

University of Tartu ICPC Team Notebook (2017-2018) November 22, 2018

Contents

- 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 $\mathcal{O}(|\text{alpha}| \sum \text{len})$
 - 11 Suffix automaton and tree $\mathcal{O}((n + q) \log(|\text{alpha}|))$
 - 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 Templatized HLD $\mathcal{O}(M(n) \log n)$ per query
 - 20 Templatized multi dimensional BIT $\mathcal{O}(\log(n)^{\dim})$ 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

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

1 25 Symmetric Submodular Functions; Queyranne's algorithm

1 Setup

```
1 set smartindent cindent
2 set ts=4 sw=4 expandtab
3 syntax enable
4 set clipboard=unnamedplus
5 "setxkbmap -option caps:escape
6 "valgrind --vgdb-error=0 ./a <inp &
7 "gdb a
8 "target remote | vgdb
```

2 crc.sh

```
1 #!/bin/env bash
2 for j in `seq 10 10 200` ; do
3     sed '/^s*$/d' $1 | head -$j | tr -d '[:space:]' | cksum | cut -f1
    ↪ -d ' ' | tail -c 4 #whist espaces don't matter.
4 done #there shouldn't be any COMMENTS.
5 #copy lines being checked to separate file.
6 # $ ./crc.sh tmp.cpp
```

3 gcc ordered set

```
1 #include<bits/stdc++.h>
2 typedef long long ll;
3 using namespace std;
4 #include<ext/pb_ds/assoc_container.hpp>
5 #include<ext/pb_ds/tree_policy.hpp>
6 using namespace __gnu_pbds;
7 template <typename T>
8 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
9   → tree_order_statistics_node_update>;
10 int main(){#221
11   ordered_set<int> cur;
12   cur.insert(1);
13   cur.insert(3);
14   cout << cur.order_of_key(2) << endl; // the number of elements in the
15   → set less than 2
16   cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in
17   → the set(0-based)
18   cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in
19   → the set(0-based)
20 }
```

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 .

$\text{dist}_l(P) = \text{side}_l(P)/\|v_l\|$ 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);
```

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) + integrate(f, m, b,
12        computing_power/2);                                #430
12 }                                                 %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; // vertexes 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)
8         return true;
9     point sp = (B - A) / (C - A);
10    double ang = M_PI/2-abs(abs(arg(sp))-M_PI/2); //positive angle with
11        the real line                                #623
11    return ang < min_delta;
12 }                                                 %446
13 point circum_center(){
14     if(collinear())
15         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);                                #385
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 }                                                 %742
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 0 + 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())
46         return point(NAN,NAN);
47     double a, b, c; //side lengths
48     a = abs(B - C);
49     b = abs(A - C);
50     c = abs(A - B);
51     //trilinear coordinates are (1,1,1)
52     //barycentric coordinates
53     double c_A = a, c_B = b, c_C = c;
54     double sum = c_A + c_B + c_C;
55     c_A /= sum;                                         #157
56     c_B /= sum;
57     c_C /= sum;
58     //cartesian coordinates of the incenter
59     return c_A * A + c_B * B + c_C * C;
60 }                                                 %980

```

8 Seg-Seg intersection, halfplane intersection area

```

1 struct Segment {
2     Vec a, b;
3     Vec d() {
4         return b-a;
5     }
6 };
7 ostream& operator<<(ostream& l, Segment r) {
8     return l << r.a << '-' << r.b;
9 }
10 Vec intersection(Segment l, Segment r) {           #355
11     Vec dl = l.d(), dr = r.d();
12     if(cross(dl, dr) == 0)
13         return {nanl(""), nanl("")};
14     long double h = cross(dr, l.a-r.a) / len(dr);
15     long double dh = cross(dr, dl) / len(dr);
16     return l.a + dl * (h / -dh);
17 }
18 //Returns the area bounded by halfplanes
19 long double getArea(vector<Segment> lines) {
20     long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;
21     vector<Segment> linesBySide[2];
22     for(auto line : lines) {
23         if(line.b.y == line.a.y) {
24             if(line.a.x < line.b.x) {
25                 lowerbound = max(lowerbound, line.a.y);
26             } else {
27                 upperbound = min(upperbound, line.a.y);
28             }
29         } else if(line.a.y < line.b.y) {

```

```

30     linesBySide[1].push_back(line);
31 } else {
32     linesBySide[0].push_back({line.b, line.a});
33 }
34 }
35 sort(linesBySide[0].begin(), linesBySide[0].end(), [] (Segment l,
36     Segment r) {
37     if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a >
38         normal(r.d())*r.a;
39     return cross(l.d(), r.d()) < 0;
40 });
41 sort(linesBySide[1].begin(), linesBySide[1].end(), [] (Segment l,
42     Segment r) {
43     if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a <
44         normal(r.d())*r.a;
45     return cross(l.d(), r.d()) > 0;
46 });
47 //Now find the application area of the lines and clean up redundant
48 // ones
49 vector<long double> applyStart[2];
50 for(int side = 0; side < 2; side++) {
51     vector<long double> &apply = applyStart[side];
52     vector<Segment> curLines;
53     for(auto line : linesBySide[side]) {
54         while(curLines.size() > 0) {
55             Segment other = curLines.back();
56             if(cross(line.d(), other.d()) != 0) {
57                 long double start = intersection(line, other).y;
58                 if(start > apply.back()) break;
59             }
60             curLines.pop_back();
61             apply.pop_back();
62         }
63         if(curLines.size() == 0) {
64             apply.push_back(-HUGE_VALL);
65         } else {
66             apply.push_back(intersection(line, curLines.back()).y);
67         }
68         curLines.push_back(line);
69     }
70     linesBySide[side] = curLines;
71 }
72 applyStart[0].push_back(HUGE_VALL);
73 applyStart[1].push_back(HUGE_VALL);
74 long double result = 0;
75 {
76     long double lb = -HUGE_VALL, ub;
77     for(int i=0, j=0; i < (int)linesBySide[0].size() && j <
78         (int)linesBySide[1].size();lb = ub) {
79         ub = min(applyStart[0][i+1], applyStart[1][j+1]);
80         long double alb = lb, aub = ub;
81         Segment l0 = linesBySide[0][i], l1 = linesBySide[1][j];
82         if(cross(l1.d(), l0.d()) > 0) {
83             alb = max(alb, intersection(l0, l1).y);
84         }
85     }
86 }
87 
```

```

78     } else if(cross(l1.d(), 10.d()) < 0) {
79         aub = min(aub, intersection(l0, l1).y);
80     }
81     alb = max(alb, lowerbound);
82     aub = min(aub, upperbound);                                #419
83     aub = max(aub, alb);
84     {
85         long double x1 = 10.a.x + (alb - 10.a.y) / 10.d().y * 10.d().x;
86         long double x2 = 10.a.x + (aub - 10.a.y) / 10.d().y * 10.d().x;
87         result -= (aub - alb) * (x1 + x2) / 2;
88     }
89     {
90         long double x1 = 11.a.x + (alb - 11.a.y) / 11.d().y * 11.d().x;
91         long double x2 = 11.a.x + (aub - 11.a.y) / 11.d().y * 11.d().x;
92         result += (aub - alb) * (x1 + x2) / 2;                  #228
93     }
94     if(applyStart[0][i+1] < applyStart[1][j+1]) {
95         i++;
96     } else {
97         j++;
98     }
99 }
100 }
101 return result;
102 }                                         %011

```

9 Convex polygon algorithms

```
1 ll dot(const pair<int, int> &v1, const pair<int, int> &v2) {
2     return (ll)v1.first * v2.first + (ll)v1.second * v2.second;
3 }
4 ll cross(const pair<int, int> &v1, const pair<int, int> &v2) {
5     return (ll)v1.first * v2.second - (ll)v2.first * v1.second;
6 }
7 ll dist_sq(const pair<int, int> &p1, const pair<int, int> &p2) {
8     return (ll)(p2.first - p1.first) * (p2.first - p1.first) +
9             (ll)(p2.second - p1.second) * (p2.second - p1.second);
10 }
11 struct Hull {
12     vector<pair<pair<int, int>, pair<int, int>>> hull;
13     vector<pair<pair<int, int>, pair<int, int>>>::iterator
14         upper_begin;
15     template <typename Iterator>
16     void extend_hull(Iterator begin, Iterator end) { // O(n)
17         vector<pair<int, int>> res;
18         for (auto it = begin; it != end; ++it) {
19             if (res.empty() || *it != res.back()) {
20                 while (res.size() >= 2) { #678
21                     auto v1 = make_pair(
22                         res[res.size() - 1].first - res[res.size() - 2].first,
23                         res[res.size() - 1].second - res[res.size() - 2].second);
24                     auto v2 =
25                         make_pair(it->first - res[res.size() - 2].first,
26                                   it->second - res[res.size() - 2].second);
```

```

27     if (cross(v1, v2) > 0) break;
28     res.pop_back();
29   }
30   res.push_back(*it);
31 }
32 }
33 for (int i = 0; i < res.size() - 1; ++i)
34   hull.emplace_back(res[i], res[i + 1]);
35 }
36 Hull(vector<pair<int, int> &vert) { // atleast 2 distinct points
37   sort(vert.begin(), vert.end()); // O(n log(n))
38   extend_hull(vert.begin(), vert.end());
39   int diff = hull.size();
40   extend_hull(vert.rbegin(), vert.rend()); #011
41   upper_begin = hull.begin() + diff;
42 } #873
43 bool contains(pair<int, int> p) { // O(log(n))
44   if (p < hull.front().first || p > upper_begin->first)
45     return false;
46   {
47     auto it_low = lower_bound(
48       hull.begin(), upper_begin,
49       make_pair(make_pair(p.first, (int)-2e9), make_pair(0, 0)));
50     if (it_low != hull.begin()) --it_low;
51     auto v1 =
52       make_pair(it_low->second.first - it_low->first.first, #477
53                   it_low->second.second - it_low->first.second);
54     auto v2 = make_pair(p.first - it_low->first.first,
55                         p.second - it_low->first.second);
56     if (cross(v1, v2) < 0) // < 0 is inclusive, <=0 is exclusive
57       return false;
58   }
59   {
60     auto it_up = lower_bound(
61       hull.rbegin(), hull.rbegin() + (hull.end() - upper_begin),
62       make_pair(make_pair(p.first, (int)2e9), make_pair(0, 0)));
63     if (it_up - hull.rbegin() == hull.end() - upper_begin) --it_up;
64     auto v1 = make_pair(it_up->first.first - it_up->second.first,
65                         it_up->first.second - it_up->second.second);
66     auto v2 = make_pair(p.first - it_up->second.first,
67                         p.second - it_up->second.second);
68     if (cross(v1, v2) > 0) // > 0 is inclusive, >=0 is exclusive
69       return false;
70   }
71   return true;
72 } #092
73 template <
74   typename T> // The function can have only one local min and max
75   // and may be constant only at min and max.
76   vector<
77     pair<pair<int, int>, pair<int, int> > >::iterator
78   max(function<

```

```

79           T(const pair<pair<int, int>, pair<int, int> &>)
80           f) { // O(log(n))
81   auto l = hull.begin();
82   auto r = hull.end();
83   vector<pair<pair<int, int>, pair<int, int> >::iterator best =
84     #479
85   hull.end();
86   T best_val;
87   while (r - l > 2) {
88     auto mid = l + (r - l) / 2;
89     T l_val = f(*l);
90     T l_nxt_val = f(*(l + 1));
91     T mid_val = f(*mid);
92     T mid_nxt_val = f(*(mid + 1));
93     if (best == hull.end() || #200
94         l_val > best_val) { // If max is at l we may remove it
95         best = l; // from
96         best_val = l_val;
97     }
98     if (l_nxt_val > l_val) {
99       if (mid_val < l_val) {
100         r = mid;
101     } else {
102       if (mid_nxt_val > mid_val) {
103         l = mid + 1;
104       } else {
105         r = mid + 1;
106       }
107     }
108   } else {
109     if (mid_val < l_val) {
110       l = mid + 1;
111     } else {
112       if (mid_nxt_val > mid_val) {
113         l = mid + 1;
114       } else {
115         r = mid + 1;
116       }
117     }
118   }
119   T l_val = f(*l);
120   if (best == hull.end() || l_val > best_val) { #369
121     best = l;
122     best_val = l_val;
123   }
124   if (r - l > 1) { #920
125     T l_nxt_val = f(*(l + 1));
126     if (best == hull.end() || l_nxt_val > best_val) {
127       best = l + 1;
128       best_val = l_nxt_val;
129     }
130   }

```

```

131     }
132     return best;
133 }
134 vector<pair<pair<int, int>, pair<int, int> >::iterator closest(
135     pair<int, int> p) { // p can't be internal(can be on border), hull
136     // must have atleast 3 points
137     const pair<pair<int, int>, pair<int, int> > &ref_p =
138         hull.front(); // O(log(n))
139     return max(
140         function<double(const pair<pair<int, int>, pair<int, int> > &)(
141             [&p,
142             &ref_p](const pair<pair<int, int>, pair<int, int> >
143                     &seg) { // accuracy of used type should be coord-2
144                 if (p == seg.first) return 10 - M_PI; #900
145                 auto v1 = make_pair(seg.second.first - seg.first.first,
146                                     seg.second.second - seg.first.second);
147                 auto v2 = make_pair(p.first - seg.first.first,
148                                     p.second - seg.first.second);
149                 ll cross_prod = cross(v1, v2);
150                 if (cross_prod > 0) { // order the backside by angle
151                     auto v1 = make_pair(ref_p.first.first - p.first,
152                                         ref_p.first.second - p.second);
153                     auto v2 = make_pair(seg.first.first - p.first,
154                                         seg.first.second - p.second); #534
155                     ll dot_prod = dot(v1, v2);
156                     ll cross_prod = cross(v2, v1);
157                     return atan2(cross_prod, dot_prod) / 2;
158                 }
159                 ll dot_prod = dot(v1, v2);
160                 double res = atan2(dot_prod, cross_prod);
161                 if (dot_prod <= 0 && res > 0) res = -M_PI;
162                 if (res > 0) {
163                     res += 20;
164                 } else { #913
165                     res = 10 - res;
166                 }
167                 return res;
168             }));
169 }
170 pair<int, int> forw_tan(
171     pair<int, int> p) { // can't be internal or on border
172     const pair<pair<int, int>, pair<int, int> > &ref_p =
173         hull.front(); // O(log(n))
174     auto best_seg = max(
175         function<double(const pair<pair<int, int>, pair<int, int> > &)(
176             [&p,
177             &ref_p](const pair<pair<int, int>, pair<int, int> >
178                     &seg) { // accuracy of used type should be coord-2
179                 auto v1 = make_pair(ref_p.first.first - p.first,
180                                     ref_p.first.second - p.second); #089
181                 auto v2 = make_pair(seg.first.first - p.first,
182                                     seg.first.second - p.second);
183                 ll dot_prod = dot(v1, v2);
184             }));
185             %331
186             %483
187             %850
188             %013
189             #662
190             #781
191             #826
192             #826
193             #826
194             #826
195             #826
196             #826
197             #826
198             #826
199             #826
200             #826
201             #826
202             #826
203             #826
204             #826
205             #826
206             #826
207             #826
208             #826
209             #826
210             #826
211             #826
212             #826
213             #826
214             #826
215             #826
216             #826
217             #826
218             #826
219             #826
220             #826
221             #826
222             #826
223             #826
224             #826
225             #826
226             #826
227             #826
228             #826
229             #826
230             #826
231             #826
232             #826
233             #826
234             #826
235             #826
236             #826
237             #826
238             #826
239             #826
240             #826
241             #826
242             #826
243             #826
244             #826
245             #826
246             #826
247             #826
248             #826
249             #826
250             #826
251             #826
252             #826
253             #826
254             #826
255             #826
256             #826
257             #826
258             #826
259             #826
260             #826
261             #826
262             #826
263             #826
264             #826
265             #826
266             #826
267             #826
268             #826
269             #826
270             #826
271             #826
272             #826
273             #826
274             #826
275             #826
276             #826
277             #826
278             #826
279             #826
280             #826
281             #826
282             #826
283             #826
284             #826
285             #826
286             #826
287             #826
288             #826
289             #826
290             #826
291             #826
292             #826
293             #826
294             #826
295             #826
296             #826
297             #826
298             #826
299             #826
300             #826
301             #826
302             #826
303             #826
304             #826
305             #826
306             #826
307             #826
308             #826
309             #826
310             #826
311             #826
312             #826
313             #826
314             #826
315             #826
316             #826
317             #826
318             #826
319             #826
320             #826
321             #826
322             #826
323             #826
324             #826
325             #826
326             #826
327             #826
328             #826
329             #826
330             #826
331             #826
332             #826
333             #826
334             #826
335             #826
336             #826
337             #826
338             #826
339             #826
340             #826
341             #826
342             #826
343             #826
344             #826
345             #826
346             #826
347             #826
348             #826
349             #826
350             #826
351             #826
352             #826
353             #826
354             #826
355             #826
356             #826
357             #826
358             #826
359             #826
360             #826
361             #826
362             #826
363             #826
364             #826
365             #826
366             #826
367             #826
368             #826
369             #826
370             #826
371             #826
372             #826
373             #826
374             #826
375             #826
376             #826
377             #826
378             #826
379             #826
380             #826
381             #826
382             #826
383             #826
384             #826
385             #826
386             #826
387             #826
388             #826
389             #826
390             #826
391             #826
392             #826
393             #826
394             #826
395             #826
396             #826
397             #826
398             #826
399             #826
400             #826
401             #826
402             #826
403             #826
404             #826
405             #826
406             #826
407             #826
408             #826
409             #826
410             #826
411             #826
412             #826
413             #826
414             #826
415             #826
416             #826
417             #826
418             #826
419             #826
420             #826
421             #826
422             #826
423             #826
424             #826
425             #826
426             #826
427             #826
428             #826
429             #826
430             #826
431             #826
432             #826
433             #826
434             #826
435             #826
436             #826
437             #826
438             #826
439             #826
440             #826
441             #826
442             #826
443             #826
444             #826
445             #826
446             #826
447             #826
448             #826
449             #826
450             #826
451             #826
452             #826
453             #826
454             #826
455             #826
456             #826
457             #826
458             #826
459             #826
460             #826
461             #826
462             #826
463             #826
464             #826
465             #826
466             #826
467             #826
468             #826
469             #826
470             #826
471             #826
472             #826
473             #826
474             #826
475             #826
476             #826
477             #826
478             #826
479             #826
480             #826
481             #826
482             #826
483             #826
484             #826
485             #826
486             #826
487             #826
488             #826
489             #826
490             #826
491             #826
492             #826
493             #826
494             #826
495             #826
496             #826
497             #826
498             #826
499             #826
500             #826
501             #826
502             #826
503             #826
504             #826
505             #826
506             #826
507             #826
508             #826
509             #826
510             #826
511             #826
512             #826
513             #826
514             #826
515             #826
516             #826
517             #826
518             #826
519             #826
520             #826
521             #826
522             #826
523             #826
524             #826
525             #826
526             #826
527             #826
528             #826
529             #826
530             #826
531             #826
532             #826
533             #826
534             #826
535             #826
536             #826
537             #826
538             #826
539             #826
540             #826
541             #826
542             #826
543             #826
544             #826
545             #826
546             #826
547             #826
548             #826
549             #826
550             #826
551             #826
552             #826
553             #826
554             #826
555             #826
556             #826
557             #826
558             #826
559             #826
560             #826
561             #826
562             #826
563             #826
564             #826
565             #826
566             #826
567             #826
568             #826
569             #826
570             #826
571             #826
572             #826
573             #826
574             #826
575             #826
576             #826
577             #826
578             #826
579             #826
580             #826
581             #826
582             #826
583             #826
584             #826
585             #826
586             #826
587             #826
588             #826
589             #826
590             #826
591             #826
592             #826
593             #826
594             #826
595             #826
596             #826
597             #826
598             #826
599             #826
600             #826
601             #826
602             #826
603             #826
604             #826
605             #826
606             #826
607             #826
608             #826
609             #826
610             #826
611             #826
612             #826
613             #826
614             #826
615             #826
616             #826
617             #826
618             #826
619             #826
620             #826
621             #826
622             #826
623             #826
624             #826
625             #826
626             #826
627             #826
628             #826
629             #826
630             #826
631             #826
632             #826
633             #826
634             #826
635             #826
636             #826
637             #826
638             #826
639             #826
640             #826
641             #826
642             #826
643             #826
644             #826
645             #826
646             #826
647             #826
648             #826
649             #826
650             #826
651             #826
652             #826
653             #826
654             #826
655             #826
656             #826
657             #826
658             #826
659             #826
660             #826
661             #826
662             #826
663             #826
664             #826
665             #826
666             #826
667             #826
668             #826
669             #826
670             #826
671             #826
672             #826
673             #826
674             #826
675             #826
676             #826
677             #826
678             #826
679             #826
680             #826
681             #826
682             #826
683             #826
684             #826
685             #826
686             #826
687             #826
688             #826
689             #826
690             #826
691             #826
692             #826
693             #826
694             #826
695             #826
696             #826
697             #826
698             #826
699             #826
700             #826
701             #826
702             #826
703             #826
704             #826
705             #826
706             #826
707             #826
708             #826
709             #826
710             #826
711             #826
712             #826
713             #826
714             #826
715             #826
716             #826
717             #826
718             #826
719             #826
720             #826
721             #826
722             #826
723             #826
724             #826
725             #826
726             #826
727             #826
728             #826
729             #826
730             #826
731             #826
732             #826
733             #826
734             #826
735             #826
736             #826
737             #826
738             #826
739             #826
740             #826
741             #826
742             #826
743             #826
744             #826
745             #826
746             #826
747             #826
748             #826
749             #826
750             #826
751             #826
752             #826
753             #826
754             #826
755             #826
756             #826
757             #826
758             #826
759             #826
760             #826
761             #826
762             #826
763             #826
764             #826
765             #826
766             #826
767             #826
768             #826
769             #826
770             #826
771             #826
772             #826
773             #826
774             #826
775             #826
776             #826
777             #826
778             #826
779             #826
780             #826
781             #826
782             #826
783             #826
784             #826
785             #826
786             #826
787             #826
788             #826
789             #826
790             #826
791             #826
792             #826
793             #826
794             #826
795             #826
796             #826
797             #826
798             #826
799             #826
800             #826
801             #826
802             #826
803             #826
804             #826
805             #826
806             #826
807             #826
808             #826
809             #826
810             #826
811             #826
812             #826
813             #826
814             #826
815             #826
816             #826
817             #826
818             #826
819             #826
820             #826
821             #826
822             #826
823             #826
824             #826
825             #826
826             #826
827             #826
828             #826
829             #826
830             #826
831             #826
832             #826
833             #826
834             #826
835             #826
836             #826
837             #826
838             #826
839             #826
840             #826
841             #826
842             #826
843             #826
844             #826
845             #826
846             #826
847             #826
848             #826
849             #826
850             #826
851             #826
852             #826
853             #826
854             #826
855             #826
856             #826
857             #826
858             #826
859             #826
860             #826
861             #826
862             #826
863             #826
864             #826
865             #826
866             #826
867             #826
868             #826
869             #826
870             #826
871             #826
872             #826
873             #826
874             #826
875             #826
876             #826
877             #826
878             #826
879             #826
880             #826
881             #826
882             #826
883             #826
884             #826
885             #826
886             #826
887             #826
888             #826
889             #826
890             #826
891             #826
892             #826
893             #826
894             #826
895             #826
896             #826
897             #826
898             #826
899             #826
900             #826
901             #826
902             #826
903             #826
904             #826
905             #826
906             #826
907             #826
908             #826
909             #826
910             #826
911             #826
912             #826
913             #826
914             #826
915             #826
916             #826
917             #826
918             #826
919             #826
920             #826
921             #826
922             #826
923             #826
924             #826
925             #826
926             #826
927             #826
928             #826
929             #826
930             #826
931             #826
932             #826
933             #826
934             #826
935             #826
936             #826
937             #826
938             #826
939             #826
940             #826
941             #826
942             #826
943             #826
944             #826
945             #826
946             #826
947             #826
948             #826
949             #826
950             #826
951             #826
952             #826
953             #826
954             #826
955             #826
956             #826
957             #826
958             #826
959             #826
960             #826
961             #826
962             #826
963             #826
964             #826
965             #826
966             #826
967             #826
968             #826
969             #826
970             #826
971             #826
972             #826
973             #826
974             #826
975             #826
976             #826
977             #826
978             #826
979             #826
980             #826
981             #826
982             #826
983             #826
984             #826
985             #826
986             #826
987             #826
988             #826
989             #826
990             #826
991             #826
992             #826
993             #826
994             #826
995             #826
996             #826
997             #826
998             #826
999             #826
1000            #826
1001            #826
1002            #826
1003            #826
1004            #826
1005            #826
1006            #826
1007            #826
1008            #826
1009            #826
1010            #826
1011            #826
1012            #826
1013            #826
1014            #826
1015            #826
1016            #826
1017            #826
1018            #826
1019            #826
1020            #826
1021            #826
1022            #826
1023            #826
1024            #826
1025            #826
1026            #826
1027            #826
1028            #826
1029            #826
1030            #826
1031            #826
1032            #826
1033            #826
1034            #826
1035            #826
1036            #826
1037            #826
1038            #826
1039            #826
1040            #826
1041            #826
1042            #826
1043            #826
1044            #826
1045            #826
1046            #826
1047            #826
1048            #826
1049            #826
1050            #826
1051            #826
1052            #826
1053            #826
1054            #826
1055            #826
1056            #826
1057            #826
1058            #826
1059            #826
1060            #826
1061            #826
1062            #826
1063            #826
1064            #826
1065            #826
1066            #826
1067            #826
1068            #826
1069            #826
1070            #826
1071            #826
1072            #826
1073            #826
1074            #826
1075            #826
1076            #826
1077            #826
1078            #826
1079            #826
1080            #826
1081            #826
1082            #826
1083            #826
1084            #826
1085            #826
1086            #826
1087            #826
1088            #826
1089            #826
1090            #826
1091            #826
1092            #826
1093            #826
1094            #826
1095            #826
1096            #826
1097            #826
1098            #826
1099            #826
1100            #826
1101            #826
1102            #826
1103            #826
1104            #826
1105            #826
1106            #826
1107            #826
1108            #826
1109            #826
1110            #826
1111            #826
1112            #826
1113            #826
1114            #826
1115            #826
1116            #826
1117            #826
1118            #826
1119            #826
1120            #826
1121            #826
1122            #826
1123            #826
1124            #826
1125            #826
1126            #826
1127            #826
1128            #826
1129            #826
1130            #826
1131            #826
1132            #826
1133            #826
1134            #826
1135            #826
1136            #826
1137            #826
1138            #826
1139            #826
1140            #826
1141            #826
1142            #826
1143            #826
1144            #826
1145            #826
1146            #826
1147            #826
1148            #826
1149            #826
1150            #826
1151            #826
1152            #826
1153            #826
1154            #826
1155            #826
1156            #826
1157            #826
1158            #826
1159            #826
1160            #826
1161            #826
1162            #826
1163            #826
1164            #826
1165            #826
1166            #826
1167            #826
1168            #826
1169            #826
1170            #826
1171            #826
1172            #826
1173            #826
1174            #826
1175            #826
1176            #826
1177            #826
1178            #826
1179            #826
1180            #826
1181            #826
1182            #826
1183            #826
1184            #826
1185            #826
1186            #826
1187            #826
1188            #826
1189            #826
1190            #826
1191            #826
1192            #826
1193            #826
1194            #826
1195            #826
1196            #826
1197            #826
1198            #826
1199            #826
1200            #826
1201            #826
1202            #826
1203            #826
1204            #826
1205            #826
1206            #826
1207            #826
1208            #826
1209            #826
1210            #826
1211            #826
1212            #826
1213            #826
1214            #826
1215            #826
1216            #826
1217            #826
1218            #826
1219            #826
1220            #826
1221            #826
1222            #826
1223            #826
1224            #826
1225            #826
1226            #826
1227            #826
1228            #826
1229            #826
1230            #826
1231            #826
1232            #826
1233            #826
1234            #826
1235            #826
1236            #826
1237            #826
1238            #826
1239            #826
1240            #826
1241            #826
1242            #826
1243            #826
1244            #826
1245            #826
1246            #826
1247            #826
1248            #826
1249            #826
1250            #826
1251            #826
1252            #826
1253            #826
1254            #826
1255            #826
1256            #826
1257            #826
1258            #826
1259            #826
1260            #826
1261            #826
1262            #826
1263            #826
1264            #826
1265            #826
1266            #826
1267            #826
1268            #826
1269            #826
1270            #826
1271            #826
1272            #826
1273            #826
1274            #826
1275            #826
1276            #826
1277            #826
1278            #826
1279            #826
1280            #826
1281            #826
1282            #826
1283            #826
1284            #826
1285            #826
1286            #826
1287            #826
1288            #826
1289            #826
1290            #
```

```

237     if (dot(dir, hull.front().first) <= opt_val) {
238         it_r2 =
239             upper_bound(hull.begin(), it_max + 1, line, inc_comp) - 1;
240             ↵ #388
241     } else {
242         it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
243     }
244     return make_pair(it_r1, it_r2);
245 } %000
246 pair<pair<int, int>, pair<int, int> > diameter() { // O(n)
247     pair<pair<int, int>, pair<int, int> > res;
248     ll dia_sq = 0;
249     auto it1 = hull.begin();
250     auto it2 = upper_begin;
251     auto v1 =
252         make_pair(hull.back().second.first - hull.back().first.first,
253                 hull.back().second.second - hull.back().first.second);
254     while (it2 != hull.begin()) {
255         auto v2 =
256             make_pair((it2 - 1)->second.first - (it2 - 1)->first.first,
257                     (it2 - 1)->second.second - (it2 - 1)->first.second);
258         ll decider = cross(v1, v2);
259         if (decider > 0) break;
260         --it2;
261     }
262     while (it2 != hull.end()) { // check all antipodal pairs
263         if (dist_sq(it1->first, it2->first) > dia_sq) {
264             res = make_pair(it1->first, it2->first);
265             dia_sq = dist_sq(res.first, res.second); #125
266         }
267         auto v1 = make_pair(it1->second.first - it1->first.first,
268                             it1->second.second - it1->first.second);
269         auto v2 = make_pair(it2->second.first - it2->first.first,
270                             it2->second.second - it2->first.second);
271         ll decider = cross(v1, v2);
272         if (decider == 0) { // report cross pairs at parallel lines.
273             if (dist_sq(it1->second, it2->first) > dia_sq) {
274                 res = make_pair(it1->second, it2->first);
275                 dia_sq = dist_sq(res.first, res.second); #860
276             }
277             if (dist_sq(it1->first, it2->second) > dia_sq) {
278                 res = make_pair(it1->first, it2->second);
279                 dia_sq = dist_sq(res.first, res.second);
280             }
281             ++it1;
282             ++it2;
283         } else if (decider < 0) {
284             ++it1;
285         } else {
286             ++it2;
287         }
288     }
289     return res;

```

```

290     }
291 }; %215


---



## 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(){
6         memset(nxt, 0, alpha_size*sizeof(node *));
7     }
8     int cnt=0;
9 };
10 node *aho_corasick(vector<vector<char> > &dict){ #480
11     node *root= new node;
12     root->suffix = 0;
13     vector<pair<vector<char> *, node *> > cur_state;
14     for(vector<char> &s : dict)
15         cur_state.emplace_back(&s, root);
16     for(int i=0; !cur_state.empty(); ++i){
17         vector<pair<vector<char> *, node *> > nxt_state;
18         for(auto &cur : cur_state){
19             node *nxt=cur.second->nxt[(*cur.first)[i]];
20             if(nxt){
21                 cur.second=nxt;
22             }else{
23                 nxt = new node;
24                 cur.second->nxt[(*cur.first)[i]] = nxt;
25                 node *suf = cur.second->suffix;
26                 cur.second = nxt;
27                 nxt->suffix = root; //set correct suffix link
28                 while(suf){
29                     if(suf->nxt[(*cur.first)[i]]){ #786
30                         nxt->suffix = suf->nxt[(*cur.first)[i]];
31                         break;
32                     }
33                     suf=suf->suffix;
34                 }
35             }
36             if(cur.first->size() > i+1)
37                 nxt_state.push_back(cur);
38         }
39         cur_state=nxt_state;
40     }
41     return root;
42 }; %064
43 //auxiliary functions for searching and counting
44 node *walk(node *cur, char c){ //longest prefix in dict that is suffix
45     ↵ of walked string.
46     while(true){
47         if(cur->nxt[c])
48             return cur->nxt[c];
49         if(!cur->suffix)

```

```

49     return cur;
50     cur = cur->suffix;
51 }
52 }
53 void cnt_matches(node *root, vector<char> &match_in){ %127
54     node *cur = root;
55     for(char c : match_in){
56         cur = walk(cur, c);
57         ++cur->cnt;
58     }
59 }
60 void add_cnt(node *root){ //After counting matches propagate ONCE to #286
61     suffixes for final counts
62     vector<node *> to_visit = {root};
63     for(int i=0; i<to_visit.size(); ++i){
64         node *cur = to_visit[i];
65         for(int j=0; j<alpha_size; ++j){
66             if(cur->nxt[j])
67                 to_visit.push_back(cur->nxt[j]);
68         }
69     for(int i=to_visit.size()-1; i>0; --i) #865
70         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
71 }
72 int main(){ %313
    //http://codeforces.com/group/s3etJR5zZK/contest/212916/problem/4
73     int n, len;
74     scanf("%d %d", &len, &n);
75     vector<char> a(len+1);
76     scanf("%s", a.data());
77     a.pop_back();
78     for(char &c : a)
79         c -= 'a';
80     vector<vector<char>> dict(n);
81     for(int i=0; i<n; ++i){
82         scanf("%d", &len);
83         dict[i].resize(len+1);
84         scanf("%s", dict[i].data());
85         dict[i].pop_back();
86         for(char &c : dict[i])
87             c -= 'a';
88     }
89     node *root = aho_corasick(dict);
90     cnt_matches(root, a);
91     add_cnt(root);
92     for(int i=0; i<n; ++i){
93         node *cur = root;
94         for(char c : dict[i])
95             cur = walk(cur, c);
96         printf("%d\n", cur->cnt);
97     }
98 }

```

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

```

1 class AutoNode {
2     private:
3         map< char, AutoNode * > nxt_char; // Map is faster than hashtable
4             → and unsorted arrays
5         int len; //Length of longest suffix in equivalence class.
6         AutoNode *suf;
7         bool has_nxt(char c) const {
8             return nxt_char.count(c);
9         }
10        AutoNode *nxt(char c) { #486
11            if (!has_nxt(c))
12                return NULL;
13            return nxt_char[c];
14        }
15        void set_nxt(char c, AutoNode *node) {
16            nxt_char[c] = node;
17        }
18        AutoNode *split(int new_len, char c) { #952
19            AutoNode *new_n = new AutoNode;
20            new_n->nxt_char = nxt_char;
21            new_n->len = new_len;
22            new_n->suf = suf;
23            suf = new_n;
24            return new_n;
25        }
26        // Extra functions for matching and counting %677
27        AutoNode *lower_depth(int depth) { //move to longest suffix of
28            current with a maximum length of depth.
29            if (suf->len >= depth)
30                return suf->lower_depth(depth);
31            return this;
32        }
33        AutoNode *walk(char c, int depth, int &match_len) { //move to longest #227
34            suffix of walked path that is a substring
35            match_len = min(match_len, len); //includes depth limit(needed for
36            finding matches)
37            if (has_nxt(c)) { //as suffixes are in classes match_len must be
38                tracked externally
39                ++match_len;
40                return nxt(c)->lower_depth(depth);
41            }
42            if (suf)
43                return suf->walk(c, depth, match_len);
44            return this;
45        }
46        int paths_to_end = 0;
47        void set_as_end() { //All suffixes of current node are marked as #955
48            endings nodes.
49            paths_to_end += 1;
50            if (suf) suf->set_as_end();
51        }
52    }
53 }
54 
```

```

47 bool vis = false;
48 void calc_paths_to_end() { //Call ONCE from ROOT. For each node
49   ↵ calculates number of ways to reach an end node.
50   if (!vis) { //paths_to_end is occurrence count for any strings in
51     ↵ current suffix equivalence class.
52     vis = true;
53     for (auto cur : nxt_char) {
54       cur.second->calc_paths_to_end();
55       paths_to_end += cur.second->paths_to_end;
56     }
57   }
58   ↵
59   //Transform into suffix tree of reverse string
60   map<char, AutoNode * > tree_links;
61   int end_dist = 1<<30;
62   int calc_end_dist(){
63     if(end_dist == 1<<30){
64       if(nxt_char.empty())
65         end_dist = 0;
66       for (auto cur : nxt_char)
67         end_dist = min(end_dist, 1+cur.second->calc_end_dist());
68     }
69     return end_dist;
70   }
71   bool vis_t = false;
72   void build_suffix_tree(string &s) { //Call ONCE from ROOT.
73     if (!vis_t) {
74       vis_t = true;
75       if(suf)
76         suf->tree_links[s[s.size()-end_dist-suf->len-1]] = this;
77       for (auto cur : nxt_char)
78         cur.second->build_suffix_tree(s);
79     }
80   struct SufAutomaton {
81     AutoNode *last;
82     AutoNode *root;
83     void extend(char new_c) {
84       AutoNode *new_end = new AutoNode;
85       new_end->len = last->len + 1;
86       AutoNode *suf_w_nxt = last;
87       while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) { #308
88         suf_w_nxt->set_nxt(new_c, new_end);
89         suf_w_nxt = suf_w_nxt->suf;
90       }
91       if (!suf_w_nxt) {
92         new_end->suf = root;
93       } else {
94         AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);
95         if (suf_w_nxt->len + 1 == max_sbstr->len) {
96           new_end->suf = max_sbstr;
97         } else {
98       }
99     }
100   };

```

```

98     AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1,
99         ↵ new_c);
100    new_end->suf = eq_sbstr;
101    AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
102    while (w_edge_to_eq_sbstr != 0 &&
103        ↵ w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
104        ↵ w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
105        ↵ w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
106    }
107    last = new_end;
108}
#035 %996 SufAutomaton(string &s) {
109    root = new AutoNode;
110    root->len = 0;
111    root->suf = NULL;
112    last = root;
113    for (char c : s) extend(c);
114    root->calc_end_dist(); //To build suffix tree use reversed string
115    root->build_suffix_tree(s);
116}
117}
118}; #412 %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):u(_u),v(_v),c(_c),rc(_rc){}
9 };
10 };
#202 #787
11 struct FlowTracker{
12 shared_ptr<ll> flow;
13 ll cap, rcap;
14 bool dir;
15 FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
16 ↵ _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){}
17 ll rem() const {
18 if(dir == 0){
19 return cap-*flow;
20 }
21 else{
22 return rcap+*flow;
23 }
24 void add_flow(ll f){
25 if(dir == 0)
26 *flow += f;
27 else
28 *flow -= f;
#308 #844
#865

```


```

```

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){ #080
41     assert(source != sink);
42 }
43 int add_edge(int u, int v, ll c, ll rc = 0){
44     edges.push_back(Edge(u,v,c,rc));
45     return edges.size()-1;
46 }
47 vector<int> now,lvl;
48 void prep(){
49     int max_id = max(source, sink);
50     for(auto edge : edges)
51         max_id = max(max_id, max(edge.u, edge.v)); #328
52     adj.resize(max_id+1);
53     cap.resize(max_id+1);
54     now.resize(max_id+1);
55     lvl.resize(max_id+1);
56     for(auto &edge : edges){
57         auto flow = make_shared<ll>(0);
58         adj[edge.u].push_back(edge.v);
59         cap[edge.u].push_back(FlowTracker(edge.c, edge.rc, flow, 0));
60         if(edge.u != edge.v){ #717
61             adj[edge.v].push_back(edge.u);
62             cap[edge.v].push_back(FlowTracker(edge.c, edge.rc, flow, 1));
63         }
64         assert(cap[edge.u].back() == edge.c);
65         edge.flow = flow;
66     }
67 }
68 bool dinic_bfs(){
69     fill(now.begin(),now.end(),0);
70     fill(lvl.begin(),lvl.end(),0); #038
71     lvl[source] = 1;
72     vector<int> bfs(1,source);
73     for(int i = 0; i < bfs.size(); ++i){
74         int u = bfs[i];
75         for(int j = 0; j < adj[u].size(); ++j){
76             int v = adj[u][j];
77             if(cap[u][j] > 0 && lvl[v] == 0){
78                 lvl[v] = lvl[u]+1;
79                 bfs.push_back(v);
80             }
81         }
82     }
#287
83     return lvl[sink] > 0;
84 }
85 ll dinic_dfs(int u, ll flow){ #014
86     if(u == sink)
87         return flow;
88     while(now[u] < adj[u].size()){
89         int v = adj[u][now[u]];
90         if(lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0){
91             ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
92             if(res > 0){
93                 cap[u][now[u]] -= res;
94                 return res;
95             }
96         }
97         ++now[u];
98     }
99     return 0;
100 }
101 ll calc_max_flow(){ #197
102     prep();
103     ll ans = 0;
104     while(dinic_bfs()){ #817
105         ll cur = 0;
106         do{
107             cur = dinic_dfs(source,INF);
108             ans += cur;
109         }while(cur > 0);
110     }
111     return ans;
112 }
113 ll flow_on_edge(int edge_index){ #583
114     assert(edge_index < edges.size());
115     return *edges[edge_index].flow;
116 }
117 };
118 int main(){
119     int n,m;
120     cin >> n >> m;
121     vector<pair<int, pair<int, int>>> graph(m);
122     for(int i=0; i<m; ++i){
123         cin >> graph[i].second.first >> graph[i].second.second >> graph[i].first;
124     }
125     ll res=0;
126     for(auto cur : graph){
127         auto mf = MaxFlow(cur.second.first,cur.second.second); // arguments
128         ↳ source and sink, memory usage O(largest node index + input
129         ↳ size), sink doesn't need to be last index
130         for(int i = 0; i < m; ++i){
131             if(graph[i].first > cur.first){
132                 mf.add_edge(graph[i].second.first,graph[i].second.second,1,1);
133                 ↳ // store edge index if care about flow value
134             }
135         }
136     }
137 }
```

```

133     res += mf.calc_max_flow();
134 }
135 cout<<res<<endl;
136 }



---


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

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]; //set [x][y] for directed capacity from x to
  ↪ y.
5 ll cost[nmax][nmax]; //set [x][y] for directed cost from x to y. SET TO
  ↪ inf IF NOT USED
6 ll min_dist[nmax];
7 int prev_node[nmax];
8 ll node_flow[nmax];
9 bool visited[nmax];
10 ll tot_cost, tot_flow; //output
11 void min_cost_max_flow(){ %576
    tot_cost=0; //Does not work with negative cycles.
12    tot_flow=0;
13    ll sink_pot=0;
14    min_dist[0] = 0; %927
15    for(int i=1; i<=v; ++i){ //in case of no negative edges Bellman-Ford
      ↪ can be removed.
16        min_dist[i]=inf;
17    }
18    for(int i=0; i<v-1; ++i){
19        for(int j=0; j<v; ++j){
20            for(int k=0; k<v; ++k){
21                if(rem_flow[j][k] > 0 && min_dist[j]+cost[j][k] < min_dist[k])
22                    min_dist[k] = min_dist[j]+cost[j][k];
23            }
24        }
25    } #599
26    for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
27        for(int j=0; j<v; ++j){
28            if(cost[i][j]!=inf){
29                cost[i][j]+=min_dist[i];
30                cost[i][j]-=min_dist[j];
31            }
32        }
33    }
34    sink_pot+=min_dist[v-1]; //Bellman-Ford end. %849
35    while(true){
36        for(int i=0; i<=v; ++i){ //node after sink is used as start value
          ↪ for Dijkstra.
37            min_dist[i]=inf;
38            visited[i]=false;
39        }
40        min_dist[0]=0;
41        node_flow[0]=inf;
42        int min_node;
43        while(true){ //Use Dijkstra to calculate potentials

```

```

45        int min_node=v; #782
46        for(int i=0; i<v; ++i){
47            if((!visited[i]) && min_dist[i]<min_dist[min_node])
48                min_node=i;
49        }
50        if(min_node==v) break;
51        visited[min_node]=true;
52        for(int i=0; i<v; ++i){
53            if((!visited[i]) && min_dist[min_node]+cost[min_node][i] <
54              min_dist[i]){
55                min_dist[i]=min_dist[min_node]+cost[min_node][i];
56                prev_node[i]=min_node; #881
57                node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
58            }
59        }
60        if(min_dist[v-1]==inf) break;
61        for(int i=0; i<v; ++i){ //Apply potentials to edge costs.
62            for(int j=0; j<v; ++j){ //Found path from source to sink becomes
              ↪ 0 cost.
63                if(cost[i][j]!=inf){
64                    cost[i][j]+=min_dist[i];
65                    cost[i][j]-=min_dist[j]; #083
66                }
67            }
68            sink_pot+=min_dist[v-1];
69            tot_flow+=node_flow[v-1];
70            tot_cost+=sink_pot*node_flow[v-1];
71            int cur=v-1;
72            while(cur!=0){ //Backtrack along found path that now has 0 cost.
73                rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
74                rem_flow[cur][prev_node[cur]]+=node_flow[v-1]; #582
75                cost[cur][prev_node[cur]]=0;
76                if(rem_flow[prev_node[cur]][cur]==0)
77                    cost[prev_node[cur]][cur]=inf;
78                cur=prev_node[cur];
79            }
80        }
81    }
82 }
83 int main(){//http://www.spoj.com/problems/GREED/
84     cin>>t;
85     for(int i=0; i<t; ++i){
86         cin>>n;
87         for(int j=0; j<nmax; ++j){
88             for(int k=0; k<nmax; ++k){
89                 cost[j][k]=inf;
90                 rem_flow[j][k]=0;
91             }
92         }
93         for(int j=1; j<=n; ++j){
94             cost[j][2*n+1]=0;
95             rem_flow[j][2*n+1]=1;

```

```

96 }
97 for(int j=1; j<=n; ++j){
98     int card;
99     cin>>card;
100    ++rem_flow[0][card];
101    cost[0][card]=0;
102 }
103 int ex_c;
104 cin>>ex_c;
105 for(int j=0; j<ex_c; ++j){
106     int a, b;
107     cin>>a>>b;
108     if(b<a) swap(a,b);
109     cost[a][b]=1;
110     rem_flow[a][b]=nmax;
111     cost[b][n+b]=0;
112     rem_flow[b][n+b]=nmax;
113     cost[n+b][a]=1;
114     rem_flow[n+b][a]=nmax;
115 }
116 v=2*n+2;
117 min_cost_max_flow();
118 cout<<tot_cost<<'\n';
119 }
120 }

28     int index;
29 };
30 deque<Node> nodes;
31 deque<Edge> edges;
32 Node* addNode() {
33     nodes.push_back(Node());
34     nodes.back().index = nodes.size()-1;
35     return &nodes.back();
36 }
37 Edge* addEdge(Node* u, Node* v, int f, int c, int cost) { #534
38     edges.push_back({u, v, f, c, cost});
39     u->conn.push_back(&edges.back());
40     v->conn.push_back(&edges.back());
41     return &edges.back();
42 }
43 //Assumes all needed flow has already been added
44 int minCostMaxFlow() { #507
45     int n = nodes.size();
46     int result = 0;
47     struct State {
48         int p;
49         Edge* used;
50     };
51     while(1) { #877
52         vector<vector<State> > state(1, vector<State>(n, {0, 0}));
53         for(int lev = 0; lev < n; lev++) {
54             state.push_back(state[lev]);
55             for(int i=0;i<n;i++){
56                 if(lev == 0 || state[lev][i].p < state[lev-1][i].p) { #42
57                     for(Edge* edge : nodes[i].conn){
58                         if(edge->getCap(&nodes[i]) > 0) {
59                             int np = state[lev][i].p + (edge->u == &nodes[i] ? #281
60                                 -edge->cost : edge->cost);
61                             int ni = edge->from(&nodes[i])->index;
62                             if(np < state[lev+1][ni].p) { #281
63                                 state[lev+1][ni].p = np;
64                                 state[lev+1][ni].used = edge;
65                             }
66                         }
67                     }
68                 }
69             }
70             //Now look at the last level
71             bool valid = false;
72             for(int i=0;i<n;i++) { #283
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> expr(n, false);

```

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)
8                 return v;
9             return u;
10        } #042
11        int getCap(Node* pos) {
12            if(pos == u)
13                return c-f;
14            return f;
15        }
16        int addFlow(Node* pos, int toAdd) { #965
17            if(pos == u) {
18                f += toAdd;
19                return toAdd * cost;
20            } else {
21                f -= toAdd;
22                return -toAdd * cost;
23            }
24        }
25    };
26    struct Node {
27        vector<Edge*> conn;

```

```

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    if(!valid) break;
105 } #954
106 return result;
107 } #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 *>> comp; //Can use inline static
18     static vector<Circle> to_process;
19     static bool no_dmst;
20     static Node *root;
21     struct Node{
22         Node *par = NULL;
23         vector<pair<int, int>> out_cands; //Circ, edge idx
24         vector<pair<ll, Edge *>> con;
25         bool in_use = false;
26         ll w = 0; //extra to add to edges in con
27         Node *anc(){
28             if(!par)
29                 return this;
30             while(par->par)
31                 par = par->par;
32             return par;
33         }
34         void clean(){
35             if(!no_dmst){
36                 in_use = false;
37                 for(auto &cur : out_cands)
38                     to_process[cur.first].clean(cur.second);
39             }
40         }
41         Node *con_to_root(){
42             if(anc() == root)
43                 return root;
44             in_use = true;
45             Node *super = this; //Will become root or the first Node
46             //encountered in a loop.
47             while(super == this){
48                 while(!con.empty() && con.front().second->tar->anc() == anc()){
49                     pop_heap(con.begin(), con.end(), comp);
50                     con.pop_back();
51                 }
52                 if(con.empty()){
53                     no_dmst = true;
54                     return root;
55                 }
56                 pop_heap(con.begin(), con.end(), comp);
57                 auto nxt = con.back();
58                 con.pop_back();
59                 w = -nxt.first;
60                 if(nxt.second->tar->in_use){ //anc() wouldn't change anything
61                     super = nxt.second->tar->anc(); #174
62                     to_process.resize(to_process.size()+1);
63                 } else {
64                     super = nxt.second->tar->con_to_root();
65                 }
66                 if(super != root){
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         }
76     };
77 };
78

```

```

73     if(super != root){ //we are some loops non first Node.
74         if(con.size() > super->con.size()){
75             swap(con, super->con); //Largest con in loop should not be
76             ↪ copied.
77             swap(w, super->w);
78         }
79         for(auto cur : con){
80             super->con.emplace_back(cur.first - super->w + w,
81             ↪ cur.second);
82             push_heap(super->con.begin(), super->con.end(), comp); #375
83         }
84     }
85     par = super; //root or anc() of first Node encountered in a loop
86     return super;
87 }
88 Node *cur_root;
89 vector<Node> graph;
90 vector<Edge> edges;
91 DMST(int n, vector<EdgeDesc> &desc, int r){ //Self loops and multiple
92     ↪ edges are okay. #076
93     graph.resize(n);
94     cur_root = &graph[r];
95     for(auto &cur : desc) //Edges are reversed internally
96         edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
97     for(int i=0; i<desc.size(); ++i)
98         graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]);
99     for(int i=0; i<n; ++i)
100        make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
101 }
102 bool find(){ #469
103     root = cur_root;
104     no_dmst = false;
105     for(auto &cur : graph){
106         cur.con_to_root();
107         to_process.clear();
108         if(no_dmst) return false;
109     }
110     return true;
111 }
112 ll weight(){ #732
113     ll res = 0;
114     for(auto &cur : edges){
115         if(cur.inc)
116             res += cur.w;
117     }
118 }
119 void DMST::Circle::clean(int idx){
120     if(!vis){
121         vis = true;
122         for(int i=0; i<contents.size(); ++i){

```

```

123             if(i != idx){
124                 contents[i]->inc = true;
125                 contents[i]->from->clean();
126             }
127         }
128     }
129 }
130 const greater<pair<ll, DMST::Edge *>> DMST::comp;
131 vector<DMST::Circle> DMST::to_process;
132 bool DMST::no_dmst;
133 DMST::Node *DMST::root; #771 %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){
7 rev = NULL;
8 }
9 vert &operator*(){
10 return *dest;
11 }
12 vert *operator->(){
13 return dest;
14 }
15 bool is_bridge();
16 };
17 struct vert{
18 deque<edge> con;
19 int val = 0;
20 int seen;
21 int dfs(int upd, edge *ban){ //handles multiple edges #336
22 if(!val){
23 val = upd;
24 seen = val;
25 for(edge &nxt : con){
26 if(nxt.exists && (&nxt) != ban)
27 seen = min(seen, nxt->dfs(upd+1, nxt.rev));
28 }
29 }
30 return seen;
31 }
32 void remove_adj_bridges(){ #673 %624
33 for(edge &nxt : con){
34 if(nxt.is_bridge())
35 nxt.exists = false;
36 }
37 }
38 int cnt_adj_bridges(){ #106
39 int res = 0;
40 for(edge &nxt : con)
41 res += nxt.is_bridge();

```


```

```

42     return res;
43 }
44 };
45 bool edge::is_bridge(){
46     return exists && (dest->seen > rev->dest->val || dest->val <
47         ~rev->dest->seen);
48 }
49 vert graph[nmax];
50 int main(){ //Mechanics Practice BRIDGES
51     int n, m;
52     cin>>n>>m;
53     for(int i=0; i<m; ++i){
54         int u, v;
55         scanf("%d %d", &u, &v);
56         graph[u].con.emplace_back(graph+v);
57         graph[v].con.emplace_back(graph+u);
58         graph[u].con.back().rev = &graph[v].con.back();
59         graph[v].con.back().rev = &graph[u].con.back();
60     }
61     graph[1].dfs(1, NULL);
62     int res = 0;
63     for(int i=1; i<=n; ++i)
64         res += graph[i].cnt_adj_bridges();
65     cout<<res/2<<endl;
}

```

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 nsiz) {
5         n = nsiz;
6         conn.resize(n);
7     }
8     void add_edge(int u, int v) {
9         conn[u].push_back(v);
10    }
11    void _topsort_dfs(int pos, vector<int> &result, vector<bool>
12        &explr, vector<vector<int> > &revconn) {
13        if(explr[pos])
14            return;
15        explr[pos] = true;
16        for(auto next : revconn[pos])
17            _topsort_dfs(next, result, explr, revconn);
18        result.push_back(pos);
19    }
20    vector<int> topsort() {
21        vector<vector<int> > revconn(n);
22        for(int u = 0; u < n; u++) {
23            for(auto v : conn[u])
24                revconn[v].push_back(u);
25        }
26        vector<int> result;
27        vector<bool> explr(n, false);

```

```

%056
%223
27        for(int i=0; i < n; i++)
28            _topsort_dfs(i, result, explr, revconn);
29        reverse(result.begin(), result.end());
30        return result;
#991
31    }
32    void dfs(int pos, vector<int> &result, vector<bool> &explr) {
33        if(explr[pos])
34            return;
35        explr[pos] = true;
36        for(auto next : conn[pos])
37            dfs(next, result, explr);
38        result.push_back(pos);
39    }
#603
40    vector<vector<int> > scc(){ // tested on
41        vector<int> order = topsort();
42        reverse(order.begin(),order.end());
43        vector<bool> explr(n, false);
44        vector<vector<int> > results;
45        for(auto it = order.rbegin(); it != order.rend(); ++it){
46            vector<int> component;
47            _topsort_dfs(*it,component,explr,conn);
48            sort(component.begin(),component.end());
49            results.push_back(component);
50        }
51        sort(results.begin(),results.end());
52        return results;
53    }
#741
54    };
#983
55    //Solution for:
56    // http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
56    int main() {
57        int n, m;
58        cin >> n >> m;
59        Graph g(2*m);
60        for(int i=0; i<n; i++) {
61            int a, sa, b, sb;
62            cin >> a >> sa >> b >> sb;
63            a--, b--;
64            g.add_edge(2*a + 1 - sa, 2*b + sb);
65            g.add_edge(2*b + 1 - sb, 2*a + sa);
66        }
67        vector<int> state(2*m, 0);
68        {
69            vector<int> order = g.toposrt();
70            vector<bool> explr(2*m, false);
71            for(auto u : order) {
72                vector<int> traversed;
73                g.dfs(u, traversed, explr);
74                if(traversed.size() > 0 && !state[traversed[0]^1]) {
75                    for(auto c : traversed)
76                        state[c] = 1;
77                }

```

```

78     }
79 }
80 for(int i=0; i < m; i++) {
81     if(state[2*i] == state[2*i+1]) {
82         cout << "IMPOSSIBLE\n";
83         return 0;
84     }
85 }
86 for(int i=0; i < m; i++) {
87     cout << state[2*i+1] << '\n';
88 }
89 return 0;
90 }

```

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, int
4     ↪ rhs_len){
5         sum = lhs_seg.sum + rhs_seg.sum;
6     }
6 } __attribute__((packed));
7 struct Lazy{
8     ll add;
9     ll assign_val; //LLONG_MIN if no assign;
10    void init(){                                #883
11        add = 0;
12        assign_val = LLONG_MIN;
13    }
14 Lazy(){ init(); }
15 void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len){
16     lhs_lazy = *this;
17     rhs_lazy = *this;
18     init();
19 }
20 void merge(Lazy &oth, int len){                #470
21     if(oth.assign_val != LLONG_MIN){
22         add = 0;
23         assign_val = oth.assign_val;
24     }
25     add += oth.add;
26 }
27 void apply_to_seg(Seg &cur, int len) const{
28     if(assign_val != LLONG_MIN){
29         cur.sum = len * assign_val;
30     }
31     cur.sum += len * add;
32 }
33 } __attribute__((packed));
34 struct Node{ //Following code should not need to be modified
35     int ver;
36     bool is_lazy = false;
37     Seg seg;
38     Lazy lazy;

```

```

39     Node *lc=NULL, *rc=NULL;
40     void init(){
41         if(!lc){
42             lc = new Node {ver};
43             rc = new Node {ver};
44         }
45     }
46     Node *upd(int L, int R, int l, int r, Lazy &val, int tar_ver){      #313
47         if(ver != tar_ver){
48             Node *rep = new Node(*this);
49             rep->ver = tar_ver;
50             return rep->upd(L, R, l, r, val, tar_ver);
51         }
52         if(L >= l && R <= r){                                         #138
53             val.apply_to_seg(seg, R-L);
54             lazy.merge(val, R-L);
55             is_lazy = true;
56         } else {
57             init();
58             int M = (L+R)/2;
59             if(is_lazy){
60                 Lazy l_val , r_val;
61                 lazy.split(l_val, r_val, R-L);
62                 lc = lc->upd(L, M, l, M, l_val, ver);
63                 rc = rc->upd(M, R, M, r, r_val, ver);          #104
64                 is_lazy = false;
65             }
66             Lazy l_val , r_val;
67             val.split(l_val, r_val, R-L);
68             if(l < M)
69                 lc = lc->upd(L, M, l, r, l_val, ver);
70             if(M < r)
71                 rc = rc->upd(M, R, l, r, r_val, ver);
72             seg.recalc(lc->seg, M-L, rc->seg, R-M);          #245
73         }
74         return this;
75     }
76     void get(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp, bool
77     ↪ last_ver){                                              #726
78         if(L >= l && R <= r){
79             tmp->recalc(*lft_res, L-l, seg, R-L);
80             swap(lft_res, tmp);
81         } else {
82             init();
83             int M = (L+R)/2;
84             if(is_lazy){                                         #726
85                 Lazy l_val , r_val;
86                 lazy.split(l_val, r_val, R-L);
87                 lc = lc->upd(L, M, l, M, l_val, ver+last_ver);
88                 lc->ver = ver;
89                 rc = rc->upd(M, R, M, r, r_val, ver+last_ver);
90                 rc->ver = ver;
91                 is_lazy = false;
92             }
93         }
94     }

```

```

91     }
92     if(l < M)
93         lc->get(L, M, l, r, lft_res, tmp, last_ver);
94     if(M < r)
95         rc->get(M, R, l, r, lft_res, tmp, last_ver);
96     }
97 }
98 } __attribute__((packed));
99 struct SegTree{ //indexes start from 0, ranges are [beg, end)
100    vector<Node *> roots; //versions start from 0
101    int len;
102    SegTree(int _len) : len(_len){
103        roots.push_back(new Node {0});
104    }
105    int upd(int l, int r, Lazy &val, bool new_ver = false){
106        Node *cur_root = roots.back()->upd(0, len, l, r, val,
107                                         roots.size()-!new_ver);
108        if(cur_root != roots.back())
109            roots.push_back(cur_root);
110        return roots.size()-1;
111    }
112    Seg get(int l, int r, int ver = -1){
113        if(ver == -1)
114            ver = roots.size()-1;
115        Seg seg1, seg2;
116        Seg *pres = &seg1, *ptmp = &seg2;
117        roots[ver]->get(0, len, l, r, pres, ptmp, roots.size()-1);
118    }
119 };
120 int main(){
121    int n, m; //solves Mechanics Practice LAZY
122    cin>>n>>m;
123    SegTree seg_tree(1<<17);
124    for(int i=0; i<n; ++i){
125        Lazy tmp;
126        scanf("%lld", &tmp.assign_val);
127        seg_tree.upd(i, i+1, tmp);
128    }
129    for(int i=0; i<m; ++i){
130        int o;
131        int l, r;
132        scanf("%d %d %d", &o, &l, &r);
133        --l;
134        if(o==1){
135            Lazy tmp;
136            scanf("%lld", &tmp.add);
137            seg_tree.upd(l, r, tmp);
138        } else if(o==2){
139            Lazy tmp;
140            scanf("%lld", &tmp.assign_val);
141            seg_tree.upd(l, r, tmp);
142        } else {
143            Seg res = seg_tree.get(l, r);
144        }
145    }
146 }
147 }
```

#696 #295 #977 #542

```

144     printf("%lld\n",res.sum);
145 }
146 }
147 }
```

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

```

1 class dummy {
2 public:
3     dummy () {}
4     dummy (int, int) {}
5     void set (int, int) {}
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
13  * group operation. It should have the following methods:
14  * * DS () - empty constructor
15  * * DS (int size, T initial) - constructs the structure with the given
16  * size,
17  * * initially filled with initial.
18  * * void set (int index, T value) - set the value at index `index` to
19  * value
20  * * T query (int left, int right) - return the "sum" of elements
21  * between left and right, inclusive.
22  */
23 template<typename T, class DS>
24 class HLD {
25     int vertexc;
26     vector<int> *adj;
27     vector<int> subtree_size;
28     DS structure;
29     DS aux;
30     void build_sizes (int vertex, int parent) {
31         subtree_size[vertex] = 1;
32         for (int child : adj[vertex]) {
33             if (child != parent) {
34                 build_sizes(child, vertex);
35                 subtree_size[vertex] += subtree_size[child];
36             }
37         }
38     }
39     int cur;
40     vector<int> ord;
41     vector<int> chain_root;
42     vector<int> par;
43     void build_hld (int vertex, int parent, int chain_source) {
44         cur++;
45         ord[vertex] = cur;
46         chain_root[vertex] = chain_source;
47         par[vertex] = parent;
48     }
49 }
```

%932 #037 #593

```

44     if (adj[vertex].size() > 1 || (vertex == 1 && adj[vertex].size() ==  

45         1)) {  

46         int big_child, big_size = -1;  

47         for (int child : adj[vertex]) {  

48             if ((child != parent) && (subtree_size[child] > big_size)) {  

49                 big_child = child;                                         #151  

50                 big_size = subtree_size[child];  

51             }  

52         build_hld(big_child, vertex, chain_source);  

53         for (int child : adj[vertex]) {  

54             if ((child != parent) && (child != big_child))  

55                 build_hld(child, vertex, child);  

56         }  

57     }  

58 }  

59 public:  

60     HLD (int _vertexc) {  

61         vertexc = _vertexc;  

62         adj = new vector<int> [vertexc + 5];  

63     }  

64     void add_edge (int u, int v) {  

65         adj[u].push_back(v);  

66         adj[v].push_back(u);  

67     }  

68     void build (T initial) {  

69         subtree_size = vector<int> (vertexc + 5);  

70         ord = vector<int> (vertexc + 5);  

71         chain_root = vector<int> (vertexc + 5);  

72         par = vector<int> (vertexc + 5);  

73         cur = 0;  

74         build_sizes(1, -1);  

75         build_hld(1, -1, 1);  

76         structure = DS (vertexc + 5, initial);  

77         aux = DS (50, initial);  

78     }  

79     void set (int vertex, int value) {  

80         structure.set(ord[vertex], value);  

81     }  

82     T query_path (int u, int v) { /* returns the "sum" of the path u->v  

83         */  

84         int cur_id = 0;  

85         while (chain_root[u] != chain_root[v]) {  

86             if (ord[u] > ord[v]) {  

87                 cur_id++;  

88                 aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));  

89                 u = par[chain_root[u]];                                         #538  

90             } else {  

91                 cur_id++;  

92                 aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));  

93                 v = par[chain_root[v]];  

94             }  

95             cur_id++;
96         aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u],  

97             ord[v])));  

98         return aux.query(1, cur_id);                                         %905
99     }  

100    void print () {  

101        for (int i = 1; i <= vertexc; i++)  

102            cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' <<  

103            par[i] << endl;
104    }
105    int main () {
106        int vertexc;
107        cin >> vertexc;
108        HLD<int, dummy> hld (vertexc);
109        for (int i = 0; i < vertexc - 1; i++) {
110            int u, v;
111            cin >> u >> v;
112            hld.add_edge(u, v);
113        }
114        hld.build(0);
115        hld.print();
116        int queryc;
117        cin >> queryc;
118        for (int i = 0; i < queryc; i++) {
119            int u, v;
120            cin >> u >> v;
121            hld.query_path(u, v);
122            cout << endl;
123        }
124    }

```

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

```

1 // Fully overloaded any dimensional BIT, use any type for coordinates,  

2 // elements, return_value.  

3 template < typename elem_t, typename coord_t, coord_t n_inf, typename  

4     > class BIT {  

5     vector< coord_t > positions;  

6     vector< elem_t > elems;  

7     bool initiated = false;  

8     public:  

9     BIT() {  

10        positions.push_back(n_inf);  

11    }  

12    void initiate() {  

13        if (initiated) {  

14            for (elem_t &c_ele : elems)  

15                c_ele.initiate();  

16        } else {  

17            initiated = true;  

18            sort(positions.begin(), positions.end());  

19            positions.resize(unique(positions.begin(), positions.end()) -  

20                           positions.begin());

```

```

20     elems.resize(positions.size());
21 }
22 } #036
23 template < typename... loc_form >
24 void update(coord_t cord, loc_form... args) {
25     if (initiated) {
26         int pos = 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 } #154
34 template < typename... loc_form >
35 ret_t query(coord_t cord, loc_form... args) { //sum in open interval
36     ← (-inf, cord)
37     ret_t res = 0;
38     int pos = (lower_bound(positions.begin(), positions.end(), cord) -
39     positions.begin())-1;
40     for (; pos > 0; pos -= pos & -pos)
41         res += elems[pos].query(args...);
42     return res;
43 }
44 template < typename internal_type >
45 struct wrapped {
46     internal_type a = 0;
47     void update(internal_type b) {
48         a += b;
49     }
50     internal_type query() {
51         return a;
52     }
53     // Should never be called, needed for compilation
54     void initiate() {
55         cerr << 'i' << endl;
56     }
57     void update() {
58         cerr << 'u' << endl;
59     }
60 };
61 int main() {
62     // return type should be same as type inside wrapped
63     BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll >
64     ← fenwick;
65     int dim = 2;
66     vector< tuple< int, int, ll > > to_insert;
67     to_insert.emplace_back(1, 1, 1);
68     // set up all positions that are to be used for update
69     for (int i = 0; i < dim; ++i) {
70         for (auto &cur : to_insert)
71             fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include
72             ← value which won't be used

```

```

73         fenwick.initiate();
74     }
75     // actual use
76     for (auto &cur : to_insert)
77         fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
78     cout << fenwick.query(2, 2) << '\n';
79 }



---



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



```

1 mt19937 randgen;
2 struct Treap {
3 struct Node {
4 int key;
5 int value;
6 unsigned int priority;
7 long long total;
8 Node* lch;
9 Node* rch;
10 Node(int new_key, int new_value) { #698
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() { #295
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) { #233
27 if(cur == 0) return {0, 0};
28 pair<Node*, Node*> result;
29 if(key <= cur->key) {
30 auto ret = split(key, cur->lch);
31 cur->lch = ret.second;
32 result = {ret.first, cur};
33 } else {
34 auto ret = split(key, cur->rch);
35 cur->rch = ret.first;
36 result = {cur, ret.second};
37 }
38 cur->update();
39 return result;
40 }
41 Node* merge(Node* left, Node* right) { #230
42 if(left == 0) return right;
43 if(right == 0) return left;
44 Node* top;

```


```

```

45     if(left->priority < right->priority) {
46         left->rch = merge(left->rch, right);
47         top = left;
48     } else {
49         right->lch = merge(left, right->lch);
50         top = right;
51     }
52     top->update();
53     return top;
54 }
55 void insert(int key, int value) {
56     nodes.push_back(Node(key, value));
57     Node* cur = &nodes.back();
58     pair<Node*, Node*> ret = split(key, root);
59     cur = merge(ret.first, cur);
60     cur = merge(cur, ret.second);
61     root = cur;
62 }
63 void erase(int key) {
64     Node *left, *mid, *right;
65     tie(left, mid) = split(key, root);
66     tie(mid, right) = split(key+1, mid);
67     root = merge(left, right);
68 }
69 long long sum_upto(int key, Node* cur) {
70     if(cur == 0) return 0;
71     if(key <= cur->key) {
72         return sum_upto(key, cur->lch);
73     } else {
74         long long result = cur->value + sum_upto(key, cur->rch);
75         if(cur->lch) result += cur->lch->total;
76         return result;
77     }
78 }
79 long long get(int l, int r) {
80     return sum_upto(r+1, root) - sum_upto(l, root); #509
81 }
82 };
83 //Solution for:
84 int main() {
85     ios_base::sync_with_stdio(false);
86     cin.tie(0);
87     int m;
88     Treap treap;
89     cin >> m;
90     for(int i=0;i<m;i++) {
91         int type;
92         cin >> type;
93         if(type == 1) {
94             int x, y;
95             cin >> x >> y;
96             treap.insert(x, y);
97         } else if(type == 2) {
#510
#760
#634
#959
#509
#947
#770
#018
#20

```

```

98         int x;
99         cin >> x;
100        treap.erase(x);
101    } else {
102        int l, r;
103        cin >> l >> r;
104        cout << treap.get(l, r) << endl;
105    }
106 }
107 return 0;
108 }
```

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(T *start, T *sec_start, int arr_size, int
4     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){ #770
20        uchar bucket_id = ((*it)>>(d*8))&mask;
21        *(locs[bucket_id]++) = *it;
22    }
23    locs = locs_mem;
24    if(d){
25        T *locs_old[256];
26        locs_old[0] = start;
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){ #018
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 }
```

```

40 }
41 }
42 const int nmax = 5e7;
43 ll arr[nmax], tmp[nmax];
44 int main(){
45     for(int i=0; i<nmax; ++i)
46         arr[i] = ((ll)rand()<<32)|rand();
47     msd_radixsort(arr, tmp, nmax);
48     assert(is_sorted(arr, arr+nmax));
49 }

```

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}; #384
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{lft.a * rgt.a - lft.b * rgt.b, lft.a * rgt.b + lft.b * #380
17        rgt.a};
18 Complex conj(const Complex &cur){
19     return Complex{cur.a, -cur.b};
20 }
21 void fft_rec(Complex *arr, Complex *root_pow, int len) { #957
22     if (len != 1) {
23         fft_rec(arr, root_pow, len >> 1);
24         fft_rec(arr + len, root_pow, len >> 1);
25     }
26     root_pow += len;
27     for (int i = 0; i < len; ++i) {
28         Complex tmp = arr[i] + root_pow[i] * arr[i + len];
29         arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
30         arr[i] = tmp; #048
31     }
32 }
33 void fft(vector< Complex > &arr, int ord, bool invert) {
34     assert(arr.size() == 1 << ord);
35     static vector< Complex > root_pow(1);
36     static int inc_pow = 1;
37     static bool is_inv = false;
38     if (inc_pow <= ord) {
39         int idx = root_pow.size();

```

```

40         root_pow.resize(1 << ord); #710
41         for (; inc_pow <= ord; ++inc_pow) {
42             for (int idx_p = 0; idx_p < 1 << (ord - 1); idx_p += 1 << (ord - #750
43                 - inc_pow), ++idx) {
44                 root_pow[idx] =
45                     Complex{cos(-idx_p * M_PI / (1 << (ord - 1))), sin(-idx_p *
46                         M_PI / (1 << (ord - 1)))};
47                 if (is_inv) root_pow[idx].b = -root_pow[idx].b;
48             }
49         }
50     }
51     if (invert != is_inv) {
52         is_inv = invert;
53         for (Complex &cur : root_pow) cur.b = -cur.b; #844
54     }
55     for (int i = 1, j=0; i < (1 << ord); ++i) {
56         int m = 1<<(ord-1);
57         bool cont = true;
58         while(cont){ #380
59             cont = j & m;
60             j ^= m;
61             m>>=1;
62         }
63         if (i < j) swap(arr[i], arr[j]);
64     }
65     fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
66     if (invert) #811
67         for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord);
68     void mult_poly_mod(vector< int > &a, vector< int > &b, vector< int > #809
69         &c) { // c += a*b
70         static vector< Complex > arr[4]; // correct upto 0.5-2M elements(mod
71         ~~= 1e9)
72         if (c.size() < 400) {
73             for (int i = 0; i < a.size(); ++i)
74                 for (int j = 0; j < b.size() && i + j < c.size(); ++j)
75                     c[i + j] = ((ll)a[i] * b[j] + c[i + j]) % mod;
76         } else {
77             int fft_ord = 32 - __builtin_clz(c.size());
78             if (arr[0].size() != 1 << fft_ord)
79                 for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord); #811
80             for (int i = 0; i < 4; ++i) fill(arr[i].begin(), arr[i].end(),
81                 Complex{});
82             for (int &cur : a)
83                 if (cur < 0) cur += mod;
84             for (int &cur : b)
85                 if (cur < 0) cur += mod;
86             const int shift = 15;
87             const int mask = (1 << shift) - 1;
88             for (int i = 0; i < min(a.size(), c.size()); ++i) {
89                 arr[0][i].a = a[i] & mask;
90                 arr[1][i].a = a[i] >> shift;
91             }
92         }
93     }

```

```

88     for (int i = 0; i < min(b.size(), c.size()); ++i) { %144
89         arr[0][i].b = b[i] & mask;
90         arr[1][i].b = b[i] >> shift;
91     }
92     for (int i = 0; i < 2; ++i) fft(arr[i], fft_ord, false);
93     for (int i = 0; i < 2; ++i) {
94         for (int j = 0; j < 2; ++j) {
95             int tar = 2 + (i + j)/2;
96             Complex mult = {0, -0.25};
97             if(i^j)
98                 mult = {0.25, 0};
99             for (int k = 0; k < (1 << fft_ord); ++k){
100                 int rev_k = ((1 << fft_ord)-k)%(1 << fft_ord);
101                 Complex ca = arr[i][k] + conj(arr[i][rev_k]);
102                 Complex cb = arr[j][k] - conj(arr[j][rev_k]);
103                 arr[tar][k] = arr[tar][k] + mult*ca*cb;
104             }
105         }
106     } #623
107     for (int i = 2; i < 4; ++i) {
108         fft(arr[i], fft_ord, true);
109         for (int k = 0; k < (int)c.size(); ++k){
110             c[k] = (c[k] + (((ll)(arr[i][k].a + 0.5) % mod) << (shift *
111             ↵ 2*(i - 2)))) % mod;
112             c[k] = (c[k] + (((ll)(arr[i][k].b + 0.5) % mod) << (shift *
113             ↵ (2*(i - 2)+1)))) % mod;
114         }
115     } %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) #290
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) #780
12    }
13    ull sqp1(ull a) { return multf(a, a) + 1; }
14 };
15    ull pow_mod(ull a, ull n, ModArithm &arithm) { #493
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 = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
24     ↵ 37}; %144
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); #356
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) { %975
45        ModArithm arithm(n);
46        int cum_cnt = 64 - __builtin_clz(n);
47        cum_cnt *= cum_cnt / 5 + 1;
48        while (true) {
49            ll lv = rand() % n;
50            ll v = arithm.sqp1(lv);
51            int idx = 1;
52            int tar = 1;
53            while (true) {
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) { #912
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                } else { #208
74                    break;
75                }
76            }
77        }
78    }

```

```

74     ll g = __gcd(cur, n);
75     if (g != 1) return g;
76 }
77 v = v_cur;
78 idx += j_stop;
79 if (idx == tar) {
80     lv = v;
81     tar *= 2;
82     v = arithm.sqp1(v);
83     ++idx;
84 }
85 }
86 }
87 }
88 map< ll, int > prime_factor(ll n, map< ll, int > *res = NULL) { // n
89     if (!res) {
90         map< ll, int > res_act;
91         for (int p : small_primes) {
92             while (!(n % p)) {
93                 ++res_act[p];
94                 n /= p;
95             }
96         }
97         if (n != 1) prime_factor(n, &res_act); #023
98         return res_act;
99     }
100    if (is_prime(n)) {
101        ++(*res)[n];
102    } else {
103        ll factor = pollard_rho(n);
104        prime_factor(factor, res);
105        prime_factor(n / factor, res);
106    }
107    return map< ll, int >(); #140
108 } //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)

```