ' ' | tail -c 5
done #there shouldn't be any COMMENTS.
#copy lines being checked to separate file.
# $ ./crc.sh tmp.cpp 999 999
# $ ./crc.sh tmp.cpp 1 333 | grep XXXX
gcc ordered set, hashtable #define DEBUG(...) cerr << __VA_ARGS__ << endl;
#define DEBUG(...) cerr << #ifndef CDEBUG
#define DEBUG(...) ((void)0); #endif
#define NDEBUG
6207
# undef DEBUG
# define DEBUG(...)((void)0);
# define NDEBUG
4737
∞

```

```

#endif
#define ran(i, a, b) for (auto i = (a); i < (b); i++)
#include <bits/stdc++.h>
typedef long long ll;
typedef long double ld;
using namespace std;
#pragma GCC optimize("Ofast") // better vectorization
#pragma GCC target("avx,avx2")
// double vectorized performance
#include <bits/extc++.h>
using namespace __gnu_pbds;
template <typename T, typename U>
using hashmap = gp_hash_table<T, U>;
// dumb, 3x faster than stl
template <typename T>
using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
int main() {
    ordered_set<int> cur;
    cur.insert(1);
    cur.insert(3);
    cout << cur.order_of_key(2) << endl;
    // the number of elements in the set less than 2
    cout << *cur.find_by_order(1) << endl;
    // the 1-st smallest number in the set(0-based)
    ordered_set<int> oth;
    oth.insert(5); // to join: cur < oth
    cur.join(oth); // cur = {1, 3, 5}, oth = {}
    cur.split(1, oth); // cur = {1}, oth = {3, 5}
    hashmap<int, int> h({}, {}, {}, {}, {1 << 16});
}

PRNGs and Hash functions
mt19937 gen;
uint64_t rand64() {
    return gen() ^ ((uint64_t)gen() << 32);
}
uint64_t rand64() {
    static uint64_t x = 1; //x != 0
    x ^= x >> 12;
    x ^= x << 25;
    x ^= x >> 27;
    return x * 0x2545f4914f6cdd1d; // can remove mult
} %6873
uint64_t mix(uint64_t x){ // deadbeef -> y allowed
variable uint64_t mem[2] = { x, 0xdeadbeeffeedbaedull };
asm volatile (
    "pxor %%xmm0, %%xmm0;" 4939
    "movdqa (%0), %%xmm1;" 
    "aesenc %%xmm0, %%xmm1;" 
    "movdqa %%xmm1, (%0);"
    :
    : "r" (&mem[0])
    : "memory"
);
    return mem[0]; // use both slots for 128 bit
} %7419
uint64_t mix64(uint64_t x) { //x != 0
    x = (x ^ (x >> 30)) * 0xbff58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    x = x ^ (x >> 31);
    return x;
} 8529
%8529
uint64_t unmix64(uint64_t x) {
    x = (x ^ (x >> 31) ^ (x >> 62)) * 0x319642b2d24d8ec3;
    x = (x ^ (x >> 27) ^ (x >> 54)) * 0x96de1b173f119089;
    x = x ^ (x >> 30) ^ (x >> 60);
    return x;
} 2960
%2960
uint64_t combine64(uint64_t x, uint64_t y) {
    if (y < x) swap(x, y); // remove for ord
    return mix64(mix64(x) + y);
} %4224
Memorypool
const int BLOCK = 8;
const int MEM_SIZE = 1 << 26;
char glob_buf[MEM_SIZE];
int glob_idx;
vector<bool> glob_used;
void init_mem() {
    glob_used.resize(MEM_SIZE / BLOCK);
    glob_used[0] = true;
}
template <typename T>
struct Ptr {
    unsigned idx;
    explicit Ptr(T *tar) { idx = (char *)tar - glob_buf; }
    Ptr() { idx = 0; }
    template <typename... Args>
    void construct(Args... args) {
        new (glob_buf + idx) T(args...);
    }
    T *operator->() {
        assert(idx);
        return (T*)(glob_buf + idx);
    }
    T &operator*() { return *operator->(); }
    bool operator==(const Ptr &oth) const {
        return idx == oth.idx;
    }
    operator unsigned() const { return idx; }
    Ptr &operator+=(int diff) {
        idx += diff * sizeof(T);
        return *this;
    }
    Ptr operator+(int diff) {
        Ptr res;
        res.idx = idx;
        return res += diff;
    }
    T &operator[](int diff) { return *operator+(diff); }
}; %1550
Radixsort 50M 64 bit integers as single array in 1 sec
template <typename T>
void deallocate(Ptr<T> ptr, int n) {
    ran(i, 0, ((int)sizeof(T) * n + BLOCK - 1) / BLOCK)
        glob_used[ptr.idx / BLOCK + i] = false;
}
template <typename T>
struct hash<Ptr<T>> {
    std::size_t operator()(const Ptr<T> &cur) const {
        return cur.idx;
    }
}; %3700
void rsort(T *a, T *b, int size, int d = sizeof(T) - 1) {
    int b_s[256]{};
    ran(i, 0, size) { ++b_s[(a[i] >> (d * 8)) & 255]; } 1466
    // ++b_s[*((uchar *)a + i) + d];
    T *mem[257];
    mem[0] = b;
    T *l_b = mem + 1;
    l_b[0] = b;
    ran(i, 0, 255) { l_b[i + 1] = l_b[i] + b_s[i]; }
    for (T *it = a; it != a + size; ++it) { 6813
        T id = ((*it) >> (d * 8)) & 255;
        *(l_b[id]++) = *it;
    }
    l_b = mem;
    if (d) {
        T *l_a[256];
        l_a[0] = a;
        ran(i, 0, 255) l_a[i + 1] = l_a[i] + b_s[i];
        ran(i, 0, 256) { 4380
            if (l_b[i + 1] - l_b[i] < 100) {
                sort(l_b[i], l_b[i + 1]);
                if (d & 1) copy(l_b[i], l_b[i + 1], l_a[i]);
            } else {
                rsort(l_b[i], l_a[i], b_s[i], d - 1);
            }
        }
    }
} %7759
} %3895

```

```

const int nmax = 5e7;
ll arr[nmax], tmp[nmax];
int main() {
    for (int i = 0; i < nmax; ++i)
        arr[i] = ((ll)rand() << 32) | rand();
    rsort(arr, tmp, nmax);
    assert(is_sorted(arr, arr + nmax));
}

FFT 10-15M length/sec
// integer c = ab is accurate if c_i < 2^49
#pragma GCC optimize ("Ofast") //10% performance
#include <complex.h>
extern "C" __complex__ double __muldc3(
    double a, double b, double c, double d){
    return a*c-b*d+I*(a*d+b*c); // 40% performance
}
#include <bits/stdc++.h>
typedef complex<double> Comp;
void fft_rec(Comp *arr, Comp *root_pow, int len) {
    if (len != 1) {
        fft_rec(arr, root_pow, len >> 1);
        fft_rec(arr + len, root_pow, len >> 1);
    }
    root_pow += len;
    ran(i, 0, len){
        tie(arr[i], arr[i + len]) = pair<Comp, Comp> {
            arr[i] + root_pow[i] * arr[i + len],
            arr[i] - root_pow[i] * arr[i + len] };
    }
}
void fft(vector<Comp> &arr, int ord, bool invert) {
    assert(arr.size() == 1 << ord);
    static vector<Comp> root_pow(1);
    static int inc_pow = 1;
    static bool is_inv = false;
    if (inc_pow <= ord) {
        int idx = root_pow.size();
        root_pow.resize(1 << ord);
        for (; inc_pow <= ord; ++inc_pow) {
            for (int idx_p = 0; idx_p < 1 << (ord - 1);
                 idx_p += 1 << (ord - inc_pow), ++idx) {
                root_pow[idx] = Comp {
                    cos(-idx_p * M_PI / (1 << (ord - 1))),
                    sin(-idx_p * M_PI / (1 << (ord - 1))) };
                if (is_inv) root_pow[idx] = conj(root_pow[idx]);
            }
        }
    }
    if (invert != is_inv) {
        is_inv = invert;
        for (Comp &cur : root_pow) cur = conj(cur);
    }
    int j = 0;
    ran(i, 1, (1<<ord)){
        int m = 1 << (ord - 1);
        bool cont = true;
        while (cont) {
            if (j >= len)
                break;
            if (arr[i] > arr[i + j])
                swap(arr[i], arr[i + j]);
            j += m;
        }
    }
}

```



```

Linear algebra
bitset<10> add(bitset<10> p, bitset<10> q) {
    return p ^ q;
}
bitset<10> mult(bitset<10> v, bool k) {
    if (k) {
        return v;
    } else {
        return bitset<10>(0);
    }
}
bitset<10> normalize(bitset<10> v, int idx) { return v; }
bitset<10> neg(bitset<10> v) { return v; }
template <typename T>
vector<T> add(vector<T> p, vector<T> q) {
    ran(i, 0, (int)p.size()) p[i] += q[i];
    return p;
}
template <typename T>
vector<T> mult(vector<T> p, T k) {
    ran(i, 0, (int)p.size()) p[i] *= k;
    return p;
}
template <typename T>
vector<T> normalize(vector<T> v, int idx) {
    return mult(v, (T)1 / v[idx]);
}
template <typename T>
vector<T> neg(vector<T> p) {
    return mult(p, (T)-1);
}
/* V is the class implementing a vector, T is the type
 * within. examples: <bitset<10>, bool>; <vector<double>,
 * double> etc. V must have an "add" operation defined */
template <typename V, typename T>
pair<vector<V>, pair<vector<int>, vector<int>>>
diagonalize(vector<V> matrix, int width) { 9747
    /* width is the number of columns we consider for
     * diagonalizing. all columns after that can be used
     * for things after equal sign etc */
    int cur_row = 0;
    vector<int> crap_columns;
    vector<int> diag_columns;
    ran(i, 0, width) {
        int row_id = -1;
        T best_val = 0; /* may want to replace with epsilon
                           if working over reals */
        ran(j, cur_row, (int)matrix.size()) {
            if (abs(matrix[j][i]) > abs(best_val)) {
                row_id = j;
                best_val = matrix[j][i];
            }
        }
        if (row_id == -1) {
            crap_columns.push_back(i);
        } else {
            diag_columns.push_back(i);
        }
    }
    swap(matrix[cur_row], matrix[row_id]);
    matrix[cur_row] = normalize(matrix[cur_row], i);
    ran(j, cur_row + 1, j < (int)matrix.size()) {
        if (matrix[j][i] != 0) { 6006
            matrix[j] = add(neg(normalize(matrix[j], i)),
                            matrix[cur_row]);
        }
    }
    cur_row++;
}
for (int i = (int)diag_columns.size() - 1; i >= 0; 1210
      --i) {
    for (int j = i - 1; j >= 0; --j) {
        matrix[j] = add(matrix[j],
                        neg(
                            mult(matrix[i], matrix[j][diag_columns[i]])));
    }
}
return {matrix, {diag_columns, crap_columns}}; %9471
template <typename V, typename T>
int matrix_rank(vector<V> matrix, int width) {
    return diagonalize<V, T>(matrix, width)
        .second.first.size(); 5622
}
template <typename V, typename T>
vector<T> one_solution(
    vector<V> matrix, int width, vector<T> y) {
    /* finds one solution to the system Ax = y.
     * each row in matrix must have width at least width
     * + 1. aborts if there is no solution (you can check
     * whether solution exists using matrix_rank) */
    assert(matrix.size() == y.size()); 8765
    ran(i, 0, (int)matrix.size()) matrix[i][width] = y[i];
    pair<vector<V>, pair<vector<int>, vector<int>>> prr =
        diagonalize<V, T>(matrix, width); 5091
    vector<V> diag = prr.first;
    vector<int> diag_cols = prr.second.first;
    vector<T> ans(width, 0);
    ran(i, 0, (int)matrix.size()) {
        if (i < (int)diag_cols.size()) {
            ans[diag_cols[i]] = diag[i][width];
        } else {
            assert(diag[i][width] == T(0));
            /* replace with epsilon if working over reals */
        }
    }
}
return ans; %4744
template <typename V, typename T>
vector<vector<T>> homog_basis(
    vector<V> matrix, int width) {
    /* finds the basis of the nullspace of matrix */
    pair<vector<V>, pair<vector<int>, vector<int>>> prr =
        diagonalize<V, T>(matrix, width); 7752
    vector<V> diag = prr.first;
    vector<int> diag_cols = prr.second.first;
    vector<vector<T>> ans;
    for (int u : crap_cols) {
        vector<T> row(width, 0);
        row[u] = 1;
        ran(i, 0, (int)diag_cols.size())
            row[diag_cols[i]] = -diag[i][u];
        ans.push_back(row);
    }
}
return ans; %5812
Polynomial roots and O(n^2) interpolation
struct Poly {
    vector<double> a;
    double operator()(double x) const {
        double val = 0;
        for (int i = (int)a.size(); i--;) (val *= x) += a[i];
        return val; 3663
    }
    void diff() {
        ran(i, 1, (int)a.size()) a[i - 1] = i * a[i];
        a.pop_back();
    }
    void divroot(double x0) {
        double b = a.back(), c;
        a.back() = 0;
        for (int i = (int)a.size() - 1; i--;) 5829
            c = a[i], a[i] = a[i + 1] * x0 + b, b = c;
        a.pop_back();
    }
}; %2226
/* Description: Finds the real roots to a polynomial.
 * Usage: poly_roots({{2,-3,1}},-1e9,1e9) // solve
 * x^2-3x+2 = 0 Time: O(n^2 \log(1/\epsilon)) */
vector<double> poly_roots(
    Poly p, double xmin, double xmax) {
    if (sz(p.a) == 2) return {-p.a[0] / p.a[1]};
    vector<double> ret; 4231
    Poly der = p;
    der.diff();
    auto dr = poly_roots(der, xmin, xmax);
    dr.push_back(xmin - 1);
    dr.push_back(xmax + 1);
    sort(dr.begin(), dr.end());
    ran(i, 0, (int)dr.size() - 1) {
        double l = dr[i], h = dr[i + 1];
        bool sign = p(l) > 0;
        if (sign ^ (p(h) > 0)) {
            ran(it, 0, 60) { // while (h - l > 1e-8)
                double m = (l + h) / 2, f = p(m);
                if ((f <= 0) ^ sign) {
                    l = m;
                } else {
                    h = m;
                }
            }
        }
    }
} 1929

```

```

    ret.push_back((l + h) / 2);
}
} %2596
/* Description: Given $n$ points $(x[i], y[i])$, computes
 * an $n-1$-degree polynomial $p$ that passes through them:
 * $p(x) = a[0]*x^0 + \dots + a[n-1]*x^{n-1}$. For
 * numerical precision, pick $x[k] = c*\cos(k/(n-1)*\pi)$,
 * $k=0 \dots n-1$. Time: $O(n^2)$ */
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
    vd res(n), temp(n);
    ran(k, 0, n - 1) ran(i, k + 1, n) y[i] =
        (y[i] - y[k]) / (x[i] - x[k]);
    double last = 0;
    temp[0] = 1;
    ran(k, 0, n) {
        ran(i, 0, n) {
            res[i] += y[k] * temp[i];
            swap(last, temp[i]);
            temp[i] -= last * x[k];
        }
    }
    return res;
} %2093
Simplex algorithm
/* Description: Solves a general linear maximization
 * problem: maximize $\mathbf{c}^T \mathbf{x}$ subject to $\mathbf{Ax} \leq \mathbf{b}$, $\mathbf{x} \geq 0$. Returns -inf if there is no solution, inf if
 * there are arbitrarily good solutions, or the maximum
 * value of $\mathbf{c}^T \mathbf{x}$ otherwise. The input vector is set to
 * an optimal $\mathbf{x}$ (or in the unbounded case, an arbitrary
 * solution fulfilling the constraints). Numerical
 * stability is not guaranteed. For better performance,
 * define variables such that $\mathbf{x} = 0$ is viable. Usage:
 * vvd A = {{1,-1}, {-1,1}, {-1,-2}};
 * vd b = {1,1,-4}, c = {-1,-1}, x;
 * T val = LPSolver(A, b, c).solve(x);
 * Time: $O(NM * \#pivots)$, where a pivot may be e.g. an
 * edge relaxation. $O(2^n)$ in the general case. Status:
 * seems to work? */
typedef double
T; // long double, Rational, double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;
const T eps = 1e-8, inf = 1 / .0;
#define MP make_pair
#define ltj(X) \
    if (s == -1 || MP(X[j], N[j]) < MP(X[s], N[s])) s = j
3913
struct LPSolver {
    int m, n;
    vi N, B;
    vvd D;
    LPSolver(const vvd& A, const vd& b, const vd& c)
        : m(sz(b)), n(sz(c)),
2418
    N(n + 1),
    B(m),
    D(m + 2, vd(n + 2)) {
        ran(i, 0, m) ran(j, 0, n) D[i][j] = A[i][j];
        ran(i, 0, m) {
            B[i] = n + i;
            D[i][n] = -1;
            D[i][n + 1] = b[i];
        }
        ran(j, 0, n) {
            N[j] = j;
            D[m][j] = -c[j];
        }
        N[n] = -1;
        D[m + 1][n] = 1;
    }
    void pivot(int r, int s) {
        T *a = D[r].data(), inv = 1 / a[s]; 1099
        ran(i, 0, m + 2) if (i != r && abs(D[i][s]) > eps) {
            T *b = D[i].data(), inv2 = b[s] * inv;
            ran(j, 0, n + 2) b[j] -= a[j] * inv2;
            b[s] = a[s] * inv2; 8058
        }
        ran(j, 0, n + 2) if (j != s) D[r][j] *= inv;
        ran(i, 0, m + 2) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }
    bool simplex(int phase) { 0674
        int x = m + phase - 1;
        for (;;) {
            int s = -1;
            ran(j, 0, n + 1) if (N[j] != -phase) ltj(D[x]);
            if (D[x][s] >= -eps) return true;
            int r = -1; 0227
            ran(i, 0, m) {
                if (D[i][s] <= eps) continue;
                if (r == -1 || MP(D[i][n + 1] / D[i][s], B[i]) <
                    MP(D[r][n + 1] / D[r][s], B[r])) 0657
                    r = i;
            }
            if (r == -1) return false;
            pivot(r, s); 2015
        }
        solve(vd& x) {
            int r = 0;
            ran(i, 1, m) if (D[i][n + 1] < D[r][n + 1]) r = i;
            if (D[r][n + 1] < -eps) {
                pivot(r, n);
                if (!simplex(2) || D[m + 1][n + 1] < -eps)
                    return -inf;
                ran(i, 0, m) if (B[i] == -1) {
                    int s = 0;
                    ran(j, 1, n + 1) ltj(D[i]);
                    pivot(i, s); 5590
                }
            }
        }
    }
    bool ok = simplex(1);
    x = vd(n);
    ran(i, 0, m) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return ok ? D[m][n + 1] : inf; 6872
} %3986
Dinic
struct MaxFlow {
    const static ll INF = 1e18;
    int source, sink;
    vector<int> start, now, lvl, adj, rcap, cap_loc, bfs;
    vector<int> cap, orig_cap; 2347
    ll sink_pot = 0;
    vector<bool> visited;
    vector<ll> cost;
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair<ll, int>> dist_que;
    void add_flow(int idx, ll flow, bool cont = true) {
        cap[idx] -= flow;
        if (cont) add_flow(rcap[idx], -flow, false); 7491
    }
    MaxFlow( 2758
        const vector<tuple<int, int, ll, ll, ll>> &edges) {
            for (auto &cur : edges) { //from, to, cap, rcap, cost
                start.resize(max(max(get<0>(cur), get<1>(cur)) + 2, 8990 6901
                    (int)start.size()));
                ++start[get<0>(cur) + 1];
                ++start[get<1>(cur) + 1];
            }
            ran(i, 1, (int)start.size()) start[i] += start[i - 1];
            now = start; 5067 7619
            adj.resize(start.back());
            cap.resize(start.back());
            rcap.resize(start.back());
            cost.resize(start.back());
            for (auto &cur : edges) {
                int u, v;
                ll c, rc, c_cost;
                tie(u, v, c, rc, c_cost) = cur;
                assert(u != v);
                adj[now[u]] = v;
                adj[now[v]] = u;
                rcap[now[u]] = now[v];
                rcap[now[v]] = now[u];
                cap_loc.push_back(now[u]);
                cost[now[u]] = c_cost;
                cost[now[v]] = -c_cost; 1155
                cap[now[u]++] = c;
                cap[now[v]++] = rc;
                orig_cap.push_back(c);
            }
        }
        bool dinic_bfs(int min_cap) { 1782
            lvl.clear();
            lvl.resize(start.size()); 1409
        }
    }
}

```

```

bfs.clear();
bfs.resize(1, source);
now = start;
lvl[source] = 1;
ran(i, 0, (int)bfs.size()) {
    int u = bfs[i];
    while (now[u] < start[u + 1]) {
        int v = adj[now[u]];
        if (cost[now[u]] == 0 &&
            cap[now[u]] >= min_cap && lvl[v] == 0) {
            lvl[v] = lvl[u] + 1;
            if(v==sink) return true;
            bfs.push_back(v);
        }
        ++now[u];
    }
    return false;
}
11 dinic_dfs(int u, ll flow, int min_cap) {
    if (u == sink) return flow;
    if (lvl[u] == lvl[sink]) return 0;
    ll res = 0;
    while (now[u] < start[u + 1]) {
        int v = adj[now[u]];
        if (lvl[v] == lvl[u] + 1 && cost[now[u]] == 0 &&
            cap[now[u]] >= min_cap) {
            ll cur = dinic_dfs(v, min(flow, (ll)cap[now[u]]),
                min_cap);
            if (cur) {
                add_flow(now[u], cur);
                flow -= cur;
                res += cur;
                if(!flow) break;
            }
        }
        ++now[u];
    }
    return res;
}
bool recalc_dist(bool check_imp = false) {
    now = start;
    visited.clear();
    visited.resize(start.size());
    dist_que.emplace(0, source);
    bool imp = false;
    while (!dist_que.empty()) {
        int u;
        ll dist;
        tie(dist, u) = dist_que.top();
        dist_que.pop();
        if (!visited[u]) {
            visited[u] = true;
            if (check_imp && dist != 0) imp = true;
            if (u == sink) sink_pot += dist;
            while (now[u] < start[u + 1]) {
                int v = adj[now[u]];
                if (!visited[v] && cap[now[u]]) {
                    dist_que.emplace(dist + cost[now[u]], v);
                    cost[now[u]] += dist;
                    cost[rCap[now[u]+]] -= dist;
                }
            }
            if (check_imp) return imp;
            return visited[sink];
        }
        // return whether there is a negative cycle
        bool recalc_dist_bellman_ford() {
            int i = 0;
            for (; i < (int)start.size() - 1 &&
                recalc_dist(true); ++i) {}
            return i == (int)start.size() - 1;
        }
        pair<ll, ll> calc_flow(int _source, int _sink) {
            source = _source;
            sink = _sink;
            assert(max(source, sink) < start.size() - 1);
            ll tot_flow = 0;
            ll tot_cost = 0;
            if (recalc_dist_bellman_ford()) {
                assert(false);
            } else {
                while (recalc_dist()){
                    ll flow = 0;
                    for(int min_cap = 1<<30; min_cap; min_cap >= 1){
                        while (dinic_bfs(min_cap)) {
                            now = start;
                            ll cur;
                            while (cur = dinic_dfs(source, INF, min_cap))
                                flow += cur;
                        }
                        tot_flow += flow;
                        tot_cost += sink_pot * flow;
                    }
                }
                return {tot_flow, tot_cost};
            }
        }
        11 flow_on_edge(int idx) {
            assert(idx < cap.size());
            return orig_cap[idx] - cap[cap_loc[idx]];
        }
    };
    const int nmax = 1055;
    int main() {
        int t;
        scanf("%d", &t);
        for (int i = 0; i < t; ++i) {
            vector<tuple<int, int, ll, ll, ll>> edges;
            int n;
            scanf("%d", &n);
            for (int j = 1; j <= n; ++j) {
                edges.emplace_back(j, 2 * n + 1, 1, 0, 0);
                7194
                4157
                0292
                9445
                6628
                3870
                6214
                7113
                0704
                0619
                8769
                9291
                6673
                8615
                9075
                %6688%9207%9075
                2399
                1146
                7076
                1736
                1746
                9547
                1565
                4534
                3762
                14
        }
    }
}

```

```

Edge* addEdge(
    Node* u, Node* v, int f, int c, int cost) {
    edges.push_back({u, v, f, c, cost});
    u->conn.push_back(&edges.back());
    v->conn.push_back(&edges.back());
    return &edges.back();
}

// Assumes all needed flow has already been added
int minCostMaxFlow() {
    int n = nodes.size();
    int result = 0;
    struct State {
        int p;
        Edge* used;
    };
    while (1) {
        vector<vector<State>> state(
            1, vector<State>(n, {0, 0}));
        for (int lev = 0; lev < n; lev++) {
            state.push_back(state[lev]);
            for (int i = 0; i < n; i++) {
                if (lev == 0 ||
                    state[lev][i].p < state[lev - 1][i].p) {
                    for (Edge* edge : nodes[i].conn) {
                        if (edge->getCap(&nodes[i]) > 0) {
                            int np =
                                state[lev][i].p + (edge->u == &nodes[i]
                                    ? edge->cost
                                    : -edge->cost);
                            int ni = edge->from(&nodes[i])->index;
                            if (np < state[lev + 1][ni].p) {
                                state[lev + 1][ni].p = np;
                                state[lev + 1][ni].used = edge;
                            }
                        }
                    }
                }
            }
        }
        // Now look at the last level
        bool valid = false;
        for (int i = 0; i < n; i++) {
            if (state[n - 1][i].p > state[n][i].p) {
                valid = true;
                vector<Edge*> path;
                int cap = 1000000000;
                Node* cur = &nodes[i];
                int clev = n;
                vector<bool> expr(n, false);
                while (!expr[cur->index]) {
                    expr[cur->index] = true;
                    State cstate = state[clev][cur->index];
                    cur = cstate.used->from(cur);
                    path.push_back(cstate.used);
                }
                reverse(path.begin(), path.end());
            }
        }
    }
}

```

```

int i = 0;
Node* cur2 = cur;
do {
    cur2 = path[i]->from(cur2);
    i++;
} while (cur2 != cur);
path.resize(i);
}
for (auto edge : path) {
    cap = min(cap, edge->getCap(cur));
    cur = edge->from(cur);
}
for (auto edge : path) {
    result += edge->addFlow(cur, cap);
    cur = edge->from(cur);
}
}
if (!valid) break;
}
return result;
}

Global Min Cut O(V^3)
<int, vi> GetMinCut(vector<vi>& weights) {
int N = sz(weights);
int used(N), cut, best_cut;
int best_weight = -1;
for (int phase = N - 1; phase >= 0; phase--) {
    vi w = weights[0], added = used;
    int prev, k = 0;
    rep(i, 0, phase) {
        prev = k;
        k = -1;
        rep(j, 1, N)
            if (!added[j] && (k == -1 || w[j] > w[k])) k = j;
        if (i == phase - 1) {
            rep(j, 0, N) weights[prev][j] += weights[k][j];
            rep(j, 0, N) weights[j][prev] = weights[prev][j];
            used[k] = true;
            cut.push_back(k);
            if (best_weight == -1 || w[k] < best_weight) {
                best_cut = cut;
                best_weight = w[k];
            }
        } else {
            rep(j, 0, N) w[j] += weights[k][j];
            added[k] = true;
        }
    }
    return {best_weight, best_cut};
}

Aho Corasick O(|alpha|*sum(len))
int alpha_size = 26;
struct Node {
    Node* nxt[alpha_size]; // May use other structures to
    // move in trie
}

```

```

Node *suffix;
Node() { memset(nxt, 0, alpha_size * sizeof(Node *)); }
int cnt = 0;
};

Node *aho_corasick(vector<vector<char>> &dict) {
    Node *root = new Node;
    root->suffix = 0;
    vector<pair<vector<char>, Node *>> state;
    for (vector<char> &s : dict)
        state.emplace_back(&s, root);
    for (int i = 0; !state.empty(); ++i) {
        vector<pair<vector<char>, Node *>> nstate;
        for (auto &cur : state) {
            Node *nxt = cur.second->nxt[(*cur.first)[i]];
            if (nxt) {
                cur.second = nxt;
            } else {
                nxt = new Node;
                cur.second->nxt[(*cur.first)[i]] = nxt;
                Node *suf = cur.second->suffix;
                cur.second = nxt;
                nxt->suffix = root; // set correct suffix link
                while (suf) {
                    if (suf->nxt[(*cur.first)[i]]) {
                        nxt->suffix = suf->nxt[(*cur.first)[i]];
                        break;
                    }
                    suf = suf->suffix;
                }
            }
            if (cur.first->size() > i + 1)
                nstate.push_back(cur);
        }
        state = nstate;
    }
    return root;
}

// auxilary functions for searching and counting
Node *walk(Node *cur,
           char c) { // longest prefix in dict that is suffix of
           // walked string.
    while (true) {
        if (cur->nxt[c]) return cur->nxt[c];
        if (!cur->suffix) return cur;
        cur = cur->suffix;
    }
}
void cnt_matches(Node *root, vector<char> &match_in) {
    Node *cur = root;
    for (char c : match_in) {
        cur = walk(cur, c);
        ++cur->cnt;
    }
}
void add_cnt(
    Node *root) { // After counting matches propagate ONCE
                  // to suffixes for final counts
}

```

```

vector<Node *> to_visit = {root};
ran(i, 0, to_visit.size()) {
    Node *cur = to_visit[i];
    ran(j, 0, alpha_size) {
        if (cur->nxt[j]) to_visit.push_back(cur->nxt[j]);
    }
}
for (int i = to_visit.size() - 1; i > 0; --i)
    to_visit[i]->suffix->cnt += to_visit[i]->cnt;
} %7950
int main() {
    int n, len;
    scanf("%d %d", &len, &n);
    vector<char> a(len + 1);
    scanf("%s", a.data());
    a.pop_back();
    for (char &c : a) c -= 'a';
    vector<vector<char>> dict(n);
    ran(i, 0, n) {
        scanf("%d", &len);
        dict[i].resize(len + 1);
        scanf("%s", dict[i].data());
        dict[i].pop_back();
        for (char &c : dict[i]) c -= 'a';
    }
    Node *root = aho_corasick(dict);
    cnt_matches(root, a);
    add_cnt(root);
    ran(i, 0, n) {
        Node *cur = root;
        for (char c : dict[i]) cur = walk(cur, c);
        printf("%d\n", cur->cnt);
    }
}
Suffix automaton and tree O((n+q)log(|alpha|)) - 10M length/s

struct Node;
typedef Ptr<Node> P;
struct Node {
    int act = 0;
    P out;
    int len; // Length of longest suffix in equivalence
    P suf; // class.
    char size = 0;
    char cap = 0;
    Node(int _len) : len(_len) {};
    Node(int &_act, P &_out, int &_len, P &_suf,
         int _size, int _cap) : act(_act), len(_len),
         suf(_suf), size(_size), cap(_cap) {
        out = alloc<P>(cap);
        ran(i, 0, size)
            out[i] = _out[i];
    }
    int has_nxt(char c) {
        return act & (1<<(c-'a'));
    }
    P nxt(char c) {
        5245
    }
}
return
    out[_builtin_popcount(act & ((1<<(c-'a'))-1))];
}
void set_nxt(char c, P nxt) {
    int idx = _builtin_popcount(act & ((1<<(c-'a'))-1));
    if(has_nxt(c)){
        out[idx] = nxt;
    } else{
        if(size == cap){
            cap *= 2;
            if(!size)
                cap = 2;
            Ptr<P> nout = alloc<P>(cap);
            ran(i, 0, idx)
                nout[i] = out[i];
            ran(i, idx, size)
                nout[i+1] = out[i];
            deallocate(out, size);
            out = nout;
        } else {
            for(int i=size; i>idx; --i)
                out[i] = out[i-1];
        }
        act |= (1<<(c-'a'));
        out[idx] = nxt;
        ++size;
    }
}
P split(int new_len) {
    return suf = alloc<Node>(1, act, out, new_len,
        suf, size, cap);
}
// Extra functions for matching and counting
P lower(int depth) {
    // move to longest suf of current with a maximum
    // length of depth.
    if (suf->len >= depth) return suf->lower(depth);
    return (P)this;
}
P walk(char c, int depth, int &match_len) {
    // move to longest suffix of walked path that is a
    // substring
    match_len = min(match_len, len);
    // includes depth limit(needed for finding matches)
    if (has_nxt(c)) { // as suffixes are in classes,
        // match_len must be tracked externally
        ++match_len;
        return nxt(c)->lower(depth);
    }
    if (suf) return suf->walk(c, depth, match_len);
    return (P)this;
}
bool vis = false;
bool vis_t = false;
int paths_to_end = 0;
void set_as_end() { // All suffixes of current node are
    paths_to_end += 1; // marked as ending nodes.
} %5014
if (suf) suf->set_as_end(); %0905
void calc_paths() {
    /* Call ONCE from ROOT. For each node calculates
     * number of ways to reach an end node. paths_to_end
     * is occurrence count for any strings in current
     * suffix equivalence class. */
    if (!vis) {
        vis = true;
        ran(i, 0, size){
            out[i]->calc_paths();
            paths_to_end += out[i]->paths_to_end;
        }
    }
    // Transform into suffix tree of reverse string
    P tree_links[26];
    int end_d_v = 1 << 30;
    int end_d() {
        if (end_d_v == 1 << 30) {
            ran(i, 0, size){
                end_d_v = min(end_d_v, 1 + out[i]->end_d());
            }
            if (end_d_v == 1 << 30)
                end_d_v = 0;
        }
        return end_d_v;
    }
    void build_suffix_tree(
        string &s) // Call ONCE from ROOT. %6985
    {
        if (!vis_t) {
            vis_t = true;
            if (suf)
                suf->tree_links[s[(int)s.size() - end_d() -
                    suf->len - 1]-'a'] = (P)this;
            ran(i, 0, size){
                out[i]->build_suffix_tree(s); %2958
            }
        }
    }
}
struct SufAuto {
    P last;
    P root;
    void extend(char new_c) {
        P nlast = alloc<Node>(1, last->len + 1);
        P swn = last;
        while (swn && !swn->has_nxt(new_c)) { %7641
            swn->set_nxt(new_c, nlast);
            swn = swn->suf;
        }
        if (!swn) {
            nlast->suf = root;
        } else {
            P max_sbstr = swn->nxt(new_c);
            if (swn->len + 1 == max_sbstr->len) {
                nlast->suf = max_sbstr;
            }
        }
    }
} %1138

```

<pre> } else { // remove for minimal DFA that matches // suffixes and crap P eq_sbstr = max_sbstr->split(swn->len + 1); nlast->suf = eq_sbstr; P x = swn; // x = with_edge_to_eq_sbstr while (x != 0 && x->nxt(new_c) == max_sbstr) { x->set_nxt(new_c, eq_sbstr); x = x->suf; } last = nlast; } SufAuto(string &s) { last = root = alloc<Node>(1, 0); for (char c : s) extend(c); // To build suffix tree use reversed string root->build_suffix_tree(s); } </pre>	<pre> return cnt[last] = cnt[link[last]] + 1; } </pre>	%3854	<pre> } pop_heap(con.begin(), con.end(), comp); auto nxt = con.back(); con.pop_back(); w = -nxt.first; if (nxt.second->tar->in_use) { super = nxt.second->tar->anc(); to_proc.resize(to_proc.size() + 1); } else { super = nxt.second->tar->con_to_root(); } if (super != root) { to_proc.back().cont.push_back(nxt.second); out_cands.emplace_back(to_proc.size() - 1, to_proc.back().cont.size() - 1); } else { // Clean circles nxt.second->inc = true; nxt.second->from->clean(); } } if (super != root) { // we are some loops non first Node. if (con.size() > super->con.size()) { swap(con, super->con); // Largest con in loop should not be copied. swap(w, super->w); } for (auto cur : con) { super->con.emplace_back(cur.first - super->w + w, cur.second); push_heap(super->con.begin(), super->con.end(), comp); } par = super; // root or anc() of first Node // encountered in a loop return super; } } Node *croot; vector<Node> graph; vector<Edge> edges; DMST(int n, vector<EdgeDesc> &desc, int r) { // Self loops and multiple edges are okay. graph.resize(n); croot = &graph[r]; for (auto &cur : desc) // Edges are reversed internally edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w}); for (int i = 0; i < desc.size(); ++i) graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]); for (int i = 0; i < n; ++i) make_heap(graph[i].con.begin(), graph[i].con.end(), comp); } </pre>	1711 6356 7094 0198 2509 0270 2636 5825 17
<pre> struct palindromic_tree { int len[MAXN], link[MAXN], cnt[MAXN]; char s[MAXN]; vector<pair<char, int>> to[MAXN]; int n, last, sz; void clear() { fill(to, to + MAXN, vector<pair<char, int>>()); memset(len, 0, sizeof(len)); memset(link, 0, sizeof(link)); memset(cnt, 0, sizeof(cnt)); memset(s, 0, sizeof(s)); n = last = 0; link[0] = 1; len[1] = -1; s[n++] = 27; sz = 2; } palindromic_tree() { clear(); } int get_link(int v) { while (s[n - len[v] - 2] != s[n - 1]) v = link[v]; return v; } int tr(int v, int c) { for (auto it : to[v]) if (it.first == c) return it.second; return 0; } int add_letter(int c) { s[n++] = c; int cur = get_link(last); if (!tr(cur, c)) { len[sz] = len[cur] + 2; link[sz] = tr(get_link(link[cur]), c); to[cur].push_back({c, sz++}); } last = tr(cur, c); } } </pre>	<pre> struct EdgeDesc { int from, to, w; }; struct DMST { struct Node; struct Edge { Node *from; Node *tar; int w; bool inc; }; struct Circle { bool vis = false; vector<Edge *> cont; void clean(int idx); }; const static greater<pair<ll, Edge *>> comp; static vector<Circle> to_proc; static bool no_dmst; static Node *root; // Can use inline static since C++17 struct Node { Node *par = NULL; vector<pair<int, int>> out_cands; // Circ, edge idx vector<pair<ll, Edge *>> con; bool in_use = false; ll w = 0; // extra to add to edges in con Node *anc() { if (!par) return this; while (par->par) par = par->par; return par; } void clean() { if (!no_dmst) { in_use = false; for (auto &cur : out_cands) to_proc[cur.first].clean(cur.second); } } Node *con_to_root() { if (anc() == root) return root; in_use = true; Node *super = this; // Will become root or the first Node encountered // in a loop. while (super == this) { while (!con.empty() && con.front().second->tar->anc() == anc()) pop_heap(con.begin(), con.end(), comp); con.pop_back(); } if (con.empty()) { no_dmst = true; return root; } } }; }; </pre>	9251	<pre> } 8693 0300 6779 1880 } } </pre>	8937 0300 6779 1880
<pre> } } } </pre>				


```

    return res;
}
};

bool edge::is_bridge() {
    return exists && (dest->seen > rev->dest->val ||
                      dest->val < rev->dest->seen);
} %3548%8614%4558
vert graph[nmax];
int main() { // Mechanics Practice BRIDGES
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; ++i) {
        int u, v;
        scanf("%d %d", &u, &v);
        graph[u].con.emplace_back(graph + v);
        graph[v].con.emplace_back(graph + u);
        graph[u].con.back().rev = &graph[v].con.back();
        graph[v].con.back().rev = &graph[u].con.back();
    }
    graph[1].dfs(1, NULL);
    int res = 0;
    for (int i = 1; i <= n; ++i)
        res += graph[i].cnt_adj_bridges();
    cout << res / 2 << endl;
}

2-Sat O(n) and SCC O(n)
struct Graph {
    int n;
    vector<vector<int> > con;
    Graph(int nsize) {
        n = nsize;
        con.resize(n);
    }
    void add_edge(int u, int v) { con[u].push_back(v); }
    void top_dfs(int pos, vector<int> &result,
                 vector<bool> &explr, vector<vector<int> > &revcon) {
        if (explr[pos]) return; 1224
        explr[pos] = true;
        for (auto next : revcon[pos])
            top_dfs(next, result, explr, revcon);
        result.push_back(pos);
    }
    vector<int> topsort() { 7420
        vector<vector<int> > revcon(n);
        ran(u, 0, n) {
            for (auto v : con[u]) revcon[v].push_back(u);
        }
        vector<int> result;
        vector<bool> explr(n, false);
        ran(i, 0, n) top_dfs(i, result, explr, revcon);
        reverse(result.begin(), result.end());
        return result;
    }
    void dfs( 5654
        int pos, vector<int> &result, vector<bool> &explr) {
        if (explr[pos]) return;
        explr[pos] = true;
        for (auto next : con[pos]) dfs(next, result, explr);
        result.push_back(pos);
    }
};

vector<vector<int> > scc() { 4882
    vector<int> order = topsort();
    reverse(order.begin(), order.end());
    vector<bool> explr(n, false);
    vector<vector<int> > res;
    for (auto it = order.rbegin(); it != order.rend();
         ++it) {
        vector<int> comp;
        top_dfs(*it, comp, explr, con); 5220
        sort(comp.begin(), comp.end());
        res.push_back(comp);
    }
    sort(res.begin(), res.end());
    return res;
};

int main() { 0503%6965%4511
    int n, m;
    cin >> n >> m;
    Graph g(2 * m);
    ran(i, 0, n) {
        int a, sa, b, sb;
        cin >> a >> sa >> b >> sb;
        a--, b--;
        g.add_edge(2 * a + 1 - sa, 2 * b + sb);
        g.add_edge(2 * b + 1 - sb, 2 * a + sa);
    }
    vector<int> state(2 * m, 0);
    {
        vector<int> order = g.topsort();
        vector<bool> explr(2 * m, false);
        for (auto u : order) {
            vector<int> traversed;
            g.dfs(u, traversed, explr);
            if (traversed.size() > 0 &&
                !state[traversed[0] ^ 1]) {
                for (auto c : traversed) state[c] = 1;
            }
        }
        ran(i, 0, m) {
            if (state[2 * i] == state[2 * i + 1]) {
                cout << "IMPOSSIBLE\n";
                return 0;
            }
        }
        ran(i, 0, m) cout << state[2 * i + 1] << '\n';
        return 0;
    }
};

Templated multi dimensional BIT O(log(n)^d) per query
// Fully overloaded any dimensional BIT, use any type for
// coordinates, elements, return_value. Includes
// coordinate compression.

```

```

template <class E_T, class C_T, C_T n_inf, class R_T>
struct BIT {
    vector<C_T> pos;
    vector<E_T> elems;
    bool act = false;
    BIT() { pos.push_back(n_inf); }
    void init() {
        if (act) {
            for (E_T &c_elem : elems) c_elem.init();
        } else {
            act = true;
            sort(pos.begin(), pos.end());
            pos.resize(
                unique(pos.begin(), pos.end()) - pos.begin());
            elems.resize(pos.size());
        }
    }
    template <typename... loc_form> 5478
    void update(C_T cx, loc_form... args) {
        if (act) {
            int x = lower_bound(pos.begin(), pos.end(), cx) -
                    pos.begin();
            for (; x < (int)pos.size(); x += x & -x)
                elems[x].update(args...);
        } else {
            pos.push_back(cx);
        }
    }
    template <typename... loc_form>
    R_T query( 9513
        C_T cx, loc_form... args) { // sum in (-inf, cx)
        R_T res = 0;
        int x = lower_bound(pos.begin(), pos.end(), cx) -
                pos.begin() - 1;
        for (; x > 0; x -= x & -x)
            res += elems[x].query(args...);
        return res;
    }
    template <typename I_T>
    struct wrapped {
        I_T a = 0;
        void update(I_T b) { a += b; }
        I_T query() { return a; }
        // Should never be called, needed for compilation
        void init() { DEBUG('i') }
        void update() { DEBUG('u') }
    };
    int main() { 0186
        // return type should be same as type inside wrapped
        BIT<BIT<wrapped<ll>, int, INT_MIN, ll>, int, INT_MIN,
        ll> fenwick;
        int dim = 2;
        vector<tuple<int, int, ll> > to_insert;
        to_insert.emplace_back(1, 1, 1);
        // set up all pos that are to be used for update
    }
}
```

```

for (int i = 0; i < dim; ++i) {
    for (auto &cur : to_insert)
        fenwick.update(get<0>(cur), get<1>(cur));
    // May include value which won't be used
    fenwick.init();
}
// actual use
for (auto &cur : to_insert)
    fenwick.update(
        get<0>(cur), get<1>(cur), get<2>(cur));
cout << fenwick.query(2, 2) << '\n';
}

Treap O(log (n)) per query
mt19937 randgen;
struct Treap {
    struct Node {
        int key;
        int value;
        unsigned int priority;
        ll total;
        Node* lch;
        Node* rch;
        Node(int new_key, int new_value) {
            key = new_key;
            value = new_value;
            priority = randgen();
            total = new_value;
            lch = 0;
            rch = 0;
        }
        void update() {
            total = value;
            if (lch) total += lch->total;
            if (rch) total += rch->total;
        }
        deque<Node> nodes;
        Node* root = 0;
        pair<Node*, Node*> split(int key, Node* cur) {
            if (cur == 0) return {0, 0};
            pair<Node*, Node*> result;
            if (key <= cur->key) {
                auto ret = split(key, cur->lch);
                cur->lch = ret.second;
                result = {ret.first, cur};
            } else {
                auto ret = split(key, cur->rch);
                cur->rch = ret.first;
                result = {cur, ret.second};
            }
            cur->update();
            return result;
        }
        Node* merge(Node* left, Node* right) {
            if (left == 0) return right;
            if (right == 0) return left;
            Node* top;
            if (left->priority < right->priority) {
                left->rch = merge(left->rch, right);
                top = left;
            } else {
                right->lch = merge(left, right->lch);
                top = right;
            }
            top->update();
            return top;
        }
        void insert(int key, int value) {
            nodes.push_back(Node(key, value));
            Node* cur = &nodes.back();
            pair<Node*, Node*> ret = split(key, root);
            cur = merge(ret.first, cur);
            cur = merge(cur, ret.second);
            root = cur;
        }
        void erase(int key) {
            Node* left, *mid, *right;
            tie(left, mid) = split(key, root);
            tie(mid, right) = split(key + 1, mid);
            root = merge(left, right);
        }
        ll sum_upto(int key, Node* cur) {
            if (cur == 0) return 0;
            if (key <= cur->key) {
                return sum_upto(key, cur->lch);
            } else {
                ll result = cur->value + sum_upto(key, cur->rch);
                if (cur->lch) result += cur->lch->total;
                return result;
            }
        }
        ll get(int l, int r) {
            return sum_upto(r + 1, root) - sum_upto(l, root);
        }
    };
    int main() {
        ios_base::sync_with_stdio(false);
        cin.tie(0);
        int m;
        Treap treap;
        cin >> m;
        for (int i = 0; i < m; i++) {
            int type;
            cin >> type;
            if (type == 1) {
                int x, y;
                cin >> x >> y;
                treap.insert(x, y);
            } else if (type == 2) {
                int x;
                cin >> x;
                treap.erase(x);
            } else {
                int l, r;
                cin >> l >> r;
                cout >> l >> r;
                cout << treap.get(l, r) << endl;
            }
        }
        return 0;
    }
};

Generic persistent compressed lazy segment tree
struct Seg {
    ll sum = 0;
    void recalc(const Seg &lhs_seg, int lhs_len,
                const Seg &rhs_seg, int rhs_len) {
        sum = lhs_seg.sum + rhs_seg.sum;
    }
} __attribute__((packed));
struct Lazy {
    ll add;
    ll assign_val; // LLONG_MIN if no assign;
    void init() {
        add = 0;
        assign_val = LLONG_MIN;
    }
    Lazy() { init(); }
    void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len) {
        lhs_lazy = *this;
        rhs_lazy = *this;
        init();
    }
    void merge(Lazy &oth, int len) {
        if (oth.assign_val != LLONG_MIN) {
            add = 0;
            assign_val = oth.assign_val;
        }
        add += oth.add;
    }
    void apply_to_seg(Seg &cur, int len) const {
        if (assign_val != LLONG_MIN) {
            cur.sum = len * assign_val;
        }
        cur.sum += len * add;
    }
} __attribute__((packed));
struct Node { // Following code should not need to be modified
    int ver;
    bool is_lazy = false;
    Seg seg;
    Lazy lazy;
    Node *lc = NULL, *rc = NULL;
    void init() {
        if (!lc) {
            lc = new Node{ver};
            rc = new Node{ver};
        }
    }
    Node *upd(
        int L, int R, int l, int r, Lazy &val, int tar_ver) {
        if (ver != tar_ver) {
            if (is_lazy) {
                if (l <= L && R <= r) {
                    seg.sum += val.sum;
                    seg.add += val.add;
                } else {
                    if (l <= L) {
                        lc = upd(lc, L, l, r, val, tar_ver);
                    }
                    if (R <= r) {
                        rc = upd(rc, R, l, r, val, tar_ver);
                    }
                }
            } else {
                if (l <= L && R <= r) {
                    seg.sum += val.sum;
                    seg.add += val.add;
                } else {
                    if (l <= L) {
                        lc = upd(lc, L, l, r, val, tar_ver);
                    }
                    if (R <= r) {
                        rc = upd(rc, R, l, r, val, tar_ver);
                    }
                }
            }
        }
        return this;
    }
} __attribute__((packed));

```

```

Node *rep = new Node(*this);
rep->ver = tar_ver;
return rep->upd(L, R, l, r, val, tar_ver);
}
if (L >= l && R <= r) {
    val.apply_to_seg(seg, R - L);
    lazy.merge(val, R - L);
    is_lazy = true;
} else {
    init();
    int M = (L + R) / 2;
    if (is_lazy) {
        Lazy l_val, r_val;
        lazy.split(l_val, r_val, R - L);
        lc = lc->upd(L, M, l, M, l_val, ver);
        rc = rc->upd(M, R, M, R, r_val, ver);
        is_lazy = false;
    }
    Lazy l_val, r_val;
    val.split(l_val, r_val, R - L);
    if (l < M) lc = lc->upd(L, M, l, r, l_val, ver);
    if (M < r) rc = rc->upd(M, R, l, r, r_val, ver);
    seg.recalc(lc->seg, M - L, rc->seg, R - M);
}
Lazy l_val, r_val;
val.split(l_val, r_val, R - L);
if (l < M) lc = lc->upd(L, M, l, r, l_val, ver);
if (M < r) rc = rc->upd(M, R, l, r, r_val, ver);
seg.recalc(lc->seg, M - L, rc->seg, R - M);
}
return this;
}

void get(int L, int R, int l, int r, Seg *&lft_res,
Seg *&tmp, bool last_ver) {
if (L >= l && R <= r) {
    tmp->recalc(*lft_res, L - l, seg, R - L);
    swap(lft_res, tmp);
}
else {
    init();
    int M = (L + R) / 2;
    if (is_lazy) {
        Lazy l_val, r_val;
        lazy.split(l_val, r_val, R - L);
        lc = lc->upd(L, M, l, M, l_val, ver + last_ver);
        lc->ver = ver;
        2185
        rc = rc->upd(M, R, M, R, r_val, ver + last_ver);
        rc->ver = ver;
        is_lazy = false;
    }
    if (l < M)
        lc->get(L, M, l, r, lft_res, tmp, last_ver);
    if (M < r)
        9755
        rc->get(M, R, l, r, lft_res, tmp, last_ver);
}
}
__attribute__((packed));
struct SegTree { // indexes start from 0, ranges are
    // [beg, end)
    vector<Node *> roots; // versions start from 0
    int len;
    SegTree(int _len) : len(_len) {
        roots.push_back(new Node{0});
    }
    8874
    int upd(
        int l, int r, Lazy &val, bool new_ver = false) {
        Node *cur_root = roots.back()->upd(
            0, len, l, r, val, roots.size() - !new_ver);
        if (cur_root != roots.back())
            roots.push_back(cur_root);
        return roots.size() - 1;
    }
    0020
    Seg get(int l, int r, int ver = -1) {
        if (ver == -1) ver = roots.size() - 1;
        Seg seg1, seg2;
        Seg *pres = &seg1, *ptmp = &seg2;
        roots[ver]->get(
            0, len, l, r, pres, ptmp, roots.size() - 1);
        return *pres;
    }
    9948];
    int main() {
        int n, m; // solves Mechanics Practice LAZY
        cin >> n >> m;
        SegTree seg_tree(1 << 17);
        for (int i = 0; i < n; ++i) {
            Lazy tmp;
            scanf("%lld", &tmp.assign_val);
            seg_tree.upd(i, i + 1, tmp);
        }
        for (int i = 0; i < m; ++i) {
            int o;
            int l, r;
            scanf("%d %d %d", &o, &l, &r);
            --l;
            if (o == 1) {
                Lazy tmp;
                scanf("%lld", &tmp.add);
                seg_tree.upd(l, r, tmp);
            } else if (o == 2) {
                Lazy tmp;
                scanf("%lld", &tmp.assign_val);
                seg_tree.upd(l, r, tmp);
            } else {
                Seg res = seg_tree.get(l, r);
                printf("%lld\n", res.sum());
            }
        }
    }
    8394
    Templatd HLD O(M(n) log n) per query
    class dummy {
    public:
        dummy() {}
        dummy(int, int) {}
        void set(int, int) {}
        int query(int left, int right) {
            cout << this << ' ' << left << ' ' << right << endl;
        }
    };
    /* T should be the type of the data stored in each
     * vertex; DS should be the underlying data structure
    */
}
* that is used to perform the group operation. It should
* have the following methods:
* * DS () - empty constructor
* * DS (int size, T initial) - constructs the structure
* with the given size, initially filled with initial.
* * void set (int index, T value) - set the value at
* index 'index' to 'value'
* * T query (int left, int right) - return the "sum" of
* elements between left and right, inclusive.
*/
template <typename T, class DS>
class HLD {
    int vertexc;
    vector<int> *adj;
    vector<int> subtree_size;
    DS structure;
    DS aux;
    void build_sizes(int vertex, int parent) {
        subtree_size[vertex] = 1;
        for (int child : adj[vertex]) {
            if (child != parent) {
                7279
                build_sizes(child, vertex);
                subtree_size[vertex] += subtree_size[child];
            }
        }
    }
    int cur;
    vector<int> ord;
    vector<int> chain_root;
    vector<int> par;
    void build_hld(
        int vertex, int parent, int chain_source) {
        cur++;
        ord[vertex] = cur;
        chain_root[vertex] = chain_source;
        par[vertex] = parent;
        if (adj[vertex].size() > 1 ||
            (vertex == 1 && adj[vertex].size() == 1)) {
            int big_child, big_size = -1;
            7661
            for (int child : adj[vertex]) {
                if ((child != parent) &&
                    (subtree_size[child] > big_size)) {
                    big_child = child;
                    big_size = subtree_size[child];
                }
            }
            build_hld(big_child, vertex, chain_source);
            for (int child : adj[vertex]) {
                if ((child != parent) && (child != big_child))
                    build_hld(child, vertex, child);
            }
        }
    }
    public:
        HLD(int _vertexc) {
            vertexc = _vertexc;
            adj = new vector<int>[vertexc + 5];
        }
}
7513
3893
0708
7393
8363
21

```

```

}

void add_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void build(T initial) {
    subtree_size = vector<int>(vertexc + 5);
    ord = vector<int>(vertexc + 5);
    chain_root = vector<int>(vertexc + 5);
    par = vector<int>(vertexc + 5);
    cur = 0;
    build_sizes(1, -1);
    build_hld(1, -1, 1);
    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();
}

8422    int queryc;
     cin >> queryc;
     for (int i = 0; i < queryc; i++) {
         int u, v;
         cin >> u >> v;
         hld.query_path(u, v);
         cout << endl;
     }

8170    }

8442    Splay Tree + Link-Cut O(NlogN)
8443    struct Tree *treev;
8444    struct Tree {
8445        struct T {
8446            int i;
8447            constexpr T() : i(-1) {}
8448            T(int _i) : i(_i) {}
8449            operator int() const { return i; }
8450            explicit operator bool() const { return i != -1; }
8451            Tree *operator->() { return treev + i; }
8452        };
8453        T c[2], p;
8454        /* insert monoid here */
8455        T link;
8456        Tree() {
8457            /* init monoid here */
8458            link = -1;
8459        }
8460        using T = Tree::T;
8461        constexpr T NIL;
8462        void update(T t) { /* recalculate the monoid here */ }
8463        void propagate(T t) {
8464            assert(t);
8465            for (T c : t->c)
8466                if (c) c->link = t->link;
8467                /* lazily propagate updates here */
8468        }
8469        void lazy_reverse(T t) { /* lazily reverse t here */ }
8470        T splay(T n) {
8471            for (;;) {
8472                propagate(n);
8473                T p = n->p;
8474                if (p == NIL) break;
8475                propagate(p);
8476                if (px = p->c[1] == n;
8477                    assert(p->c[px] == n);
8478                    T g = p->p;
8479                    if (g == NIL) { /* zig */
8480                        p->c[px] = n->c[px ^ 1];
8481                        p->c[px]->p = p;
8482                        n->c[px ^ 1] = p;
8483                        n->c[px ^ 1]->p = n;
8484                        n->p = NIL;
8485                        update(p);
8486                        update(n);
8487                    }
8488                }
8489            }
8490        }
8491    }
8492    break;
8493    propagate(g);
8494    if (gx = g->c[1] == p;
8495        assert(g->c[gx] == p);
8496        T gg = g->p;
8497        if (gg) assert(gg->c[ggx] == g);
8498        if (gx == px) { /* zig zig */
8499            g->c[gx] = p->c[gx ^ 1];
8500            g->c[gx]->p = g;
8501            p->c[gx ^ 1] = g;
8502            p->c[gx ^ 1]->p = p;
8503            p->c[gx] = n->c[gx ^ 1];
8504            p->c[gx]->p = p;
8505            n->c[gx ^ 1] = p;
8506            n->c[gx ^ 1]->p = n;
8507        } else { /* zig zag */
8508            g->c[gx] = n->c[gx ^ 1];
8509            g->c[gx]->p = g;
8510            n->c[gx ^ 1] = g;
8511            n->c[gx ^ 1]->p = n;
8512            p->c[gx ^ 1] = n->c[gx];
8513            p->c[gx ^ 1]->p = p;
8514            n->c[gx] = p;
8515            n->c[gx]->p = n;
8516        }
8517        if (gg) gg->c[ggx] = n;
8518        n->p = gg;
8519        update(g);
8520        update(p);
8521        update(n);
8522        if (gg) update(gg);
8523    }
8524    return n;
8525}
8526extreme(T t, int x) {
8527    while (t->c[x]) t = t->c[x];
8528    return t;
8529}
8530set_child(T t, int x, T a) {
8531    T o = t->c[x];
8532    t->c[x] = a;
8533    update(t);
8534    o->p = NIL;
8535    a->p = t;
8536    return o;
8537}
8538***** Link-Cut Tree: *****
8539expose(T t) {
8540    set_child(splay(t), 1, NIL);
8541    T leader = splay(extreme(t, 0));
8542    if (leader->link == NIL) return t;
8543    set_child(splay(leader), 0, expose(leader->link));
8544    return splay(t);
8545}
8546void link(T t, T p) {
8547}

```

```
assert(t->link == NIL);
t->link = p;
}
T cut(T t) {
T p = t->link;
if (p) expose(p);

    t->link = NIL;
    return p;
}

void make_root(T t) {
    expose(t);
    lazy_reverse(extreme(splay(t), 0));
}

%7295%6269
```


