

University of Tartu ICPC Team Notebook

(2018-2019) November 22, 2018

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

23 FFT 5M length/sec 20

24 Fast mod mult, Rabbin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$ 22

25 Symmetric Submodular Functions; Queyrannes's algorithm 23

1	1 Setup	
2	1 set smartindent cindent	
2	2 set ts=4 sw=4 expandtab	
2	3 syntax enable	
4	4 set clipboard=unnamedplus	
3	5 # setwkmap -option caps:escape	
3	6 # valgrind --vgdb-error=0 ./a <inp &	
3	7 # gdb a	
3	8 # target remote vgdb	
3	2 crc.sh	
4	1 #!/bin/envbash	
4	2 for j in `seq 10 10 200`; do	
7	3 sed '/~\s*\$/d' \$1 head -\$j tr -d '[:space:]' cksum cut -f1	
7	↪ -d ' ' tail -c 4 #whitespaces don't matter.	
8	4 done #there shouldn't be any COMMENTS.	
8	5 #copy lines being checked to separate file.	
9	6 # \$./crc.sh tmp.cpp	
10	3 gcc ordered set	
11	1 #include <bits/stdc++.h>	
11	2 typedef long long ll;	
13	3 using namespace std;	
13	4 #include <ext/pb_ds/assoc_container.hpp>	
14	5 #include <ext/pb_ds/tree_policy.hpp>	
14	6 using namespace __gnu_pbds;	
14	7 template <typename T>	
14	8 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,	
14	9 tree_order_statistics_node_update>;	
14	10 int main() {	#558
15	11 ordered_set<int> cur;	
15	12 cur.insert(1);	
15	13 cur.insert(3);	
17	14 cout << cur.order_of_key(2)	
17	15 << endl; // the number of elements in the set less than 2	
18	16 cout << *cur.find_by_order(0)	
18	17 << endl; // the 0-th smallest number in the set(0-based)	
19	18 cout << *cur.find_by_order(1)	
19	19 << endl; // the 1-th smallest number in the set(0-based)	
20	20 }	%574

4 2D geometry

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

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

Line through P and Q has $\bar{v} = \overline{PQ}$ and $c = \bar{v} \times P$.

$\text{side}_l(P) = \bar{v}_l \times P - c_l$ sign determines which side P is on from l .

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

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

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

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

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

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

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

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

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

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

Circumcircle center:

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

Circle-line intersect:

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

Circle-circle intersect:

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

```
if (d2 == 0) {assert(r1 != r2); return 0;} //
    // concentric circles
    double pd = (d2 + r1*r1 - r2*r2)/2; // = |O_1P
    | * d
    double h2 = r1*r1 - pd*pd/d2; // = h^2
    if (h2 >= 0) {
        pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2)
        ;
        out = {p-h, p+h};}
    return 1 + sgn(h2);
```

Tangent lines:

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

5 3D geometry

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

S above PQR iff > 0 .

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

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

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

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

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

Plane-plane intersection of has direction $\bar{n}_1 \times \bar{n}_2$ and goes

through $((d_1 \bar{n}_2 - d_2 \bar{n}_1) \times \bar{d}) / \|\bar{d}\|^2$.

Line-line distance:

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

Spherical to Cartesian:

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

Sphere-line intersection:

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

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

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

Spherical segment intersection:

```
bool properInter(p3 a, p3 b, p3 c, p3 d, p3 &out
    ) {
    p3 ab = a*b, cd = c*d; // normals of planes
    // OAB and OCD
    int oa = sgn(cd|a),
        ob = sgn(cd|b),
        oc = sgn(ab|c),
        od = sgn(ab|d);
    out = ab*cd*od; // four multiplications =>
    // careful with overflow !
    return (oa != ob && oc != od && oa != oc);
}
bool onSphSegment(p3 a, p3 b, p3 p) {
    p3 n = a*b;
    if (n == zero)
        return a*p == zero && (a|p) > 0;
    return (n|p) == 0 && (n|a*p) >= 0 && (n|b*p)
        <= 0;
}
struct directionSet : vector<p3> {
    using vector::vector; // import constructors
    void insert(p3 p) {
        for (p3 q : *this) if (p*q == zero) return;
        push_back(p);
    }
};
directionSet intersSph(p3 a, p3 b, p3 c, p3 d) {
    assert(validSegment(a, b) && validSegment(c, d
        ));
    p3 out;
    if (properInter(a, b, c, d, out)) return {out
        };
    directionSet s;
    if (onSphSegment(c, d, a)) s.insert(a);
    if (onSphSegment(c, d, b)) s.insert(b);
    if (onSphSegment(a, b, c)) s.insert(c);
    if (onSphSegment(a, b, d)) s.insert(d);
    return s;
}
```

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

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

Area of a spherical polygon:

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

6 Numerical integration with Simpson's rule

```

1 // computing power = how many times function integrate gets called
2 template <typename T>
3 double simps(T f, double a, double b) {
4     return (f(a) + 4 * f((a + b) / 2) + f(b)) * (b - a) / 6;
5 }
6 template <typename T>
7 double integrate(T f, double a, double b, double computing_power) {
8     double m = (a + b) / 2;
9     double l = simps(f, a, m), r = simps(f, m, b), tot = simps(f, a, b);
10    if (computing_power < 1) return tot;
11    return integrate(f, a, m, computing_power / 2) +           #567
12           integrate(f, m, b, computing_power / 2);
13 }                                                                %360

```

7 Triangle centers

```

1 const double min_delta = 1e-13;
2 const double coord_max = 1e6;
3 typedef complex<double> point;
4 point A, B, C; // vertices of the triangle
5 bool collinear() {
6     double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7     if (min_diff < coord_max * min_delta) return true;
8     point sp = (B - A) / (C - A);
9     double ang =
10         M_PI / 2 -                                     #638
11         abs(abs(arg(sp)) - M_PI / 2); // positive angle with the real line
12     return ang < min_delta;
13 }                                                                %446
14 point circum_center() {
15     if (collinear()) return point(NAN, NAN);
16     // squared lengths of sides
17     double a2, b2, c2;
18     a2 = norm(B - C);
19     b2 = norm(A - C);
20     c2 = norm(A - B);
21     // barycentric coordinates of the circumcenter
22     double c_A, c_B, c_C;
23     c_A = a2 * (b2 + c2 - a2); // sin(2 * alpha) may be used as well
24     c_B = b2 * (a2 + c2 - b2);
25     c_C = c2 * (a2 + b2 - c2);
26     double sum = c_A + c_B + c_C;
27     c_A /= sum;
28     c_B /= sum;
29     c_C /= sum;
30     // cartesian coordinates of the circumcenter
31     return c_A * A + c_B * B + c_C * C;
32 }                                                                #403
33 point centroid() { // center of mass
34     return (A + B + C) / 3.0;
35 }
36 point ortho_center() { // euler line

```

```

37     point O = circum_center();
38     return O + 3.0 * (centroid() - O);
39 };
40 point nine_point_circle_center() { // euler line
41     point O = circum_center();
42     return O + 1.5 * (centroid() - O);
43 };
44 point in_center() {
45     if (collinear()) return point(NAN, NAN);
46     double a, b, c; // side lengths
47     a = abs(B - C);
48     b = abs(A - C);
49     c = abs(A - B);
50     // trilinear coordinates are (1,1,1)
51     // barycentric coordinates
52     double c_A = a, c_B = b, c_C = c;
53     double sum = c_A + c_B + c_C;
54     c_A /= sum;
55     c_B /= sum;
56     c_C /= sum;
57     // cartesian coordinates of the incenter
58     return c_A * A + c_B * B + c_C * C;
59 }

```

#193
%031

#812

%980

8 Seg-Seg intersection, halfplane intersection area

```

1 struct Seg {
2     Vec a, b;
3     Vec d() { return b - a; }
4 };
5 Vec intersection(Seg l, Seg r) {
6     Vec dl = l.d(), dr = r.d();
7     if (cross(dl, dr) == 0) return {nanl(""), nanl("")};
8     double h = cross(dr, l.a - r.a) / len(dr);
9     double dh = cross(dr, dl) / len(dr);
10    return l.a + dl * (h / -dh);
11 }
12 // Returns the area bounded by halfplanes
13 double calc_area(vector<Seg> lines) {
14     double lb = -HUGE_VAL, ub = HUGE_VAL;
15     vector<Seg> linesBySide[2];
16     for (auto line : lines) {
17         if (line.b.y == line.a.y) {
18             if (line.a.x < line.b.x) {
19                 lb = max(lb, line.a.y);
20             } else {
21                 ub = min(ub, line.a.y);
22             }
23         } else if (line.a.y < line.b.y) {
24             linesBySide[1].push_back(line);
25         } else {
26             linesBySide[0].push_back({line.b, line.a});
27         }

```

#893

#029

```

28 }
29 sort(
30     linesBySide[0].begin(), linesBySide[0].end(), [](Seg l, Seg r) {
31         if (cross(l.d(), r.d()) == 0) #123
32             return normal(l.d()) * l.a > normal(r.d()) * r.a;
33         return cross(l.d(), r.d()) < 0;
34     });
35 sort(
36     linesBySide[1].begin(), linesBySide[1].end(), [](Seg l, Seg r) {
37         if (cross(l.d(), r.d()) == 0)
38             return normal(l.d()) * l.a < normal(r.d()) * r.a;
39         return cross(l.d(), r.d()) > 0;
40     });
41 // Now find the application area of the lines and clean up redundant
42 // ones
43 vector<double> applyStart[2]; #597
44 for (int side = 0; side < 2; side++) {
45     vector<double> &apply = applyStart[side];
46     vector<Seg> curLines;
47     for (auto line : linesBySide[side]) {
48         while (curLines.size() > 0) {
49             Seg other = curLines.back();
50             if (cross(line.d(), other.d()) != 0) {
51                 double start = intersection(line, other).y;
52                 if (start > apply.back()) break;
53             }
54             curLines.pop_back();
55             apply.pop_back();
56         }
57         if (curLines.size() == 0) {
58             apply.push_back(-HUGE_VAL);
59         } else {
60             apply.push_back(intersection(line, curLines.back()).y);
61         }
62         curLines.push_back(line);
63     }
64     linesBySide[side] = curLines;
65 }
66 applyStart[0].push_back(HUGE_VALL);
67 applyStart[1].push_back(HUGE_VALL);
68 double result = 0;
69 {
70     double lb = -HUGE_VALL, ub;
71     for (int i = 0, j = 0; i < (int)linesBySide[0].size() &&
72         j < (int)linesBySide[1].size();
73         lb = ub) {
74         ub = min(applyStart[0][i + 1], applyStart[1][j + 1]);
75         double alb = lb, aub = ub;
76         Seg l0 = linesBySide[0][i], l1 = linesBySide[1][j];
77         if (cross(l1.d(), l0.d()) > 0) {
78             alb = max(alb, intersection(l0, l1).y);
79         } else if (cross(l1.d(), l0.d()) < 0) {

```

```

80         aub = min(aub, intersection(l0, l1).y);
81     }
82     alb = max(alb, lb);
83     aub = min(aub, ub); #839
84     aub = max(aub, alb);
85     {
86         double x1 = l0.a.x + (alb - l0.a.y) / l0.d().y * l0.d().x;
87         double x2 = l0.a.x + (aub - l0.a.y) / l0.d().y * l0.d().x;
88         result -= (aub - alb) * (x1 + x2) / 2;
89     }
90     {
91         double x1 = l1.a.x + (alb - l1.a.y) / l1.d().y * l1.d().x;
92         double x2 = l1.a.x + (aub - l1.a.y) / l1.d().y * l1.d().x;
93         result += (aub - alb) * (x1 + x2) / 2; #717
94     }
95     if (applyStart[0][i + 1] < applyStart[1][j + 1]) {
96         i++;
97     } else {
98         j++;
99     }
100 }
101 }
102 return result;
103 } %103

```

9 Convex polygon algorithms

```

1 typedef pair<int, int> Vec;
2 typedef pair<Vec, Vec> Seg;
3 typedef vector<Seg>::iterator SegIt;
4 #define F first
5 #define S second
6 #define MP(x, y) make_pair(x, y)
7 ll dot(const Vec &v1, const Vec &v2) {
8     return (ll)v1.F * v2.F + (ll)v1.S * v2.S;
9 }
10 ll cross(const Vec &v1, const Vec &v2) {
11     return (ll)v1.F * v2.S - (ll)v2.F * v1.S;
12 }
13 ll dist_sq(const Vec &p1, const Vec &p2) {
14     return (ll)(p2.F - p1.F) * (p2.F - p1.F) +
15         (ll)(p2.S - p1.S) * (p2.S - p1.S);
16 } %623
17 struct Hull {
18     vector<Seg> hull;
19     SegIt upper_begin;
20     template <typename It>
21     void extend_hull(It begin, It end) { // O(n)
22         vector<Vec> res;
23         for (auto it = begin; it != end; ++it) {
24             if (res.empty() || *it != res.back()) {
25                 while (res.size() >= 2) {

```

```

26     Vec v1 = {res[res.size() - 1].F - res[res.size() - 2].F,
    ↪ #854
27     res[res.size() - 1].S - res[res.size() - 2].S};
28     Vec v2 = {it->F - res[res.size() - 2].F,
29     it->S - res[res.size() - 2].S};
30     if (cross(v1, v2) > 0) break;
31     res.pop_back();
32 }
33 res.push_back(*it);
34 }
35 }
36 for (int i = 0; i < res.size() - 1; ++i) #114
37     hull.emplace_back(res[i], res[i + 1]);
38 }
39 Hull(vector<Vec> &vert) { // atleast 2 distinct points
40     sort(vert.begin(), vert.end()); //  $O(n \log(n))$ 
41     extend_hull(vert.begin(), vert.end());
42     int diff = hull.size();
43     extend_hull(vert.rbegin(), vert.rend());
44     upper_begin = hull.begin() + diff;
45 } %039
46 bool contains(Vec p) { //  $O(\log(n))$ 
47     if (p < hull.front().F || p > upper_begin->F) return false;
48     {
49         auto it_low = lower_bound(
50             hull.begin(), upper_begin, MP(MP(p.F, (int)-2e9), MP(0, 0)));
51         if (it_low != hull.begin()) --it_low;
52         Vec v1 = {it_low->S.F - it_low->F.F, it_low->S.S - it_low->F.S};
53         Vec v2 = {p.F - it_low->F.F, p.S - it_low->F.S};
54         if (cross(v1, v2) < 0) // < 0 is inclusive, <= 0 is exclusive #287
55             return false;
56     }
57     {
58         auto it_up = lower_bound(hull.rbegin(),
59             hull.rbegin() + (hull.end() - upper_begin),
60             MP(MP(p.F, (int)2e9), MP(0, 0)));
61         if (it_up - hull.rbegin() == hull.end() - upper_begin) --it_up;
62         Vec v1 = {it_up->F.F - it_up->S.F, it_up->F.S - it_up->S.S};
63         Vec v2 = {p.F - it_up->S.F, p.S - it_up->S.S};
64         if (cross(v1, v2) > 0) // > 0 is inclusive, >= 0 is exclusive #906
65             return false;
66     }
67     return true;
68 } %673
69 // The function can have only one local min and max
70 // and may be constant only at min and max.
71 template <typename T>
72 SegIt max(function<T(const Seg &)> f) { //  $O(\log(n))$ 
73     auto l = hull.begin();
74     auto r = hull.end();
75     SegIt best = hull.end();

```

```

76     T best_val;
77     while (r - l > 2) {
78         auto mid = l + (r - l) / 2;
79         T l_val = f(*l);
80         T l_nxt_val = f(*(l + 1));
81         T mid_val = f(*mid);
82         T mid_nxt_val = f(*(mid + 1));
83         if (best == hull.end() ||
84             l_val > best_val) { // If max is at l we may remove it from
85                                 // the range.
86             best = l;
87             best_val = l_val;
88         }
89         if (l_nxt_val > l_val) {
90             if (mid_val < l_val) {
91                 r = mid; #397
92             } else {
93                 if (mid_nxt_val > mid_val) {
94                     l = mid + 1;
95                 } else {
96                     r = mid + 1;
97                 }
98             }
99         } else {
100             if (mid_val < l_val) {
101                 l = mid + 1; #634
102             } else {
103                 if (mid_nxt_val > mid_val) {
104                     l = mid + 1;
105                 } else {
106                     r = mid + 1;
107                 }
108             }
109         }
110     }
111     T l_val = f(*l);
112     if (best == hull.end() || l_val > best_val) { #470
113         best = l;
114         best_val = l_val;
115     }
116     if (r - l > 1) {
117         T l_nxt_val = f(*(l + 1));
118         if (best == hull.end() || l_nxt_val > best_val) {
119             best = l + 1;
120             best_val = l_nxt_val;
121         }
122     }
123     return best;
124 }
125 SegIt closest(Vec p) { // p can't be internal(can be on border),
126                        // hull must have atleast 3 points

```



```

127 const Seg &ref_p = hull.front(); // O(log(n))
128 return max(function<double(const Seg &)>(
129     [&p, &ref_p](
130         const Seg &seg) { // accuracy of used type should be coord^2
131             if (p == seg.F) return 10 - M_PI;
132             Vec v1 = {seg.S.F - seg.F.F, seg.S.S - seg.F.S};
133             Vec v2 = {p.F - seg.F.F, p.S - seg.F.S};
134             ll cross_prod = cross(v1, v2);
135             if (cross_prod > 0) { // order the backside by angle #083
136                 Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
137                 Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
138                 ll dot_prod = dot(v1, v2);
139                 ll cross_prod = cross(v2, v1);
140                 return atan2(cross_prod, dot_prod) / 2;
141             }
142             ll dot_prod = dot(v1, v2);
143             double res = atan2(dot_prod, cross_prod);
144             if (dot_prod <= 0 && res > 0) res = -M_PI;
145             if (res > 0) { #195
146                 res += 20;
147             } else {
148                 res = 10 - res;
149             }
150             return res;
151         }));
152     } #368
153 template <int DIRECTION> // 1 or -1
154 Vec tan_point(Vec p) { // can't be internal or on border
155     // -1 iff CCW rotation of ray from p to res takes it away from
156     // polygon?
157     const Seg &ref_p = hull.front(); // O(log(n))
158     auto best_seg = max(function<double(const Seg &)>(
159         [&p, &ref_p](
160             const Seg &seg) { // accuracy of used type should be coord^2
161                 Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
162                 Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
163                 ll dot_prod = dot(v1, v2);
164                 ll cross_prod = DIRECTION * cross(v2, v1); #867
165                 return atan2(cross_prod, dot_prod); // order by signed angle
166             }));
167     return best_seg->F; #101
168 }
169 SegIt max_in_dir(Vec v) { // first is the ans. O(log(n))
170     return max(function<ll(const Seg &)>(
171         [&v](const Seg &seg) { return dot(v, seg.F); }));
172 } #861
173 pair<SegIt, SegIt> intersections(Seg line) { // O(log(n))
174     int x = line.S.F - line.F.F;
175     int y = line.S.S - line.F.S;
176     Vec dir = {-y, x};
177     auto it_max = max_in_dir(dir);

```

```

178     auto it_min = max_in_dir(MP(y, -x));
179     ll opt_val = dot(dir, line.F);
180     if (dot(dir, it_max->F) < opt_val ||
181         dot(dir, it_min->F) > opt_val)
182         return MP(hull.end(), hull.end()); #292
183 SegIt it_r1, it_r2;
184 function<bool(const Seg &, const Seg &)> inc_comp(
185     [&dir](const Seg &lft, const Seg &rgt) {
186         return dot(dir, lft.F) < dot(dir, rgt.F);
187     });
188 function<bool(const Seg &, const Seg &)> dec_comp(
189     [&dir](const Seg &lft, const Seg &rgt) {
190         return dot(dir, lft.F) > dot(dir, rgt.F);
191     });
192 if (it_min <= it_max) { #402
193     it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
194     if (dot(dir, hull.front().F) >= opt_val) {
195         it_r2 =
196             upper_bound(hull.begin(), it_min + 1, line, dec_comp) - 1;
197     } else {
198         it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
199     }
200 } else {
201     it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
202     if (dot(dir, hull.front().F) <= opt_val) { #421
203         it_r2 =
204             upper_bound(hull.begin(), it_max + 1, line, inc_comp) - 1;
205     } else {
206         it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
207     }
208 }
209 return MP(it_r1, it_r2); #567
210 }
211 Seg diameter() { // O(n)
212     Seg res;
213     ll dia_sq = 0;
214     auto it1 = hull.begin();
215     auto it2 = upper_begin;
216     Vec v1 = {hull.back().S.F - hull.back().F.F,
217         hull.back().S.S - hull.back().F.S};
218     while (it2 != hull.begin()) {
219         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F,
220             (it2 - 1)->S.S - (it2 - 1)->F.S}; #386
221         ll decider = cross(v1, v2);
222         if (decider > 0) break;
223         --it2;
224     }
225     while (it2 != hull.end()) { // check all antipodal pairs
226         if (dist_sq(it1->F, it2->F) > dia_sq) {
227             res = {it1->F, it2->F};
228             dia_sq = dist_sq(res.F, res.S);

```

```

229     }
230     Vec v1 = {it1->S.F - it1->F.F, it1->S.S - it1->F.S};
231     Vec v2 = {it2->S.F - it2->F.F, it2->S.S - it2->F.S};
232     ll decider = cross(v1, v2);
233     if (decider == 0) { // report cross pairs at parallel lines.
234         if (dist_sq(it1->S, it2->F) > dia_sq) {
235             res = {it1->S, it2->F};
236             dia_sq = dist_sq(res.F, res.S);
237         }
238         if (dist_sq(it1->F, it2->S) > dia_sq) {
239             res = {it1->F, it2->S};
240             dia_sq = dist_sq(res.F, res.S);
241         }
242         ++it1;
243         ++it2;
244     } else if (decider < 0) {
245         ++it1;
246     } else {
247         ++it2;
248     }
249 }
250 return res;
251 }
252 };

```

#607
#980
#686
%781

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

```

1 const int alpha_size = 26;
2 struct node {
3     node *nxt[alpha_size]; // May use other structures to move in trie
4     node *suffix;
5     node() { memset(nxt, 0, alpha_size * sizeof(node *)); }
6     int cnt = 0;
7 };
8 node *aho_corasick(vector<vector<char>> &dict) {
9     node *root = new node;
10    root->suffix = 0;
11    vector<pair<vector<char> *, node *>> cur_state;
12    for (vector<char> &s : dict) cur_state.emplace_back(&s, root);
13    for (int i = 0; !cur_state.empty(); ++i) {
14        vector<pair<vector<char> *, node *>> nxt_state;
15        for (auto &cur : cur_state) {
16            node *nxt = cur.second->nxt[(*cur.first)[i]];
17            if (nxt) {
18                cur.second = nxt;
19            } else {
20                nxt = new node;
21                cur.second->nxt[(*cur.first)[i]] = nxt;
22                node *suf = cur.second->suffix;
23                cur.second = nxt;
24                nxt->suffix = root; // set correct suffix link
25                while (suf) {
26                    if (suf->nxt[(*cur.first)[i]]) {

```

#911
#003

```

27            nxt->suffix = suf->nxt[(*cur.first)[i]];
28            break;
29        }
30        suf = suf->suffix;
31    }
32    }
33    if (cur.first->size() > i + 1) nxt_state.push_back(cur);
34    }
35    cur_state = nxt_state;
36    }
37    return root;
38 }
39 // auxiliary functions for searching and counting
40 node *walk(node *cur,
41 char c) { // longest prefix in dict that is suffix of walked string.
42 while (true) {
43     if (cur->nxt[c]) return cur->nxt[c];
44     if (!cur->suffix) return cur;
45     cur = cur->suffix;
46 }
47 }
48 void cnt_matches(node *root, vector<char> &match_in) {
49     node *cur = root;
50     for (char c : match_in) {
51         cur = walk(cur, c);
52         ++cur->cnt;
53     }
54 }
55 void add_cnt(node *root) { // After counting matches propagate ONCE to
56 // suffixes for final counts
57 vector<node *> to_visit = {root};
58 for (int i = 0; i < to_visit.size(); ++i) {
59     node *cur = to_visit[i];
60     for (int j = 0; j < alpha_size; ++j) {
61         if (cur->nxt[j]) to_visit.push_back(cur->nxt[j]);
62     }
63 }
64 for (int i = to_visit.size() - 1; i > 0; --i)
65     to_visit[i]->suffix->cnt += to_visit[i]->cnt;
66 }
67 int main() { //
68     ↪ http://codeforces.com/group/s3etJR5zZK/contest/212916/problem/4
69     int n, len;
70     scanf("%d %d", &n, &len);
71     vector<char> a(len + 1);
72     scanf("%s", a.data());
73     a.pop_back();
74     for (char &c : a) c -= 'a';
75     vector<vector<char>> dict(n);
76     for (int i = 0; i < n; ++i) {
77         scanf("%d", &len);

```

#378
%064
%127
%286
#354
%313

```

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

```

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

```

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

```

#308

#890
%677

#091

```

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

```

%955

#035

%996

#188

#748


```

87 new_end->len = last->len + 1;
88 AutoNode *suf_w_nxt = last;
89 while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {
90     suf_w_nxt->set_nxt(new_c, new_end);
91     suf_w_nxt = suf_w_nxt->suf;
92 }
93 if (!suf_w_nxt) {
94     new_end->suf = root;
95 } else {
96     AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);
97     if (suf_w_nxt->len + 1 == max_sbstr->len) {
98         new_end->suf = max_sbstr;
99     } else {
100         AutoNode *eq_sbstr =
101             max_sbstr->split(suf_w_nxt->len + 1, new_c);
102         new_end->suf = eq_sbstr;
103         AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
104         while (w_edge_to_eq_sbstr != 0 &&
105             w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
106             w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
107             w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
108         }
109     }
110 }
111 last = new_end;
112 }
113 SufAutomaton(string &s) {
114     root = new AutoNode;
115     root->len = 0;
116     root->suf = NULL;
117     last = root;
118     for (char c : s) extend(c);
119     root->calc_end_dist(); // To build suffix tree use reversed string
120     root->build_suffix_tree(s);
121 }
122 };

```

#705

#169

#356
%628

%034

12 Dinic

```

1 struct MaxFlow {
2     typedef long long ll;
3     const ll INF = 1e18;
4     struct Edge {
5         int u, v;
6         ll c, rc;
7         shared_ptr<ll> flow;
8         Edge(int _u, int _v, ll _c, ll _rc = 0)
9             : u(_u), v(_v), c(_c), rc(_rc) {}
10    };
11    struct FlowTracker {
12        shared_ptr<ll> flow;
13        ll cap, rcap;
14        bool dir;

```

#787

```

15 FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int _dir)
16     : cap(_cap), rcap(_rcap), flow(_flow), dir(_dir) {}
17 ll rem() const {
18     if (dir == 0) {
19         return cap - *flow;
20     } else {
21         return rcap + *flow;
22     }
23 }
24 void add_flow(ll f) {
25     if (dir == 0)
26         *flow += f;
27     else
28         *flow -= f;
29     assert(*flow <= cap);
30     assert(-*flow <= rcap);
31 }
32 operator ll() const { return rem(); }
33 void operator--(ll x) { add_flow(x); }
34 void operator+=(ll x) { add_flow(-x); }
35 };
36 int source, sink;
37 vector<vector<int>> > adj;
38 vector<vector<FlowTracker>> > cap;
39 vector<Edge> edges;
40 MaxFlow(int _source, int _sink) : source(_source), sink(_sink) {
41     ↪ #080
42     assert(source != sink);
43 }
44 int add_edge(int u, int v, ll c, ll rc = 0) {
45     edges.push_back(Edge(u, v, c, rc));
46     return edges.size() - 1;
47 }
48 vector<int> now, lvl;
49 void prep() {
50     int max_id = max(source, sink);
51     for (auto &edge : edges) max_id = max(max_id, max(edge.u, edge.v));
52     ↪ #638
53     adj.resize(max_id + 1);
54     cap.resize(max_id + 1);
55     now.resize(max_id + 1);
56     lvl.resize(max_id + 1);
57     for (auto &edge : edges) {
58         auto flow = make_shared<ll>(0);
59         adj[edge.u].push_back(edge.v);
60         cap[edge.u].push_back(FlowTracker(edge.c, edge.rc, flow, 0));
61         if (edge.u != edge.v) {
62             adj[edge.v].push_back(edge.u);
63             cap[edge.v].push_back(FlowTracker(edge.c, edge.rc, flow, 1));
64         }
65     }
66     assert(cap[edge.u].back() == edge.c);

```

#844

#287

#789

```

64     edge.flow = flow;
65 }
66 }
67 bool dinic_bfs() {
68     fill(now.begin(), now.end(), 0);
69     fill(lvl.begin(), lvl.end(), 0);
70     lvl[source] = 1;
71     vector<int> bfs(1, source);
72     for (int i = 0; i < bfs.size(); ++i) {
73         int u = bfs[i];
74         for (int j = 0; j < adj[u].size(); ++j) {
75             int v = adj[u][j];
76             if (cap[u][j] > 0 && lvl[v] == 0) {
77                 lvl[v] = lvl[u] + 1;
78                 bfs.push_back(v);
79             }
80         }
81     }
82     return lvl[sink] > 0;
83 }
84 ll dinic_dfs(int u, ll flow) {
85     if (u == sink) return flow;
86     while (now[u] < adj[u].size()) {
87         int v = adj[u][now[u]];
88         if (lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0) {
89             ll res = dinic_dfs(v, min(flow, (ll)cap[u][now[u]]));
90             if (res > 0) {
91                 cap[u][now[u]] -= res;
92                 return res;
93             }
94         }
95         ++now[u];
96     }
97     return 0;
98 }
99 ll calc_max_flow() {
100     prep();
101     ll ans = 0;
102     while (dinic_bfs()) {
103         ll cur = 0;
104         do {
105             cur = dinic_dfs(source, INF);
106             ans += cur;
107         } while (cur > 0);
108     }
109     return ans;
110 }
111 ll flow_on_edge(int edge_index) {
112     assert(edge_index < edges.size());
113     return *edges[edge_index].flow;
114 }
115 };

```

#448

#722

#459

#054

#346

%583

```

116 int main() {
117     int n, m;
118     cin >> n >> m;
119     auto mf = MaxFlow(
120         1, n); // arguments source and sink, memory usage O(largest node
121               // index + input size), sink doesn't need to be last index
122     int edge_index;
123     for (int i = 0; i < m; ++i) {
124         int a, b, c;
125         cin >> a >> b >> c;
126         // mf.add_edge(a,b,c); // for directed edges
127         edge_index = mf.add_edge(
128             a, b, c, c); // store edge index if care about flow value
129     }
130     cout << mf.calc_max_flow() << '\n';
131     // cout << mf.flow_on_edge(edge_index) << endl; // return flow on
132     // this edge
133 }

```

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

```

1 const int nmax = 1055;
2 const ll inf = 1e14;
3 int t, n, v; // 0 is source, v-1 sink
4 ll rem_flow[nmax][nmax];
5 // set [x][y] for directed capacity from x to y.
6 ll cost[nmax][nmax]; // set [x][y] for directed cost from x to y. SET
7                       // TO inf IF NOT USED
8 ll min_dist[nmax];
9 int prev_node[nmax];
10 ll node_flow[nmax];
11 bool visited[nmax];
12 ll tot_cost, tot_flow; // output
13 void min_cost_max_flow() {
14     tot_cost = 0; // Does not work with negative cycles.
15     tot_flow = 0;
16     ll sink_pot = 0;
17     min_dist[0] = 0;
18     for (int i = 1; i <= v; ++i) { // incase of no negative edges
19                                     // Bellman-Ford can be removed.
20         min_dist[i] = inf;
21     }
22     for (int i = 0; i < v - 1; ++i) {
23         for (int j = 0; j < v; ++j) {
24             for (int k = 0; k < v; ++k) {
25                 if (rem_flow[j][k] > 0 &&
26                     min_dist[j] + cost[j][k] < min_dist[k])
27                     min_dist[k] = min_dist[j] + cost[j][k];
28             }
29         }
30     }
31     for (int i = 0; i < v; ++i) { // Apply potentials to edge costs.

```

%576

%927

#040

```

32     for (int j = 0; j < v; ++j) {
33         if (cost[i][j] != inf) {
34             cost[i][j] += min_dist[i];
35             cost[i][j] -= min_dist[j];
36         }
37     }
38 }
39 sink_pot += min_dist[v - 1]; // Bellman-Ford end.
40 while (true) {
41     for (int i = 0; i <= v; ++i) { // node after sink is used as start
42                                     // value for Dijkstra.
43         min_dist[i] = inf;
44         visited[i] = false;
45     }
46     min_dist[0] = 0;
47     node_flow[0] = inf;
48     int min_node;
49     while (true) { // Use Dijkstra to calculate potentials
50         int min_node = v;
51         for (int i = 0; i < v; ++i) {
52             if ((!visited[i]) && min_dist[i] < min_dist[min_node])
53                 min_node = i;
54         }
55         if (min_node == v) break;
56         visited[min_node] = true;
57         for (int i = 0; i < v; ++i) {
58             if ((!visited[i]) &&
59                 min_dist[min_node] + cost[min_node][i] < min_dist[i]) {
60                 min_dist[i] = min_dist[min_node] + cost[min_node][i];
61                 prev_node[i] = min_node;
62                 node_flow[i] =
63                     min(node_flow[min_node], rem_flow[min_node][i]);
64             }
65         }
66     }
67     if (min_dist[v - 1] == inf)
68         break;
69     for (int i = 0; i < v; ++i) { // Apply potentials to edge costs.
70         for (int j = 0; j < v; ++j) { // Found path from source to sink becomes 0
71                                         // cost.
72             if (cost[i][j] != inf) {
73                 cost[i][j] += min_dist[i];
74                 cost[i][j] -= min_dist[j];
75             }
76         }
77     }
78     sink_pot += min_dist[v - 1];
79     tot_flow += node_flow[v - 1];
80     tot_cost += sink_pot * node_flow[v - 1];
81     int cur = v - 1;
82     while (cur != 0) {
83         // Backtrack along found path that now has 0 cost.

```

#630
%849

#782

#881

#664

#946

```

83     rem_flow[prev_node[cur]][cur] -= node_flow[v - 1];
84     rem_flow[cur][prev_node[cur]] += node_flow[v - 1];
85     cost[cur][prev_node[cur]] = 0;
86     if (rem_flow[prev_node[cur]][cur] == 0)
87         cost[prev_node[cur]][cur] = inf;
88     cur = prev_node[cur];
89 }
90 }
91 }
92 int main() { // http://www.spoj.com/problems/GREED/
93     cin >> t;
94     for (int i = 0; i < t; ++i) {
95         cin >> n;
96         for (int j = 0; j < nmax; ++j) {
97             for (int k = 0; k < nmax; ++k) {
98                 cost[j][k] = inf;
99                 rem_flow[j][k] = 0;
100             }
101         }
102         for (int j = 1; j <= n; ++j) {
103             cost[j][2 * n + 1] = 0;
104             rem_flow[j][2 * n + 1] = 1;
105         }
106         for (int j = 1; j <= n; ++j) {
107             int card;
108             cin >> card;
109             ++rem_flow[0][card];
110             cost[0][card] = 0;
111         }
112         int ex_c;
113         cin >> ex_c;
114         for (int j = 0; j < ex_c; ++j) {
115             int a, b;
116             cin >> a >> b;
117             if (b < a) swap(a, b);
118             cost[a][b] = 1;
119             rem_flow[a][b] = nmax;
120             cost[b][n + b] = 0;
121             rem_flow[b][n + b] = nmax;
122             cost[n + b][a] = 1;
123             rem_flow[n + b][a] = nmax;
124         }
125         v = 2 * n + 2;
126         min_cost_max_flow();
127         cout << tot_cost << '\n';
128     }
129 }

```

%803

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

```

1 struct Network {
2     struct Node;

```

```

3  struct Edge {
4      Node *u, *v;
5      int f, c, cost;
6      Node* from(Node* pos) {
7          if (pos == u) return v;
8          return u;
9      }
10     int getCap(Node* pos) {
11         if (pos == u) return c - f;
12         return f;
13     }
14     int addFlow(Node* pos, int toAdd) {
15         if (pos == u) {
16             f += toAdd;
17             return toAdd * cost;
18         } else {
19             f -= toAdd;
20             return -toAdd * cost;
21         }
22     }
23 };
24 struct Node {
25     vector<Edge*> conn;
26     int index;
27 };
28 deque<Node> nodes;
29 deque<Edge> edges;
30 Node* addNode() {
31     nodes.push_back(Node());
32     nodes.back().index = nodes.size() - 1;
33     return &nodes.back();
34 }
35 Edge* addEdge(Node* u, Node* v, int f, int c, int cost) {
36     edges.push_back({u, v, f, c, cost});
37     u->conn.push_back(&edges.back());
38     v->conn.push_back(&edges.back());
39     return &edges.back();
40 }
41 // Assumes all needed flow has already been added
42 int minCostMaxFlow() {
43     int n = nodes.size();
44     int result = 0;
45     struct State {
46         int p;
47         Edge* used;
48     };
49     while (1) {
50         vector<vector<State>> state(1, vector<State>(n, {0, 0}));
51         for (int lev = 0; lev < n; lev++) {
52             state.push_back(state[lev]);
53             for (int i = 0; i < n; i++) {
54                 if (lev == 0 || state[lev][i].p < state[lev - 1][i].p) {

```

#145

#987

#057

#692

#158

```

55         for (Edge* edge : nodes[i].conn) {
56             if (edge->getCap(&nodes[i]) > 0) {
57                 int np =
58                     state[lev][i].p +
59                     (edge->u == &nodes[i] ? edge->cost : -edge->cost);
60                 int ni = edge->from(&nodes[i])->index;
61                 if (np < state[lev + 1][ni].p) {
62                     state[lev + 1][ni].p = np;
63                     state[lev + 1][ni].used = edge;
64                 }
65             }
66         }
67     }
68 }
69 }
70 // Now look at the last level
71 bool valid = false;
72 for (int i = 0; i < n; i++)
73     if (state[n - 1][i].p > state[n][i].p) {
74         valid = true;
75         vector<Edge*> path;
76         int cap = 1000000000;
77         Node* cur = &nodes[i];
78         int clev = n;
79         vector<bool> explr(n, false);
80         while (!explr[cur->index]) {
81             explr[cur->index] = true;
82             State cstate = state[clev][cur->index];
83             cur = cstate.used->from(cur);
84             path.push_back(cstate.used);
85         }
86         reverse(path.begin(), path.end());
87         {
88             int i = 0;
89             Node* cur2 = cur;
90             do {
91                 cur2 = path[i]->from(cur2);
92                 i++;
93             } while (cur2 != cur);
94             path.resize(i);
95         }
96         for (auto edge : path) {
97             cap = min(cap, edge->getCap(cur));
98             cur = edge->from(cur);
99         }
100         for (auto edge : path) {
101             result += edge->addFlow(cur, cap);
102             cur = edge->from(cur);
103         }
104     }
105     if (!valid) break;

```

#281

#283

#954

#990

#599

```

106     }
107     return result;
108 }
109 };

```

%900

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

```

1 struct EdgeDesc {
2     int from, to, w;
3 };
4 struct DMST {
5     struct Node;
6     struct Edge {
7         Node *from;
8         Node *tar;
9         int w;
10        bool inc;
11    };
12    struct Circle {
13        bool vis = false;
14        vector<Edge *> contents;
15        void clean(int idx);
16    };
17    const static greater<pair<ll, Edge *> >
18        comp; // Can use inline static since C++17
19    static vector<Circle> to_process;
20    static bool no_dmst;
21    static Node *root;
22    struct Node {
23        Node *par = NULL;
24        vector<pair<int, int> > out_cands; // Circ, edge idx
25        vector<pair<ll, Edge *> > con;
26        bool in_use = false;
27        ll w = 0; // extra to add to edges in con
28        Node *anc() {
29            if (!par) return this;
30            while (par->par) par = par->par;
31            return par;
32        }
33        void clean() {
34            if (!no_dmst) {
35                in_use = false;
36                for (auto &cur : out_cands)
37                    to_process[cur.first].clean(cur.second);
38            }
39        }
40        Node *con_to_root() {
41            if (anc() == root) return root;
42            in_use = true;
43            Node *super = this; // Will become root or the first Node
44                                // encountered in a loop.
45            while (super == this) {
46                while (

```

#186

#478

#721

#488

```

47         !con.empty() && con.front().second->tar->anc() == anc()) {
48             pop_heap(con.begin(), con.end(), comp);
49             con.pop_back();
50         }
51         if (con.empty()) {
52             no_dmst = true;
53             return root;
54         }
55         pop_heap(con.begin(), con.end(), comp);
56         auto nxt = con.back();
57         con.pop_back();
58         w = -nxt.first;
59         if (nxt.second->tar
60             ->in_use) { // anc() wouldn't change anything
61             super = nxt.second->tar->anc();
62             to_process.resize(to_process.size() + 1);
63         } else {
64             super = nxt.second->tar->con_to_root();
65         }
66         if (super != root) {
67             to_process.back().contents.push_back(nxt.second);
68             out_cands.emplace_back(to_process.size() - 1,
69                                   to_process.back().contents.size() - 1);
70         } else { // Clean circles
71             nxt.second->inc = true;
72             nxt.second->from->clean();
73         }
74     }
75     if (super != root) { // we are some loops non first Node.
76         if (con.size() > super->con.size()) {
77             swap(con,
78                  super->con); // Largest con in loop should not be copied.
79             swap(w, super->w);
80         }
81         for (auto cur : con) {
82             super->con.emplace_back(
83                 cur.first - super->w + w, cur.second);
84             push_heap(super->con.begin(), super->con.end(), comp);
85         }
86     }
87     par = super; // root or anc() of first Node encountered in a
88                 // loop
89     return super;
90 }
91 };
92 Node *cur_root;
93 vector<Node> graph;
94 vector<Edge> edges;
95 DMST(int n, vector<EdgeDesc> &desc,
96      int r) { // Self loops and multiple edges are okay.
97     graph.resize(n);

```

#506

#174

#848

#064

#995


```

98     cur_root = &graph[r];
99     for (auto &cur : desc) // Edges are reversed internally
100         edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
101     for (int i = 0; i < desc.size(); ++i)
102         graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]); #895
103     for (int i = 0; i < n; ++i)
104         make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
105 }
106 bool find() {
107     root = cur_root;
108     no_dmst = false;
109     for (auto &cur : graph) {
110         cur.con_to_root();
111         to_process.clear();
112         if (no_dmst) return false; #405
113     }
114     return true;
115 }
116 ll weight() { #732
117     ll res = 0;
118     for (auto &cur : edges) {
119         if (cur.inc) res += cur.w;
120     }
121     return res;
122 } #477
123 };
124 void DMST::Circle::clean(int idx) {
125     if (!vis) {
126         vis = true;
127         for (int i = 0; i < contents.size(); ++i) {
128             if (i != idx) {
129                 contents[i]->inc = true;
130                 contents[i]->from->clean();
131             }
132         }
133     } #711
134 }
135 const greater<pair<ll, DMST::Edge *> > DMST::comp;
136 vector<DMST::Circle> DMST::to_process;
137 bool DMST::no_dmst;
138 DMST::Node *DMST::root; #771

```

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

```

1 struct vert;
2 struct edge {
3     bool exists = true;
4     vert *dest;
5     edge *rev;
6     edge(vert *_dest) : dest(_dest) { rev = NULL; }
7     vert &operator*() { return *dest; }
8     vert *operator->() { return dest; }
9     bool is_bridge();

```

```

10 }; #116
11 struct vert {
12     deque<edge> con;
13     int val = 0;
14     int seen;
15     int dfs(int upd, edge *ban) { // handles multiple edges
16         if (!val) {
17             val = upd;
18             seen = val;
19             for (edge &nxt : con) {
20                 if (nxt.exists && (&nxt) != ban) #866
21                     seen = min(seen, nxt->dfs(upd + 1, nxt.rev));
22             }
23         }
24         return seen; #624
25     }
26     void remove_adj_bridges() {
27         for (edge &nxt : con) {
28             if (nxt.is_bridge()) nxt.exists = false;
29         }
30     } #106
31     int cnt_adj_bridges() {
32         int res = 0;
33         for (edge &nxt : con) res += nxt.is_bridge();
34         return res;
35     } #056
36 };
37 bool edge::is_bridge() {
38     return exists &&
39         (dest->seen > rev->dest->val || dest->val < rev->dest->seen); #223
40 }
41 vert graph[nmax];
42 int main() { // Mechanics Practice BRIDGES
43     int n, m;
44     cin >> n >> m;
45     for (int i = 0; i < m; ++i) {
46         int u, v;
47         scanf("%d %d", &u, &v);
48         graph[u].con.emplace_back(graph + v);
49         graph[v].con.emplace_back(graph + u);
50         graph[u].con.back().rev = &graph[v].con.back();
51         graph[v].con.back().rev = &graph[u].con.back();
52     }
53     graph[1].dfs(1, NULL);
54     int res = 0;
55     for (int i = 1; i <= n; ++i) res += graph[i].cnt_adj_bridges();
56     cout << res / 2 << endl;
57 }

```

17 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$

```

1 struct Graph {
2     int n;

```

```

3  vector<vector<int> > conn;
4  Graph(int nsize) {
5      n = nsize;
6      conn.resize(n);
7  }
8  void add_edge(int u, int v) { conn[u].push_back(v); }
9  void _topsort_dfs(int pos, vector<int> &result, vector<bool> &explr,
10     vector<vector<int> > &revconn) { #592
11     if (explr[pos]) return;
12     explr[pos] = true;
13     for (auto next : revconn[pos])
14         _topsort_dfs(next, result, explr, revconn);
15     result.push_back(pos);
16 }
17 vector<int> topsort() {
18     vector<vector<int> > revconn(n);
19     for (int u = 0; u < n; u++) {
20         for (auto v : conn[u]) revconn[v].push_back(u); #775
21     }
22     vector<int> result;
23     vector<bool> explr(n, false);
24     for (int i = 0; i < n; i++)
25         _topsort_dfs(i, result, explr, revconn);
26     reverse(result.begin(), result.end());
27     return result;
28 }
29 void dfs(int pos, vector<int> &result, vector<bool> &explr) { #591
30     if (explr[pos]) return;
31     explr[pos] = true;
32     for (auto next : conn[pos]) dfs(next, result, explr);
33     result.push_back(pos);
34 } %603
35 vector<vector<int> > scc() {
36     vector<int> order = topsort();
37     reverse(order.begin(), order.end());
38     vector<bool> explr(n, false);
39     vector<vector<int> > results;
40     for (auto it = order.rbegin(); it != order.rend(); ++it) {
41         vector<int> component;
42         _topsort_dfs(*it, component, explr, conn);
43         sort(component.begin(), component.end());
44         results.push_back(component); #741
45     }
46     sort(results.begin(), results.end());
47     return results;
48 }
49 }; %983
50 // Solution for:
51 // http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
52 int main() {
53     int n, m;

```

```

54     cin >> n >> m;
55     Graph g(2 * m);
56     for (int i = 0; i < n; i++) {
57         int a, sa, b, sb;
58         cin >> a >> sa >> b >> sb;
59         a--, b--;
60         g.add_edge(2 * a + 1 - sa, 2 * b + sb);
61         g.add_edge(2 * b + 1 - sb, 2 * a + sa);
62     }
63     vector<int> state(2 * m, 0);
64     {
65         vector<int> order = g.topsort();
66         vector<bool> explr(2 * m, false);
67         for (auto u : order) {
68             vector<int> traversed;
69             g.dfs(u, traversed, explr);
70             if (traversed.size() > 0 && !state[traversed[0] ^ 1]) {
71                 for (auto c : traversed) state[c] = 1;
72             }
73         }
74     }
75     for (int i = 0; i < m; i++) {
76         if (state[2 * i] == state[2 * i + 1]) {
77             cout << "IMPOSSIBLE\n";
78             return 0;
79         }
80     }
81     for (int i = 0; i < m; i++) {
82         cout << state[2 * i + 1] << '\n';
83     }
84     return 0;
85 }

```

18 Generic persistent compressed lazy segment tree

```

1 struct Seg {
2     ll sum = 0;
3     void recalc(const Seg &lhs_seg, int lhs_len, const Seg &rhs_seg,
4         int rhs_len) {
5         sum = lhs_seg.sum + rhs_seg.sum;
6     }
7 } __attribute__((packed));
8 struct Lazy {
9     ll add;
10    ll assign_val; // LLONG_MIN if no assign; #529
11    void init() {
12        add = 0;
13        assign_val = LLONG_MIN;
14    }
15    Lazy() { init(); }
16    void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len) {
17        lhs_lazy = *this;
18        rhs_lazy = *this;

```

```

19     init();
20 }
21 void merge(Lazy &oth, int len) {
22     if (oth.assign_val != LLONG_MIN) {
23         add = 0;
24         assign_val = oth.assign_val;
25     }
26     add += oth.add;
27 }
28 void apply_to_seg(Seg &cur, int len) const {
29     if (assign_val != LLONG_MIN) {
30         cur.sum = len * assign_val;
31     }
32     cur.sum += len * add;
33 }
34 } __attribute__((packed));
35 struct Node { // Following code should not need to be modified
36     int ver;
37     bool is_lazy = false;
38     Seg seg;
39     Lazy lazy;
40     Node *lc = NULL, *rc = NULL;
41     void init() {
42         if (!lc) {
43             lc = new Node{ver};
44             rc = new Node{ver};
45         }
46     }
47     Node *upd(int L, int R, int l, int r, Lazy &val, int tar_ver) {
48         if (ver != tar_ver) {
49             Node *rep = new Node(*this);
50             rep->ver = tar_ver;
51             return rep->upd(L, R, l, r, val, tar_ver);
52         }
53         if (L >= l && R <= r) {
54             val.apply_to_seg(seg, R - L);
55             lazy.merge(val, R - L);
56             is_lazy = true;
57         } else {
58             init();
59             int M = (L + R) / 2;
60             if (is_lazy) {
61                 Lazy l_val, r_val;
62                 lazy.split(l_val, r_val, R - L);
63                 lc = lc->upd(L, M, L, M, l_val, ver);
64                 rc = rc->upd(M, R, M, R, r_val, ver);
65                 is_lazy = false;
66             }
67             Lazy l_val, r_val;
68             val.split(l_val, r_val, R - L);
69             if (l < M) lc = lc->upd(L, M, l, r, l_val, ver);
70             if (M < r) rc = rc->upd(M, R, l, r, r_val, ver);

```

#953

#204

%625

#313

#138

#104

```

71     seg.recalc(lc->seg, M - L, rc->seg, R - M);
72 }
73 return this;
74 }
75 void get(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp,
76     bool last_ver) {
77     if (L >= l && R <= r) {
78         tmp->recalc(*lft_res, L - l, seg, R - L);
79         swap(lft_res, tmp);
80     } else {
81         init();
82         int M = (L + R) / 2;
83         if (is_lazy) {
84             Lazy l_val, r_val;
85             lazy.split(l_val, r_val, R - L);
86             lc = lc->upd(L, M, L, M, l_val, ver + last_ver);
87             lc->ver = ver;
88             rc = rc->upd(M, R, M, R, r_val, ver + last_ver);
89             rc->ver = ver;
90             is_lazy = false;
91         }
92         if (l < M) lc->get(L, M, l, r, lft_res, tmp, last_ver);
93         if (M < r) rc->get(M, R, l, r, lft_res, tmp, last_ver);
94     }
95 }
96 } __attribute__((packed));
97 struct SegTree { // indexes start from 0, ranges are [beg, end)
98     vector<Node *> roots; // versions start from 0
99     int len;
100     SegTree(int _len) : len(_len) { roots.push_back(new Node{0}); }
101     int upd(int l, int r, Lazy &val, bool new_ver = false) {
102         Node *cur_root =
103             roots.back()->upd(0, len, l, r, val, roots.size() - !new_ver);
104         if (cur_root != roots.back()) roots.push_back(cur_root);
105         return roots.size() - 1;
106     }
107     Seg get(int l, int r, int ver = -1) {
108         if (ver == -1) ver = roots.size() - 1;
109         Seg seg1, seg2;
110         Seg *pres = &seg1, *ptmp = &seg2;
111         roots[ver]->get(0, len, l, r, pres, ptmp, roots.size() - 1);
112         return *pres;
113     }
114 };
115 int main() {
116     int n, m; // solves Mechanics Practice LAZY
117     cin >> n >> m;
118     SegTree seg_tree(1 << 17);
119     for (int i = 0; i < n; ++i) {
120         Lazy tmp;
121         scanf("%lld", &tmp.assign_val);

```

#441

#803

#770

#700

%542

```

122     seg_tree.upd(i, i + 1, tmp);
123 }
124 for (int i = 0; i < m; ++i) {
125     int o;
126     int l, r;
127     scanf("%d %d %d", &o, &l, &r);
128     --l;
129     if (o == 1) {
130         Lazy tmp;
131         scanf("%lld", &tmp.add);
132         seg_tree.upd(l, r, tmp);
133     } else if (o == 2) {
134         Lazy tmp;
135         scanf("%lld", &tmp.assign_val);
136         seg_tree.upd(l, r, tmp);
137     } else {
138         Seg res = seg_tree.get(l, r);
139         printf("%lld\n", res.sum);
140     }
141 }
142 }

```

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

```

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

```

%932

```

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

```

#037

#593

#461

#587

```

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

```

#638

#052

#041

%905

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

```

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

```

#330

#620

#542

#549


```

51 // Should never be called, needed for compilation
52 void initiate() { cerr << 'i' << endl; }
53 void update() { cerr << 'u' << endl; }
54 };
55 int main() {
56     // return type should be same as type inside wrapped
57     BIT<BIT<wrapped<ll>, int, INT_MIN, ll>, int, INT_MIN, ll> fenwick;
58     int dim = 2;
59     vector<tuple<int, int, ll> > to_insert;
60     to_insert.emplace_back(1, 1, 1);
61     // set up all positions that are to be used for update
62     for (int i = 0; i < dim; ++i) {
63         for (auto &cur : to_insert)
64             fenwick.update(get<0>(cur),
65                             get<1>(cur)); // May include value which won't be used
66     fenwick.initiate();
67 }
68 // actual use
69 for (auto &cur : to_insert)
70     fenwick.update(get<0>(cur), get<1>(cur), get<2>(cur));
71 cout << fenwick.query(2, 2) << '\n';
72 }

```

#636
%714

21 Treap $O(\log n)$ per query

```

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

```

#698

#295

```

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

```

#233

#230

#510

#760

#634

```

80     return sum_upto(r + 1, root) - sum_upto(l, root);          #509
81 }
82 };                                                            %959
83 // Solution for:
84 // http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/TREAP
85 int main() {
86     ios_base::sync_with_stdio(false);
87     cin.tie(0);
88     int m;
89     Treap treap;
90     cin >> m;
91     for (int i = 0; i < m; i++) {
92         int type;
93         cin >> type;
94         if (type == 1) {
95             int x, y;
96             cin >> x >> y;
97             treap.insert(x, y);
98         } else if (type == 2) {
99             int x;
100             cin >> x;
101             treap.erase(x);
102         } else {
103             int l, r;
104             cin >> l >> r;
105             cout << treap.get(l, r) << endl;
106         }
107     }
108     return 0;
109 }

```

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

```

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

```

#772

```

21     *(locs[bucket_id]++) = *it;
22 }
23 locs = locs_mem;
24 if (d) {
25     T *locs_old[256];
26     locs_old[0] = start;
27     for (int j = 0; j < 255; ++j) {
28         locs_old[j + 1] = locs_old[j] + bucket_sizes[j];
29     }
30     for (int j = 0; j < 256; ++j) {
31         if (locs[j + 1] - locs[j] < msd_radix_lim) {
32             std::sort(locs[j], locs[j + 1]);
33             if (d & 1) {
34                 copy(locs[j], locs[j + 1], locs_old[j]);
35             }
36         } else {
37             msd_radixsort(locs[j], locs_old[j], bucket_sizes[j], d - 1);
38         }
39     }
40 }
41 }
42 const int nmax = 5e7;
43 ll arr[nmax], tmp[nmax];
44 int main() {
45     for (int i = 0; i < nmax; ++i) arr[i] = ((ll)rand() << 32) | rand();
46     msd_radixsort(arr, tmp, nmax);
47     assert(is_sorted(arr, arr + nmax));
48 }

```

#867

%225

23 FFT 5M length/sec

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

```

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

```

#384

#385

```

21 if (len != 1) {
22     fft_rec(arr, root_pow, len >> 1);
23     fft_rec(arr + len, root_pow, len >> 1);
24 }
25 root_pow += len;
26 for (int i = 0; i < len; ++i) {
27     Complex tmp = arr[i] + root_pow[i] * arr[i + len];
28     arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
29     arr[i] = tmp;
30 }
31 }
32 void fft(vector<Complex> &arr, int ord, bool invert) {
33     assert(arr.size() == 1 << ord);
34     static vector<Complex> root_pow(1);
35     static int inc_pow = 1;
36     static bool is_inv = false;
37     if (inc_pow <= ord) {
38         int idx = root_pow.size();
39         root_pow.resize(1 << ord);
40         for (; inc_pow <= ord; ++inc_pow) {
41             for (int idx_p = 0; idx_p < 1 << (ord - 1);
42                 idx_p += 1 << (ord - inc_pow), ++idx_p) {
43                 root_pow[idx] = Complex{cos(-idx_p * M_PI / (1 << (ord - 1))),
44                     sin(-idx_p * M_PI / (1 << (ord - 1)))};
45                 if (is_inv) root_pow[idx].b = -root_pow[idx].b;
46             }
47         }
48     }
49     if (invert != is_inv) {
50         is_inv = invert;
51         for (Complex &cur : root_pow) cur.b = -cur.b;
52     }
53     for (int i = 1, j = 0; i < (1 << ord); ++i) {
54         int m = 1 << (ord - 1);
55         bool cont = true;
56         while (cont) {
57             cont = j & m;
58             j ^= m;
59             m >>= 1;
60         }
61         if (i < j) swap(arr[i], arr[j]);
62     }
63     fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
64     if (invert)
65         for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord);
66 }
67 void mult_poly_mod(
68     vector<int> &a, vector<int> &b, vector<int> &c) { // c += a*b
69     static vector<Complex>
70         arr[4]; // correct upto 0.5-2M elements(mod ~= 1e9)
71     if (c.size() < 400) {
72         for (int i = 0; i < a.size(); ++i)

```

#249

#517

#750

#844

%380

```

73     for (int j = 0; j < b.size() && i + j < c.size(); ++j)
74         c[i + j] = ((1l)a[i] * b[j] + c[i + j]) % mod;
75 } else {
76     int fft_ord = 32 - __builtin_clz(c.size());
77     if (arr[0].size() != 1 << fft_ord)
78         for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord);
79     for (int i = 0; i < 4; ++i)
80         fill(arr[i].begin(), arr[i].end(), Complex{});
81     for (int &cur : a)
82         if (cur < 0) cur += mod;
83     for (int &cur : b)
84         if (cur < 0) cur += mod;
85     const int shift = 15;
86     const int mask = (1 << shift) - 1;
87     for (int i = 0; i < min(a.size(), c.size()); ++i) {
88         arr[0][i].a = a[i] & mask;
89         arr[1][i].a = a[i] >> shift;
90     }
91     for (int i = 0; i < min(b.size(), c.size()); ++i) {
92         arr[0][i].b = b[i] & mask;
93         arr[1][i].b = b[i] >> shift;
94     }
95     for (int i = 0; i < 2; ++i) fft(arr[i], fft_ord, false);
96     for (int i = 0; i < 2; ++i) {
97         for (int j = 0; j < 2; ++j) {
98             int tar = 2 + (i + j) / 2;
99             Complex mult = {0, -0.25};
100             if (i ^ j) mult = {0.25, 0};
101             for (int k = 0; k < (1 << fft_ord); ++k) {
102                 int rev_k = ((1 << fft_ord) - k) % (1 << fft_ord);
103                 Complex ca = arr[i][k] + conj(arr[i][rev_k]);
104                 Complex cb = arr[j][k] - conj(arr[j][rev_k]);
105                 arr[tar][k] = arr[tar][k] + mult * ca * cb;
106             }
107         }
108     }
109     for (int i = 2; i < 4; ++i) {
110         fft(arr[i], fft_ord, true);
111         for (int k = 0; k < (int)c.size(); ++k) {
112             c[k] = (c[k] + (((1l)(arr[i][k].a + 0.5) % mod)
113                 << (shift * 2 * (i - 2)))) %
114                 mod;
115             c[k] = (c[k] + (((1l)(arr[i][k].b + 0.5) % mod)
116                 << (shift * (2 * (i - 2) + 1)))) %
117                 mod;
118         }
119     }
120 }
121 }

```

#629

#625

#644

#471

#108

%231

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

```

1 struct ModArithm {
2     ull n;
3     ld rec;
4     ModArithm(ull _n) : n(_n) { // n in [2, 1<<63)
5         rec = 1.0L / n;
6     }
7     ull multf(ull a, ull b) { // a, b in [0, min(2*n, 1<<63))
8         ull mult = (ld)a * b * rec + 0.5L;
9         ll res = a * b - mult * n;
10        if (res < 0) res += n;
11        return res; // in [0, n-1)
12    }
13    ull sqp1(ull a) { return multf(a, a) + 1; }
14};
15 ull pow_mod(ull a, ull n, ModArithm &arithm) {
16     ull res = 1;
17     for (ull i = 1; i <= n; i <= 1) {
18         if (n & i) res = arithm.multf(res, a);
19         a = arithm.multf(a, a);
20     }
21     return res;
22 }
23 vector<char> small_primes = {
24     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
25 bool is_prime(ull n) { // n <= 1<<63, 1M rand/s
26     ModArithm arithm(n);
27     if (n == 2 || n == 3) return true;
28     if (!(n & 1) || n == 1) return false;
29     ull s = __builtin_ctz(n - 1);
30     ull d = (n - 1) >> s;
31     for (ull a : small_primes) {
32         if (a >= n) break;
33         a = pow_mod(a, d, arithm);
34         if (a == 1 || a == n - 1) continue;
35         for (ull r = 1; r < s; ++r) {
36             a = arithm.multf(a, a);
37             if (a == 1) return false;
38             if (a == n - 1) break;
39         }
40         if (a != n - 1) return false;
41     }
42     return true;
43 }
44 ll pollard_rho(ll n) {
45     ModArithm arithm(n);
46     int cum_cnt = 64 - __builtin_clz(n);
47     cum_cnt *= cum_cnt / 5 + 1;
48     while (true) {
49         ll lv = rand() % n;

```

#780

%493

%144

#402

#806

%975

```

50     ll v = arithm.sqp1(lv);
51     int idx = 1;
52     int tar = 1;
53     while (true) {
54         ll cur = 1;
55         ll v_cur = v;
56         int j_stop = min(cum_cnt, tar - idx);
57         for (int j = 0; j < j_stop; ++j) {
58             cur = arithm.multf(cur, abs(v_cur - lv));
59             v_cur = arithm.sqp1(v_cur);
60             ++idx;
61         }
62         if (!cur) {
63             for (int j = 0; j < cum_cnt; ++j) {
64                 ll g = __gcd(abs(v - lv), n);
65                 if (g == 1) {
66                     v = arithm.sqp1(v);
67                 } else if (g == n) {
68                     break;
69                 } else {
70                     return g;
71                 }
72             }
73             break;
74         } else {
75             ll g = __gcd(cur, n);
76             if (g != 1) return g;
77         }
78         v = v_cur;
79         idx += j_stop;
80         if (idx == tar) {
81             lv = v;
82             tar *= 2;
83             v = arithm.sqp1(v);
84             ++idx;
85         }
86     }
87 }
88 }
89 map<ll, int> prime_factor(ll n,
90     map<ll, int> *res = NULL) { // n <= 1<<61, ~1000/s (<500/s on CF)
91     if (!res) {
92         map<ll, int> res_act;
93         for (int p : small_primes) {
94             while (!(n % p)) {
95                 ++res_act[p];
96                 n /= p;
97             }
98         }
99         if (n != 1) prime_factor(n, &res_act);
100        return res_act;

```

#290

#912

#208

#174

%542

#612

```

101 }
102 if (is_prime(n)) {
103     ++(*res)[n];
104 } else {
105     ll factor = pollard_rho(n);
106     prime_factor(factor, res);
107     prime_factor(n / factor, res);
108 }                                     #350
109 return map<ll, int>();
110 } // Usage: fact = prime_factor(n);    %477

```

25 Symmetric Submodular Functions; Queyrannes's algorithm

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

```

def minimize():
    s = merge_all_loops()
    while size >= 3:
        t, u = find_pp()
        {u} is a possible minimizer
        tu = merge(t, u)
        if tu not in I:
            s = merge(tu, s)

    for x in V:
        {x} is a possible minimizer

def find_pp():
    W = {s} # s as in minimizer()
    todo = V/W
    ord = []
    while len(todo) > 0:
        x = min(todo, key=lambda x: f(W+{x}) - f({x}))
        W += {x}
        todo -= {x}
        ord.append(x)
    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)

```