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

Contents

1 Setup

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

2 crc.sh

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

3 gcc ordered set

```
#include <bits/stdc++.h>
typedef long long   ll;
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
int main(){
  ordered_set<int>  cur;                                          #221
  cur.insert(1);
  cur.insert(3);
  cout << cur.order_of_key(2) << endl; // the number of elements in the set less than 2
  cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in the set(0-based)
  cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in the set(0-based)
}                                                                 %574
```

4 2D geometry

Define $\mathbf{orient}(A,B,C) = \overline{AB} \times \overline{AC}$. CCW iff > 0.
Define $\mathbf{perp}((a,b)) = (-b,a)$. The vectors are orthogonal.
For line $ax + by = c$ def $\overline{v} = (-b,a)$.
Line through P and Q has $\overline{v} = \overline{PQ}$ and $c = \overline{v} \times P$.
$\mathbf{side}_l(P) = \overline{v_l} \times P - c_l$ sign determines which side P is on from l.
$\mathbf{dist}_l(P) = \mathbf{side}_l(P)/\|v_l\|$ squared is integer.
Sorting points along a line: comparator is $\overline{v} \cdot A < \overline{v} \cdot B$.
Translating line by $\overline{t}$: new line has $c' = c + \overline{v} \times \overline{t}$.
Line intersection: is $(c_l\overline{v}_m - c_m\overline{v}_l)/(\overline{v}_l \times \overline{v}_m)$.
Project P onto l: is $P - \mathbf{perp}(v)\,\mathbf{side}_l(P)/\|v\|^2$.
Angle bisectors: $\overline{v} = \overline{v}_l/\|\overline{v}_l\| + \overline{v}_m/\|\overline{v}_m\|$
$c = c_l/\|\overline{v}_l\| + c_m/\|\overline{v}_m\|$.
P is on segment AB iff $\mathbf{orient}(A,B,P) = 0$ and $\overline{PA} \cdot \overline{PB} \leqslant 0$.
Proper intersection of AB and CD exists iff $\mathbf{orient}(C,D,A)$ and $\mathbf{orient}(C,D,B)$ have opp. signs and $\mathbf{orient}(A,B,C)$ and $\mathbf{orient}(A,B,D)$ have opp. signs. Coordinates:

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

Circumcircle center:

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

Circle-line intersect:

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

Circle-circle intersect:

```
int circleCircle(pt o1, double r1, pt o2, double r2, pair<pt,pt> &out) {
  pt d=o2-o1; double d2=sq(d);
  if (d2 == 0) {assert(r1 != r2); return 0;} // concentric circles
  double pd = (d2 + r1*r1 - r2*r2)/2; // = |O_1P| * d
  double h2 = r1*r1 - pd*pd/d2; // = h^2
  if (h2 >= 0) {
    pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
    out = {p-h, p+h};}
  return 1 + sgn(h2);
```

Tangent lines:

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

5 3D geometry

$\mathbf{orient}(P,Q,R,S) = (\overline{PQ} \times \overline{PR}) \cdot \overline{PS}$.
S above PQR iff > 0.
For plane $ax + by + cz = d$ def $\overline{n} = (a,b,c)$.
Line with normal $\overline{n}$ through point P has $d = \overline{n} \cdot P$.
$\mathbf{side}_\Pi(P) = \overline{n} \cdot P - d$ sign determines side from Π.
$\mathbf{dist}_\Pi(P) = \mathbf{side}_\Pi(P)/\|\overline{n}\|$.
Translating plane by $\overline{t}$ makes $d' = d + \overline{n} \cdot \overline{t}$.
Plane-plane intersection of has direction $\overline{n}_1 \times \overline{n}_2$ and goes through $((d_1\overline{n}_2 - d_2\overline{n}_1) \times \overline{d})/\|\overline{d}\|^2$.
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

```
//computing power = how many times function integrate gets called
template<typename T>
double simps(T f, double a, double b) {
  return (f(a) + 4*f((a+b)/2) + f(b))*(b-a)/6;
}
template<typename T>
double integrate(T f, double a, double b, double computing_power){
  double m = (a+b)/2;
  double l = simps(f,a,m), r = simps(f,m,b), tot=simps(f,a,b);
  if (computing_power < 1) return tot;
  return integrate(f, a, m, computing_power/2) + integrate(f, m, b, computing_power/2);   #430
}   %360
```

7 Triangle centers

```
const double min_delta = 1e-13;
const double coord_max = 1e6;
typedef complex < double > point;
point A, B, C; // vertixes of the triangle
bool collinear(){
  double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
  if(min_diff < coord_max * min_delta)
    return true;
  point sp = (B - A) / (C - A);
  double ang = M_PI/2-abs(abs(arg(sp))-M_PI/2); //positive angle with the real line   #623
  return ang < min_delta;
}   %446
point circum_center(){
  if(collinear())
    return point(NAN,NAN);
  //squared lengths of sides
  double a2, b2, c2;
  a2 = norm(B - C);
  b2 = norm(A - C);
  c2 = norm(A - B);
  //barycentric coordinates of the circumcenter
  double c_A, c_B, c_C;
  c_A = a2 * (b2 + c2 - a2);//sin(2 * alpha) may be used as well
  c_B = b2 * (a2 + c2 - b2);   #385
  c_C = c2 * (a2 + b2 - c2);
  double sum = c_A + c_B + c_C;
  c_A /= sum;
  c_B /= sum;
  c_C /= sum;
  // cartesian coordinates of the circumcenter
  return c_A * A + c_B * B + c_C * C;
}   %742
point centroid(){ //center of mass
  return (A + B + C) / 3.0;
}
point ortho_center(){ //euler line
  point O = circum_center();
  return O + 3.0 * (centroid() - O);
};
point nine_point_circle_center(){ //euler line
  point O = circum_center();
  return O + 1.5 * (centroid() - O);   #193
};   %031
point in_center(){
  if(collinear())
    return point(NAN,NAN);
  double a, b, c; //side lengths
  a = abs(B - C);
  b = abs(A - C);
  c = abs(A - B);
  //trilinear coordinates are (1,1,1)
  //barycentric coordinates
  double c_A = a, c_B = b, c_C = c;
  double sum = c_A + c_B + c_C;
  c_A /= sum;   #157
  c_B /= sum;
  c_C /= sum;
  // cartesian coordinates of the incenter
  return c_A * A + c_B * B + c_C * C;
}   %980
```

8 Seg-Seg intersection, halfplane intersection area

```
struct Segment {
  Vec a, b;
  Vec d() {
    return b-a;
  }
};
ostream& operator<<(ostream& l, Segment r) {
  return l << r.a << '-' << r.b;
}
Vec intersection(Segment l, Segment r) {   #355
  Vec dl = l.d(), dr = r.d();
  if(cross(dl, dr) == 0)
    return {nanl(""), nanl("")};
  long double h = cross(dr, l.a-r.a) / len(dr);
  long double dh = cross(dr, dl) / len(dr);
  return l.a + dl * (h / -dh);
}
//Returns the area bounded by halfplanes
long double getArea(vector&lt;Segment&gt; lines) {
  long double lowerbound = -HUGE_VALL, upperbound = HUGE_VALL;
  vector&lt;Segment&gt; linesBySide[2];   #658
  for(auto line : lines) {
    if(line.b.y == line.a.y) {
      if(line.a.x < line.b.x) {
        lowerbound = max(lowerbound, line.a.y);
      } else {
        upperbound = min(upperbound, line.a.y);
      }
    } else if(line.a.y < line.b.y) {
```

```
      linesBySide[1].push_back(line);
    } else {                                                    #049
      linesBySide[0].push_back({line.b, line.a});
    }
  }
  sort(linesBySide[0].begin(), linesBySide[0].end(), [](Segment l, Segment r) {
    if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a > normal(r.d())*r.a;
    return cross(l.d(), r.d()) < 0;
  });
  sort(linesBySide[1].begin(), linesBySide[1].end(), [](Segment l, Segment r) {
    if(cross(l.d(), r.d()) == 0) return normal(l.d())*l.a < normal(r.d())*r.a;
    return cross(l.d(), r.d()) > 0;                             #434
  });
  //Now find the application area of the lines and clean up redundant ones
  vector<long double> applyStart[2];
  for(int side = 0; side < 2; side++) {
    vector<long double> &apply = applyStart[side];
    vector&lt;Segment&gt; curLines;
    for(auto line : linesBySide[side]) {
      while(curLines.size() > 0) {
        Segment other = curLines.back();
        if(cross(line.d(), other.d()) != 0) {
          long double start = intersection(line, other).y;      #501
          if(start > apply.back()) break;
        }
        curLines.pop_back();
        apply.pop_back();
      }
      if(curLines.size() == 0) {
        apply.push_back(-HUGE_VALL);
      } else {
        apply.push_back(intersection(line, curLines.back()).y);
      }                                                         #060
      curLines.push_back(line);
    }
    linesBySide[side] = curLines;
  }
  applyStart[0].push_back(HUGE_VALL);
  applyStart[1].push_back(HUGE_VALL);
  long double result = 0;
  {
    long double lb = -HUGE_VALL, ub;
    for(int i=0, j=0; i < (int)linesBySide[0].size() && j < (int)linesBySide[1].size();lb = ub) {   #349
      ub = min(applyStart[0][i+1], applyStart[1][j+1]);
      long double alb = lb, aub = ub;
      Segment l0 = linesBySide[0][i], l1 = linesBySide[1][j];
      if(cross(l1.d(), l0.d()) > 0) {
        alb = max(alb, intersection(l0, l1).y);
      } else if(cross(l1.d(), l0.d()) < 0) {
        aub = min(aub, intersection(l0, l1).y);
      }
      alb = max(alb, lowerbound);
      aub = min(aub, upperbound);                               #419
      aub = max(aub, alb);
      {
        long double x1 = l0.a.x + (alb - l0.a.y) / l0.d().y * l0.d().x;
        long double x2 = l0.a.x + (aub - l0.a.y) / l0.d().y * l0.d().x;
        result -= (aub - alb) * (x1 + x2) / 2;
      }
      {
        long double x1 = l1.a.x + (alb - l1.a.y) / l1.d().y * l1.d().x;
        long double x2 = l1.a.x + (aub - l1.a.y) / l1.d().y * l1.d().x;
        result += (aub - alb) * (x1 + x2) / 2;                  #228
      }
      if(applyStart[0][i+1] < applyStart[1][j+1]) {
        i++;
      } else {
        j++;
      }
    }
  }
  return result;
}                                                               %011
```

9 Convex polygon algorithms

```
ll dot(const pair<int, int> &v1, const pair<int, int> &v2) {
  return (ll)v1.first * v2.first + (ll)v1.second * v2.second;
}
ll cross(const pair<int, int> &v1, const pair<int, int> &v2) {
  return (ll)v1.first * v2.second - (ll)v2.first * v1.second;
}
ll dist_sq(const pair<int, int> &p1, const pair<int, int> &p2) {
  return (ll)(p2.first - p1.first) * (p2.first - p1.first) +
         (ll)(p2.second - p1.second) * (p2.second - p1.second);
}                                                               %025
struct Hull {
  vector<pair<pair<int, int>, pair<int, int> > > hull;
  vector<pair<pair<int, int>, pair<int, int> > >::iterator
    upper_begin;
  template <typename Iterator>
  void extend_hull(Iterator begin, Iterator end) { // O(n)
    vector<pair<int, int> > res;
    for (auto it = begin; it != end; ++it) {
      if (res.empty() || *it != res.back()) {
        while (res.size() >= 2) {                               #678
          auto v1 = make_pair(
            res[res.size() - 1].first - res[res.size() - 2].first,
            res[res.size() - 1].second - res[res.size() - 2].second);
          auto v2 =
            make_pair(it->first - res[res.size() - 2].first,
                      it->second - res[res.size() - 2].second);
```

```
        if (cross(v1, v2) > 0) break;
        res.pop_back();
      }
      res.push_back(*it);                                    #906
    }
  }
  for (int i = 0; i < res.size() - 1; ++i)
    hull.emplace_back(res[i], res[i + 1]);
}
Hull(vector<pair<int, int> > &vert) { // atleast 2 distinct points
  sort(vert.begin(), vert.end());     // O(n log(n))
  extend_hull(vert.begin(), vert.end());
  int diff = hull.size();
  extend_hull(vert.rbegin(), vert.rend());                   #011
  upper_begin = hull.begin() + diff;
}                                                            %873
bool contains(pair<int, int> p) { // O(log(n))
  if (p < hull.front().first || p > upper_begin->first)
    return false;
  {
    auto it_low = lower_bound(
      hull.begin(), upper_begin,
      make_pair(make_pair(p.first, (int)-2e9), make_pair(0, 0)));
    if (it_low != hull.begin()) --it_low;
    auto v1 =
      make_pair(it_low->second.first - it_low->first.first,  #477
                it_low->second.second - it_low->first.second);
    auto v2 = make_pair(p.first - it_low->first.first,
                        p.second - it_low->first.second);
    if (cross(v1, v2) < 0) // < 0 is inclusive, <=0 is exclusive
      return false;
  }
  {
    auto it_up = lower_bound(
      hull.rbegin(), hull.rbegin() + (hull.end() - upper_begin),
      make_pair(make_pair(p.first, (int)2e9), make_pair(0, 0)));
      ↪ #129
    if (it_up - hull.rbegin() == hull.end() - upper_begin) --it_up;
    auto v1 = make_pair(it_up->first.first - it_up->second.first,
                        it_up->first.second - it_up->second.second);
    auto v2 = make_pair(p.first - it_up->second.first,
                        p.second - it_up->second.second);
    if (cross(v1, v2) > 0) // > 0 is inclusive, >=0 is exclusive
      return false;
  }
  return true;
}                                                            %092
template <
  typename T> // The function can have only one local min and max
              // and may be constant only at min and max.
              vector<
                pair<pair<int, int>, pair<int, int> > >::iterator
              max(function<
                  T(const pair<pair<int, int>, pair<int, int> > &)>
                    f) { // O(log(n))
  auto l = hull.begin();
  auto r = hull.end();
  vector<pair<pair<int, int>, pair<int, int> > >::iterator best =
  ↪ #479
    hull.end();
  T best_val;
  while (r - l > 2) {
    auto mid = l + (r - l) / 2;
    T l_val = f(*l);
    T l_nxt_val = f(*(l + 1));
    T mid_val = f(*mid);
    T mid_nxt_val = f(*(mid + 1));
    if (best == hull.end() ||
        l_val > best_val) { // If max is at l we may remove it
        ↪ from                                                 #200
                            // the range.
      best = l;
      best_val = l_val;
    }
    if (l_nxt_val > l_val) {
      if (mid_val < l_val) {
        r = mid;
      } else {
        if (mid_nxt_val > mid_val) {
          l = mid + 1;
        } else {                                               #848
          r = mid + 1;
        }
      }
    } else {
      if (mid_val < l_val) {
        l = mid + 1;
      } else {
        if (mid_nxt_val > mid_val) {
          l = mid + 1;
        } else {                                               #369
          r = mid + 1;
        }
      }
    }
  }
  T l_val = f(*l);
  if (best == hull.end() || l_val > best_val) {
    best = l;
    best_val = l_val;
  }                                                            #920
  if (r - l > 1) {
    T l_nxt_val = f(*(l + 1));
    if (best == hull.end() || l_nxt_val > best_val) {
      best = l + 1;
      best_val = l_nxt_val;
    }
```

```
  }
  return best;
}                                                                   %331
vector<pair<pair<int, int>, pair<int, int> > >::iterator closest(
  pair<int, int> p) { // p can't be internal(can be on border), hull
                      // must have atleast 3 points
  const pair<pair<int, int>, pair<int, int> > &ref_p =
    hull.front(); // O(log(n))
  return max(
    function<double(const pair<pair<int, int>, pair<int, int> > &)>(
      [&p,
       &ref_p](const pair<pair<int, int>, pair<int, int> >
                  &seg) { // accuracy of used type should be coord^2
        if (p == seg.first) return 10 - M_PI;                       #900
        auto v1 = make_pair(seg.second.first - seg.first.first,
                            seg.second.second - seg.first.second);
        auto v2 = make_pair(p.first - seg.first.first,
                            p.second - seg.first.second);
        ll cross_prod = cross(v1, v2);
        if (cross_prod > 0) { // order the backside by angle
          auto v1 = make_pair(ref_p.first.first - p.first,
                              ref_p.first.second - p.second);
          auto v2 = make_pair(seg.first.first - p.first,
                              seg.first.second - p.second);      #534
          ll dot_prod = dot(v1, v2);
          ll cross_prod = cross(v2, v1);
          return atan2(cross_prod, dot_prod) / 2;
        }
        ll dot_prod = dot(v1, v2);
        double res = atan2(dot_prod, cross_prod);
        if (dot_prod <= 0 && res > 0) res = -M_PI;
        if (res > 0) {
          res += 20;
        } else {                                                    #913
          res = 10 - res;
        }
        return res;
      }));
}                                                                   %483
pair<int, int> forw_tan(
  pair<int, int> p) { // can't be internal or on border
  const pair<pair<int, int>, pair<int, int> > &ref_p =
    hull.front(); // O(log(n))
  auto best_seg = max(
    function<double(const pair<pair<int, int>, pair<int, int> > &)>(
      [&p,
       &ref_p](const pair<pair<int, int>, pair<int, int> >
                  &seg) { // accuracy of used type should be coord^2
        auto v1 = make_pair(ref_p.first.first - p.first,            #089
                            ref_p.first.second - p.second);
        auto v2 = make_pair(seg.first.first - p.first,
                            seg.first.second - p.second);
        ll dot_prod = dot(v1, v2);
```

```
        ll cross_prod =
          cross(v2, v1); // cross(v1, v2) for back_tan!!!
        return atan2(cross_prod, dot_prod); // order by signed angle
      }));
  return best_seg->first;
}                                                                   %850
vector<pair<pair<int, int>, pair<int, int> > >::iterator max_in_dir(
  pair<int, int> v) { // first is the ans. O(log(n))
  return max(
    function<ll(const pair<pair<int, int>, pair<int, int> > &)>(
      [&v](const pair<pair<int, int>, pair<int, int> > &seg) {
        return dot(v, seg.first);
      }));
}
pair<vector<pair<pair<int, int>, pair<int, int> > >::iterator,
     vector<pair<pair<int, int>, pair<int, int> > >::iterator>  %013
intersections(
  pair<pair<int, int>, pair<int, int> > line) { // O(log(n))
  int x = line.second.first - line.first.first;
  int y = line.second.second - line.first.second;
  auto dir = make_pair(-y, x);
  auto it_max = max_in_dir(dir);
  auto it_min = max_in_dir(make_pair(y, -x));
  ll opt_val = dot(dir, line.first);
  if (dot(dir, it_max->first) < opt_val ||
      dot(dir, it_min->first) > opt_val)                            #662
    return make_pair(hull.end(), hull.end());
  vector<pair<pair<int, int>, pair<int, int> > >::iterator it_r1,
    it_r2;
  function<bool(const pair<pair<int, int>, pair<int, int> > &,
                const pair<pair<int, int>, pair<int, int> > &)>
    inc_comp(
      [&dir](const pair<pair<int, int>, pair<int, int> > &lft,
             const pair<pair<int, int>, pair<int, int> > &rgt) {
        return dot(dir, lft.first) < dot(dir, rgt.first);
      });                                                           #781
  function<bool(const pair<pair<int, int>, pair<int, int> > &,
                const pair<pair<int, int>, pair<int, int> > &)>
    dec_comp(
      [&dir](const pair<pair<int, int>, pair<int, int> > &lft,
             const pair<pair<int, int>, pair<int, int> > &rgt) {
        return dot(dir, lft.first) > dot(dir, rgt.first);
      });
  if (it_min <= it_max) {
    it_r1 = upper_bound(it_min, it_max + 1, line, inc_comp) - 1;
    if (dot(dir, hull.front().first) >= opt_val) {                  #826
      it_r2 =
        upper_bound(hull.begin(), it_min + 1, line, dec_comp) - 1;
    } else {
      it_r2 = upper_bound(it_max, hull.end(), line, dec_comp) - 1;
    }
  } else {
    it_r1 = upper_bound(it_max, it_min + 1, line, dec_comp) - 1;
```

```
    if (dot(dir, hull.front().first) <= opt_val) {
      it_r2 =
        upper_bound(hull.begin(), it_max + 1, line, inc_comp) - 1;
        ↪ #388
    } else {
      it_r2 = upper_bound(it_min, hull.end(), line, inc_comp) - 1;
    }
  }
  return make_pair(it_r1, it_r2);
}                                                                   %000
pair<pair<int, int>, pair<int, int> > diameter() { // O(n)
  pair<pair<int, int>, pair<int, int> > res;
  ll dia_sq = 0;
  auto it1 = hull.begin();
  auto it2 = upper_begin;
  auto v1 =
    make_pair(hull.back().second.first - hull.back().first.first,
              hull.back().second.second - hull.back().first.second);
  while (it2 != hull.begin()) {
    auto v2 =                                                       #919
      make_pair((it2 - 1)->second.first - (it2 - 1)->first.first,
                (it2 - 1)->second.second - (it2 - 1)->first.second);
    ll decider = cross(v1, v2);
    if (decider > 0) break;
    --it2;
  }
  while (it2 != hull.end()) { // check all antipodal pairs
    if (dist_sq(it1->first, it2->first) > dia_sq) {
      res = make_pair(it1->first, it2->first);
      dia_sq = dist_sq(res.first, res.second);                      #125
    }
    auto v1 = make_pair(it1->second.first - it1->first.first,
                        it1->second.second - it1->first.second);
    auto v2 = make_pair(it2->second.first - it2->first.first,
                        it2->second.second - it2->first.second);
    ll decider = cross(v1, v2);
    if (decider == 0) { // report cross pairs at parallel lines.
      if (dist_sq(it1->second, it2->first) > dia_sq) {
        res = make_pair(it1->second, it2->first);
        dia_sq = dist_sq(res.first, res.second);                    #860
      }
      if (dist_sq(it1->first, it2->second) > dia_sq) {
        res = make_pair(it1->first, it2->second);
        dia_sq = dist_sq(res.first, res.second);
      }
      ++it1;
      ++it2;
    } else if (decider < 0) {
      ++it1;
    } else {                                                        #530
      ++it2;
    }
  }
  return res;
  }
};                                                                  %215
```

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

```
const int alpha_size=26;
struct node{
  node *nxt[alpha_size]; //May use other structures to move in trie
  node *suffix;
  node(){
    memset(nxt, 0, alpha_size*sizeof(node *));
  }
  int cnt=0;
};
node *aho_corasick(vector<vector<char> > &dict){                    #480
  node *root= new node;
  root->suffix = 0;
  vector<pair<vector<char> *, node *> > cur_state;
  for(vector<char> &s : dict)
    cur_state.emplace_back(&s, root);
  for(int i=0; !cur_state.empty(); ++i){
    vector<pair<vector<char> *, node *> > nxt_state;
    for(auto &cur : cur_state){
      node *nxt=cur.second->nxt[(*cur.first)[i]];
      if(nxt){                                                      #888
        cur.second=nxt;
      }else{
        nxt = new node;
        cur.second->nxt[(*cur.first)[i]] = nxt;
        node *suf = cur.second->suffix;
        cur.second = nxt;
        nxt->suffix = root; //set correct suffix link
        while(suf){
          if(suf->nxt[(*cur.first)[i]]){
            nxt->suffix = suf->nxt[(*cur.first)[i]];                #786
            break;
          }
          suf=suf->suffix;
        }
      }
      if(cur.first->size() > i+1)
        nxt_state.push_back(cur);
    }
    cur_state=nxt_state;
  }                                                                 #940
  return root;
}                                                                   %064
//auxilary functions for searhing and counting
node *walk(node *cur, char c){ //longest prefix in dict that is suffix
  ↪ of walked string.
  while(true){
    if(cur->nxt[c])
      return cur->nxt[c];
    if(!cur->suffix)
```

```cpp
      return cur;
    cur = cur->suffix;
  }
}                                                                    %127
void cnt_matches(node *root, vector<char> &match_in){
  node *cur = root;
  for(char c : match_in){
    cur = walk(cur, c);
    ++cur->cnt;
  }
}                                                                    %286
void add_cnt(node *root){ //After counting matches propagete ONCE to suffixes for final counts
  vector<node *> to_visit = {root};
  for(int i=0; i<to_visit.size(); ++i){
    node *cur = to_visit[i];
    for(int j=0; j<alpha_size; ++j){
      if(cur->nxt[j])
        to_visit.push_back(cur->nxt[j]);
    }
  }
  for(int i=to_visit.size()-1; i>0; --i)                            #865
    to_visit[i]->suffix->cnt += to_visit[i]->cnt;
}                                                                    %313
int main(){ //http://codeforces.com/group/s3etJR5zZK/contest/212916/problem/4
  int n, len;
  scanf("%d %d", &len, &n);
  vector<char> a(len+1);
  scanf("%s", a.data());
  a.pop_back();
  for(char &c : a)
    c -= 'a';
  vector<vector<char> > dict(n);
  for(int i=0; i<n; ++i){
    scanf("%d", &len);
    dict[i].resize(len+1);
    scanf("%s", dict[i].data());
    dict[i].pop_back();
    for(char &c : dict[i])
      c -= 'a';
  }
  node *root = aho_corasick(dict);
  cnt_matches(root, a);
  add_cnt(root);
  for(int i=0; i<n; ++i){
    node *cur = root;
    for(char c : dict[i])
      cur = walk(cur, c);
    printf("%d\n", cur->cnt);
  }
}
```

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

```cpp
class AutoNode {
 private:
  map< char, AutoNode * > nxt_char;  // Map is faster than hashtable and unsorted arrays
 public:
  int len; //Length of longest suffix in equivalence class.
  AutoNode *suf;
  bool has_nxt(char c) const {
    return nxt_char.count(c);
  }
  AutoNode *nxt(char c) {                                            #486
    if (!has_nxt(c))
      return NULL;
    return nxt_char[c];
  }
  void set_nxt(char c, AutoNode *node) {
    nxt_char[c] = node;
  }
  AutoNode *split(int new_len, char c) {
    AutoNode *new_n = new AutoNode;
    new_n->nxt_char = nxt_char;                                      #952
    new_n->len = new_len;
    new_n->suf = suf;
    suf = new_n;
    return new_n;
  }                                                                  %677
  // Extra functions for matching and counting
  AutoNode *lower_depth(int depth) { //move to longest suffix of current with a maximum length of depth.
    if (suf->len >= depth)
      return suf->lower_depth(depth);
    return this;
  }
  AutoNode *walk(char c, int depth, int &match_len) { //move to longest suffix of walked path that is a substring
    match_len = min(match_len, len); //includes depth limit(needed for finding matches)
    if (has_nxt(c)) {  //as suffixes are in classes match_len must be tracked externally
      ++match_len;
      return nxt(c)->lower_depth(depth);                             #227
    }
    if (suf)
      return suf->walk(c, depth, match_len);
    return this;
  }                                                                  %955
  int paths_to_end = 0;
  void set_as_end() { //All suffixes of current node are marked as ending nodes.
    paths_to_end += 1;
    if (suf) suf->set_as_end();
  }
```

```cpp
  bool vis = false;
  void calc_paths_to_end() { //Call ONCE from ROOT. For each node calculates number of ways to reach an end node.
    if (!vis) {  //paths_to_end is ocurence count for any strings in current suffix equivalence class.
      vis = true;
      for (auto cur : nxt_char) {                                    #035
        cur.second->calc_paths_to_end();
        paths_to_end += cur.second->paths_to_end;
      }
    }
  }                                                                  %996
  //Transform into suffix tree of reverse string
  map<char, AutoNode * > tree_links;
  int end_dist = 1<<30;
  int calc_end_dist(){
    if(end_dist == 1<<30){
      if(nxt_char.empty())
        end_dist = 0;
      for (auto cur : nxt_char)
        end_dist = min(end_dist, 1+cur.second->calc_end_dist());
    }
    return end_dist;                                                 #412
  }
  bool vis_t = false;
  void build_suffix_tree(string &s) { //Call ONCE from ROOT.
    if (!vis_t) {
      vis_t = true;
      if(suf)
        suf->tree_links[s[s.size()-end_dist-suf->len-1]] = this;
      for (auto cur : nxt_char)
        cur.second->build_suffix_tree(s);
    }                                                                #202
  }
};
struct SufAutomaton {
  AutoNode *last;
  AutoNode *root;
  void extend(char new_c) {
    AutoNode *new_end = new AutoNode;
    new_end->len = last->len + 1;
    AutoNode *suf_w_nxt = last;
    while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {                #308
      suf_w_nxt->set_nxt(new_c, new_end);
      suf_w_nxt = suf_w_nxt->suf;
    }
    if (!suf_w_nxt) {
      new_end->suf = root;
    } else {
      AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);
      if (suf_w_nxt->len + 1 == max_sbstr->len) {
        new_end->suf = max_sbstr;
      } else {                                                       #865
        AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
        new_end->suf = eq_sbstr;
        AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
        while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
          w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
          w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
        }
      }
    }
    last = new_end;                                                  #356
  }                                                                  %628
  SufAutomaton(string &s) {
    root = new AutoNode;
    root->len = 0;
    root->suf = NULL;
    last = root;
    for (char c : s) extend(c);
    root->calc_end_dist(); //To build suffix tree use reversed string
    root->build_suffix_tree(s);
  }
};                                                                   %034
```

12 Dinic

```cpp
struct MaxFlow{
  typedef long long ll;
  const ll INF = 1e18;
  struct Edge{
    int u,v;
    ll c,rc;
    shared_ptr<ll> flow;
    Edge(int _u, int _v, ll _c, ll _rc = 0):u(_u),v(_v),c(_c),rc(_rc){
    }
  };                                                                 #787
  struct FlowTracker{
    shared_ptr<ll> flow;
    ll cap, rcap;
    bool dir;
    FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
    ll rem() const {
      if(dir == 0){
        return cap-*flow;
      }
      else{                                                          #844
        return rcap+*flow;
      }
    }
    void add_flow(ll f){
      if(dir == 0)
        *flow += f;
      else
        *flow -= f;
```

```cpp
      assert(*flow <= cap);
      assert(-*flow <= rcap);                                    #287
    }
    operator ll() const { return rem(); }
    void operator-=(ll x){ add_flow(x); }
    void operator+=(ll x){ add_flow(-x); }
  };
  int source,sink;
  vector<vector<int> > adj;
  vector<vector<FlowTracker> > cap;
  vector<Edge> edges;
  MaxFlow(int _source, int _sink):source(_source),sink(_sink){    #080
    assert(source != sink);
  }
  int add_edge(int u, int v, ll c, ll rc = 0){
    edges.push_back(Edge(u,v,c,rc));
    return edges.size()-1;
  }
  vector<int> now,lvl;
  void prep(){
    int max_id = max(source, sink);
    for(auto edge : edges)                                       #328
      max_id = max(max_id, max(edge.u, edge.v));
    adj.resize(max_id+1);
    cap.resize(max_id+1);
    now.resize(max_id+1);
    lvl.resize(max_id+1);
    for(auto &edge : edges){
      auto flow = make_shared<ll>(0);
      adj[edge.u].push_back(edge.v);
      cap[edge.u].push_back(FlowTracker(edge.c, edge.rc, flow, 0));
      if(edge.u != edge.v){                                      #717
        adj[edge.v].push_back(edge.u);
        cap[edge.v].push_back(FlowTracker(edge.c, edge.rc, flow, 1));
      }
      assert(cap[edge.u].back() == edge.c);
      edge.flow = flow;
    }
  }
  bool dinic_bfs(){
    fill(now.begin(),now.end(),0);
    fill(lvl.begin(),lvl.end(),0);                               #038
    lvl[source] = 1;
    vector<int> bfs(1,source);
    for(int i = 0; i < bfs.size(); ++i){
      int u = bfs[i];
      for(int j = 0; j < adj[u].size(); ++j){
        int v = adj[u][j];
        if(cap[u][j] > 0 && lvl[v] == 0){
          lvl[v] = lvl[u]+1;
          bfs.push_back(v);
        }                                                        #010
      }
    }
    return lvl[sink] > 0;
  }
  ll dinic_dfs(int u, ll flow){
    if(u == sink)
      return flow;
    while(now[u] < adj[u].size()){
      int v = adj[u][now[u]];
      if(lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0){           #014
        ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
        if(res > 0){
          cap[u][now[u]] -= res;
          return res;
        }
      }
      ++now[u];
    }
    return 0;
  }                                                              #197
  ll calc_max_flow(){
    prep();
    ll ans = 0;
    while(dinic_bfs()){
      ll cur = 0;
      do{
        cur = dinic_dfs(source,INF);
        ans += cur;
      }while(cur > 0);
    }                                                            #817
    return ans;
  }
  ll flow_on_edge(int edge_index){
    assert(edge_index < edges.size());
    return *edges[edge_index].flow;
  }
};                                                               %583
int main(){
  int n,m;
  cin >> n >> m;
  vector<pair<int, pair<int, int> > > graph(m);
  for(int i=0; i<m; ++i){
    cin>>graph[i].second.first>>graph[i].second.second>>graph[i].first;
  }
  ll res=0;
  for(auto cur : graph){
    auto mf = MaxFlow(cur.second.first,cur.second.second); // arguments source and sink, memory usage O(largest node index + input size), sink doesn't need to be last index
    for(int i = 0; i < m; ++i){
      if(graph[i].first > cur.first){
        mf.add_edge(graph[i].second.first,graph[i].second.second,1,1); // store edge index if care about flow value
      }
    }
```

```
    res +=  mf.calc_max_flow();
  }
  cout<<res<<endl;
}
```

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

```
const int nmax=1055;
const ll inf=1e14;
int t, n, v; //0 is source, v-1 sink
ll rem_flow[nmax][nmax]; //set [x][y] for directed capacity from x to y.
ll cost[nmax][nmax]; //set [x][y] for directed cost from x to y. SET TO inf IF NOT USED
ll min_dist[nmax];
int prev_node[nmax];
ll node_flow[nmax];
bool visited[nmax];                                             %576
ll tot_cost, tot_flow; //output
void min_cost_max_flow(){
  tot_cost=0;              //Does not work with negative cycles.
  tot_flow=0;
  ll sink_pot=0;
  min_dist[0] = 0;                                              %927
  for(int i=1; i<=v; ++i){ //incase of no negative edges Bellman-Ford can be removed.
    min_dist[i]=inf;
  }
  for(int i=0; i<v-1; ++i){
    for(int j=0; j<v; ++j){
      for(int k=0; k<v; ++k){
        if(rem_flow[j][k] > 0 && min_dist[j]+cost[j][k] < min_dist[k])
          min_dist[k] = min_dist[j]+cost[j][k];
      }
    }                                                           #599
  }
  for(int i=0; i<v; ++i){   //Apply potentials to edge costs.
    for(int j=0; j<v; ++j){
      if(cost[i][j]!=inf){
        cost[i][j]+=min_dist[i];
        cost[i][j]-=min_dist[j];
      }
    }
  }
  sink_pot+=min_dist[v-1]; //Bellman-Ford end.                  %849
  while(true){
    for(int i=0; i<=v; ++i){ //node after sink is used as start value for Dijkstra.
      min_dist[i]=inf;
      visited[i]=false;
    }
    min_dist[0]=0;
    node_flow[0]=inf;
    int min_node;
    while(true){ //Use Dijkstra to calculate potentials
      int min_node=v;                                           #782
      for(int i=0; i<v; ++i){
        if((!visited[i]) && min_dist[i]<min_dist[min_node])
          min_node=i;
      }
      if(min_node==v) break
      visited[min_node]=true;
      for(int i=0; i<v; ++i){
        if((!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){
          min_dist[i]=min_dist[min_node]+cost[min_node][i];
          prev_node[i]=min_node;                                #881
          node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
        }
      }
    }
    if(min_dist[v-1]==inf) break
    for(int i=0; i<v; ++i){   //Apply potentials to edge costs.
      for(int j=0; j<v; ++j){ //Found path from source to sink becomes 0 cost.
        if(cost[i][j]!=inf){
          cost[i][j]+=min_dist[i];
          cost[i][j]-=min_dist[j];                              #083
        }
      }
    }
    sink_pot+=min_dist[v-1];
    tot_flow+=node_flow[v-1];
    tot_cost+=sink_pot*node_flow[v-1];
    int cur=v-1;
    while(cur!=0){ //Backtrack along found path that now has 0 cost.
      rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
      rem_flow[cur][prev_node[cur]]+=node_flow[v-1];            #582
      cost[cur][prev_node[cur]]=0;
      if(rem_flow[prev_node[cur]][cur]==0)
        cost[prev_node[cur]][cur]=inf;
      cur=prev_node[cur];
    }
  }
}                                                               %803
int main(){//http://www.spoj.com/problems/GREED/
  cin>>t;
  for(int i=0; i<t; ++i){
    cin>>n;
    for(int j=0; j<nmax; ++j){
      for(int k=0; k<nmax; ++k){
        cost[j][k]=inf;
        rem_flow[j][k]=0;
      }
    }
    for(int j=1; j<=n; ++j){
      cost[j][2*n+1]=0;
      rem_flow[j][2*n+1]=1;
```

```cpp
    }
    for(int j=1; j<=n; ++j){
      int card;
      cin>>card;
      ++rem_flow[0][card];
      cost[0][card]=0;
    }
    int ex_c;
    cin>>ex_c;
    for(int j=0; j<ex_c; ++j){
      int a, b;
      cin>>a>>b;
      if(b<a) swap(a,b);
      cost[a][b]=1;
      rem_flow[a][b]=nmax;
      cost[b][n+b]=0;
      rem_flow[b][n+b]=nmax;
      cost[n+b][a]=1;
      rem_flow[n+b][a]=nmax;
    }
    v=2*n+2;
    min_cost_max_flow();
    cout<<tot_cost<<'\n';
  }
}
```

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

```cpp
struct Network {
  struct Node;
  struct Edge {
    Node *u, *v;
    int f, c, cost;
    Node* from(Node* pos) {
      if(pos == u)
        return v;
      return u;
    }                                                    #042
    int getCap(Node* pos) {
      if(pos == u)
        return c-f;
      return f;
    }
    int addFlow(Node* pos, int toAdd) {
      if(pos == u) {
        f += toAdd;
        return toAdd * cost;
      } else {                                           #965
        f -= toAdd;
        return -toAdd * cost;
      }
    }
  };
  struct Node {
    vector<Edge*> conn;
    int index;
  };
  deque<Node> nodes;                                     #534
  deque<Edge> edges;
  Node* addNode() {
    nodes.push_back(Node());
    nodes.back().index = nodes.size()-1;
    return &nodes.back();
  }
  Edge* addEdge(Node* u, Node* v, int f, int c, int cost) {
    edges.push_back({u, v, f, c, cost});
    u->conn.push_back(&edges.back());
    v->conn.push_back(&edges.back());                    #507
    return &edges.back();
  }
  //Assumes all needed flow has already been added
  int minCostMaxFlow() {
    int n = nodes.size();
    int result = 0;
    struct State {
      int p;
      Edge* used;
    };
    while(1) {                                           #877
      vector<vector<State> > state(1, vector<State>(n, {0, 0}));
      for(int lev = 0; lev < n; lev++) {
        state.push_back(state[lev]);
        for(int i=0;i<n;i++){
          if(lev == 0 || state[lev][i].p < state[lev-1][i].p) {
            for(Edge* edge : nodes[i].conn){
              if(edge->getCap(&nodes[i]) > 0) {
                int np = state[lev][i].p + (edge->u == &nodes[i] ? edge->cost : -edge->cost);
                int ni = edge->from(&nodes[i])->index;
                if(np < state[lev+1][ni].p) {            #281
                  state[lev+1][ni].p = np;
                  state[lev+1][ni].used = edge;
                }
              }
            }
          }
        }
      }
      //Now look at the last level
      bool valid = false;
      for(int i=0;i<n;i++)                               #283
        if(state[n-1][i].p > state[n][i].p) {
          valid = true;
          vector<Edge*> path;
          int cap = 1000000000;
          Node* cur = &nodes[i];
          int clev = n;
          vector<bool> explr(n, false);
```

```
          while(!explr[cur->index]) {
            explr[cur->index] = true;
            State cstate = state[clev][cur->index];                 #954
            cur = cstate.used->from(cur);
            path.push_back(cstate.used);
          }
          reverse(path.begin(), path.end() );
          {
            int i=0;
            Node* cur2 = cur;
            do {
              cur2 = path[i]->from(cur2);
              i++;                                                  #990
            } while(cur2 != cur);
            path.resize(i);
          }
          for(auto edge : path) {
            cap = min(cap, edge->getCap(cur));
            cur = edge->from(cur);
          }
          for(auto edge : path) {
            result += edge->addFlow(cur, cap);
            cur = edge->from(cur);                                  #599
          }
        }
      if(!valid) break;
    }
    return result;
  }
};                                                                  %900
```

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

```
struct EdgeDesc{
  int from, to, w;
};
struct DMST{
  struct Node;
  struct Edge{
    Node *from;
    Node *tar;
    int w;
    bool inc;                                                       #186
  };
  struct Circle{
    bool vis = false;
    vector<Edge *> contents;
    void clean(int idx);
  };
  const static greater<pair<ll, Edge *> > comp; //Can use inline static since C++17
  static vector<Circle> to_process;
  static bool no_dmst;
  static Node *root;                                                #536
  struct Node{
    Node *par = NULL;
    vector<pair<int, int> > out_cands; //Circ, edge idx
    vector<pair<ll, Edge *> > con;
    bool in_use = false;
    ll w = 0; //extra to add to edges in con
    Node *anc(){
      if(!par)
        return this;
      while(par->par)                                               #425
        par = par->par;
      return par;
    }
    void clean(){
      if(!no_dmst){
        in_use = false;
        for(auto &cur : out_cands)
          to_process[cur.first].clean(cur.second);
      }
    }                                                               #561
    Node *con_to_root(){
      if(anc() == root)
        return root;
      in_use = true;
      Node *super = this; //Will become root or the first Node encountered in a loop.
      while(super == this){
        while(!con.empty() && con.front().second->tar->anc() == anc()){
          pop_heap(con.begin(), con.end(), comp);
          con.pop_back();
        }                                                           #522
        if(con.empty()){
          no_dmst = true;
          return root;
        }
        pop_heap(con.begin(), con.end(), comp);
        auto nxt = con.back();
        con.pop_back();
        w = -nxt.first;
        if(nxt.second->tar->in_use){ //anc() wouldn't change anything
          super = nxt.second->tar->anc();                           #174
          to_process.resize(to_process.size()+1);
        } else {
          super = nxt.second->tar->con_to_root();
        }
        if(super != root){
          to_process.back().contents.push_back(nxt.second);
          out_cands.emplace_back(to_process.size()-1, to_process.back().contents.size()-1);
        } else { //Clean circles
          nxt.second->inc = true;
          nxt.second->from->clean();                                #629
        }
      }
```

```
    if(super != root){ //we are some loops non first Node.
      if(con.size() > super->con.size()){
        swap(con, super->con); //Largest con in loop should not be copied.
        swap(w, super->w);
      }
      for(auto cur : con){
        super->con.emplace_back(cur.first - super->w + w, cur.second);
        push_heap(super->con.begin(), super->con.end(), comp);  #375
      }
    }
    par = super; //root or anc() of first Node encountered in a loop
    return super;
  }
};
Node *cur_root;
vector<Node> graph;
vector<Edge> edges;
DMST(int n, vector<EdgeDesc> &desc, int r){ //Self loops and multiple edges are okay.  #076
  graph.resize(n);
  cur_root = &graph[r];
  for(auto &cur : desc) //Edges are reversed internally
    edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
  for(int i=0; i<desc.size(); ++i)
    graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]);
  for(int i=0; i<n; ++i)
    make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
}
bool find(){  #469
  root = cur_root;
  no_dmst = false;
  for(auto &cur : graph){
    cur.con_to_root();
    to_process.clear();
    if(no_dmst) return false;
  }
  return true;
}  %732
ll weight(){
  ll res = 0;
  for(auto &cur : edges){
    if(cur.inc)
      res += cur.w;
  }
  return res;
}  %477
};
void DMST::Circle::clean(int idx){
  if(!vis){
    vis = true;
    for(int i=0; i<contents.size(); ++i){
      if(i != idx){
        contents[i]->inc = true;
        contents[i]->from->clean();
      }
    }  #711
  }
}
const greater<pair<ll, DMST::Edge *> > DMST::comp;
vector<DMST::Circle> DMST::to_process;
bool DMST::no_dmst;
DMST::Node *DMST::root;  %771
```

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

```
struct vert;
struct edge{
  bool exists = true;
  vert *dest;
  edge *rev;
  edge(vert *_dest) : dest(_dest){
    rev = NULL;
  }
  vert &operator*(){
    return *dest;  #955
  }
  vert *operator->(){
    return dest;
  }
  bool is_bridge();
};
struct vert{
  deque<edge> con;
  int val = 0;
  int seen;  #336
  int dfs(int upd, edge *ban){ //handles multiple edges
    if(!val){
      val = upd;
      seen = val;
      for(edge &nxt : con){
        if(nxt.exists  && (&nxt) != ban)
          seen = min(seen, nxt->dfs(upd+1, nxt.rev));
      }
    }
    return seen;  #673
  }  %624
  void remove_adj_bridges(){
    for(edge &nxt : con){
      if(nxt.is_bridge())
        nxt.exists = false;
    }
  }  %106
  int cnt_adj_bridges(){
    int res = 0;
    for(edge &nxt : con)
      res += nxt.is_bridge();
```

```
    return res;
  }                                                                      %056
};
bool edge::is_bridge(){
  return exists && (dest->seen > rev->dest->val || dest->val < rev->dest->seen);
}                                                                        %223
vert graph[nmax];
int main(){ //Mechanics Practice BRIDGES
  int n, m;
  cin>>n>>m;
  for(int i=0; i<m; ++i){
    int u, v;
    scanf("%d %d", &u, &v);
    graph[u].con.emplace_back(graph+v);
    graph[v].con.emplace_back(graph+u);
    graph[u].con.back().rev = &graph[v].con.back();
    graph[v].con.back().rev = &graph[u].con.back();
  }
  graph[1].dfs(1, NULL);
  int res = 0;
  for(int i=1; i<=n; ++i)
    res += graph[i].cnt_adj_bridges();
  cout<<res/2<<endl;
}
```

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

```
struct Graph {
    int n;
    vector<vector<int> > conn;
    Graph(int nsize) {
        n = nsize;
        conn.resize(n);
    }
    void add_edge(int u, int v) {
        conn[u].push_back(v);
    }                                                                    #078
    void _topsort_dfs(int pos, vector<int> &result, vector<bool> &explr, vector<vector<int> > &revconn) {
        if(explr[pos])
            return;
        explr[pos] = true;
        for(auto next : revconn[pos])
            _topsort_dfs(next, result, explr, revconn);
        result.push_back(pos);
    }
    vector<int> topsort() {
        vector<vector<int> > revconn(n);                                 #346
        for(int u = 0; u < n; u++) {
            for(auto v : conn[u])
                revconn[v].push_back(u);
        }
        vector<int> result;
        vector<bool> explr(n, false);
        for(int i=0; i < n; i++)
            _topsort_dfs(i, result, explr, revconn);
        reverse(result.begin(), result.end());
        return result;                                                   #991
    }
    void dfs(int pos, vector<int> &result, vector<bool> &explr) {
        if(explr[pos])
            return;
        explr[pos] = true;
        for(auto next : conn[pos])
            dfs(next, result, explr);
        result.push_back(pos);
    }                                                                    %603
  vector<vector<int> > scc(){ // tested on https://www.hackerearth.com/practice/algorithms/graphs/strongly-connect
        vector<int> order = topsort();
    reverse(order.begin(),order.end());
        vector<bool> explr(n, false);
    vector<vector<int> > results;
    for(auto it = order.rbegin(); it != order.rend(); ++it){
      vector<int> component;
      _topsort_dfs(*it,component,explr,conn);
      sort(component.begin(),component.end());
      results.push_back(component);                                      #741
    }
    sort(results.begin(),results.end());
    return results;
  }
};                                                                       %983
//Solution for: http://codeforces.com/group/PjzGiggT71/contest/221700/problem/C
int main() {
    int n, m;
    cin >> n >> m;
    Graph g(2*m);
    for(int i=0; i<n; i++) {
        int a, sa, b, sb;
        cin >> a >> sa >> b >> sb;
        a--, b--;
        g.add_edge(2*a + 1 - sa, 2*b + sb);
        g.add_edge(2*b + 1 - sb, 2*a + sa);
    }
    vector<int> state(2*m, 0);
    {
        vector<int> order = g.topsort();
        vector<bool> explr(2*m, false);
        for(auto u : order) {
            vector<int> traversed;
            g.dfs(u, traversed, explr);
            if(traversed.size() > 0 && !state[traversed[0]^1]) {
                for(auto c : traversed)
                    state[c] = 1;
            }
```

```
        }
    }
    for(int i=0; i < m; i++) {
        if(state[2*i] == state[2*i+1]) {
            cout << "IMPOSSIBLE\n";
            return 0;
        }
    }
    for(int i=0; i < m; i++) {
        cout << state[2*i+1] << '\n';
    }
    return 0;
}
```

18 Generic persistent compressed lazy segment tree

```
struct Seg{
  ll sum=0;
  void recalc(const Seg &lhs_seg, int lhs_len, const Seg &rhs_seg, int rhs_len){
    sum = lhs_seg.sum + rhs_seg.sum;
  }
} __attribute__((packed));
struct Lazy{
  ll add;
  ll assign_val; //LLONG_MIN if no assign;
  void init(){                                                    #883
    add = 0;
    assign_val = LLONG_MIN;
  }
  Lazy(){ init(); }
  void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len){
    lhs_lazy = *this;
    rhs_lazy = *this;
    init();
  }
  void merge(Lazy &oth, int len){                                 #470
    if(oth.assign_val != LLONG_MIN){
      add = 0;
      assign_val = oth.assign_val;
    }
    add += oth.add;
  }
  void apply_to_seg(Seg &cur, int len) const{
    if(assign_val != LLONG_MIN){
      cur.sum =  len * assign_val;
    }                                                             #216
    cur.sum += len * add;
  }
} __attribute__((packed));                                        %625
struct Node{ //Following code should not need to be modified
  int ver;
  bool is_lazy = false;
  Seg seg;
  Lazy lazy;
  Node *lc=NULL, *rc=NULL;
  void init(){
    if(!lc){
      lc = new Node {ver};
      rc = new Node {ver};                                        #313
    }
  }
  Node *upd(int L, int R, int l, int r, Lazy &val, int tar_ver){
    if(ver != tar_ver){
      Node *rep = new Node(*this);
      rep->ver = tar_ver;
      return rep->upd(L, R, l, r, val, tar_ver);
    }
    if(L >= l && R <= r){
      val.apply_to_seg(seg, R-L);                                 #138
      lazy.merge(val, R-L);
      is_lazy = true;
    } else {
      init();
      int M = (L+R)/2;
      if(is_lazy){
        Lazy l_val , r_val;
        lazy.split(l_val, r_val, R-L);
        lc = lc->upd(L, M, L, M, l_val, ver);
        rc = rc->upd(M, R, M, R, r_val, ver);                     #104
        is_lazy = false;
      }
      Lazy l_val , r_val;
      val.split(l_val, r_val, R-L);
      if(l < M)
        lc = lc->upd(L, M, l, r, l_val, ver);
      if(M < r)
        rc = rc->upd(M, R, l, r, r_val, ver);
      seg.recalc(lc->seg, M-L, rc->seg, R-M);
    }                                                             #245
    return this;
  }
  void get(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp, bool last_ver){
    if(L >= l && R <= r){
      tmp->recalc(*lft_res, L-l, seg, R-L);
      swap(lft_res, tmp);
    } else {
      init();
      int M = (L+R)/2;
      if(is_lazy){                                                #726
        Lazy l_val , r_val;
        lazy.split(l_val, r_val, R-L);
        lc = lc->upd(L, M, L, M, l_val, ver+last_ver);
        lc->ver = ver;
        rc = rc->upd(M, R, M, R, r_val, ver+last_ver);
        rc->ver = ver;
        is_lazy = false;
```

```
      }
      if(l < M)
        lc->get(L, M, l, r, lft_res, tmp, last_ver);                #696
      if(M < r)
        rc->get(M, R, l, r, lft_res, tmp, last_ver);
    }
  }
} __attribute__((packed));
struct SegTree{ //indexes start from 0, ranges are [beg, end)
  vector<Node *> roots; //versions start from 0
  int len;
  SegTree(int _len) : len(_len){
    roots.push_back(new Node {0});                                  #295
  }
  int upd(int l, int r, Lazy &val, bool new_ver = false){
    Node *cur_root = roots.back()->upd(0, len, l, r, val,
      roots.size()-!new_ver);
    if(cur_root != roots.back())
      roots.push_back(cur_root);
    return roots.size()-1;
  }
  Seg get(int l, int r, int ver = -1){
    if(ver == -1)
      ver = roots.size()-1;                                         #977
    Seg seg1, seg2;
    Seg *pres = &seg1, *ptmp = &seg2;
    roots[ver]->get(0, len, l, r, pres, ptmp, roots.size()-1);
    return *pres;
  }
};                                                                  %542
int main(){
  int n, m; //solves Mechanics Practice LAZY
  cin>>n>>m;
  SegTree seg_tree(1<<17);
  for(int i=0; i<n; ++i){
    Lazy tmp;
    scanf("%lld", &tmp.assign_val);
    seg_tree.upd(i, i+1, tmp);
  }
  for(int i=0; i<m; ++i){
    int o;
    int l, r;
    scanf("%d %d %d", &o, &l, &r);
    --l;
    if(o==1){
      Lazy tmp;
      scanf("%lld", &tmp.add);
      seg_tree.upd(l, r, tmp);
    } else if(o==2){
      Lazy tmp;
      scanf("%lld", &tmp.assign_val);
      seg_tree.upd(l, r, tmp);
    } else {
      Seg res = seg_tree.get(l, r);
      printf("%lld\n",res.sum);
    }
  }
}
```

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

```
class dummy {
public:
  dummy () {}
  dummy (int, int) {}
  void set (int, int) {}
  int query (int left, int right) {
    cout << this << ' ' << left << ' ' << right << endl;
  }
};                                                                  %932
/* T should be the type of the data stored in each vertex;
 * DS should be the underlying data structure that is used to peform
   the
 * group operation. It should have the following methods:
 * * DS () - empty constructor
 * * DS (int size, T initial) - constructs the structure with the given
   size,
 *   initially filled with initial.
 * * void set (int index, T value) - set the value at index `index` to
   `value`
 * * T query (int left, int right) - return the "sum" of elements
   between left and right, inclusive.
 */
template<typename T, class DS>
class HLD {
  int vertexc;
  vector<int> *adj;
  vector<int> subtree_size;
  DS structure;
  DS aux;
  void build_sizes (int vertex, int parent) {
    subtree_size[vertex] = 1;
    for (int child : adj[vertex]) {                                 #037
      if (child != parent) {
        build_sizes(child, vertex);
        subtree_size[vertex] += subtree_size[child];
      }
    }
  }
  int cur;
  vector<int> ord;
  vector<int> chain_root;
  vector<int> par;                                                  #593
  void build_hld (int vertex, int parent, int chain_source) {
    cur++;
    ord[vertex] = cur;
    chain_root[vertex] = chain_source;
    par[vertex] = parent;
```

```cpp
    if (adj[vertex].size() > 1 || (vertex == 1 && adj[vertex].size() == 1)) {
      int big_child, big_size = -1;
      for (int child : adj[vertex]) {
        if ((child != parent) && (subtree_size[child] > big_size)) {
          big_child = child;                                              #151
          big_size = subtree_size[child];
        }
      }
      build_hld(big_child, vertex, chain_source);
      for (int child : adj[vertex]) {
        if ((child != parent) && (child != big_child))
          build_hld(child, vertex, child);
      }
    }
  }                                                                       #562
public:
  HLD (int _vertexc) {
    vertexc = _vertexc;
    adj = new vector<int> [vertexc + 5];
  }
  void add_edge (int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  void build (T initial) {                                                #566
    subtree_size = vector<int> (vertexc + 5);
    ord = vector<int> (vertexc + 5);
    chain_root = vector<int> (vertexc + 5);
    par = vector<int> (vertexc + 5);
    cur = 0;
    build_sizes(1, -1);
    build_hld(1, -1, 1);
    structure = DS (vertexc + 5, initial);
    aux = DS (50, initial);
  }                                                                       #758
  void set (int vertex, int value) {
    structure.set(ord[vertex], value);
  }
  T query_path (int u, int v) { /* returns the "sum" of the path u->v */
    int cur_id = 0;
    while (chain_root[u] != chain_root[v]) {
      if (ord[u] > ord[v]) {
        cur_id++;
        aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
        u = par[chain_root[u]];                                           #538
      } else {
        cur_id++;
        aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
        v = par[chain_root[v]];
      }
    }
    cur_id++;
    aux.set(cur_id, structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
    return aux.query(1, cur_id);
  }                                                                       %905
  void print () {
    for (int i = 1; i <= vertexc; i++)
      cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' ' << par[i] << endl;
  }
};
int main () {
  int vertexc;
  cin >> vertexc;
  HLD<int, dummy> hld (vertexc);
  for (int i = 0; i < vertexc - 1; i++) {
    int u, v;
    cin >> u >> v;
    hld.add_edge(u, v);
  }
  hld.build(0);
  hld.print();
  int queryc;
  cin >> queryc;
  for (int i = 0; i < queryc; i++) {
    int u, v;
    cin >> u >> v;
    hld.query_path(u, v);
    cout << endl;
  }
}
```

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

```cpp
// Fully overloaded any dimensional BIT, use any type for coordinates, elements, return_value.
// Includes coordinate compression.
template < typename elem_t, typename coord_t, coord_t n_inf, typename ret_t >
class BIT {
  vector< coord_t > positions;
  vector< elem_t > elems;
  bool initiated = false;
 public:
  BIT() {
    positions.push_back(n_inf);
  }
  void initiate() {                                                       #448
    if (initiated) {
      for (elem_t &c_elem : elems)
        c_elem.initiate();
    } else {
      initiated = true;
      sort(positions.begin(), positions.end());
      positions.resize(unique(positions.begin(), positions.end()) - positions.begin());
```

```
      elems.resize(positions.size());
    }
  }                                                          #036
  template < typename... loc_form >
  void update(coord_t cord, loc_form... args) {
    if (initiated) {
      int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
      for (; pos < positions.size(); pos += pos & -pos)
        elems[pos].update(args...);
    } else {
      positions.push_back(cord);
    }
  }                                                          #154
  template < typename... loc_form >
  ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
    ret_t res = 0;
    int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
    for (; pos > 0; pos -= pos & -pos)
      res += elems[pos].query(args...);
    return res;
  }
};
template < typename internal_type >                          #895
struct wrapped {
  internal_type a = 0;
  void update(internal_type b) {
    a += b;
  }
  internal_type query() {
    return a;
  }
  // Should never be called, needed for compilation
  void initiate() {
    cerr << 'i' << endl;                                     #560
  }
  void update() {
    cerr << 'u' << endl;
  }
};                                                           %714
int main() {
  // retun type should be same as type inside wrapped
  BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll > fenwick;
  int dim = 2;
  vector< tuple< int, int, ll > > to_insert;
  to_insert.emplace_back(1, 1, 1);
  // set up all positions that are to be used for update
  for (int i = 0; i < dim; ++i) {
    for (auto &cur : to_insert)
      fenwick.update(get< 0 >(cur), get< 1 >(cur));  // May include value which won't be used
    fenwick.initiate();
  }
  // actual use
  for (auto &cur : to_insert)
    fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
  cout << fenwick.query(2, 2)<<'\n';
}
```

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

```
mt19937 randgen;
struct Treap {
    struct Node {
        int key;
        int value;
        unsigned int priority;
        long long total;
        Node* lch;
        Node* rch;
        Node(int new_key, int new_value) {                   #698
            key = new_key;
            value = new_value;
            priority = randgen();
            total = new_value;
            lch = 0;
            rch = 0;
        }
        void update() {
            total = value;
            if(lch) total += lch->total;                     #295
            if(rch) total += rch->total;
        }
    };
    deque<Node> nodes;
    Node* root = 0;
    pair<Node*, Node*> split(int key, Node* cur) {
        if(cur == 0) return {0, 0};
        pair<Node*, Node*> result;
        if(key <= cur->key) {
            auto ret = split(key, cur->lch);                 #233
            cur->lch = ret.second;
            result = {ret.first, cur};
        } else {
            auto ret = split(key, cur->rch);
            cur->rch = ret.first;
            result = {cur, ret.second};
        }
        cur->update();
        return result;
    }                                                        #230
    Node* merge(Node* left, Node* right) {
        if(left == 0) return right;
        if(right == 0) return left;
        Node* top;
```

```
        if(left->priority < right->priority) {
            left->rch = merge(left->rch, right);
            top = left;
        } else {
            right->lch = merge(left, right->lch);
            top = right;                                              #510
        }
        top->update();
        return top;
    }
    void insert(int key, int value) {
        nodes.push_back(Node(key, value));
        Node* cur = &nodes.back();
        pair<Node*, Node*> ret = split(key, root);
        cur = merge(ret.first, cur);
        cur = merge(cur, ret.second);                                 #760
        root = cur;
    }
    void erase(int key) {
        Node *left, *mid, *right;
        tie(left, mid) = split(key, root);
        tie(mid, right) = split(key+1, mid);
        root = merge(left, right);
    }
    long long sum_upto(int key, Node* cur) {
        if(cur == 0) return 0;                                        #634
        if(key <= cur->key) {
            return sum_upto(key, cur->lch);
        } else {
            long long result = cur->value + sum_upto(key, cur->rch);
            if(cur->lch) result += cur->lch->total;
            return result;
        }
    }
    long long get(int l, int r) {
        return sum_upto(r+1, root) - sum_upto(l, root);               #509
    }
};                                                                    %959
//Solution for: http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/TREAP
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    int m;
    Treap treap;
    cin >> m;
    for(int i=0;i<m;i++) {
        int type;
        cin >> type;
        if(type == 1) {
            int x, y;
            cin >> x >> y;
            treap.insert(x, y);
        } else if(type == 2) {
            int x;
            cin >> x;
            treap.erase(x);
        } else {
            int l, r;
            cin >> l >> r;
            cout << treap.get(l, r) << endl;
        }
    }
    return 0;
}
```

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

```
typedef unsigned char uchar;
template<typename T>
void msd_radixsort(T *start, T *sec_start, int arr_size, int d=sizeof(T)-1){
  const int msd_radix_lim = 100;
  const T mask = 255;
  int bucket_sizes[256]{};
  for(T *it = start; it!=start+arr_size; ++it){
    ++bucket_sizes[((*it)>>(d*8))&mask];
    //++bucket_sizes[*((uchar*)it + d)];
  }
  T *locs_mem[257];                                                   #947
  locs_mem[0] = sec_start;
  T **locs = locs_mem+1;
  locs[0] = sec_start;
  for(int j=0; j<255; ++j){
    locs[j+1] = locs[j]+bucket_sizes[j];
  }
  for(T *it = start; it!=start+arr_size; ++it){
    uchar bucket_id = ((*it)>>(d*8))&mask;
    *(locs[bucket_id]++) = *it;
  }                                                                   #770
  locs = locs_mem;
  if(d){
    T *locs_old[256];
    locs_old[0] = start;
    for(int j=0; j<255; ++j){
      locs_old[j+1] = locs_old[j]+bucket_sizes[j];
    }
    for(int j=0; j<256; ++j){
      if(locs[j+1]-locs[j] < msd_radix_lim){
        std::sort(locs[j], locs[j+1]);                                #018
        if(d & 1){
          copy(locs[j], locs[j+1], locs_old[j]);
        }
      }
      else{
        msd_radixsort(locs[j], locs_old[j], bucket_sizes[j], d-1);
      }
    }
```

```
  }
}                                                    %225
const int nmax = 5e7;
ll arr[nmax], tmp[nmax];
int main(){
  for(int i=0; i<nmax; ++i)
    arr[i] = ((ll)rand()<<32)|rand();
  msd_radixsort(arr, tmp, nmax);
  assert(is_sorted(arr, arr+nmax));
}
```

23 FFT 5M length/sec

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

```
struct Complex {
  double a = 0, b = 0;
  Complex &operator/=(const int &oth) {
    a /= oth;
    b /= oth;
    return *this;
  }
};
Complex operator+(const Complex &lft, const Complex &rgt) {
  return Complex{lft.a + rgt.a, lft.b + rgt.b};            #384
}
Complex operator-(const Complex &lft, const Complex &rgt) {
  return Complex{lft.a - rgt.a, lft.b - rgt.b};
}
Complex operator*(const Complex &lft, const Complex &rgt) {
  return Complex{lft.a * rgt.a - lft.b * rgt.b, lft.a * rgt.b + lft.b * rgt.a};
}
Complex conj(const Complex &cur){
  return Complex{cur.a, -cur.b};
}                                                          #957
void fft_rec(Complex *arr, Complex *root_pow, int len) {
  if (len != 1) {
    fft_rec(arr, root_pow, len >> 1);
    fft_rec(arr + len, root_pow, len >> 1);
  }
  root_pow += len;
  for (int i = 0; i < len; ++i) {
    Complex tmp = arr[i] + root_pow[i] * arr[i + len];
    arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
    arr[i] = tmp;                                          #048
  }
}
void fft(vector< Complex > &arr, int ord, bool invert) {
  assert(arr.size() == 1 << ord);
  static vector< Complex > root_pow(1);
  static int inc_pow = 1;
  static bool is_inv = false;
  if (inc_pow <= ord) {
    int idx = root_pow.size();
    root_pow.resize(1 << ord);                             #710
    for (; inc_pow <= ord; ++inc_pow) {
      for (int idx_p = 0; idx_p < 1 << (ord - 1); idx_p += 1 << (ord - inc_pow), ++idx) {
        root_pow[idx] =
            Complex{cos(-idx_p * M_PI / (1 << (ord - 1))), sin(-idx_p * M_PI / (1 << (ord - 1)))};
        if (is_inv) root_pow[idx].b = -root_pow[idx].b;
      }
    }
  }
  if (invert != is_inv) {
    is_inv = invert;                                       #750
    for (Complex &cur : root_pow) cur.b = -cur.b;
  }
  for (int i = 1, j=0; i < (1 << ord); ++i) {
    int m = 1<<(ord-1);
    bool cont = true;
    while(cont){
      cont = j & m;
      j ^= m;
      m>>=1;
    }                                                      #844
    if (i < j) swap(arr[i], arr[j]);
  }
  fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
  if (invert)
    for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord);
}                                                          %380
void mult_poly_mod(vector< int > &a, vector< int > &b, vector< int > &c) {  // c += a*b
  static vector< Complex > arr[4];  // correct upto 0.5-2M elements(mod ~= 1e9)
  if (c.size() < 400) {
    for (int i = 0; i < a.size(); ++i)
      for (int j = 0; j < b.size() && i + j < c.size(); ++j)
        c[i + j] = ((ll)a[i] * b[j] + c[i + j]) % mod;
  } else {
    int fft_ord = 32 - __builtin_clz(c.size());
    if (arr[0].size() != 1 << fft_ord)
      for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord);    #811
    for (int i = 0; i < 4; ++i) fill(arr[i].begin(), arr[i].end(), Complex{});
    for (int &cur : a)
      if (cur < 0) cur += mod;
    for (int &cur : b)
      if (cur < 0) cur += mod;
    const int shift = 15;
    const int mask = (1 << shift) - 1;
    for (int i = 0; i < min(a.size(), c.size()); ++i) {
      arr[0][i].a = a[i] & mask;
      arr[1][i].a = a[i] >> shift;                         #809
    }
```

```
    for (int i = 0; i < min(b.size(), c.size()); ++i) {
      arr[0][i].b = b[i] & mask;
      arr[1][i].b = b[i] >> shift;
    }
    for (int i = 0; i < 2; ++i) fft(arr[i], fft_ord, false);
    for (int i = 0; i < 2; ++i) {
      for (int j = 0; j < 2; ++j) {
        int tar = 2 + (i + j)/2;
        Complex mult = {0, -0.25};                          #066
        if(i^j)
          mult = {0.25, 0};
        for (int k = 0; k < (1 << fft_ord); ++k){
          int rev_k = ((1 << fft_ord)-k)%(1 << fft_ord);
          Complex ca = arr[i][k] + conj(arr[i][rev_k]);
          Complex cb = arr[j][k] - conj(arr[j][rev_k]);
          arr[tar][k] = arr[tar][k] + mult*ca*cb;
        }
      }
    }                                                       #623
    for (int i = 2; i < 4; ++i) {
      fft(arr[i], fft_ord, true);
      for (int k = 0; k < (int)c.size(); ++k){
        c[k] = (c[k] + (((ll)(arr[i][k].a + 0.5) % mod) << (shift * 2*(i - 2)))) % mod;
        c[k] = (c[k] + (((ll)(arr[i][k].b + 0.5) % mod) << (shift * (2*(i - 2)+1)))) % mod;
      }
    }
  }
}                                                           %231
```

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

```
struct ModArithm {
  ull n;
  ld rec;
  ModArithm(ull _n) : n(_n) {  // n in [2, 1<<63)
    rec = 1.0L/n;
  }
  ull multf(ull a, ull b) {  // a, b in [0, min(2*n, 1<<63))
    ull mult = (ld)a*b*rec+0.5L;
    ll res = a*b-mult*n;
    if(res < 0) res += n;                                   #780
    return res; // in [0, n-1)
  }
  ull sqp1(ull a) { return multf(a, a) + 1; }
};                                                          %493
ull pow_mod(ull a, ull n, ModArithm &arithm) {
  ull res = 1;
  for (ull i = 1; i <= n; i <<= 1) {
    if (n & i) res = arithm.multf(res, a);
    a = arithm.multf(a, a);
  }
  return res;
}                                                           %144
vector< char > small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
bool is_prime(ull n) {  // n <= 1<<63, 1M rand/s
  ModArithm arithm(n);
  if (n == 2 || n == 3) return true;
  if (!(n & 1) || n == 1) return false;
  ull s = __builtin_ctz(n - 1);
  ull d = (n - 1) >> s;
  for (ull a : small_primes) {
    if (a >= n) break;
    a = pow_mod(a, d, arithm);                              #356
    if (a == 1 || a == n - 1) continue;
    for (ull r = 1; r < s; ++r) {
      a = arithm.multf(a, a);
      if (a == 1) return false;
      if (a == n - 1) break;
    }
    if (a != n - 1) return false;
  }
  return true;
}                                                           %975
ll pollard_rho(ll n) {
  ModArithm arithm(n);
  int cum_cnt = 64 - __builtin_clz(n);
  cum_cnt *= cum_cnt / 5 + 1;
  while (true) {
    ll lv = rand() % n;
    ll v = arithm.sqp1(lv);
    int idx = 1;
    int tar = 1;
    while (true) {                                          #290
      ll cur = 1;
      ll v_cur = v;
      int j_stop = min(cum_cnt, tar-idx);
      for (int j = 0; j < j_stop; ++j) {
        cur = arithm.multf(cur, abs(v_cur -lv));
        v_cur = arithm.sqp1(v_cur);
        ++idx;
      }
      if (!cur) {
        for (int j = 0; j < cum_cnt; ++j) {                 #912
          ll g = __gcd(abs(v-lv), n);
          if (g == 1) {
            v = arithm.sqp1(v);
          } else if (g == n) {
            break;
          } else {
            return g;
          }
        }
        break;                                              #208
      } else {
```

```
      ll g = __gcd(cur, n);
      if (g != 1) return g;
    }
    v = v_cur;
    idx += j_stop;
    if (idx == tar) {
      lv = v;
      tar *= 2;
      v = arithm.sqp1(v);                                        #174
      ++idx;
    }
  }
 }
}                                                                %542
map< ll, int > prime_factor(ll n, map< ll, int > *res = NULL) {  // n <= 1<<61, ~1000/s (<500/s on CF)
 if (!res) {
   map< ll, int > res_act;
   for (int p : small_primes) {
     while (!(n % p)) {
       ++res_act[p];
       n /= p;
     }
   }
   if (n != 1) prime_factor(n, &res_act);                        #023
   return res_act;
 }
 if (is_prime(n)) {
   ++(*res)[n];
 } else {
   ll factor = pollard_rho(n);
   prime_factor(factor, res);
   prime_factor(n / factor, res);
 }
 return map< ll, int >();                                        #140
} //Usage: fact = prime_factor(n);                               %477
```

25 Symmetric Submodular Functions; Queyrannes's algorithm

SSF: such function $f : V \to R$ that satisfies $f(A) = f(V/A)$ and for all $x \in V, X \subseteq Y \subseteq V$ it holds that $f(X+x) - f(X) \le f(Y+x) - f(Y)$. **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)
```