University of Tartu ICPC Team Notebook (2018-2019) March 22, 2019


```
alias g++='g++ -g -Wall -Wshadow -Wconversion \
    -fsanitize=undefined,address -DCDEBUG'              #.bashrc
alias a='setxkbmap us -option'                          #.bashrc
alias m='setxkbmap us -option caps:escape'              #.bashrc
alias ma='setxkbmap us -variant dvp -option caps:escape' #.bashrc
#settings
gsettings set org.compiz.core:/org/compiz/profiles/Default/plugins/core/ hsize 4
gsettings set org.gnome.desktop.wm.preferences focus-mode 'sloppy'
set si cin          #.vimrc
set ts=4 sw=4 noet #.vimrc
set cb=unnamedplus #.vimrc
(global-set-key (kbd "C-x <next>") 'other-window) #.emacs
(global-set-key (kbd "C-x <prior>") 'previous-multiframe-window)
(global-set-key (kbd "C-M-z") 'ansi-term)
(global-linum-mode 1)
(column-number-mode 1)
(show-paren-mode 1)
(setq-default indent-tabs-mode nil) #.emacs
valgrind --vgdb-error=0 ./a <inp & #valgrind
gdb a                              #valgrind
target remote | vgdb               #valgrind
```

1 crc.sh

```
#!/bin/envbash
for j in `seq 1 1 200`; do
  sed '/^\s*$/d' $1 | head -$j | tr -d '[[:space:]]' | cksum | cut -f1 -d ' ' | tail -c 5 #whitespaces don't matter.
done #there shouldn't be any COMMENTS.
#copy lines being checked to separate file.
# $ ./crc.sh tmp.cpp | grep XXXX
```

2 2D geometry

Define $\textbf{orient}(A,B,C) = \overline{AB} \times \overline{AC}$. CCW iff > 0.
Define $\textbf{perp}((a,b)) = (-b,a)$. 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 $c = \overline{v} \times P$.
$\textbf{side}_l(P) = \overline{v_l} \times P - c_l$ sign determines which side P is on from l.
$\textbf{dist}_l(P) = \textbf{side}_l(P)/\|\overline{v_l}\|$ 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 $P - \textbf{perp}(v)\,\textbf{side}_l(P)/\|v\|^2$.
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 $\textbf{orient}(A,B,P) = 0$ and $\overline{PA} \cdot \overline{PB} \leqslant 0$.
Proper intersection of AB and CD exists iff $\textbf{orient}(C,D,A)$ and $\textbf{orient}(C,D,B)$ have opp. signs and $\textbf{orient}(A,B,C)$ and $\textbf{orient}(A,B,D)$ have opp. signs. Coordinates:

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

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);
```

3 3D geometry

$\textbf{orient}(P,Q,R,S) = (\overline{PQ} \times \overline{PR}) \cdot \overline{PS}$.
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 $d = \overline{n} \cdot P$.
$\textbf{side}_\Pi(P) = \overline{n} \cdot P - d$ sign determines side from Π.
$\textbf{dist}_\Pi(P) = \textbf{side}_\Pi(P)/\|\overline{n}\|$.
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 $((d_1\overline{n}_2 - d_2\overline{n}_1) \times \overline{d})/\left\|\overline{d}\right\|^2$.
Line-line distance:

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

Spherical to Cartesian:
$(r\cos\varphi\cos\lambda, r\cos\varphi\sin\lambda, r\sin\varphi)$.
Sphere-line intersection:

```
int sphereLine(p3 o, double r, line3d l, pair<p3,p3> &out) {
  double h2 = r*r - l.sqDist(o);
  if (h2 < 0) return 0; // the line doesn't touch the sphere
  p3 p = l.proj(o); // point P
  p3 h = l.d*sqrt(h2)/abs(l.d); // vector parallel to l, of length h
  out = {p-h, p+h};
  return 1 + (h2 > 0);
```

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

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

Angle between spherical segments AB and AC is angle between $A \times B$ and $A \times C$.
Oriented angle: subtract from 2π if mixed product is negative.
Area of a spherical polygon:

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

4 Maxflow Complexity

$\boldsymbol{\mathcal{O}(V^2E)}$ – Dinic

$\boldsymbol{\Theta(VE\log U)}$ – Capacity scaling

$\boldsymbol{\Theta(\mathbf{flow}E)}$ – Small flow

$\boldsymbol{\Theta(\min\{V^{\frac{2}{3}}, E^{\frac{1}{2}}\}E)}$ – Unitary capacities

$\boldsymbol{\Theta(\sqrt{V}E)}$ – Each vertex other than S,T has only a single incoming unitary edge or outgoing one (bipartite matching)

$\boldsymbol{\Theta(\mathbf{flow}E\log V)}$ – Min-cost-max flow

5 Min Rotation of string

```
int a=0, N=s.size();
s += s;
ran(b,0,N){
  ran(i,0,N) {
    if (a+i == b || s[a+i] < s[b+i]) {
      b += max(0, i-1);
      break;
    }
    if (s[a+i] > s[b+i]) {
      a = b;
      break;
    }
  }
}
return a;
```

6 gcc ordered set, hashtable

```
#define DEBUG(...) cerr << __VA_ARGS__ << endl;
#ifndef CDEBUG
#undef DEBUG
#define DEBUG(...) ((void)0);
#define NDEBUG
#endif
#define ran(i, a, b) for (auto i = (a); i < (b); i++)
#include <bits/stdc++.h>
typedef long long ll;                                              #1736
typedef long double ld;
using namespace std;                                               %1386
#pragma GCC optimize("Ofast")  // better vectorization
#pragma GCC target("avx,avx2") // double vectorized performance
#include <bits/extc++.h>
using namespace __gnu_pbds;
template <typename T, typename U>
using hashmap = gp_hash_table<T, U>; // dumb, 3x faster than stl
template <typename T>                                              #1392
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
  tree_order_statistics_node_update>;                              %3907
int main() {
  ordered_set<int> cur;
  cur.insert(1);
  cur.insert(3);
  cout << cur.order_of_key(2)
       << endl; // the number of elements in the set less than 2
  cout << *cur.find_by_order(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)
  ordered_set<int> oth;
  oth.insert(5); // to join: cur < oth
  cur.join(oth); // cur = 1, 3, 5, oth =
  cur.split(1, oth); // cur = 1, oth = 3, 5
  hashmap<int, int> h({}, {}, {}, {}, {1 << 16});
}
```

7 PRNGs and Hash functions

```
mt19937 gen;
uint64_t rand64() { return gen() ^ ((uint64_t)gen() << 32); }     %4798
uint64_t rand64() {
  static uint64_t x = 1; //x != 0
  x ^= x >> 12;
  x ^= x << 25;
  x ^= x >> 27;                                                    #6087
  return x * 0x2545f4914f6cdd1d; // can remove mult
}                                                                  %9038
uint64_t mix(uint64_t x){ //can replace constant with variable
        uint64_t mem[2] = { x, 0xdeadbeeffeebdaedull };
        asm volatile (
                "pxor %%xmm0, %%xmm0;"
                "movdqa (%0), %%xmm1;"                             #2024
                "aesenc %%xmm0, %%xmm1;"
                "movdqa %%xmm1, (%0);"
                :
                : "r" (&mem[0])
                : "memory"                                         #6956
        );
        return mem[0]; // use both slots for 128 bit hash
}                                                                  %9499
uint64_t mix(uint64_t x) { //x != 0
  x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
  x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
  x = x ^ (x >> 31);
  return x;                                                        #7126
}                                                                  %1575
uint64_t unmix(uint64_t x) {
  x = (x ^ (x >> 31) ^ (x >> 62)) * 0x319642b2d24d8ec3;
  x = (x ^ (x >> 27) ^ (x >> 54)) * 0x96de1b173f119089;
  x = x ^ (x >> 30) ^ (x >> 60);
  return x;                                                        #4780
}                                                                  %2094
uint64_t combine(uint64_t x, uint64_t y) {
  if (y < x) swap(x, y); // remove for ord
  return mix(mix(x) + y);
}                                                                  %1466
```

8 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() {                                                 #0823
  double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
  if (min_diff < coord_max * min_delta) return true;
  point sp = (B - A) / (C - A);
  double ang = M_PI / 2 - abs(abs(arg(sp)) - M_PI / 2);
  return ang < min_delta; // positive angle with the real line
}                               #8446                              %8446
point circum_center() {
  if (collinear()) return point(NAN, NAN);
  // squared lengths of sides
  double a2 = norm(B - C);
  double b2 = norm(A - C);
  double c2 = norm(A - B);                                         #6715
  // barycentric coordinates of the circumcenter
  double c_A = a2 * (b2 + c2 - a2); // sin(2 * alpha) works also
  double c_B = b2 * (a2 + c2 - b2);
  double c_C = c2 * (a2 + b2 - c2);
  double sum = c_A + c_B + c_C;
  c_A /= sum;                                                      #9407
  c_B /= sum;
  c_C /= sum;
```

```
  return c_A * A + c_B * B + c_C * C; // cartesian
}                                                                    %6856
point centroid() { // center of mass
  return (A + B + C) / 3.0;
}
point ortho_center() { // euler line
  point O = circum_center();                                         #3895
  return O + 3.0 * (centroid() - O);
};
point nine_point_circle_center() { // euler line
  point O = circum_center();
  return O + 1.5 * (centroid() - O);                                 #8193
};                                                                   %3031
point in_center() {
  if (collinear()) return point(NAN, NAN);
  double a = abs(B - C); // side lenghts
  double b = abs(A - C);
  double c = abs(A - B);                                             #5954
  // trilinear coordinates are (1,1,1)
  double sum = a + b + c;
  a /= sum;
  b /= sum;
  c /= sum;                     // barycentric
  return a * A + b * B + c * C; // cartesian
}                               #4892                                %4892
```

9 Seg-Seg intersection, halfplane intersection area

```
struct Seg {
  Vec a, b;
  Vec d() { return b - a; }
};
Vec intersection(Seg l, Seg r) {                                     #6327
  Vec dl = l.d(), dr = r.d();
  if (cross(dl, dr) == 0) return {nanl(""), nanl("")};
  double h = cross(dr, l.a - r.a) / len(dr);
  double dh = cross(dr, dl) / len(dr);
  return l.a + dl * (h / -dh);                                       #8893
}
// Returns the area bounded by halfplanes
double calc_area(const vector<Seg>& lines) {
  double lb = -HUGE_VAL, ub = HUGE_VAL;
  vector<Seg> slines[2];
  for (auto line : lines) {                                          #1804
    if (line.b.y == line.a.y) {
      if (line.a.x < line.b.x) {
        lb = max(lb, line.a.y);
      } else {
        ub = min(ub, line.a.y);                                      #6288
      }
    } else if (line.a.y < line.b.y) {
      slines[1].push_back(line);
    } else {
      slines[0].push_back({line.b, line.a});                         #3607
    }
  }
  ran(i, 0, 2) {
    sort(slines[i].begin(), slines[i].end(), [&](Seg l, Seg r) {
      if (cross(l.d(), r.d()) == 0)                                  #4919
        return normal(l.d()) * l.a > normal(r.d()) * r.a;
      return (1 - 2 * i) * cross(l.d(), r.d()) < 0;
    });
  }
  // Now find the application area of the lines and clean up redundant
  // ones
  vector<double> ap_s[2];                                            #9949
  ran(side, 0, 2) {
    vector<double>& apply = ap_s[side];
    vector<Seg> clines;
    for (auto line : slines[side]) {
      while (clines.size() > 0) {                                    #3099
        Seg other = clines.back();
        if (cross(line.d(), other.d()) != 0) {
          double start = intersection(line, other).y;
          if (start > apply.back()) break;
        }                                                            #7856
        clines.pop_back();
        apply.pop_back();
      }
      if (clines.size() == 0) {
        apply.push_back(-HUGE_VAL);                                  #0868
      } else {
        apply.push_back(intersection(line, clines.back()).y);
      }
      clines.push_back(line);
    }                                                                #8545
    slines[side] = clines;
  }
  ap_s[0].push_back(HUGE_VALL);
  ap_s[1].push_back(HUGE_VALL);
  double result = 0;                                                 #3234
  {
    double lb = -HUGE_VALL, ub;
    for (int i = 0, j = 0;
         i < (int)slines[0].size() && j < (int)slines[1].size();
         lb = ub) {                                                  #4531
      ub = min(ap_s[0][i + 1], ap_s[1][j + 1]);
      double alb = lb, aub = ub;
      Seg l[2] = {slines[0][i], slines[1][j]};
      if (cross(l[1].d(), l[0].d()) > 0) {
        alb = max(alb, intersection(l[0], l[1]).y);
      } else if (cross(l[1].d(), l[0].d()) < 0) {                    #2627
        aub = min(aub, intersection(l[0], l[1]).y);
      }
```

```
      alb = max(alb, lb);
      aub = min(aub, ub);
      aub = max(aub, alb);                                     #8493
      ran(k, 0, 2) {
        double x1 =
          l[0].a.x + (alb - l[0].a.y) / l[0].d().y * l[0].d().x;
        double x2 =
          l[0].a.x + (aub - l[0].a.y) / l[0].d().y * l[0].d().x;
        result += (-1 + 2 * k) * (aub - alb) * (x1 + x2) / 2;
      }                                                        #9267
      if (ap_s[0][i + 1] < ap_s[1][j + 1]) {
        i++;
      } else {
        j++;
      }                                                        #3074
    }
  }
  return result;
}                                                              %0513
```

10 Convex polygon algorithms

```
typedef pair<int, int> Vec;
typedef pair<Vec, Vec> Seg;
typedef vector<Seg>::iterator SegIt;
#define F first
#define S second
#define MP(x, y) make_pair(x, y)
Vec sub(const Vec &v1, const Vec &v2) {                        #7360
  return MP(v1.F - v2.F, v1.S - v2.S);
}
ll dot(const Vec &v1, const Vec &v2) {
  return (ll)v1.F * v2.F + (ll)v1.S * v2.S;
}                                                              #9034
ll cross(const Vec &v1, const Vec &v2) {
  return (ll)v1.F * v2.S - (ll)v2.F * v1.S;
}
ll dist_sq(const Vec &p1, const Vec &p2) {
  return (ll)(p2.F - p1.F) * (p2.F - p1.F) +                   #3379
         (ll)(p2.S - p1.S) * (p2.S - p1.S);
}                                                              %0923
struct Point;
multiset<Point>::iterator end_node;
struct Point {
  Vec p;
  typename multiset<Point>::iterator get_it() const {
    tuple<void *> tmp = {(void *)this - 32}; // gcc rb_tree dependent
    return *(multiset<Point>::iterator *)(&tmp);               #7198
  }
  bool operator<(const Point &rhs) const {
    return (p.F < rhs.p.F); // sort by x
  }
  bool operator<(const Vec &q) const {                         #2890
    auto nxt = next(get_it());
    if (nxt == end_node) return 0; // nxt == end()
    return q.S * dot(p, {q.F, 1}) <
           q.S * dot(nxt->p, {q.F, 1}); // convex hull trick
  }                                                            #9972
};
template <int part> // 1 = upper, -1 = lower
struct HullDynamic : public multiset<Point, less<> > {
  bool bad(iterator y) {
    if (y == begin()) return 0;                                #5204
    auto x = prev(y);
    auto z = next(y);
    if (z == end()) return y->p.F == x->p.F && y->p.S <= x->p.S;
    return part * cross(sub(y->p, x->p), sub(y->p, z->p)) <= 0;
  }                                                            #5415
  void insert_point(int m, int b) { // O(log(N))
    auto y = insert({{m, b}});
    if (bad(y)) {
      erase(y);
      return;                                                  #1418
    }
    while (next(y) != end() && bad(next(y))) erase(next(y));
    while (y != begin() && bad(prev(y))) erase(prev(y));
  }
  ll eval(int x) {    // O(log(N)) upper maximize dot(x, 1, v)
    end_node = end(); //       lower minimize dot(x, 1, v)
    auto it = lower_bound((Vec){x, part});                     #4908
    return (ll)it->p.F * x + it->p.S;
  }
};                                                             %9769
struct Hull {
  vector<Seg> hull;
  SegIt up_beg;
  template <typename It>
  void extend(It beg, It end) { // O(n)
    vector<Vec> r;                                             #4033
    for (auto it = beg; it != end; ++it) {
      if (r.empty() || *it != r.back()) {
        while (r.size() >= 2) {
          int n = r.size();
          Vec v1 = {r[n - 1].F - r[n - 2].F, r[n - 1].S - r[n - 2].S};
          Vec v2 = {it->F - r[n - 2].F, it->S - r[n - 2].S};
          if (cross(v1, v2) > 0) break;                        #3588
          r.pop_back();
        }
        r.push_back(*it);
      }
    }                                                          #6639
    ran(i, 0, (int)r.size() - 1) hull.emplace_back(r[i], r[i + 1]);
  }
  Hull(vector<Vec> &vert) {          // atleast 2 distinct points
```

```
  sort(vert.begin(), vert.end()); // O(n log(n))
  extend(vert.begin(), vert.end());                          #6560
  int diff = hull.size();
  extend(vert.rbegin(), vert.rend());
  up_beg = hull.begin() + diff;
}                                                            %0722
bool contains(Vec p) { // O(log(n))
  if (p < hull.front().F || p > up_beg->F) return false;
  {
    auto it_low = lower_bound(
      hull.begin(), up_beg, MP(MP(p.F, (int)-2e9), MP(0, 0)));
    if (it_low != hull.begin()) --it_low;                    #3373
    Vec a = {it_low->S.F - it_low->F.F, it_low->S.S - it_low->F.S};
    Vec b = {p.F - it_low->F.F, p.S - it_low->F.S};
    if (cross(a, b) < 0) // < 0 is inclusive, <=0 is exclusive
      return false;
  }                                                          #2197
  {
    auto it_up = lower_bound(hull.rbegin(),
      hull.rbegin() + (hull.end() - up_beg),
      MP(MP(p.F, (int)2e9), MP(0, 0)));
    if (it_up - hull.rbegin() == hull.end() - up_beg) --it_up;
    Vec a = {it_up->F.F - it_up->S.F, it_up->F.S - it_up->S.S};
    Vec b = {p.F - it_up->S.F, p.S - it_up->S.S};
    if (cross(a, b) > 0) // > 0 is inclusive, >=0 is exclusive
      return false;                                          #7227
  }
  return true;
}                                                            %1826
// The function can have only one local min and max
// and may be constant only at min and max.
template <typename T>
SegIt max(function<T(Seg &)> f) { // O(log(n))
  auto l = hull.begin();
  auto r = hull.end();
  SegIt b = hull.end();                                      #8566
  T b_v;
  while (r - l > 2) {
    auto m = l + (r - l) / 2;
    T l_v = f(*l);
    T l_n_v = f(*(l + 1));                                   #3586
    T m_v = f(*m);
    T m_n_v = f(*(m + 1));
    if (b == hull.end() || l_v > b_v) {
      b = l; // If max is at l we may remove it from the range.
      b_v = l_v;                                             #7332
    }
    if (l_n_v > l_v) {
      if (m_v < l_v) {
        r = m;
      } else {                                               #7279
        if (m_n_v > m_v) {
          l = m + 1;
        } else {
          r = m + 1;
        }                                                    #0656
      }
    } else {
      if (m_v < l_v) {
        l = m + 1;
      } else {                                               #7311
        if (m_n_v > m_v) {
          l = m + 1;
        } else {
          r = m + 1;
        }                                                    #4469
      }
    }
  }
  T l_v = f(*l);
  if (b == hull.end() || l_v > b_v) {                        #9864
    b = l;
    b_v = l_v;
  }
  if (r - l > 1) {
    T l_n_v = f(*(l + 1));                                   #5972
    if (b == hull.end() || l_n_v > b_v) {
      b = l + 1;
      b_v = l_n_v;
    }
  }                                                          #9086
  return b;
}                                                            %9504
SegIt closest(Vec p) { // p can't be internal(can be on border),
                       // hull must have atleast 3 points
  Seg &ref_p = hull.front(); // O(log(n))
  return max(function<double(Seg &)>(
    [&p, &ref_p](
      Seg &seg) { // accuracy of used type should be coord^2
      if (p == seg.F) return 10 - M_PI;                      #0134
      Vec v1 = {seg.S.F - seg.F.F, seg.S.S - seg.F.S};
      Vec v2 = {p.F - seg.F.F, p.S - seg.F.S};
      ll c_p = cross(v1, v2);
      if (c_p > 0) { // order the backside by angle
        Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
        Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
        ll d_p = dot(v1, v2);                                #5063
        ll c_p = cross(v2, v1);
        return atan2(c_p, d_p) / 2;
      }
      ll d_p = dot(v1, v2);
      double res = atan2(d_p, c_p);                          #0469
```

```
      if (d_p <= 0 && res > 0) res = -M_PI;
      if (res > 0) {
        res += 20;
      } else {
        res = 10 - res;                                              #7417
      }
      return res;
    }));
}                                                                    %8283
template <int DIRECTION> // 1 or -1
Vec tan_point(Vec p) {   // can't be internal or on border
  //-1 iff CCW rotation of ray from p to res takes it away from
  // polygon?
  Seg &ref_p = hull.front(); // O(log(n))
  auto best_seg = max(function<double(Seg &)>(
    [&p, &ref_p](                                                    #5209
      Seg &seg) { // accuracy of used type should be coord^2
      Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
      Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
      ll d_p = dot(v1, v2);
      ll c_p = DIRECTION * cross(v2, v1);                            #9762
      return atan2(c_p, d_p); // order by signed angle
    }));
  return best_seg->F;
}                                                                    %5037
SegIt max_in_dir(Vec v) { // first is the ans. O(log(n))
  return max(
    function<ll(Seg &)>([&v](Seg &seg) { return dot(v, seg.F); }));
}                                                                    %9596
pair<SegIt, SegIt> intersections(Seg l) { // O(log(n))
  int x = l.S.F - l.F.F;
  int y = l.S.S - l.F.S;
  Vec dir = {-y, x};
  auto it_max = max_in_dir(dir);                                     #4740
  auto it_min = max_in_dir(MP(y, -x));
  ll opt_val = dot(dir, l.F);
  if (dot(dir, it_max->F) < opt_val ||
      dot(dir, it_min->F) > opt_val)
    return MP(hull.end(), hull.end());                               #0276
  SegIt it_r1, it_r2;
  function<bool(const Seg &, const Seg &)> inc_c(
    [&dir](const Seg &lft, const Seg &rgt) {
      return dot(dir, lft.F) < dot(dir, rgt.F);
    });                                                              #1828
  function<bool(const Seg &, const Seg &)> dec_c(
    [&dir](const Seg &lft, const Seg &rgt) {
      return dot(dir, lft.F) > dot(dir, rgt.F);
    });
  if (it_min <= it_max) {                                            #8337
    it_r1 = upper_bound(it_min, it_max + 1, l, inc_c) - 1;
    if (dot(dir, hull.front().F) >= opt_val) {
      it_r2 = upper_bound(hull.begin(), it_min + 1, l, dec_c) - 1;
    } else {
      it_r2 = upper_bound(it_max, hull.end(), l, dec_c) - 1;
    }                                                                #1848
  } else {
    it_r1 = upper_bound(it_max, it_min + 1, l, dec_c) - 1;
    if (dot(dir, hull.front().F) <= opt_val) {
      it_r2 = upper_bound(hull.begin(), it_max + 1, l, inc_c) - 1;
    } else {                                                         #3840
      it_r2 = upper_bound(it_min, hull.end(), l, inc_c) - 1;
    }
  }
  return MP(it_r1, it_r2);
}                                   #5268                            %5268
Seg diameter() { // O(n)
  Seg res;
  ll dia_sq = 0;
  auto it1 = hull.begin();
  auto it2 = up_beg;                                                 #2632
  Vec v1 = {hull.back().S.F - hull.back().F.F,
    hull.back().S.S - hull.back().F.S};
  while (it2 != hull.begin()) {
    Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F,
      (it2 - 1)->S.S - (it2 - 1)->F.S};                              #5150
    if (cross(v1, v2) > 0) break;
    --it2;
  }
  while (it2 != hull.end()) { // check all antipodal pairs
    if (dist_sq(it1->F, it2->F) > dia_sq) {                          #1246
      res = {it1->F, it2->F};
      dia_sq = dist_sq(res.F, res.S);
    }
    Vec v1 = {it1->S.F - it1->F.F, it1->S.S - it1->F.S};
    Vec v2 = {it2->S.F - it2->F.F, it2->S.S - it2->F.S};
    if (cross(v1, v2) == 0) {                                        #9381
      if (dist_sq(it1->S, it2->F) > dia_sq) {
        res = {it1->S, it2->F};
        dia_sq = dist_sq(res.F, res.S);
      }
      if (dist_sq(it1->F, it2->S) > dia_sq) {                        #7011
        res = {it1->F, it2->S};
        dia_sq = dist_sq(res.F, res.S);
      } // report cross pairs at parallel lines.
      ++it1;
      ++it2;                                                         #5626
    } else if (cross(v1, v2) < 0) {
      ++it1;
    } else {
      ++it2;
    }                                                                #4406
  }
```

```
    return res;
  }
};                                                          %9383
```

11 Delaunay triangulation $\mathcal{O}(n \log n)$

```
const int max_co = (1 << 28) - 5;
struct Vec {
  int x, y;
  bool operator==(const Vec &oth) { return x == oth.x && y == oth.y; }
  bool operator!=(const Vec &oth) { return !operator==(oth); }
  Vec operator-(const Vec &oth) { return {x - oth.x, y - oth.y}; }
};                                                          #2919
ll cross(Vec a, Vec b) { return (ll)a.x * b.y - (ll)a.y * b.x; }
ll dot(Vec a, Vec b) { return (ll)a.x * b.x + (ll)a.y * b.y; }
struct Edge {
  Vec tar;
  Edge *nxt;                                                #8008
  Edge *inv = NULL;
  Edge *rep = NULL;
  bool vis = false;
};
struct Seg {                                                #7311
  Vec a, b;
  bool operator==(const Seg &oth) { return a == oth.a && b == oth.b; }
  bool operator!=(const Seg &oth) { return !operator==(oth); }
};
ll orient(Vec a, Vec b, Vec c) {                            #6432
  return (ll)a.x * (b.y - c.y) + (ll)b.x * (c.y - a.y) +
         (ll)c.x * (a.y - b.y);
}                                                           %6334
bool in_c_circle(Vec *arr, Vec d) {
  if (cross(arr[1] - arr[0], arr[2] - arr[0]) == 0)
    return true; // degenerate
  ll m[3][3];
  ran(i, 0, 3) {                                            #4264
    m[i][0] = arr[i].x - d.x;
    m[i][1] = arr[i].y - d.y;
    m[i][2] = m[i][0] * m[i][0];
    m[i][2] += m[i][1] * m[i][1];
  }                                                         #7305
  __int128 res = 0;
  res += (__int128)(m[0][0] * m[1][1] - m[0][1] * m[1][0]) * m[2][2];
  res += (__int128)(m[1][0] * m[2][1] - m[1][1] * m[2][0]) * m[0][2];
  res -= (__int128)(m[0][0] * m[2][1] - m[0][1] * m[2][0]) * m[1][2];
  return res > 0;                                           #1845
}                                                           %6793
Edge *add_triangle(Edge *a, Edge *b, Edge *c) {
  Edge *old[] = {a, b, c};
  Edge *tmp = new Edge[3];
  ran(i, 0, 3) {
    old[i]->rep = tmp + i;                                  #8219
    tmp[i] = {old[i]->tar, tmp + (i + 1) % 3, old[i]->inv};
    if (tmp[i].inv) tmp[i].inv->inv = tmp + i;
  }
  return tmp;
}                                                           #8178
Edge *add_point(Vec p, Edge *cur) { // returns outgoing edge
  Edge *triangle[] = {cur, cur->nxt, cur->nxt->nxt};
  ran(i, 0, 3) {
    if (orient(triangle[i]->tar, triangle[(i + 1) % 3]->tar, p) < 0)
      return NULL;                                          #0233
  }
  ran(i, 0, 3) {
    if (triangle[i]->rep) {
      Edge *res = add_point(p, triangle[i]->rep);
      if (res)                                              #5636
        return res; // unless we are on last layer we must exit here
    }
  }
  Edge p_as_e{p};
  Edge tmp{cur->tar};                                       #1432
  tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
  Edge *res = tmp.inv->nxt;
  tmp.tar = cur->tar;
  tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
  tmp.tar = cur->tar;                                       #8359
  res->inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
  res->inv->inv = res;
  return res;
}
Edge *delaunay(vector<Vec> &points) {                       #3029
  random_shuffle(points.begin(), points.end());
  Vec arr[] = {{4 * max_co, 4 * max_co}, {-4 * max_co, max_co},
    {max_co, -4 * max_co}};
  Edge *res = new Edge[3];
  ran(i, 0, 3) res[i] = {arr[i], res + (i + 1) % 3};
  for (Vec &cur : points) {                                 #4575
    Edge *loc = add_point(cur, res);
    Edge *out = loc;
    arr[0] = cur;
    while (true) {
      arr[1] = out->tar;                                    #3471
      arr[2] = out->nxt->tar;
      Edge *e = out->nxt->inv;
      if (e && in_c_circle(arr, e->nxt->tar)) {
        Edge tmp{cur};
        tmp.inv = add_triangle(&tmp, out, e->nxt);
        tmp.tar = e->nxt->tar;                              #9851
        tmp.inv->inv = add_triangle(&tmp, e->nxt->nxt, out->nxt->nxt);
        out = tmp.inv->nxt;
        continue;
      }
      out = out->nxt->nxt->inv;                             #0151
```

```
      if (out->tar == loc->tar) break;
    }
  }
  return res;
}                                  #6769                                %6769
void extract_triangles(Edge *cur, vector<vector<Seg> > &res) {
  if (!cur->vis) {
    bool inc = true;
    Edge *it = cur;
    do {                                                                #3769
      it->vis = true;
      if (it->rep) {
        extract_triangles(it->rep, res);
        inc = false;
      }                                                                 #2104
      it = it->nxt;
    } while (it != cur);
    if (inc) {
      Edge *triangle[3] = {cur, cur->nxt, cur->nxt->nxt};
      res.resize(res.size() + 1);                                       #6207
      vector<Seg> &tar = res.back();
      ran(i, 0, 3) {
        if ((abs(triangle[i]->tar.x) < max_co &&
              abs(triangle[(i + 1) % 3]->tar.x) < max_co))
          tar.push_back(                                                #3011
            {triangle[i]->tar, triangle[(i + 1) % 3]->tar});
      }
      if (tar.empty()) res.pop_back();
    }
  }                                                                     #8602
}                                                                       %5626
```

12 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;                                                          #1006
};
Node *aho_corasick(vector<vector<char> > &dict) {
  Node *root = new Node;
  root->suffix = 0;
  vector<pair<vector<char> *, Node *> > state;                          #9056
  for (vector<char> &s : dict) state.emplace_back(&s, root);
  for (int i = 0; !state.empty(); ++i) {
    vector<pair<vector<char> *, Node *> > nstate;
    for (auto &cur : state) {
      Node *nxt = cur.second->nxt[(*cur.first)[i]];
      if (nxt) {                                                        #1331
        cur.second = nxt;
      } else {
        nxt = new Node;
        cur.second->nxt[(*cur.first)[i]] = nxt;
        Node *suf = cur.second->suffix;                                 #5283
        cur.second = nxt;
        nxt->suffix = root; // set correct suffix link
        while (suf) {
          if (suf->nxt[(*cur.first)[i]]) {
            nxt->suffix = suf->nxt[(*cur.first)[i]];
            break;                                                      #3580
          }
          suf = suf->suffix;
        }
      }
      if (cur.first->size() > i + 1) nstate.push_back(cur);
    }                                                                   #3263
    state = nstate;
  }
  return root;
}                                                                       %2882
// 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) return cur;                                       #5414
    cur = cur->suffix;
  }
}                                                                       %1529
void cnt_matches(Node *root, vector<char> &match_in) {
  Node *cur = root;
  for (char c : match_in) {
    cur = walk(cur, c);
    ++cur->cnt;                                                         #0015
  }
}                                                                       %8156
void add_cnt(Node *root) { // After counting matches propagete ONCE to
                           // suffixes for final counts
  vector<Node *> to_visit = {root};
  ran(i, 0, to_visit.size()) {
    Node *cur = to_visit[i];
    ran(j, 0, alpha_size) {                                             #0662
      if (cur->nxt[j]) to_visit.push_back(cur->nxt[j]);
    }
  }
  for (int i = to_visit.size() - 1; i > 0; --i)
    to_visit[i]->suffix->cnt += to_visit[i]->cnt;                       #7950
}                                                                       %0488
int main() {
  int n, len;
  scanf("%d %d", &len, &n);
  vector<char> a(len + 1);
```

```
  scanf("%s", a.data());
  a.pop_back();
  for (char &c : a) c -= 'a';
  vector<vector<char> > dict(n);
  ran(i, 0, n) {
    scanf("%d", &len);
    dict[i].resize(len + 1);
    scanf("%s", dict[i].data());
    dict[i].pop_back();
    for (char &c : dict[i]) c -= 'a';
  }
  Node *root = aho_corasick(dict);
  cnt_matches(root, a);
  add_cnt(root);
  ran(i, 0, n) {
    Node *cur = root;
    for (char c : dict[i]) cur = walk(cur, c);
    printf("%d\n", cur->cnt);
  }
}
```

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

```
struct Node {
  map<char, Node *> nxt_char;
  // Map is faster than hashtable and unsorted arrays
  int len; // Length of longest suffix in equivalence class.
  Node *suf;
  bool has_nxt(char c) const { return nxt_char.count(c); }
  Node *nxt(char c) {                                          #9664
    if (!has_nxt(c)) return NULL;
    return nxt_char[c];
  }
  void set_nxt(char c, Node *node) { nxt_char[c] = node; }
  Node *split(int new_len, char c) {                           #8305
    Node *new_n = new Node;
    new_n->nxt_char = nxt_char;
    new_n->len = new_len;
    new_n->suf = suf;
    suf = new_n;                                               #4595
    return new_n;
  }                                                            %3114
  // Extra functions for matching and counting
  Node *lower(int depth) {
    // move to longest suf of current with a maximum length of depth.
    if (suf->len >= depth) return suf->lower(depth);
    return this;
  }
  Node *walk(char c, int depth, int &match_len) {              #2130
    // 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);
    }                                                          #9589
    if (suf) return suf->walk(c, depth, match_len);
    return this;
  }                                                            %2262
  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();
  }                                                            #3041
  bool vis = false;
  void calc_paths() {
    /* Call ONCE from ROOT. For each node  calculates number of ways
     * to reach an end node. paths_to_end is ocurence count for any
     * strings in current suffix equivalence class. */
    if (!vis) {
      vis = true;
      for (auto cur : nxt_char) {                              #2404
        cur.second->calc_paths();
        paths_to_end += cur.second->paths_to_end;
      }
    }
  }                                  #7906                     %7906
  // Transform into suffix tree of reverse string
  map<char, Node *> tree_links;
  int end_dist = 1 << 30;
  int calc_end_dist() {
    if (end_dist == 1 << 30) {
      if (nxt_char.empty()) end_dist = 0;                      #7524
      for (auto cur : nxt_char)
        end_dist = min(end_dist, 1 + cur.second->calc_end_dist());
    }
    return end_dist;
  }                                                            #2021
  bool vis_t = false;
  void build_suffix_tree(string &s) { // Call ONCE from ROOT.
    if (!vis_t) {
      vis_t = true;
      if (suf)                                                 #6270
        suf->tree_links[s[s.size() - end_dist - suf->len - 1]] = this;
      for (auto cur : nxt_char) cur.second->build_suffix_tree(s);
    }
  }
};                                   #1268                     %1268
struct SufAuto {
  Node *last;
  Node *root;
  void extend(char new_c) {
```

```
    Node *nlast = new Node;                                    #0936
    nlast->len = last->len + 1;
    Node *swn = last;
    while (swn && !swn->has_nxt(new_c)) {
      swn->set_nxt(new_c, nlast);
      swn = swn->suf;                                          #1831
    }
    if (!swn) {
      nlast->suf = root;
    } else {
      Node *max_sbstr = swn->nxt(new_c);                       #0855
      if (swn->len + 1 == max_sbstr->len) {
        nlast->suf = max_sbstr;
      } else { // remove for minimal DFA that matches suffixes and
               // crap
        Node *eq_sbstr = max_sbstr->split(swn->len + 1, new_c);
        nlast->suf = eq_sbstr;                                 #1749
        Node *x = swn; // x = with_edge_to_eq_sbstr
        while (x != 0 && x->nxt(new_c) == max_sbstr) {
          x->set_nxt(new_c, eq_sbstr);
          x = x->suf;
        }                                                      #4310
      }
    }
    last = nlast;
  }                                                            %6146
  SufAuto(string &s) {
    root = new Node;
    root->len = 0;
    root->suf = NULL;
    last = root;                                               #9604
    for (char c : s) extend(c);
    root->calc_end_dist(); // To build suffix tree use reversed string
    root->build_suffix_tree(s);
  }
};                               #6251                         %6251
```

14 Dinic

```
struct MaxFlow {
  const static ll INF = 1e18;
  int source, sink;
  vector<int> start, now, lvl, adj, rcap, cap_loc, bfs;
  vector<int> cap, orig_cap; /*ly*/
  ll sink_pot = 0;
  vector<bool> visited;
  vector<ll> cost;
  priority_queue<pair<ll, int>, vector<pair<ll, int> >,
    greater<pair<ll, int> > >
    dist_que; /*ry*/
  void add_flow(int idx, ll flow, bool cont = true) {
    cap[idx] -= flow;
    if (cont) add_flow(rcap[idx], -flow, false);
  }
  MaxFlow(
    const vector<tuple<int, int, ll, ll /*ly*/, ll /*ry*/> > &edges) {
    for (auto &cur : edges) { // from, to, cap, rcap/*ly*/, cost/*ry*/
      start.resize(
        max(max(get<0>(cur), get<1>(cur)) + 2, (int)start.size()));
      ++start[get<0>(cur) + 1];
      ++start[get<1>(cur) + 1];
    }
    ran(i, 1, (int)start.size()) start[i] += start[i - 1];
    now = start;
    adj.resize(start.back());
    cap.resize(start.back());
    rcap.resize(start.back()); /*ly*/
    cost.resize(start.back()); /*ry*/
    for (auto &cur : edges) {
      int u, v;
      ll c, rc /*ly*/, c_cost /*ry*/;
      tie(u, v, c, rc /*ly*/, c_cost /*ry*/) = cur;
      assert(u != v);
      adj[now[u]] = v;
      adj[now[v]] = u;
      rcap[now[u]] = now[v];
      rcap[now[v]] = now[u];
      cap_loc.push_back(now[u]); /*ly*/
      cost[now[u]] = c_cost;
      cost[now[v]] = -c_cost; /*ry*/
      cap[now[u]++] = c;
      cap[now[v]++] = rc;
      orig_cap.push_back(c);
    }
  }
  bool dinic_bfs(int min_cap) {
    lvl.clear();
    lvl.resize(start.size());
    bfs.clear();
    bfs.resize(1, source);
    now = start;
    lvl[source] = 1;
    ran(i, 0, (int)bfs.size()) {
      int u = bfs[i];
      while (now[u] < start[u + 1]) {
        int v = adj[now[u]];
        if (/*ly*/ cost[now[u]] == 0 &&
            /*ry*/ cap[now[u]] >= min_cap && lvl[v] == 0) {
          lvl[v] = lvl[u] + 1;
          if (v == sink) return true;
          bfs.push_back(v);
        }
        ++now[u];
      }
```

```
    }
    return false;
  }
  ll dinic_dfs(int u, ll flow, int min_cap) {
    if (u == sink) return flow;
    if (lvl[u] == lvl[sink]) return 0;
    ll res = 0;
    while (now[u] < start[u + 1]) {
      int v = adj[now[u]];
      if (lvl[v] == lvl[u] + 1 /*ly*/ && cost[now[u]] == 0 /*ry*/ &&
          cap[now[u]] >= min_cap) {
        ll cur = dinic_dfs(v, min(flow, (ll)cap[now[u]]), min_cap);
        if (cur) {
          add_flow(now[u], cur);
          flow -= cur;
          res += cur;
          if (!flow) break;
        }
      }
      ++now[u];
    }
    return res;
  } /*ly*/
  bool recalc_dist(bool check_imp = false) {
    now = start;
    visited.clear();
    visited.resize(start.size());
    dist_que.emplace(0, source); /*lp*/
    bool imp = false;            /*rp*/
    while (!dist_que.empty()) {
      int u;
      ll dist;
      tie(dist, u) = dist_que.top();
      dist_que.pop();
      if (!visited[u]) {
        visited[u] = true;                   /*lp*/
        if (check_imp && dist != 0) imp = true; /*rp*/
        if (u == sink) sink_pot += dist;
        while (now[u] < start[u + 1]) {
          int v = adj[now[u]];
          if (!visited[v] && cap[now[u]])
            dist_que.emplace(dist + cost[now[u]], v);
          cost[now[u]] += dist;
          cost[rcap[now[u]++]] -= dist;
        }
      }
    }                          /*lp*/
    if (check_imp) return imp; /*rp*/
    return visited[sink];
  } /*ry*/                         /*lp*/
  bool recalc_dist_bellman_ford() { // return whether there is
                                    // a negative cycle
    int i = 0;
    for (; i < (int)start.size() - 1 && recalc_dist(true); ++i) {
    }
    return i == (int)start.size() - 1;
  } /*rp*/
  /*ly*/ pair<ll, /*ry*/ ll /*ly*/> /*ry*/ calc_flow(
    int _source, int _sink) {
    source = _source;
    sink = _sink;
    assert(max(source, sink) < start.size() - 1); /*ly*/
    ll tot_flow = 0;
    ll tot_cost = 0; /*lp*/
    if (recalc_dist_bellman_ford()) {
      assert(false);
    } else {                  /*rp*/
      while (recalc_dist()) { /*ry*/
        ll flow = 0;
        for (int min_cap = 1 << 30; min_cap; min_cap >>= 1) {
          while (dinic_bfs(min_cap)) {
            now = start;
            ll cur;
            while (cur = dinic_dfs(source, INF, min_cap)) flow += cur;
          }
        } /*ly*/
        tot_flow += flow;
        tot_cost += sink_pot * flow; /*ry*/
      }
    }
    return /*ly*/ {tot_ /*ry*/ flow /*ly*/, tot_cost} /*ry*/;
  }
  ll flow_on_edge(int idx) {
    assert(idx < cap.size());
    return orig_cap[idx] - cap[cap_loc[idx]];
  }
};
const int nmax = 1055;
int main() {
  int t;
  scanf("%d", &t);
  for (int i = 0; i < t; ++i) {
    vector<tuple<int, int, ll, ll, ll> > edges;
    int n;
    scanf("%d", &n);
    for (int j = 1; j <= n; ++j) {
      edges.emplace_back(j, 2 * n + 1, 1, 0, 0);
    }
    for (int j = 1; j <= n; ++j) {
      int card;
      scanf("%d", &card);
      edges.emplace_back(0, card, 1, 0, 0);
    }
```

```
    int ex_c;
    scanf("%d", &ex_c);
    for (int j = 0; j < ex_c; ++j) {
      int a, b;
      scanf("%d %d", &a, &b);
      if (b < a) swap(a, b);
      edges.emplace_back(a, b, nmax, 0, 1);
      edges.emplace_back(b, n + b, nmax, 0, 0);
      edges.emplace_back(n + b, a, nmax, 0, 1);
    }
    int v = 2 * n + 2;
    MaxFlow mf(edges);
    printf("%d\n", (int)mf.calc_flow(0, v - 1).second);
    // cout << mf.flow_on_edge(edge_index) << endl;
  }
}
```

15 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\textbf{cap} \cdot nm)$

```
struct Network {
  struct Node;
  struct Edge {
    Node *u, *v;
    int f, c, cost;                                        #2965
    Node* from(Node* pos) {
      if (pos == u) return v;
      return u;
    }
    int getCap(Node* pos) {                                #4145
      if (pos == u) return c - f;
      return f;
    }
    int addFlow(Node* pos, int toAdd) {
      if (pos == u) {                                      #6369
        f += toAdd;
        return toAdd * cost;
      } else {
        f -= toAdd;
        return -toAdd * cost;                              #8987
      }
    }
  };
  struct Node {
    vector<Edge*> conn;                                    #1577
    int index;
  };
  deque<Node> nodes;
  deque<Edge> edges;
  Node* addNode() {                                        #5057
    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});                   #5123
    u->conn.push_back(&edges.back());
    v->conn.push_back(&edges.back());
    return &edges.back();
  }
  // Assumes all needed flow has already been added
  int minCostMaxFlow() {                                   #0927
    int n = nodes.size();
    int result = 0;
    struct State {
      int p;
      Edge* used;                                          #7358
    };
    while (1) {
      vector<vector<State> > state(1, vector<State>(n, {0, 0}));
      for (int lev = 0; lev < n; lev++) {
        state.push_back(state[lev]);                       #0078
        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 =                                   #7871
                  state[lev][i].p +
                  (edge->u == &nodes[i] ? edge->cost : -edge->cost);
                int ni = edge->from(&nodes[i])->index;
                if (np < state[lev + 1][ni].p) {
                  state[lev + 1][ni].p = np;               #3940
                  state[lev + 1][ni].used = edge;
                }
              }
            }
          }                                                #3693
        }
      }
      // Now look at the last level
      bool valid = false;
      for (int i = 0; i < n; i++)
        if (state[n - 1][i].p > state[n][i].p) {           #5398
          valid = true;
          vector<Edge*> path;
          int cap = 1000000000;
          Node* cur = &nodes[i];
          int clev = n;                                    #6663
          vector<bool> explr(n, false);
          while (!explr[cur->index]) {
            explr[cur->index] = true;
            State cstate = state[clev][cur->index];
            cur = cstate.used->from(cur);                  #3984
            path.push_back(cstate.used);
          }
```

```
        reverse(path.begin(), path.end());
        {
          int i = 0;                                          #9784
          Node* cur2 = cur;
          do {
            cur2 = path[i]->from(cur2);
            i++;
          } while (cur2 != cur);                              #9838
          path.resize(i);
        }
        for (auto edge : path) {
          cap = min(cap, edge->getCap(cur));
          cur = edge->from(cur);                              #8867
        }
        for (auto edge : path) {
          result += edge->addFlow(cur, cap);
          cur = edge->from(cur);
        }                                                     #4467
      }
      if (!valid) break;
    }
    return result;
  }                                                           #4029
};                                                            %2900
```

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

```
struct EdgeDesc {
  int from, to, w;
};
struct DMST {
  struct Node;                                                #6091
  struct Edge {
    Node *from;
    Node *tar;
    int w;
    bool inc;                                                 #2186
  };
  struct Circle {
    bool vis = false;
    vector<Edge *> cont;
    void clean(int idx);                                      #4353
  };
  const static greater<pair<ll, Edge *> > comp;
  static vector<Circle> to_proc;
  static bool no_dmst;
  static Node *root; // Can use inline static since C++17
  struct Node {                                               #9916
    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() {                                             #0564
      if (!par) return this;
      while (par->par) par = par->par;
      return par;
    }
    void clean() {                                            #0300
      if (!no_dmst) {
        in_use = false;
        for (auto &cur : out_cands)
          to_proc[cur.first].clean(cur.second);
      }                                                       #0747
    }
    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) {                                 #3927
        while (
          !con.empty() && con.front().second->tar->anc() == anc()) {
          pop_heap(con.begin(), con.end(), comp);
          con.pop_back();
        }                                                     #2561
        if (con.empty()) {
          no_dmst = true;
          return root;
        }
        pop_heap(con.begin(), con.end(), comp);               #8600
        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();                     #6612
          to_proc.resize(to_proc.size() + 1);
        } else {
          super = nxt.second->tar->con_to_root();
        }
        if (super != root) {                                  #7005
          to_proc.back().cont.push_back(nxt.second);
          out_cands.emplace_back(
            to_proc.size() - 1, to_proc.back().cont.size() - 1);
        } else { // Clean circles
          nxt.second->inc = true;                             #1096
          nxt.second->from->clean();
        }
      }
      if (super != root) { // we are some loops non first Node.
        if (con.size() > super->con.size()) {                 #2844
          swap(con,
            super->con); // Largest con in loop should not be copied.
```

```
          swap(w, super->w);
        }
        for (auto cur : con) {                                  #3498
          super->con.emplace_back(
            cur.first - super->w + w, cur.second);
          push_heap(super->con.begin(), super->con.end(), comp);
        }
      }                                                         #6348
      par = super; // root or anc() of first Node encountered in a
                   // loop
      return super;
    }
  };
  Node *croot;                                                  #0309
  vector<Node> graph;
  vector<Edge> edges;
  DMST(int n, vector<EdgeDesc> &desc,
    int r) { // Self loops and multiple edges are okay.
    graph.resize(n);                                            #8100
    croot = &graph[r];
    for (auto &cur : desc) // Edges are reversed internally
      edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
    for (int i = 0; i < desc.size(); ++i)
      graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]);
    for (int i = 0; i < n; ++i)                                 #8811
      make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
  }
  bool find() {
    root = croot;
    no_dmst = false;                                            #5307
    for (auto &cur : graph) {
      cur.con_to_root();
      to_proc.clear();
      if (no_dmst) return false;
    }                                                           #6725
    return true;
  }                                                             %1568
  ll weight() {
    ll res = 0;
    for (auto &cur : edges) {
      if (cur.inc) res += cur.w;
    }                                                           #6369
    return res;
  }                                                             %1477
};
void DMST::Circle::clean(int idx) {
  if (!vis) {
    vis = true;
    for (int i = 0; i < cont.size(); ++i) {                     #6503
      if (i != idx) {
        cont[i]->inc = true;
        cont[i]->from->clean();
      }
    }                                                           #8144
  }
}
const greater<pair<ll, DMST::Edge *> > DMST::comp;
vector<DMST::Circle> DMST::to_proc;
bool DMST::no_dmst;                                             #2354
DMST::Node *DMST::root;                                         %2870
```

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

```
struct vert;
struct edge {
  bool exists = true;
  vert *dest;
  edge *rev;                                                    #8922
  edge(vert *_dest) : dest(_dest) { rev = NULL; }
  vert &operator*() { return *dest; }
  vert *operator->() { return dest; }
  bool is_bridge();
};                                                              #0116
struct vert {
  deque<edge> con;
  int val = 0;
  int seen;
  int dfs(int upd, edge *ban) { // handles multiple edges
    if (!val) {                                                 #1288
      val = upd;
      seen = val;
      for (edge &nxt : con) {
        if (nxt.exists && (&nxt) != ban)
          seen = min(seen, nxt->dfs(upd + 1, nxt.rev));
      }                                                         #8194
    }
    return seen;
  }                                                             %8624
  void remove_adj_bridges() {
    for (edge &nxt : con) {
      if (nxt.is_bridge()) nxt.exists = false;
    }
  }                             #7106                           %7106
  int cnt_adj_bridges() {
    int res = 0;
    for (edge &nxt : con) res += nxt.is_bridge();
    return res;
  }                             #9056                           %9056
};
bool edge::is_bridge() {
  return exists &&
         (dest->seen > rev->dest->val || dest->val < rev->dest->seen);
}                               #5223                           %5223
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;
}
```

18 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$

```
struct Graph {
  int n;
  vector<vector<int> > con;
  Graph(int nsize) {
    n = nsize;                                                #0321
    con.resize(n);
  }
  void add_edge(int u, int v) { con[u].push_back(v); }
  void top_dfs(int pos, vector<int> &result, vector<bool> &explr,
    vector<vector<int> > &revcon) {                           #2422
    if (explr[pos]) return;
    explr[pos] = true;
    for (auto next : revcon[pos])
      top_dfs(next, result, explr, revcon);
    result.push_back(pos);                                    #2081
  }                                                           %7763
  vector<int> topsort() {
    vector<vector<int> > revcon(n);
    ran(u, 0, n) {
      for (auto v : con[u]) revcon[v].push_back(u);
    }                                                         #3875
    vector<int> result;
    vector<bool> explr(n, false);
    ran(i, 0, n) top_dfs(i, result, explr, revcon);
    reverse(result.begin(), result.end());
    return result;                                            #7568
  }                                                           %5339
  void dfs(int pos, vector<int> &result, vector<bool> &explr) {
    if (explr[pos]) return;
    explr[pos] = true;
    for (auto next : con[pos]) dfs(next, result, explr);
    result.push_back(pos);                                    #6880
  }                                                           %3565
  vector<vector<int> > scc() {
    vector<int> order = topsort();
    reverse(order.begin(), order.end());
    vector<bool> explr(n, false);
    vector<vector<int> > res;                                 #9931
    for (auto it = order.rbegin(); it != order.rend(); ++it) {
      vector<int> comp;
      top_dfs(*it, comp, explr, con);
      sort(comp.begin(), comp.end());
      res.push_back(comp);                                    #2243
    }
    sort(res.begin(), res.end());
    return res;
  }
};                          #0543                             %0543
int main() {
  int n, m;
  cin >> n >> m;
  Graph g(2 * m);
  ran(i, 0, n) {
    int a, sa, b, sb;
    cin >> a >> sa >> b >> sb;
    a--, b--;
    g.add_edge(2 * a + 1 - sa, 2 * b + sb);
    g.add_edge(2 * b + 1 - sb, 2 * a + sa);
  }
  vector<int> state(2 * m, 0);
  {
    vector<int> order = g.topsort();
    vector<bool> explr(2 * m, false);
    for (auto u : order) {
      vector<int> traversed;
      g.dfs(u, traversed, explr);
      if (traversed.size() > 0 && !state[traversed[0] ^ 1]) {
        for (auto c : traversed) state[c] = 1;
      }
    }
  }
  ran(i, 0, m) {
    if (state[2 * i] == state[2 * i + 1]) {
      cout << "IMPOSSIBLE\n";
      return 0;
    }
  }
  ran(i, 0, m) cout << state[2 * i + 1] << '\n';
  return 0;
}
```

19 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;                                  #7684
  }
} __attribute__((packed));
struct Lazy {
  ll add;
  ll assign_val; // LLONG_MIN if no assign;
  void init() {                                                         #7883
    add = 0;
    assign_val = LLONG_MIN;
  }
  Lazy() { init(); }
  void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len) {
    lhs_lazy = *this;                                                   #7654
    rhs_lazy = *this;
    init();
  }
  void merge(Lazy &oth, int len) {
    if (oth.assign_val != LLONG_MIN) {                                  #0050
      add = 0;
      assign_val = oth.assign_val;
    }
    add += oth.add;
  }                                                                     #2924
  void apply_to_seg(Seg &cur, int len) const {
    if (assign_val != LLONG_MIN) {
      cur.sum = len * assign_val;
    }
    cur.sum += len * add;                                               #6280
  }
} __attribute__((packed));
 %0625 struct Node { // Following code should not need to be modified
  int ver;
  bool is_lazy = false;
  Seg seg;
  Lazy lazy;                                                            #6321
  Node *lc = NULL, *rc = NULL;
  void init() {
    if (!lc) {
      lc = new Node{ver};
      rc = new Node{ver};                                               #5313
    }
  }
  Node *upd(int L, int R, int l, int r, Lazy &val, int tar_ver) {
    if (ver != tar_ver) {
      Node *rep = new Node(*this);                                      #8874
      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);                                     #2138
      lazy.merge(val, R - L);
      is_lazy = true;
    } else {
      init();
      int M = (L + R) / 2;                                              #8209
      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);                           #8104
        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);                       #8581
    }
    return this;
  }
  void get(int L, int R, int l, int r, Seg *&lft_res, Seg *&tmp,
    bool last_ver) {                                                    #9373
    if (L >= l && R <= r) {
      tmp->recalc(*lft_res, L - l, seg, R - L);
      swap(lft_res, tmp);
    } else {
      init();                                                           #6654
      int M = (L + R) / 2;
      if (is_lazy) {
        Lazy l_val, r_val;
        lazy.split(l_val, r_val, R - L);
        lc = lc->upd(L, M, L, M, l_val, ver + last_ver);
        lc->ver = ver;                                                  #2185
        rc = rc->upd(M, R, M, R, r_val, ver + last_ver);
        rc->ver = ver;
        is_lazy = false;
      }
      if (l < M) lc->get(L, M, l, r, lft_res, tmp, last_ver);
      if (M < r) rc->get(M, R, l, r, lft_res, tmp, last_ver);
    }                                                                   #4770
  }
} __attribute__((packed));
struct SegTree {        // indexes start from 0, ranges are [beg, end)
  vector<Node *> roots; // versions start from 0
  int len;                                                              #4873
  SegTree(int _len) : len(_len) { roots.push_back(new Node{0}); }
  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;                                            #1461
  }
  Seg get(int l, int r, int ver = -1) {
```

```
    if (ver == -1) ver = roots.size() - 1;
    Seg seg1, seg2;
    Seg *pres = &seg1, *ptmp = &seg2;                          #9427
    roots[ver]->get(0, len, l, r, pres, ptmp, roots.size() - 1);
    return *pres;
  }
};
                                              %7542 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);
    }
  }
}
```

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

```
class dummy {
 public:
  dummy() {}
  dummy(int, int) {}
  void set(int, int) {}                                      #9531
  int query(int left, int right) {
    cout << this << ' ' << left << ' ' << right << endl;
  }
};                                                           %7932
/* 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;                                  #6178
  DS structure;
  DS aux;
  void build_sizes(int vertex, int parent) {
    subtree_size[vertex] = 1;
    for (int child : adj[vertex]) {                          #2037
      if (child != parent) {
        build_sizes(child, vertex);
        subtree_size[vertex] += subtree_size[child];
      }
    }                                                        #6759
  }
  int cur;
  vector<int> ord;
  vector<int> chain_root;
  vector<int> par;                                           #9593
  void build_hld(int vertex, int parent, int chain_source) {
    cur++;
    ord[vertex] = cur;
    chain_root[vertex] = chain_source;
    par[vertex] = parent;                                    #0432
    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;                                 #9151
          big_size = subtree_size[child];
        }
      }
      build_hld(big_child, vertex, chain_source);
      for (int child : adj[vertex]) {                        #3027
        if ((child != parent) && (child != big_child))
          build_hld(child, vertex, child);
      }
    }
  }                                                          #8562
 public:
  HLD(int _vertexc) {
    vertexc = _vertexc;
    adj = new vector<int>[vertexc + 5];
  }                                                          #3486
```

```
  void add_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  void build(T initial) {                                              #4566
    subtree_size = vector<int>(vertexc + 5);
    ord = vector<int>(vertexc + 5);
    chain_root = vector<int>(vertexc + 5);
    par = vector<int>(vertexc + 5);
    cur = 0;                                                           #2693
    build_sizes(1, -1);
    build_hld(1, -1, 1);
    structure = DS(vertexc + 5, initial);
    aux = DS(50, initial);
  }                                                                    #7758
  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;                                                    #4754
    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]];                                        #4538
      } else {
        cur_id++;
        aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
        v = par[chain_root[v]];
      }                                                                #1595
    }
    cur_id++;
    aux.set(cur_id,
      structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
    return aux.query(1, cur_id);                                       #7150
  }                                                                    %1905
  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;
  }
}
```

21 Splay Tree + Link-Cut $O(N \log N)$

```
struct Tree *treev;
struct Tree {
  struct T {
    int i;
    constexpr T() : i(-1) {}                                           #7635
    T(int _i) : i(_i) {}
    operator int() const { return i; }
    explicit operator bool() const { return i != -1; }
    Tree *operator->() { return treev + i; }
  };                                                                   #2337
  T c[2], p;
  /* insert monoid here */
  /*lg*/ T link; /*rg*/
  Tree() {
    /* init monoid here */
    /*lg*/ link = -1; /*rg*/
  }                                                                    #0939
};
using T = Tree::T;
constexpr T NIL;
void update(T t) { /* recalculate the monoid here */
}                                                                      #3006
void propagate(T t) {
  assert(t);
  /*lg*/
  for (T c : t->c)
    if (c) c->link = t->link;
  /*rg*/
  /* lazily propagate updates here */
}                                                                      #8514
/*lp*/
void lazy_reverse(T t) { /* lazily reverse t here */
}
/*rp*/
T splay(T n) {
  for (;;) {
    propagate(n);                                                      #3792
    T p = n->p;
```

```
    if (p == NIL) break;
    propagate(p);
    ll px = p->c[1] == n;
    assert(p->c[px] == n);                                    #0245
    T g = p->p;
    if (g == NIL) { /* zig */
      p->c[px] = n->c[px ^ 1];
      p->c[px]->p = p;
      n->c[px ^ 1] = p;                                       #4750
      n->c[px ^ 1]->p = n;
      n->p = NIL;
      update(p);
      update(n);
      break;                                                  #8981
    }
    propagate(g);
    ll gx = g->c[1] == p;
    assert(g->c[gx] == p);
    T gg = g->p;                                              #5659
    ll ggx = gg && gg->c[1] == g;
    if (gg) assert(gg->c[ggx] == g);
    if (gx == px) { /* zig zig */
      g->c[gx] = p->c[gx ^ 1];
      g->c[gx]->p = g;                                        #5608
      p->c[gx ^ 1] = g;
      p->c[gx ^ 1]->p = p;
      p->c[gx] = n->c[gx ^ 1];
      p->c[gx]->p = p;
      n->c[gx ^ 1] = p;                                       #1439
      n->c[gx ^ 1]->p = n;
    } else { /* zig zag */
      g->c[gx] = n->c[gx ^ 1];
      g->c[gx]->p = g;
      n->c[gx ^ 1] = g;                                       #9140
      n->c[gx ^ 1]->p = n;
      p->c[gx ^ 1] = n->c[gx];
      p->c[gx ^ 1]->p = p;
      n->c[gx] = p;
      n->c[gx]->p = n;                                        #2928
    }
    if (gg) gg->c[ggx] = n;
    n->p = gg;
    update(g);
    update(p);                                                #9645
    update(n);
    if (gg) update(gg);
  }
  return n;
}                                                             #4920
T extreme(T t, int x) {
  while (t->c[x]) t = t->c[x];
  return t;
}
T set_child(T t, int x, T a) {                                #2821
  T o = t->c[x];
  t->c[x] = a;
  update(t);
  o->p = NIL;
  a->p = t;                                                   #4262
  return o;
}
/************* Link-Cut Tree: *************/
T expose(T t) {
  set_child(splay(t), 1, NIL);
  T leader = splay(extreme(t, 0));
  if (leader->link == NIL) return t;
  set_child(splay(leader), 0, expose(leader->link));
  return splay(t);
}
void link(T t, T p) {
  assert(t->link == NIL);
  t->link = p;
}
T cut(T t) {
  T p = t->link;
  if (p) expose(p);
  t->link = NIL;
  return p;
}
/*lp*/
void make_root(T t) {
  expose(t);
  lazy_reverse(extreme(splay(t), 0));                         #4240
}
/*rp*/                                                        %8430
```

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

```
// Fully overloaded any dimensional BIT, use any type for coordinates,
// elements, return_value. Includes coordinate compression.
template <typename E_T, typename C_T, C_T n_inf, typename R_T>
struct BIT {
  vector<C_T> pos;
  vector<E_T> elems;
  bool act = false;                                           #3273
  BIT() { pos.push_back(n_inf); }
  void init() {
    if (act) {
      for (E_T &c_elem : elems) c_elem.init();
    } else {                                                  #2594
      act = true;
      sort(pos.begin(), pos.end());
      pos.resize(unique(pos.begin(), pos.end()) - pos.begin());
      elems.resize(pos.size());
```

```
    }                                                                #7774
  }
  template <typename... loc_form>
  void update(C_T cx, loc_form... args) {
    if (act) {
      int x = lower_bound(pos.begin(), pos.end(), cx) - pos.begin();
      for (; x < (int)pos.size(); x += x & -x)                       #7303
        elems[x].update(args...);
    } else {
      pos.push_back(cx);
    }
  }                                                                  #8505
  template <typename... loc_form>
  R_T query(C_T cx, loc_form... args) { // sum in (-inf, cx)
    R_T res = 0;
    int x = lower_bound(pos.begin(), pos.end(), cx) - pos.begin() - 1;
    for (; x > 0; x -= x & -x) res += elems[x].query(args...);
    return res;                                                      #2526
  }
};
template <typename I_T>
struct wrapped {
  I_T a = 0;                                                         #6509
  void update(I_T b) { a += b; }
  I_T query() { return a; }
  // Should never be called, needed for compilation
  void init() { DEBUG('i') }
  void update() { DEBUG('u') }
};                                                                   #2858
                                                    %2858 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 pos that are to be used for update
  for (int i = 0; i < dim; ++i) {
    for (auto &cur : to_insert)
      fenwick.update(get<0>(cur), get<1>(cur));
    // May include value which won't be used
    fenwick.init();
  }
  // actual use
  for (auto &cur : to_insert)
    fenwick.update(get<0>(cur), get<1>(cur), get<2>(cur));
  cout << fenwick.query(2, 2) << '\n';
}
```

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

```
mt19937 randgen;
struct Treap {
  struct Node {
    int key;
    int value;                                                       #5615
    unsigned int priority;
    long long total;
    Node* lch;
    Node* rch;
    Node(int new_key, int new_value) {                               #5698
      key = new_key;
      value = new_value;
      priority = randgen();
      total = new_value;
      lch = 0;                                                       #7232
      rch = 0;
    }
    void update() {
      total = value;
      if (lch) total += lch->total;                                  #4295
      if (rch) total += rch->total;
    }
  };
  deque<Node> nodes;
  Node* root = 0;                                                    #9633
  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);                               #5233
      cur->lch = ret.second;
      result = {ret.first, cur};
    } else {
      auto ret = split(key, cur->rch);
      cur->rch = ret.first;                                          #6988
      result = {cur, ret.second};
    }
    cur->update();
    return result;
  }                                                                  #7230
  Node* merge(Node* left, Node* right) {
    if (left == 0) return right;
    if (right == 0) return left;
    Node* top;
    if (left->priority < right->priority) {                          #6282
      left->rch = merge(left->rch, right);
      top = left;
    } else {
      right->lch = merge(left, right->lch);
      top = right;                                                   #3510
    }
    top->update();
    return top;
  }
```

```
  void insert(int key, int value) {                                  #8918
    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);                                     #9760
    root = cur;
  }
  void erase(int key) {
    Node *left, *mid, *right;
    tie(left, mid) = split(key, root);                                #1416
    tie(mid, right) = split(key + 1, mid);
    root = merge(left, right);
  }
  long long sum_upto(int key, Node* cur) {
    if (cur == 0) return 0;                                           #7634
    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;                        #8122
      return result;
    }
  }
  long long get(int l, int r) {
    return sum_upto(r + 1, root) - sum_upto(l, root);
  }                                                                   #0094
};                                                                    %4959
// 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;
}
```

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

```
template <typename T>
void rsort(T *a, T *b, int size, int d = sizeof(T) - 1) {
  int b_s[256]{};
  ran(i, 0, size) { ++b_s[(a[i] >> (d * 8)) & 255]; }
  // ++b_s[*((uchar *)(a + i) + d)];
  T *mem[257];                                                        #5369
  mem[0] = b;
  T **l_b = mem + 1;
  l_b[0] = b;
  ran(i, 0, 255) { l_b[i + 1] = l_b[i] + b_s[i]; }
  for (T *it = a; it != a + size; ++it) {                             #6813
    T id = ((*it) >> (d * 8)) & 255;
    *(l_b[id]++) = *it;
  }
  l_b = mem;
  if (d) {                                                            #5681
    T *l_a[256];
    l_a[0] = a;
    ran(i, 0, 255) l_a[i + 1] = l_a[i] + b_s[i];
    ran(i, 0, 256) {
      if (l_b[i + 1] - l_b[i] < 100) {                                #1162
        sort(l_b[i], l_b[i + 1]);
        if (d & 1) copy(l_b[i], l_b[i + 1], l_a[i]);
      } else {
        rsort(l_b[i], l_a[i], b_s[i], d - 1);
      }                                                               #7759
    }
  }
}                                                                     %0571
const int nmax = 5e7;
ll arr[nmax], tmp[nmax];
int main() {
  for (int i = 0; i < nmax; ++i) arr[i] = ((ll)rand() << 32) | rand();
  rsort(arr, tmp, nmax);
  assert(is_sorted(arr, arr + nmax));
}
```

25 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;                                                         #1139
    return *this;
  }
```

```
};
Complex operator+(const Complex &lft, const Complex &rgt) {
  return Complex{lft.a + rgt.a, lft.b + rgt.b};                #8384
}
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{                                              #5371
    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}; }
void fft_rec(Complex *arr, Complex *root_pow, int len) {
  if (len != 1) {                                              #7637
    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) {                              #0670
    Complex tmp = arr[i] + root_pow[i] * arr[i + len];
    arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
    arr[i] = tmp;
  }
}                                                              #7078
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;                                  #0102
  if (inc_pow <= ord) {
    int idx = root_pow.size();
    root_pow.resize(1 << ord);
    for (; inc_pow <= ord; ++inc_pow) {
      for (int idx_p = 0; idx_p < 1 << (ord - 1);              #3349
           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;
      }                                                        #6357
    }
  }
  if (invert != is_inv) {
    is_inv = invert;
    for (Complex &cur : root_pow) cur.b = -cur.b;              #7526
  }
  for (int i = 1, j = 0; i < (1 << ord); ++i) {
    int m = 1 << (ord - 1);
    bool cont = true;
    while (cont) {                                             #0510
      cont = j & m;
      j ^= m;
      m >>= 1;
    }
    if (i < j) swap(arr[i], arr[j]);                           #0506
  }
  fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
  if (invert)
    for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord);
}                                   #4380                      %4380
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) {                                        #8811
    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());                #4629
    if (arr[0].size() != 1 << fft_ord)
      for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord);
    for (int i = 0; i < 4; ++i)
      fill(arr[i].begin(), arr[i].end(), Complex{});
    for (int &cur : a)                                         #9591
      if (cur < 0) cur += mod;
    for (int &cur : b)
      if (cur < 0) cur += mod;
    const int shift = 15;
    const int mask = (1 << shift) - 1;                         #2625
    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;
    }
    for (int i = 0; i < min(b.size(), c.size()); ++i) {
      arr[0][i].b = b[i] & mask;                               #3501
      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) {                            #9971
        int tar = 2 + (i + j) / 2;
        Complex mult = {0, -0.25};
        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;
        }                                                      #4471
      }
    }
    for (int i = 2; i < 4; ++i) {
      fft(arr[i], fft_ord, true);
```

```
      for (int k = 0; k < (int)c.size(); ++k) {            #8403
        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;                                     #8289
      }
    }
  }
}                                                           %1231
```

26 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;                                         #0237
  }
  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;                                  #0780
    return res; // in [0, n-1)
  }
  ull sqp1(ull a) { return multf(a, a) + 1; }
};                                                          %9493
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);                                 #1758
  }
  return res;
}                                                           %2144
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;                        #8104
  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;                                      #6402
    a = pow_mod(a, d, arithm);
    if (a == 1 || a == n - 1) continue;
    for (ull r = 1; r < s; ++r) {
      a = arithm.multf(a, a);
      if (a == 1) return false;                             #0876
      if (a == n - 1) break;
    }
    if (a != n - 1) return false;
  }
  return true;                                              #4806
}                                                           %0975
ll pollard_rho(ll n) {
  ModArithm arithm(n);
  int cum_cnt = 64 - __builtin_clz(n);
  cum_cnt *= cum_cnt / 5 + 1;
  while (true) {                                            #2118
    ll lv = rand() % n;
    ll v = arithm.sqp1(lv);
    int idx = 1;
    int tar = 1;
    while (true) {                                          #5290
      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));           #4468
        v_cur = arithm.sqp1(v_cur);
        ++idx;
      }
      if (!cur) {
        for (int j = 0; j < cum_cnt; ++j) {                 #7912
          ll g = __gcd(abs(v - lv), n);
          if (g == 1) {
            v = arithm.sqp1(v);
          } else if (g == n) {
            break;                                          #0906
          } else {
            return g;
          }
        }
        break;                                              #7208
      } else {
        ll g = __gcd(cur, n);
        if (g != 1) return g;
      }
      v = v_cur;                                            #2298
      idx += j_stop;
      if (idx == tar) {
        lv = v;
        tar *= 2;
        v = arithm.sqp1(v);                                 #1174
        ++idx;
      }
    }
  }
}                                #3542                      %3542
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) {                                #3770
      while (!(n % p)) {
        ++res_act[p];
        n /= p;
      }
    }                                                           #4612
    if (n != 1) prime_factor(n, &res_act);
    return res_act;
  }
  if (is_prime(n)) {
    ++(*res)[n];                                                #1963
  } else {
    ll factor = pollard_rho(n);
    prime_factor(factor, res);
    prime_factor(n / factor, res);
  }                                                             #5350
  return map<ll, int>();
} // Usage: fact = prime
_factor(n);                                                     %5477
```

27 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 $v \notin I$. breaklines

```
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)
```

28 Berlekamp-Massey $O(\mathcal{L}N)$

```
template <typename K>
static vector<K> berlekamp_massey(vector<K> ss) {
  vector<K> ts(ss.size());
  vector<K> cs(ss.size());
  cs[0] = K::unity;                                             #0349
  fill(cs.begin() + 1, cs.end(), K::zero);
  vector<K> bs = cs;
  int l = 0, m = 1;
  K b = K::unity;
  for (int k = 0; k < (int)ss.size(); k++) {                    #4390
    K d = ss[k];
    assert(l <= k);
    for (int i = 1; i <= l; i++) d += cs[i] * ss[k - i];
    if (d == K::zero) {
      m++;                                                      #8445
    } else if (2 * l <= k) {
      K w = d / b;
      ts = cs;
      for (int i = 0; i < (int)cs.size() - m; i++)
        cs[i + m] -= w * bs[i];                                 #9661
      l = k + 1 - l;
      swap(bs, ts);
      b = d;
      m = 1;
    } else {                                                    #2815
      K w = d / b;
      for (int i = 0; i < (int)cs.size() - m; i++)
        cs[i + m] -= w * bs[i];
      m++;
    }                                                           #8888
  }
  cs.resize(l + 1);
  while (cs.back() == K::zero) cs.pop_back();
  return cs;
}                                  #6267                        %6267
```