Monash University Team Reference Document

1 Geometry **2**
2D Computational Geometry 2
3D Computational Geometry 6
Delauney Triangulation 7
Great Circle Distance 7
Integer Computational Geometry 7
Area of Union of Rectangles 8

2 Number Theory **8**
Extended Euclidean Algorithm 8
Modular Multiplicative Inverse 8
Chinese Remainder Algorithm 8
Euler's Totient Function 8
Sieve of Eratosthenes 8
Primality Testing 8
Prime Factorisation 9
Primitive Roots 9
Discrete Roots 9
Discrete Logarithm 9
Fast Convolution using Number Theoretic Transform 9

3 Combinatorics **10**
Generate Set Partitions 10
Generate Integer Partitions 10
Generate Subsets and Combinations 10

4 Dynamic Programming **10**
Longest Increasing Subsequence 10
Longest Palindromic Substring 10
Monotonic Queue 11
Convex Hull Trick 11
Divide and Conquer Optimisation 11
Knuth's Optimisation 11

5 Graph Algorithms **11**
Bellman-Ford Algorithm 11
Articulation Points, Bridges, Biconnected Components and Bridge-Connected Components 12
Eulerian Path and Tour 12
Lowest Common Ancestor and Tree Distances 12
Maximum Flow (Dinic's Algorithm in $\mathcal{O}(V^2E)$) 13
Maximum Flow (Push Relabel in $\mathcal{O}(V^2\sqrt{E})$) 13
Maximum Bipartite Matching 14
Maximum Matching on a General Graph 14
Stable Matching / Stable Marriage Problem 14
Hungarian Algorithm (Minimum weight bipartite matching) 14
Minimum Cost Flow 14
Minimum Cut 15
Strongly Connected Components 15
Minimum Mean Cycle 15
Minimum Cost Arborescence 16

6 Tree Decomposition Techniques **16**
Heavy-Light Decomposition 16
Centroid Decomposition 16

7 Linear Algebra **16**
Reduced Row Echelon Form 16
System of Linear Diophantine Equations Solver 17
Fraction Free Integral Linear System Solver 17
Floating-Point Linear Solver Using LU-Decomposition 17

8 Data Structures **17**
Fenwick Tree 17
Sparse Table for Static Range Minimum Query 18
Segment Tree for Dynamic Range Minimum Query 18
Segment Tree with Lazy Propagation 18
Multidimensional Segment Tree 18
Union Find 19
Link-Cut Tree 19
Treap 20
Implicit Key Treap (Implicit Cartesian Tree / Rope) 20
GNU Policy-Based Data Structures (Prefix Trie, Rope, Order Statistics Tree) 21

9 String Processing **21**
KMP 21
Z-Algorithm 21
Suffix Tree 21
Suffix Automaton 22
Suffix Array ($\mathcal{O}(N)$ and $\mathcal{O}(N\log(N))$) 22
Aho-Corasick Algorithm 23

10 Miscellaneous **23**
Date Conversion 23
2-SATISFIABILITY 23
Cubic Equation Solver 23
Hashing Custom Types 23
Conversion to and from Roman Numerals 24
Josephus Problem 24
Fast Convolution using Fast Fourier Transform 24
Numerical Integration and Differentiation 24
Linear Programming (Simplex Algorithm) 24

11 Formulas and Theorems **25**

```
clear; clear
g++ $1 -g -Og -std=gnu++14 -Wall -Wextra -Wconversion -Wshadow -Wfatal-errors -
    fsanitize=address,undefined -o sol || exit

for i in *.in; do
  echo --- $i
  ./sol < $i > o && (diff -y o ${i::-3}.[ao]?? > t || cat t) || cat o
done
```

```
// Monash University
// Peter, Xin Wei, Daniel
// << LOCATION >>
#include<bits/stdc++.h>
using namespace std;

#define X first
#define Y second

#define debug(a) cerr << #a << " = " << (a) << endl;

typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef vector<vi> vvi;

template<typename T> ostream& operator<<(ostream& o, const vector<T>& v) {
  int b=0; for (const auto& a : v) o << (b++ ? " " : "") << a; return o;
}

int main() {
  ios::sync_with_stdio(0);  cin.tie(0);

}
```

```
chmod +x run
f = (sample/*)
for i in {A..Z}; do
  mkdir $i
  cp t.cpp $i/$i.cpp
  cp ${f[n++]}/* $i
  ls $i/*
done
setxkbmap us,us ,colemak grp:rwin_toggle
```

```
chmod +x run
for i in {A..Z}; do
  mkdir $i
  cp t.cpp $i/$i.cpp
  cp sample/$i* $i
  cp sample/${i,}* $i
  ls $i/*
done
setxkbmap us,us ,colemak grp:rwin_toggle
```

1 Geometry

```
// -------------------------- 2D Computational Geometry ------------------------------
#define x real()
#define y imag()
#define cpt const Pt&

const double EPS = 1e-9;
const double pi=acos(-1);
const double inf=1e100;
bool deq(double a,double b) {return abs(a-b)<EPS;}

typedef complex<double> cpx;
struct Pt : cpx {
  Pt() = default; using cpx::cpx;
  Pt(cpx a) : cpx(a) {}
  double& x const { return (double&)*this; }
  double& y const { return ((double*)this)[1]; }
  bool operator ==(cpt b) const {return abs(*this-b) < EPS; }
  bool operator <(cpt b) const {return x<b.x || (x==b.x && y<b.y); }
};

bool epsless(cpt a,cpt b) {return a.x+EPS<b.x || (deq(a.x,b.x) && a.y<b.y);}

double dot(cpt a, cpt b) {return (conj(a) * b).x;} // Dot product
double det(cpt a, cpt b) {return (conj(a) * b).y;} // Determinant/"Cross Product"
double angle(cpt a, cpt b) {return arg(b - a);} // [-pi,pi] a to b with x axis
double angle (cpt a, cpt b, cpt c) {return arg((a-b)/(c-b));} //[-pi,pi]
double slope(cpt a, cpt b) {return (b.y-a.y)/(b.x-a.x);}

Pt rotate(cpt a, double theta) {return a * polar((double)1.0, theta);}
// rotate a around p by theta anticlockwise
Pt rotate(cpt a, cpt p, double theta) {return rotate(a - p,theta) + p;}
Pt project(cpt p, cpt v) {return v * dot(p, v) / norm(v);} // p onto v
Pt project(cpt p, cpt a, cpt b) {return a+project(p-a,b-a);} // p onto line (a,b)
// reflect p across the line (a,b)
Pt reflect(cpt p, cpt a, cpt b) {return a + conj((p - a) / (b - a)) * (b - a);}

bool collinear(Pt a, Pt b, Pt c) {return deq(det(b-a,c-b),0);}
bool areperp(cpt a,cpt b,cpt p,cpt q) { return deq(dot(b-a,q-p),0); }
bool arepara(cpt a, cpt b, cpt p, cpt q) { return deq(det(b-a,q-p),0); }

// Orientation test (1 anticlockwise, -1 clockwise, 0 collinear)
int orient(cpt a, cpt b, cpt c) {
  double d=det(b-a,c-b);
  return d>EPS?1:d<-EPS?-1:0;
}

//Compare points by principal argument (-pi,pi] breaking ties by norm.
//0 is considered less than everything else.
bool argcomp(cpt a,cpt b) {
  if (b==0) return 0;
  if (a==0) return 1;
  double a1=arg(a),a2=arg(b);
  if (a1<-pi+EPS/2) a1+=2*pi;
  if (a2<-pi+EPS/2) a2+=2*pi;
  return a1+EPS<a2 || (deq(a1,a2) && norm(a)<norm(b));
}

// Point on line segment (including endpoints)
bool ptonseg(cpt a, cpt b, cpt p) {
  Pt u=b-a,v=p-a;
  return a==p || b==p || ((0<dot(u,v) && dot(u,v)<norm(u)) && deq(det(u,v),0));
}

// Signed area of polygon. Positive for anticlockwise orientation.
double polygonarea(const vector<Pt>& p) {
  double r=0; int n=p.size();
  for (int j=0,i=n-1;j<n;i=j++) r+=det(p[i],p[j]);
  return r/2;
}

// Convex hull O(NlogN). Be careful of duplicate or very close points.
// if all points are colinear the middle points come up twice forwards and
// backwards e.g. a-b-c-d becomes a-b-c-d-c-b
// To remove colinear points change <-EPS and >EPS to <EPS and >-EPS.
vector<Pt> convexhull(vector<Pt> p) {
  sort(p.begin(),p.end(),epsless); p.resize(unique(p.begin(),p.end())-p.begin());
  int l=0,u=0; vector<Pt> L(p),U(p);
  if (p.size()<=2) return p;
  for (Pt& i:p) {
    while (l>1 && det(i-L[l-1],L[l-2]-i)<-EPS) l--;
    while (u>1 && det(i-U[u-1],U[u-2]-i)>EPS) u--;
    L[l++]=U[u++]=i;
  }
```

```
  L.resize(l+u-2); copy(U.rend()-u+1,U.rend()-1,L.begin()+1);
  return L;
}

// Point in polygon test O(N)
// Returns: 0 if not in polygon, 1 if on boundary, 2 if in interior
int ptinpoly(const vector<Pt>& p, cpt q) {
  int n=p.size(), i,j,r=0;
  for (j=0,i=n-1;j<n;i=j++) {
    if (ptonseg(p[i],p[j],q)) return 1;
    if (((p[i].y <= q.y && q.y < p[j].y) || (p[j].y <= q.y && q.y < p[i].y))
      && q.x < (p[j].x-p[i].x) * (q.y-p[i].y)/(p[j].y-p[i].y) + p[i].x) r^=2;
  }
  return r;
}

// Point in polygon test for convex polygons. P must not contain colinear points.
// boundary = true if points on the boundary are considered to be in the polygon.
// Complexity: O(log(N))
bool point_in_convex_polygon(const vector<Pt>& P, const Pt& p, bool boundary) {
  int a = 1, b = (int)P.size()-1;
  if (ptonseg(P[a],P[0],p) || ptonseg(P[b],P[0],p))
    return boundary; else if (orient(P[a],P[0],P[b]) > 0) swap(a,b);
  if (orient(P[a],P[0],p) > 0 || orient(P[b],P[0],p) < 0) return false;
  while (abs(a-b) > 1) {
    int c = (a+b)/2;
    if (orient(P[c],P[0],p) > 0) b = c; else a = c;
  }
  return orient(P[b],P[a],p) < 0 || (orient(P[b],P[a],p)==0 && boundary);
}

Pt solve(cpt a, cpt b, cpt v) { // solves [a b]x==v with Cramer's rule.
  return Pt(det(v,b)/det(a,b),det(a,v)/det(a,b));
}

//Intersection of 2 line segments. Divides by 0 if they are parallel.
//Returns {nan,nan} if they don't intersect.
//Remove if statements below to get infinite lines.
Pt intersectline(Pt a, Pt b, Pt p, Pt q) {
  Pt ab=b-a,qp=p-q,ap=p-a;
  double s=det(ap,qp)/det(ab,qp),t=det(ab,ap)/det(ab,qp);
  if (-EPS<s && s<1+EPS //Answer is on ab
    && -EPS<t && t<1+EPS) //Answer is on pq
      return a+s*ab;
  return Pt(NAN,NAN);
}

Pt intersectlineexact(Pt a, Pt b, Pt p, Pt q) {
  Pt ab=b-a,qp=p-q,ap=p-a;
  double s=det(ap,qp)/det(ab,qp),t=det(ab,ap)/det(ab,qp);
  if (0<s && s<1 //Answer is on ab
    && 0<t && t<1) //Answer is on pq
      return a+s*ab;
  return Pt(NAN,NAN);
}

//Distance between infinite line and point.
double distlinept(cpt a, cpt b, cpt p) { return abs(det(b-a,p-a)/abs(b-a)); }

//Distance between finite line and point
double distfinitelinept(Pt a, Pt b, Pt p) {
  b-=a;p-=a;  Pt closest;  double sp=(p/b).x; //dot(b,p)/norm(b);
  if (sp>=0) {
    if (sp>1) closest=b;
    else closest=sp*b;
  }
  return abs(closest-p); // Note that actual closest Pt on line is closest + a
}

//Distance between 2 finite lines
double distfinitelineline(cpt a,cpt b,cpt p,cpt q) {
  if (!arepara(a,b,p,q) && !std::isnan(intersectlineexact(a,b,p,q).x)) return 0;
  return min({distfinitelinept(a,b,p),distfinitelinept(a,b,q),
    distfinitelinept(p,q,a),distfinitelinept(p,q,b)});
}

struct Circle {
  Pt c;double r;
  bool operator==(const Circle& b) const {return c==b.c && deq(r,b.r);}
};

// Number of intersections, pair containing intersections
// 3 means infinitely many intersections. This also happens with identical
// radius 0 circles.
pair<int,pair<Pt,Pt>> intersect(const Circle& a,const Circle& b) {
  Pt v=b.c-a.c; // disjoint || one inside other
  if (a.r+b.r+EPS<abs(v)    || abs(a.r-b.r)>abs(v)+EPS) return {0,{}};
  if (abs(v)<EPS) return {3,{}};
  double X=(norm(a.r)-norm(b.r)+norm(v))/(2.0*abs(v)), Ysq=norm(a.r)-norm(X),Y;
  v/=abs(v);
  if (Ysq<0 || (Y=sqrt(Ysq))<EPS) return {1,{Pt{X,0}*v+a.c,{}}};
  return {2,{Pt{X,Y}*v+a.c,Pt{X,-Y}*v+a.c}};
}

pair<int,pair<Pt,Pt>> intersectfinitelinecircle(cpt a,cpt b,Circle c) {
  Pt v=b-a; v/=abs(v);  c.c=(c.c-a)/v;
  if (c.r+EPS<abs(c.c.y)) return {0,{}};
  double offsq=norm(c.r)-norm(c.c.y),off;
  if (offsq<0 || (off=sqrt(offsq))<EPS)
  if (-EPS<c.c.x && c.c.x<abs(v)+EPS) return {1,{Pt{c.c.x,0}*v+a,{}}};
  pair<int,pair<Pt,Pt>> ans;
  for (int sgn=-1;sgn<2;sgn+=2) {
    double X=c.c.x+sgn*off;
    if (-EPS<X && X<abs(v)+EPS) { // line bounds check
      if (ans.X==0) ans.Y.X=Pt{X,0}*v+a;
      else ans.Y.Y=Pt{X,0}*v+a;
      ans.X++;
    }
  }
  return ans;
}

Circle circlefrom3points(cpt a,cpt b,cpt c) {
  Pt v=b-a; double X=abs(v); v/=X; Pt p=(c-a)/v;
  if (deq(det(v,c-a),0)) return {0,-1}; // Not unique or infinite if collinear
  Pt q(X/2,(norm(p.x)-norm(p.y)-p.x*X)/(2*p.y));
  return {q*v+a,abs(q)};
}

// Peter's custom array
template<class T,int maxn> struct Arr { int n=0; T a[maxn]={}; };

// Each tangent is two points in Arr.
// These points represent where the tangent touches each circle. If these points
// are the same then the second point is to the right of the first (when looking
// from the center of the first circle), and the distance between the two points
// is the distance between the centers of the circles.
// Outer tangents are before inner tangents since they occur whenever inner
// tangents do. The first tangent in each group is the one which intersects the
// first circle to the left of the second circle (when looking from the center
// of the first circle).
// The radii should be positive. 0 radii should work but give multiple identical lines.
Arr<Pt,8> commontangents(const Circle& a,const Circle& b) {
  Arr<Pt,8> ans; Pt v=b.c-a.c; double X=abs(v); v/=norm(X); int &n=ans.n;
```

```
  if (a==b) {ans.n=9; return ans;} // infinitely many
  for (int sgn=-1;sgn<2;sgn+=2) {
    Pt u=a.r+sgn*b.r;
    if (X+EPS<abs(u.x)) break;
    u.y=norm(X)-norm(u.x), u.y=u.y>0?sqrt(u.y):0;
    ans.a[n++]=a.r*u, ans.a[n++]=(a.r+(u.y<EPS?X:u.y)*Pt(0,-1))*u;
    if (u.y>=EPS) ans.a[n++]=a.r*conj(u), ans.a[n++]=(a.r-u.y*Pt(0,-1))*conj(u);
  }
  for (int i=0;i<n;i++) ans.a[i]=ans.a[i]*v+a.c;
  return ans;
}

// Find the max dot product of a point in p with v. p must be a convex polygon
// where no three points are collinear and dot(p[l],v)<dot(p[l+1],v). O(log n)
int maxdot(int l,int r,const vector<Pt>& p,cpt v) {
  if (r-l<10) {
    int i=l;
    for (int j=l+1;j<r;j++) if (dot(p[i],v)<dot(p[j],v)) i=j;
    return i;
  }
  int m1=(2*l+r)/3,m2=(l+2*r)/3; double d1=dot(p[m1],v),d2=dot(p[m2],v);
  if (d1<dot(p[l],v)) return maxdot(l,m1,p,v);
  if (d1>d2) return maxdot(l,m2,p,v);
  return maxdot(m1+1,r,p,v);
}

// Min and max dot product of a point in p with v. p must be a convex polygon
// where no three points are collinear. Indices are returned. O(log n)
pii minmaxdot(const vector<Pt>& p,cpt v) {
  int i=deq(dot(p[0],v),dot(p[1],v)), n=p.size(),m,M;
  if (dot(p[i],v)<dot(p[i+1],v)) M=maxdot(i,n,p,v),m=maxdot(M,n,p,-v);
  else m=maxdot(i,n,p,-v),M=maxdot(m,n,p,v);
  for (int j=0;j<=i;j++) {
    if (dot(p[j],v)>dot(p[M],v)) M=j;
    if (dot(p[j],v)<dot(p[m],v)) m=j;
  }
  return {m,M};
}

//Returns convex hull of all points x within the convex polygon p, which satisfy
//det(b-a,x-a)>=0. Returned polygon may be degenerate if the cut runs across an
//edge. For p ordered counterclockwise, the cut polygon is on the left of a->b
vector<Pt> convexcut(cpt a,cpt b,const vector<Pt>& p) {
  int n=p.size(); vector<Pt> r;
  for (int i=n-1,j=0;j<n;i=j++) {
    double d1=det(b-a,p[i]-a),d2=det(b-a,p[j]-a);
    if (d1>-EPS) r.push_back(p[i]);
    if ((d1>EPS && d2<-EPS) || (d1<-EPS && d2>EPS))
      r.push_back(intersectline(a,b,p[i],p[j])); //infinite lines
  }
  return r;
}

// Facilitates queries for the pair of furthest visible points on a convex polygon from
// some external point. P must be a non-degenerate convex polygon. Returns an empty
// interval for p inside P. Complexity: O(N log(N)) pre-process, O(log(N)) per query.
struct ExtremePoints {
  vector<Pt> P; Pt c; int n; vi ids;
  ExtremePoints(vector<Pt> poly) : P(move(poly)), n(P.size()), ids(2*n) {
    for (auto p : P) c += 1.0/n * p;
    for (auto& p : P) p -= c;  sort(P.begin(), P.end(), argcomp);
    iota(ids.begin(),ids.begin()+n,0), iota(ids.begin()+n,ids.end(),0);
  }
  pii query(Pt p) { // Returns {i,j} such that P[i..j] are visible to p
    int a = lower_bound(P.begin(),P.end(),p-c,argcomp)-P.begin();
    int b = lower_bound(P.begin(),P.end(),-(p-c),argcomp)-P.begin();
    if (b < a) b += n;
    auto seen = [&](int i) { return orient(P[i?i-1:n-1],P[i],p-c) < 0; };
    int r = *partition_point(ids.begin()+a,ids.begin()+b,seen);
    int l = *partition_point(ids.rbegin()+n-a,ids.rbegin()+2*n-b,seen);
    return {l,r-1+(r?0:n)};
  }
  Pt operator[](int i) { return P[i]+c; }  // Get the (untranslated) i'th point
};

//Signed Area of polygon and circle intersection. Sign is determined by
//orientation of polygon. Divides by 0 if adjacent points are identical.
double areapolygoncircle(vector<Pt> p,Circle c) {
  int n=p.size(); double r=0;
  for (int i=n-1,j=0;j<n;i=j++) {
    Pt v=abs(p[j]-p[i])/(p[j]-p[i]), a=(p[i]-c.c)*v,b=(p[j]-c.c)*v;
    if (deq(a.y,0)) continue;
    double d=sqrt(max(0.0, norm(c.r)-norm(a.y)));
    r+=norm(c.r)*(atan2(b.y,min(b.x,-d))-atan2(a.y,min(a.x,-d))
        +atan2(b.y,max(b.x,d))-atan2(a.y,max(a.x,d)))
      +a.y*(min(max(a.x,-d),d)-min(max(b.x,-d),d));
  }
  return r/2;
}

// Closest pair of points. Complexity: O(N log(N))
pair<Pt,Pt> closest_pair(vector<Pt> P) {
  sort(P.begin(), P.end(), [](auto p1, auto p2) { return p1.y < p2.y; });
  set<Pt> a; double d = inf; pair<Pt,Pt> cp{{0,0},{inf,0}};
  for (auto p = P.begin(), c = p; c != P.end(); c++) {
    while (p != c && p->y < c->y-d) a.erase(*p++);
    for (auto i=a.lower_bound(Pt{c->x-d,0}); i != a.end() && i->x < c->x+d; i++)
      if (abs(*c - *i) < d) d = abs(*c - *i), cp = {*c, *i};
    a.insert(*c);
  }
  return cp;
}

//Diameter of convex polygon. Complexity: O(N)
double polygondiameter(const vector<Pt>& p) {
  int i=min_element(p.begin(),p.end())-p.begin(),ic=0,n=p.size(),ni=(i+1)%n;
  int j=max_element(p.begin(),p.end())-p.begin(),jc=0,nj=(j+1)%n;
  double r=0;
  while (ic<n || jc<n) {
    r=max(r,abs(p[j]-p[i]));
    if (det(p[ni]-p[i],p[j]-p[nj])>0) {
      i=ni++;ic++;
      if (ni==n) ni=0;
    }
    else {
      j=nj++;jc++;
      if (nj==n) nj=0;
    }
  }
  return r;
}

//Minimum width of a bounding rectangle of a convex polygon O(n)
//The polygon must have positive signed area.
double minboundingwidth(const vector<Pt>& p) {
  double r=DBL_MAX; int n=p.size();
  for (int i=n-1,j=0,k=0,nk;j<n;i=j++) {
    Pt v=p[j]-p[i];v/=abs(v);
    for (;det(v,p[nk=k+1==n?0:k+1]-p[i])>det(v,p[k]-p[i]);k=nk);
    r=min(r,det(v,p[k]-p[i]));
  }
  return r;
}
```

```
//Minkowski sum of convex polygons O(n)
//Polygon is returned with the minimum number of points. i.e. No three points
//will be collinear. The input polygons must have positive signed area.
vector<Pt> minkowskisum(const vector<Pt>& p,const vector<Pt>& q) {
  vector<Pt> r;  int n=p.size(),m=q.size();
  int i=min_element(p.begin(),p.end(),epsless)-p.begin(),oi=i,ni=(i+1)%n;
  int j=min_element(q.begin(),q.end(),epsless)-q.begin(),oj=j,nj=(j+1)%m;
  do {
    r.push_back(p[i]+q[j]);
    Pt v=det(p[ni]-p[i],q[nj]-q[j])>0?p[ni]-p[i]:q[nj]-q[j];
    while (det(v,p[ni]-p[i])<EPS) {
      i=ni++;
      if (ni==n) ni=0;
    }
    while (det(v,q[nj]-q[j])<EPS) {
      j=nj++;
      if (nj==m) nj=0;
    }
  } while (i!=oi || j!=oj);
  return r;
}

// Returns true if the given point is contained within the given circle
bool point_in_circle(const Pt& p, const Circle& c) { return abs(p - c.c) <= c.r + EPS; }
// Construct a circle from two antipodal points on the boundary
Circle circle_from_diameter(cpt a, cpt b) { return {0.5*(a+b), abs(0.5*(a+b) - a)}; }

// Find the smallest circle that encloses all of the given points. Complexity: O(N)
Circle minimum_enclosing_circle(vector<Pt> P) {
  int N = (int)P.size(); random_shuffle(P.begin(), P.end());
  Circle c{P[0], 0};
  for (int i=1; i<N; i++) if (!point_in_circle(P[i], c)) {
    c = Circle{P[i],0};
    for (int j=0; j<i; j++) if (!point_in_circle(P[j], c)) {
      c = circle_from_diameter(P[i],P[j]);
      for (int k=0; k<j; k++) if (!point_in_circle(P[k], c))
        c = circlefrom3points(P[i],P[j],P[k]);
    }
  }
  return c;
}

// Find the area of the union of the given circles. Complexity: O(n^2 log(n))
double circle_union_area(const vector<Circle>& cir) {
  int n = (int)cir.size(); vector<bool> ok(n, 1); double ans = 0.0;
  for (int i=0; i<n; i++) for (int j=0; j<n; j++) if (i != j && ok[j])
    if (abs(cir[i].c - cir[j].c)+cir[i].r-cir[j].r < EPS) { ok[i] = false; break; }
  for (int i=0; i<n; i++) if (ok[i]) {
    bool flag = false;  vector<pair<double,double>> reg;
    for (int j=0; j<n; j++) if (i != j && ok[j]) {
      auto p = intersect(cir[i], cir[j]);
      if (p.X < 2) continue; else flag = true;
      auto ang1 = arg(p.Y.Y - cir[i].c), ang2 = arg(p.Y.X - cir[i].c);
      if (ang1 < 0) ang1 += 2*pi;
      if (ang2 < 0) ang2 += 2*pi;
      if (ang1 > ang2) reg.emplace_back(ang1, 2*pi), reg.emplace_back(0, ang2);
      else reg.emplace_back(ang1, ang2);
    }
    if (!flag) { ans += pi*cir[i].r*cir[i].r; continue; }
    int cnt = 1; sort(reg.begin(), reg.end());
    for (int j=1; j<(int)reg.size(); j++)
      if (reg[cnt-1].Y >= reg[j].X) reg[cnt-1].Y = max(reg[cnt-1].Y, reg[j].Y);
      else reg[cnt++] = reg[j];
    reg.emplace_back(0,0);  reg[cnt] = reg[0];
    for (int j=0; j<cnt; j++) {
      auto p1 = cir[i].c + polar(cir[i].r, reg[j].Y);
      auto p2 = cir[i].c + polar(cir[i].r, reg[j+1].X);
      ans += det(p1, p2) / 2.0;
      double ang = reg[j+1].X - reg[j].Y;
      if (ang < 0) ang += 2*pi;
      ans += 0.5 * cir[i].r*cir[i].r * (ang - sin(ang));
    }
  }
  return ans;
}

// Find a pair of intersecting lines. Complexity: O(N log(N))
#define cl const Line&
struct Line {
  Pt u,v; //Endpoints
  double m() const {return (u.y-v.y)/(u.x-v.x);}
  double c() const {return yv(0);}
  double yv(double X) const {return det(u-X,v-X)/(u.x-v.x);}
  bool operator<(cl b) const {return u<b.u || (!(b.u<u) && v<b.v);}
};
namespace FindIntersection {
  typedef pair<double,int> pdi;
  const int maxn=300000;  Line segs[maxn];  pdi ord[2*maxn];
  int sgndiff(double a,double b) {return (a+EPS<b)-(b+EPS<a);}
  bool comp(int i,int j) {
    cl a=segs[i],b=segs[j];
    int by,bg;
    if (deq(a.u.x,b.u.x)) by=sgndiff(a.u.y,b.u.y);
    else if (a.u.x<b.u.x) by=sgndiff(0,det(a.v-a.u,b.u-a.u));
    else by=sgndiff(det(b.v-b.u,a.u-b.u),0);
    bg=sgndiff(0,det(a.v-a.u,b.v-b.u));
    return by==1 || (by==0 && (bg==1 || (bg==0 && i<j)));
  }
  set<int,bool(*)(int,int)> L(comp);
  pii checkpair(int i,int j) {
    cl a=segs[i],b=segs[j]; Pt ab=a.v-a.u,qp=b.u-b.v,ap=b.u-a.u;
    double d1=det(ap,qp),d2=det(ab,ap),d3=det(ab,qp);
    if (d3<0) d1*=-1,d2*=-1,d3*=-1;
    if (deq(d3,0)) {// Parallel
      Pt v=ab/abs(ab);  double c=dot(v,b.u),d=dot(v,b.v);
      if (d<c) swap(c,d);
      return {max(c,dot(v,a.u))+EPS<min(d,dot(v,a.v)) && deq(d1,0)?i:-1,j};
    }
    if (-EPS<d1 && d1<d3+EPS && -EPS<d2 && d2<d3+EPS) {
      if (EPS<d1 && d1+EPS<d3) return {i,j};
      if (EPS<d2 && d2+EPS<d3) return {j,i};
    }
    return {-1,0};
  }
  // Returns a pair of indices such that the first segment's interior
  // intersects with the other segment. First item is -1 if there are no such
  // segments.
  pii findintersection(const vector<Line>& lines) {
    int n=lines.size();copy(lines.begin(),lines.end(),segs);L.clear(); pii r;
    for (int i=0;i<n;i++) {
      if (epsless(segs[i].v,segs[i].u)) swap(segs[i].u,segs[i].v);
      ord[2*i]={segs[i].u.x,i};
      ord[2*i+1]={segs[i].v.x,i+n};
    }
    sort(ord,ord+2*n,[](const pdi& a,const pdi& b) {
      return a.X+EPS<b.X || (deq(a.X,b.X) && a.Y<b.Y); });
    for (int i=0;i<2*n;i++) {
      int j=ord[i].Y;
      if (j<n) {
        auto oit=L.insert(j).X,it=oit++;
        if (oit!=L.end() && (r=checkpair(*it,*oit)).X!=-1) return r;
        if (it!=L.begin() && (r=checkpair(*prev(it),*it)).X!=-1) return r;
      }
      else {
```

```
        auto it=L.erase(L.find(j-n));
        if (it!=L.begin() && it!=L.end()
          && (r=checkpair(*prev(it),*it)).X!=-1) return r;
      }
    }
    return {-1,0};
  }
}

// Split convex hull into lower and uppper hull. Endpoints included
pair<vector<Pt>,vector<Pt>> splithull(vector<Pt> p) {
  rotate(p.begin(),min_element(p.begin(),p.end()),p.end());
  auto it=max_element(p.begin(),p.end());
  vector<Pt> L(p.begin(),it+1),U(it,p.end());
  U.push_back(p[0]), reverse(U.begin(),U.end());
  return {L,U};
}

//Intersect convex polygons O(n)
//Run convex hull to remove collinear points if required
//Beware of very steep but not vertical lines when polygon coordinates can
//differ by less than EPS without being equal. Undef defs if you want.
vector<Pt> intersectpolygons(const vector<Pt>&P,const vector<Pt>& Q) {
#define u(j) h[j][i[j]]
#define v(j) h[j][i[j]+1]
#define b(j) i[j]+1<h[j].size()
#define loop for (int j=0;j<4;j++)
#define yv(j) b(j)?det(u(j)-X,v(j)-X)/(u(j).x-v(j).x):u(j).x
  auto c=splithull(P),d=splithull(Q); vector<Pt> h[]{c.X,d.X,c.Y,d.Y},L,U;
  int i[4]{}; double l=-inf,r=inf,X=-inf,nX;
  loop { r=min(r,h[j].back().x), l=max(l,u(j).x); }
  while (1) {
    nX=inf;
    loop if (b(j) && v(j).x>X+EPS) nX=min(nX,v(j).x);
    loop for (int k=j+1;k<4;k++) if (b(j) && b(k)) {
      double p=intersectline(u(j),v(j),u(k),v(k)).x;
      if (!std::isnan(p) && p>X+EPS) nX=min(nX,p);
    }
    if ((X=max(nX,l))>r+EPS) break;
    loop while (b(j) && v(j).x<X+(1-2*deq(X,r))*EPS) i[j]++;
    double m=max(yv(0),yv(1)),M=min(yv(2),yv(3));
    if (m<M+EPS) {
      L.emplace_back(X,m);
      if (!deq(m,M)) U.emplace_back(X,M);
    }
  }
  L.insert(L.end(),U.rbegin(),U.rend());
  return L;
}

// -------------------------- 3D Computational Geometry ------------------------------
using namespace rel_ops;

#define x first
#define y second

#define cpt const Pt&
#define cpt2 const Pt2&

const double EPS=1e-8;
const double pi=acos(-1);
bool deq(double a,double b) {return abs(a-b)<EPS;}

struct Pt {
  double x=0,y=0,z=0;
  bool operator==(cpt b) const { return deq(x,b.x) && deq(y,b.y) && deq(z,b.z); }
  bool operator<(cpt b) const {
    return x<b.x || (x==b.x && (y<b.y || (y==b.y && z<b.z)));
  }
  double& operator[](int i) {return i==0?x:i==1?y:z;}
  Pt operator+=(cpt b) {return {x+=b.x,y+=b.y,z+=b.z};}
  Pt operator-=(cpt b) {return {x-=b.x,y-=b.y,z-=b.z};}
  Pt operator*=(double c) {return {x*=c,y*=c,z*=c};}
  Pt operator/=(double c) {return {x/=c,y/=c,z/=c};}
};
Pt operator+(cpt a,cpt b) {return {a.x+b.x,a.y+b.y,a.z+b.z};}
Pt operator-(cpt a) {return {-a.x,-a.y,-a.z};}
Pt operator-(cpt a,cpt b) {return {a.x-b.x,a.y-b.y,a.z-b.z};}
Pt operator*(double c,cpt a) {return {c*a.x,c*a.y,c*a.z};}
Pt operator*(cpt a,double c) {return {c*a.x,c*a.y,c*a.z};}
Pt operator/(cpt a,double c) {return {a.x/c,a.y/c,a.z/c};}
double operator*(cpt a,cpt b) {return a.x*b.x+a.y*b.y+a.z*b.z;}

Pt cross(cpt a,cpt b) {return {a.y*b.z-a.z*b.y,a.z*b.x-a.x*b.z,a.x*b.y-a.y*b.x};}
double det(cpt a,cpt b,cpt c) {return a*cross(b,c);}
double norm(cpt a) {return a*a;}
double abs(cpt a) {return sqrt(norm(a));}

bool areperp(cpt a,cpt b,cpt p,cpt q) { return deq((b-a)*(q-p),0); }
bool arepara(cpt a,cpt b,cpt p,cpt q) { return cross(b-a,q-p)==Pt{0,0,0}; }

typedef pair<double,double> Pt2;

double det(cpt2 a,cpt2 b) {return a.x*b.y-a.y*b.x;}

// Finds a line that is perpendicular to two lines. Divides by 0 if they are
// parallel. The if statments below ensure the resulting line intersects
// the lines taken as segments. The first point returned lies on ab and the
// second on pq. If the given lines intersect then the points returned are the same.
pair<Pt,Pt> perpline(Pt a,Pt b,Pt p,Pt q) {
  Pt ab=b-a,qp=p-q,ap=p-a;
  Pt2 c{ab*ab,ab*qp},d{ab*qp,qp*qp},e{ab*ap,qp*ap};//[c,d] is Gram matrix
  double s=det(e,d)/det(c,d),t=det(c,e)/det(c,d);
  if (-EPS<s && s<1+EPS //Answer intersects ab
    && -EPS<t && t<1+EPS) //Answer intersects pq
    return {a+s*ab,p-t*qp};
  return {Pt{NAN,NAN,NAN},{}};
}

//Distance between line and point (Infinite line and line segment respectively)
double distlinept(cpt a,cpt b,cpt p) { return abs(cross(b-a,p-a))/abs(b-a); }
double distsegpt(Pt a,Pt b,Pt p) {
  b-=a;p-=a;  double sp=b*p/norm(b);  Pt closest;
  if (sp>=0) {
    if (sp>1) closest=b;
    else closest=sp*b;
  }
  return abs(closest-p); // Note that actual closest Pt on line is closest + a
}

//Project p onto the plane through the origin spanned by a and b. Coordinates
//are given with respect to the basis {a,b}. Divides by 0 if a and b are
//parallel.
Pt2 projectplanept(cpt a,cpt b,cpt p) {
  Pt2 c{a*a,a*b},d{a*b,b*b},e{a*p,b*p};
  return {det(e,d)/det(c,d),det(c,e)/det(c,d)};
}

//Divides by 0 if a, b and c are collinear.
double disttrianglept(Pt a,Pt b,Pt c,Pt p) {
  b-=a;c-=a;p-=a;
  double s,t; tie(s,t)=projectplanept(b,c,p);
  if (0<s && 0<t && s+t<1) return abs(s*b+t*c-p); // Projection within tri.
  return min({distsegpt(a,b,p),distsegpt(a,c,p),distsegpt(b,c,p)});
```

```
}

//Distance between two finite lines. Modify perpline to get infinite lines
double distfinitelineline(cpt a,cpt b,cpt p,cpt q) {
  if (!arepara(a,b,p,q)) {
    Pt u,v; tie(u,v)=perpline(a,b,p,q);
    if (!std::isnan(u.x)) return abs(v-u);
  }
  return min({distsegpt(a,b,p),distsegpt(a,b,q),distsegpt(p,q,a),distsegpt(p,q,b)});
}

//Rotate a point around a line by theta radians. Anticlockwise when looking from b to a.
Pt rotatelinept(Pt a,Pt b,double theta,Pt p) {
  b-=a;p-=a;  b/=abs(b);  double C=cos(theta);
  return C*p+(1-C)*(b*p)*b+sin(theta)*cross(b,p)+a;
}

//Use quaternions when composition of 3D rotations is required. Note that both a
//quaternion and its negative represent the same rotation.
typedef pair<double,Pt> Quaternion;
#define cq const Quaternion&

// Gives the rotation equivalent to doing the b rotation then the a rotation.
Quaternion operator*(cq a,cq b) {
  return {a.x*b.x-a.y*b.y,a.x*b.y+a.y*b.x+cross(a.y,b.y)};
}

double norm(cq a) {return norm(a.x)+norm(a.y);}
double abs(cq a) {return sqrt(norm(a));}
Quaternion operator/(cq a,double c) {return {a.x/c,a.y/c};}

// Careful of divide by zero if you invert this
Quaternion quaternionforrotation(Pt a,double theta) {
  return {cos(theta/2),sin(theta/2)/abs(a)*a};
}

Pt rotatept(Quaternion q,cpt p) {
  q=q/abs(q); // Need this only if quaternion not already normalized.
  return p+cross(2*q.y,cross(q.y,p)+q.x*p);
}

// 3D Convex Hull O(n^2)
// faces is an array of triangles covering the convex hull. f is the number of
// faces. Edges and Tris store indices of p. For any Tri of the hull,
// (p[b]-p[a]) X (p[c]-p[a]) points outward.
// Fun fact: Any triangulation of a (non-degenerate) polyhedron with n vertices
// has 3*(n-2) edges and 2*(n-2) faces.
// If f is two after running convexhull then all points lie in the plane
// described by the two faces but they are not necessarily touching the
// triangle. If f is 0 then all points are collinear.
namespace Hull {  // Set maxn to max number of points
  const int maxn=1000; int f,inh[2*maxn],in[maxn],out[maxn],modif[maxn];
  struct Tri {int a,b,c;} faces[2*maxn];
  struct Edge { int u,v,f[2]; int& operator[](int i) {return i?v:u;} } edges[3*maxn];
  void convexhull(const vector<Pt>& p) {
    int n=p.size(),m=0,i,j,k; f=0; fill(modif,modif+n,-1);
    for (i=1;i<n;i++) {
      if (m==0 && p[i]!=p[0]) edges[m++]={0,i,0,0};
      bool use=m==1 && cross(p[edges[0][1]]-p[0],p[i]-p[0])!=Pt{0,0,0};
      for (j=0;j<f;j++) {
        Tri &t=faces[j];
        if (inh[j] && det(p[t.a]-p[i],p[t.b]-p[i],p[t.c]-p[i])<-EPS)
          inh[j]=0,use=1;
      }
      if (!use) continue;
      for (j=0,k=0;j<m;j++) {
        int nk=1; Edge &e=edges[j];
        if (m==1 || (nk=(int)inh[e.f[0]]+inh[e.f[1]])==1)
          for (int c=0;c<2;c++) if (m==1 || !inh[e.f[c]]) {
            for (;inh[k] && k<f;k++);
            Tri &t=faces[k]={e[c],e[1-c],i};
            e.f[c]=k, in[t.b]=out[t.a]=k, modif[t.b]=i, k++;
          }
        if (nk==0) e=edges[--m], j--;
      }
      bool reset=f==2; f=max(f,k);
      for (j=0;j<n;j++) if (modif[j]==i)
        edges[m++]={i,j,out[j],in[j]},inh[in[j]]=1;
      if (reset) i=0;
    }
    for (i=0,j=0;j<f;j++) if (inh[j]) faces[i++]=faces[j];
    f=i;
  }
}

struct Tri {Pt a,b,c;};

//Signed Volume of polyhedron. Positive when (b-a) X (c-a) points outward for each Tri.
double volume(const vector<Tri>& poly) {
  double r=0;
  for (const Tri &t:poly) r+=det(t.a,t.b,t.c);
  return r/6;
}
//Surface area of polyhedron
double surfacearea(const vector<Tri>& poly) {
  double r=0;
  for (const Tri &t:poly) r+=abs(cross(t.b-t.a,t.c-t.a));
  return r/2;
}

// Delauney Triangulation O(n^2)
// Triangulation of a set of points so that no point p is inside the
// circumcircle of any triangle. Maximizes the minimum angle of all angles of
// the triangles in the triangulation. Each Tri in the result holds 3 indices of
// p. The indices are such that det(p[b]-p[a],p[c]-p[a]) is positive. If all
// points are collinear, then the triangulation will be empty.
vector<Hull::Tri> delauneytriangulation(const vector<Pt2>& p) {
  using namespace Hull;
  vector<Pt> q(p.size());
  for (int i=0;i<p.size();i++) q[i]={p[i].x,p[i].y,-norm(p[i].x)-norm(p[i].y)};
  convexhull(q);
  for (int i=0;i<f;i++) {
    Hull::Tri &t=faces[i];
    if (cross(q[t.b]-q[t.a],q[t.c]-q[t.a]).z<EPS) faces[i--]=faces[--f];
  }
  return {faces,faces+f};
}

double greatcircledist(cpt a,cpt b) {
  return abs(a)*acos((a*b)/(abs(a)*abs(b)));
}

// -------------------------- Integer Computational Geometry ---------------------------
#define x real()
#define y imag()
#define cpt const Pt&

typedef complex<ll> cpx;
struct Pt : cpx {
  Pt() = default; using cpx::cpx;
  Pt(cpx a) : cpx(a) {}
  ll& x const { return (ll&)*this; }
  ll& y const { return ((ll*)this)[1]; }
  bool operator<(cpt b) {return x<b.x || (x==b.x && y<b.y);}
```

```
};

ll dot(cpt a,cpt b) { return (conj(a)*b).x; }
ll det(cpt a,cpt b) { return (conj(a)*b).y; }

//Compare points by principal argument (-pi,pi] breaking ties by norm.
//0 is considered less than everything else.
bool argcomp(cpt a, cpt b) {
  if (b==(ll)0) return 0;
  if (a==(ll)0) return 1;
  bool r1=a.y>0 || (a.y==0 && a.x<0), r2=b.y>0 || (b.y==0 && b.x<0); ll d=det(a,b);
  return r1<r2 || (r1==r2 && (d>0 || (d==0 && norm(a)<norm(b))));
}

// Area of union of rectangles. Include SegmentTree code. Complexity: O(N log(N))
template<class T> struct UnionOfRect {
  int m=0,U=0;  T nm=1,l=1;
  void op(UnionOfRect& b,UnionOfRect& c) {
    if (b.m<c.m) m=b.m, nm=b.nm;
    else { m=c.m, nm=c.nm; if (b.m==c.m) nm+=b.nm; }
    l=b.l+c.l;
  }
  void us(int v) { m+=v, U+=v; }
  void NU() {U=0;}
  T nonzerolen() { return l-(m?0:nm); }
};

template<class T> struct Rect { pair<T,T> l,u; }; // lower left and upper right corners

// You will get runtime error if all y values are the same.
template<class T> T areaofunionofrect(const vector<Rect<T>>& rect) {
  int n=rect.size(),m;  T r=0;
  vector<T> ys(2*n);  vector<pair<pair<T,int>,pair<T,T>>> sides(2*n);
  for (int i=0;i<n;i++) {
    ys[2*i]=rect[i].l.Y, ys[2*i+1]=rect[i].u.Y;
    sides[2*i]={{rect[i].l.X,1},{rect[i].l.Y,rect[i].u.Y}};
    sides[2*i+1]={{rect[i].u.X,-1},{rect[i].l.Y,rect[i].u.Y}};
  }
  sort(ys.begin(),ys.end());  sort(sides.begin(),sides.end());
  ys.resize(unique(ys.begin(),ys.end())-ys.begin());
  vector<UnionOfRect<T>> stinit(m=ys.size()-1);
  for (int i=0;i<m;i++) stinit[i].l=stinit[i].nm=ys[i+1]-ys[i];
  SegmentTree<UnionOfRect<T>,int> st(stinit);  // Include SegmentTree code
  T x = sides[0].X.X;
  for (auto &i:sides) {
    r+=(i.X.X-x)*st.query(0,m).nonzerolen();  x=i.X.X;
    int a=lower_bound(ys.begin(),ys.end(),i.Y.X)-ys.begin();
    int b=lower_bound(ys.begin(),ys.end(),i.Y.Y)-ys.begin();
    if (a!=b) st.update(a,b,i.X.Y);
  }
  return r;
}
```

2 Number Theory

```
typedef __int128 big;  // Use this if necessary. Mainly needed for huge prime testing.

// Binary exponentiation - compute a^b mod m. Complexity O(log(n))
ll expmod(big a, big b, big m) {
  big res=1%m;  a %= m;
  for(; b; b /= 2) { if (b&1) res=res*a%m;  a=a*a%m; }
  return res;
}

// Extended Euclidean Algorithm. Finds x,y such that
// ax + by = gcd(a,b). Returns gcd(a,b). Compexity: O(log(min(a,b)))
ll gcd(ll a, ll b, ll& x, ll& y) {
  if (b == 0) { y = 0; x = (a < 0) ? -1 : 1; return (a < 0) ? -a : a; }
  else { ll g = gcd(b, a%b, y, x); y -= a/b*x; return g; }
}

// Multiplicative inverse of a mod m, for a,m coprime. Complexity: O(log(a))
ll inv(ll a, ll m) { ll x, y; gcd(m,a,x,y); return ((y % m) + m) % m; }

// Chinese Remainder Algorithm. Solves x = a[i] mod m[i] for x mod lcm(m)
// for m[i] pairwise coprime. In general x = x0 + t*lcm(m) for all t.
ll cra(vi& a, vi& m) {
  int n = (int)a.size();  big u = a[0], v = m[0]; ll p, q, r, t;
  for (int i = 1; i < n; ++i) {
    r = gcd(v, m[i], p, q); t = v;
    if ((a[i] - u) % r != 0) { return -1; }  // no solution!
    v = v/r * m[i];  u = ((a[i] - u)/r * p * t + u) % v;
  }
  if (u < 0) u += v;
  return u;
}

// Euler Phi Function - Count the integers coprime to n. Facts:
// (1) If p is prime, phi(p) = p - 1. (2) If p is prime, then
// phi(p^k) = p^k - p^(k-1). (3) If a and b are relatively
// prime, then phi(ab) = phi(a)phi(b). (4) If a and b are relatively
// prime, then a^phi(m) = 1 mod m.  Complexity: O(sqrt(n))
ll phi(ll n) {
  ll res = n;
  for (ll i = 2; i*i <= n; ++i) if (n % i == 0) {
    while (n % i == 0) n /= i;
    res -= res / i;
  }
  if (n > 1) res -= res / n;
  return res;
}

// Sieve for primality testing up to 10^8. Complexity: O(n log(log(n)))
vector<bool> isprime;
void sieve(int n) {
  isprime.assign(n + 1, 1); isprime[0] = isprime[1] = 0;
  for (ll i = 2; i * i <= n; ++i) if (isprime[i])
    for (ll j = i*i; j <= n; j += i) isprime[j] = 0;
}

// Sieve for factoring up to 10^7. Complexity: O(n)
// fac contains a prime factor, pr is a list of primes.
vi fac, pr;
void fast_sieve(int n) {
  fac.assign(n + 1, 0);
  for (ll i = 2; i <= n; ++i) {
    if (fac[i] == 0) fac[i] = i, pr.push_back(i);
    for (int p : pr) if (p > fac[i] || i * p > n) break; else fac[i * p] = p;
  }
}

// Deterministic Miller-Rabin primality test. Complexity: O(log(n))
vi val = {2, 7, 61};                                      // n <= 2^32
vi val = {2, 13, 23, 1662803};                            // n <= 10^12
vi val = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};  // n <= 2^64 (Needs __int128)

bool is_prime(ll n) {
  if (n < 2) return false;
  ll s = __builtin_ctzll(n-1), d = (n-1) >> s;
  for (int v : val) {
    if (v >= n) break;
    ll x = expmod(v, d, n);
    if (x == 1 || x == n - 1) continue;
    for (ll r=1; r<s; r++) if ((x = ((big(x)*x) % n)) == n - 1) goto nextPr;
```

```
    return false;
    nextPr:;
  }
  return true;
}

// Prime factors in O(log(n)) using precomputed fast_sieve(N >= n).
vi fast_factors(int n) {
  vi res;
  while (n > 1) {
    int f = fac[n];
    while (n % f == 0) n /= f;  // remove while to include duplicates
    res.push_back(f);
  }
  return res;
}

// Prime factors in O(sqrt(n)) with no precomputation.
vector<ll> slow_factors(ll n) {
  vector<ll> res;
  for (ll i = 2; i*i <= n; ++i) if (n % i == 0) {  // change if to while for duplicates
    res.push_back(i);
    while (n % i == 0) n /= i;  // remove while to include duplicates
  }
  if (n > 1) res.push_back(n);
  return res;
}

// Finds one (not necessarily prime) factor of n.
// Works best on semi-primes (n = pq for p, q distinct primes)
// Does not work well on perfect powers -- check those separately.
// Expected complexity: O(n^(1/4)) (only a heuristic)
ll F(ll x,ll n,ll c){ x=big(x)*x%n-c; return (x < 0 ? x + n : x); }
ll pollardRho(ll n) {
  ll i,c,b,x,y,z,g;
  for(g=0, c=3; g%n == 0; c++)
    for (g=b=x=y=z=1; g == 1; b *= 2, g = __gcd(z,n), z = 1, y = x)
      for (i=0; i<b; i++) { x = F(x,n,c); z = (big)z * abs(x-y) % n; }
  return g;
}

// Factorise a huge number (n <= 10^18). Expected Complexity: O(n^(1/3))
vector<ll> factor_huge(ll n) {
  vector<ll> res;
  for (ll i = 2; i*i*i <= n; ++i) if (n % i == 0) { // change if to while for duplicates
    res.push_back(i);
    while (n % i == 0) n /= i;  // remove while to include duplicates
  } // Below, push_back(sqrt(n)) twice for duplicates
  if (ll(sqrt(n))*ll(sqrt(n)) == n) return res.push_back(sqrt(n)), res;
  if (is_prime(n)) return res.push_back(n), res;
  ll q = pollardRho(n);  res.push_back(q);  res.push_back(n/q);
  return res;
}

// Find a primitive root modulo n. g is a primitive root modulo n if
// all coprimes to n are congruent to a power of g (mod n), ie. for any a
// such that gcd(a,n) = 1, there is k such that g^k = a (mod n) where k
// is the index or discrete logarithm of a to g (mod n). A primitive root
// exists only if n = 1,2,4 or n is a power of an odd prime or twice
// the power of an odd prime. The number of primitive roots is phi(phi(n)).
// Complexity: O(g log(phi(n)) log(n)). Returns -1 if no root exists.
ll primitive_root(ll n) {
  ll tot = phi(n);                    // if n is prime, can use tot = n - 1
  auto fact = slow_factors(tot);      // use fast_factors if you need
  for (ll res=2; res<n; ++res) {
    bool ok = __gcd(res, n) == 1;
    for (int i = 0; i < (int)fact.size() && ok; ++i)
      ok &= expmod(res, tot / fact[i], n) != 1;
    if (ok) return res;  // Can add to a vector and find all of them if needed
  }
  return -1;
}

// Discrete root solver - Given a prime n and integers a, k, we want
// to find all x satisfying x^k = a (mod n). Complexity: O(sqrt(n) log(n))
vector<ll> discrete_root(ll n, ll k, ll a) {
  ll g = primitive_root(n);   // n must be prime
  ll sq = (ll)sqrt(n) + 1;
  vector<pair<ll,ll>> dec(sq);
  for (ll i = 1; i <= sq; ++i)
    dec[i-1] = {expmod(g, i * sq * k % (n - 1), n), i};
  sort(dec.begin(), dec.end());
  ll ans = -1;
  for (ll i=0; i<sq; ++i) {
    ll my = expmod(g, i * k % (n - 1), n) * a % n;
    auto it = lower_bound(dec.begin(), dec.end(), make_pair(my, 0LL));
    if (it != dec.end() && it->first == my) {
      ans = it->second * sq - i;  break;
    }
  }
  // Optional: if you only need one solution, return ans
  vector<ll> res;  if (ans == -1) return res;
  ll delta = (n-1) / __gcd (k, n-1);
  for (ll cur = ans % delta; cur < n - 1; cur += delta)
    res.push_back(expmod(g, cur, n));
  return res;
}

// Discrete Logarithm. Complexity: O(sqrt(M)log(M))
// Solves a^x == b (mod mod) for integer x. The smallest non-negative x is chosen.
// Returns -1 if there is no such x. x is assumed to be strictly less than M;
// To optimise set M to
//   phi(mod/gcd(mod,lcm(a^lg(mod),b^lg(mod)))) + lg(mod)
//   lg(mod) is the maximum multiplicity of a prime factor. This is length of
//   path before entering cycle.
// Can also derive extra conditions to determine when solution exists before
// running algorithm.
ll discrete_log(ll a,ll b,ll mod) {
  static pair<ll,ll> seen[5000000]; // Must be at least ceil(sqrt(M))
  ll M=mod, s=0, as=1, bas; // step size, a^s, ba^s
  for (;s*s<M;s++) as=mult(as,a,mod), bas=mult(b,as,mod), seen[s]={bas,s+1};
  sort(seen,seen+s);
  for (ll i=1,ap=1,ct=0,p;i<=s && ct<=s;i++) {
    ap=mult(ap,as,mod); //(ll)ap*as%mod;
    int j=lower_bound(seen,seen+s,pair<ll,ll>{ap+1,0})-seen;
    for (;--j>=0 && seen[j].X==ap && ct<=s;ct++)
      if (expmod(a,p=(ll)i*s-seen[j].Y,mod)==b) return p;
  }
  return -1;
}

// Integer convolution mod m using number theoretic transform.
// m = modulo, r = a primitive root, ord = order of the root
// (Must be a power of two). The length of the given input
// vectors must not exceed n = ord.   Complexity: O(n log(n))
//
// Usable coefficients::
//    m                       | r             | ord         | __int128 required
//------------------------|-------------|-----------|--------------------
//   7340033                | 5             | 1 << 20     | No
//   469762049              | 13            | 1 << 25     | No
//   998244353              | 31            | 1 << 23     | No
//   1107296257             | 8             | 1 << 24     | No
//   10000093151233         | 366508        | 1 << 26     | Yes
```

```
//  1000000523862017      | 2127080     | 1 << 26   | Yes
//  1000000000949747713   | 465958852   | 1 << 26   | Yes
//
// In general, you may use mod = c * 2^k + 1 which has a primitive
// root of order 2^k, then use number theory to find a generator.
template<typename T> struct convolution {
  const T m, r, ord;
  T mult(T x, T y) { return big(x) * y % m; }
  void ntt(vector<T> & a, int invert = 0) {
    int n = (int)a.size(); T ninv = inv(n, m), rinv = inv(r, m);  // Modular inverses
    for (int i=1, j=0; i<n; ++i) {
      int bit = n >> 1; for (; j>=bit; bit>>=1) j -= bit;
      j += bit; if (i < j) swap (a[i], a[j]);
    }
    for (int len=2; len<=n; len<<=1) {
      T wlen = invert ? rinv : r;
      for (int i=len; i<ord; i<<=1) wlen = mult(wlen, wlen);
      for (int i=0; i<n; i+=len) {
        T w = 1;
        for (int j=0; j<len/2; ++j) {
          T u = a[i+j],  v = mult(a[i+j+len/2], w);
          a[i+j] = u + v < m ? u + v : u + v - m;
          a[i+j+len/2] = u - v >= 0 ? u - v : u - v + m;
          w = mult(w, wlen);
        }
      }
    }
    if (invert) for (int i=0; i<n; ++i) a[i] = mult(a[i], ninv);
  }
  // Compute the convolution a * b -- Complexity: O(n log(n))
  vector<T> multiply(vector<T>& a, vector<T>& b) {
    vector<T> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;  while (n < 2 * (int)max(a.size(), b.size())) n*=2;
    fa.resize(n), fb.resize(n); ntt(fa), ntt(fb);
    for(int i=0;i<n;i++) fa[i] = mult(fa[i], fb[i]);
    ntt(fa, 1); fa.resize(n);
    return fa;
  }
};
```

3 Combinatorics

```
// Generates set partitions for a set of size n in gray code order. A set partition
// is represented as a vector of size n where P[i] = the index of the set that
// element i belongs to. Safe to use for n <= 13, where B_n = 27 million.
struct set_partition_generator {
  int n;  vi a, b, d;  bool done = false;
  set_partition_generator(int n) : n(n), a(n), b(n, 1), d(n, 1) { }
  void fix(int j, int m) { fill(b.begin() + j + 1, b.end(), m); }
  bool has_next() { return !done; }
  vi next_partition() {
    vi ans = a;  int j = n - 1;
    while (a[j] == d[j]) d[j--] ^= 1;
    if (j == 0) done = true;
    else if (d[j] != 0) {
      if (a[j] == 0) { a[j] = b[j]; fix(j, a[j] + 1); }
      else if (a[j] == b[j]) { a[j] = b[j] - 1; fix(j, b[j]); }
      else a[j]--;
    }
    else {
      if (a[j] == b[j] - 1) { a[j] = b[j]; fix(j, a[j] + 1); }
      else if (a[j] == b[j]) { a[j] = 0; fix(j, b[j]); }
      else a[j]++;
    }
    return ans;
  }
};
```

```
// Generates the lexicographically next integer partition of sum(p). Start with
// p = [1,1,1,1,...] to generate all integer partitions in gray code order.
bool next_partition(vi& p) {
  int n = p.size(), i = n - 2;
  if (n <= 1) return false;
  int s = p.back() - 1; p.pop_back();
  while (i > 0 && p[i] == p[i - 1]) { s += p[i--]; p.pop_back(); }
  p[i]++;
  while (s-- > 0) p.push_back(1);
  return true;
}

// Generate the lexicographically previous subset of mask. To generate all subsets,
// the initial subset should be sub = mask.
template<typename T> bool next_subset(T& sub, T mask) {
  if (sub == 0) return false; else return sub = (sub - 1) & mask, true;
}

// Generate the lexicographically next combination of n choose k. To generate all
// combinations, the initial combination is comb = (1 << k) - 1.
template<typename T> bool next_combination(T& comb, int n, int k) {
  T x = comb & -comb, y = comb + x;  comb = (((comb ^ y) >> 2) / x) | y;
  return comb < (T(1) << n);
}
```

4 Dynamic Programming

```
// Returns the length of the longest (strictly) increasing subsequence of v
// Reverse the input for longest decreasing subsequence. Complexity: O(n log(n))
template<typename T> int lis_len(vector<T>& v) {
  vector<T> s(v.size()); int k=0;
  for (int i=0; i<(int)v.size(); i++) {  // Change to upper_bound for non-decreasing
    auto it = lower_bound(s.begin(), s.begin()+k, v[i]); *it = v[i];
    if (it == s.begin()+k) k++;
  }
  return k;
}

// Returns the longest (strictly) increasing subsequence of v
// Reverse the input for longest decreasing subsequence. Complexity: O(n log(n))
template<typename T> vector<T> lis(vector<T>& v) {
  int n = v.size(), len = 0;  vi tail(n), prev(n); T val[n];
  for (int i=0; i < n; i++) { // Change to upper_bound for non-decreasing
    int pos = lower_bound(val, val + len, v[i]) - val;
    len = max(len, pos + 1), prev[i] = (pos > 0 ? tail[pos - 1] : -1);
    tail[pos] = i;  val[pos] = v[i];
  }
  vector<T> res(len);
  for (int i = tail[len - 1]; i >= 0; i = prev[i]) res[--len] = v[i];
  return res;
}

// Finds the longest palindromic substrings of a. Returns the longest length and a
// vector of positions at which longest palindromes occur. Complexity: O(n)
pair<int, vi> longest_palindrome(const vi& a) {
  int n = 2 * a.size() + 1, b = 0, m = 0, res = 0; vi pos;
  vector<pii> R(n);  vi p(n, -1);  // -1 should be something not in the input
  for (int i = 1; i < n; i += 2) p[i] = a[i/2];
  for (int i = 1; i < n; i++) {
    int w = i < b ? min(R[2 * m - i].Y, b-i) : 0;
    for (int l = i-w-1, u = i+w+1; l >= 0 && u < n && p[l--] == p[u++]; w++);
    R[i] = {(i - w)/2, w};
    if (i + w > b) b = i + w, m = i;
    if (w > res) res = R[i].Y;
  }
  for (auto& x : R) if (x.Y == res) pos.push_back(x.X);
  return {res, pos};
```

```
}

// A queue that supports amortized O(1) insertion and query
// for the minimum element. Change <= to >= for max element.
template<typename T> struct MonotonicQueue {
  deque<pair<int,T>> q, mins; int cnt = 0;
  void push(T x) {
    while (!mins.empty() && x <= mins.back().Y) mins.pop_back();
    mins.emplace_back(cnt,x), q.emplace_back(cnt++,x);
  }
  void pop() {
    if (mins.front().X == q.front().X) mins.pop_front();
    q.pop_front();
  }
  T front() { return q.front().Y; }
  T min() { return mins.front().Y; }
  bool empty() { return q.empty(); }
};

// Monotonic convex hull trick. Find the maximum value on the upper envelope
// of a dynamic set of lines such that the queries and slopes are monotonically
// non-decreasing. To maintain a lower hull, just negate the values.
// Complexity: amortized O(1) per operation. T = type of slope/intercept.
template<typename T> struct MonotoneHull {
  struct Line { T a, b; double x; };  deque<Line> lines;
  void add_line(T a, T b) { // Add a line of the form y = ax + b
    double x = -1e200;
    while (!lines.empty()) {
      if (a == lines.back().a) x *= b < lines.back().b ? -1 : 1;
      else x = 1.0 * (lines.back().b - b) / (a - lines.back().a);
      if (x < lines.back().x) lines.pop_back();
      else break;
    }
    lines.push_back({a,b,x});
  }
  T query(T x) {     // Find min A[i]x + B[i]. Can alter this to binary search if the
    while (lines.size() > 1 && lines[1].x <= x) lines.pop_front();    // query points
    return lines[0].a * x + lines[0].b;  // are not monotone but the slopes still are
  }
};

// Dynamic upper convex hull trick. Maintains the upper hull of a dynamic
// set of lines. To maintain a lower hull, just negate the values.
// Complexity: O(log(N)) per operation. T = type of slope / intercept.
template<typename T> struct DynamicHull {
  struct Line {
    typedef typename multiset<Line>::iterator It;
    T a, b; mutable It me, endit, none;
    Line(T a, T b, It endit) : a(a), b(b), endit(endit) { }
    bool operator<(const Line& rhs) const {
      if (rhs.endit != none) return a < rhs.a;
      if (next(me) == endit) return 0;
      return (b - next(me)->b) < (next(me)->a - a) * rhs.a;
    }
  };
  multiset<Line> lines;
  void add_line(T a, T b) {
    auto bad = [&](auto y) {
      auto z = next(y);
      if (y == lines.begin()) {
        if (z == lines.end()) return false;
        return y->a == z->a && z->b >= y->b;
      }
      auto x = prev(y);
      if (z == lines.end()) return y->a == x->a && x->b >= y->b;
      return (x->b-y->b)*(z->a-y->a) >= (y->b-z->b)*(y->a-x->a);
    }; // WARNING: Change above comparison to doubles if you fear overflow
    auto it = lines.emplace(a, b, lines.end()); it->me = it;
    if (bad(it)) { lines.erase(it); return; }
    while (next(it) != lines.end() && bad(next(it))) lines.erase(next(it));
    while (it != lines.begin() && bad(prev(it))) lines.erase(prev(it));
  }
  T query(T x) {
    auto it = lines.lower_bound(Line{x,0,{}});
    return it->a * x + it->b;
  }
};

// Divide and conquer optimisation for dynamic programs of the form
//   DP[i][j] = min(DP[i-1][k] + C[k][j]) for k < j.  i <= K, j <= N
// The minimiser must be monotonic (satisfy opt[i][j] <= opt[i][j+1]).
// To use: -- define cost function cost(k,j) = C[k][j]
//         -- fill base cases DP[0][0], DP[0][j], DP[i][0]
//         -- fill the rest DP[i][j] = INF
//         -- compute each row: for(int i=1; i<=K; i++) compute(i,1,N,0,N)
// Complexity: O(KN log(N))
void compute(int i, int l, int r, int optL, int optR) {
  if (r < l) return;  int mid = (l + r) / 2, opt = optL;
  for (int k=optL; k<=min(mid-1,optR); k++) {
    ll new_cost = DP[i-1][k] + cost(k,mid);
    if (new_cost < DP[i][mid]) DP[i][mid] = new_cost, opt = k;
  }
  compute(i, l, mid-1, optL, opt), compute(i, mid+1, r, opt, optR);
}

// Knuth optimisation for problems of the form:
//   DP[i][j] = min(DP[i][k-1] + DP[k+1][j]) + C[i][j] for i <= k <= j
// The minimiser must be monotonic (satisfy opt[i][j-1] <= opt[i][j] <= opt[i+1][j])
// Alternatively, also applicable if instead the following conditions are met:
//   1. C[a][c] + C[b][d] <= C[a][d] + C[b][c] (quadrangle inequality)
//   2. C[b][c] <= C[a][d]                     (monotonicity)
// for all a <= b <= c <= d
// To use: -- define cost function cost(i,j) = C[i][j]
//         -- Compute base cases DP[i][i] for all 0 < i < n
//         -- Fill the rest of DP[i][j] = INF
//         -- Compute knuth(DP);
// Returns the optimal split points k = opt[i][j].  Complexity: O(N^2)
template<typename T> vvi knuth(vector<vector<T>>& DP) {
  int n = (int)DP.size();  vvi opt(n, vi(n));
  for (int i=0; i<n; i++) opt[i][i] = i;
  for (int len=1; len<n; len++) for (int i=0; i+len<n; i++) {
    int j = i + len;
    for (int k=opt[i][j-1]; k <= opt[i+1][j]; k++) {
      T new_cost = (k-1>=i ? DP[i][k-1] : 0) + (k+1<=j ? DP[k+1][j] : 0) + cost(i,j);
      if (new_cost < DP[i][j]) DP[i][j] = new_cost, opt[i][j] = k;
    }
  }
  return opt;
}
```

5 Graph Algorithms

```
// Shortest paths and negative cycle finding in graphs with any weights.
// dist[u] = INF if u is not reachable. dist[u] = -INF if u is reachable via a
// negative cycle. T is the type of the edge weights / costs. Complexity: O(VE)
template <typename T> struct BellmanFord {
  typedef pair<T, int> pti;  vector<vector<pti>> adj;
  int n, last = -1; const T INF = numeric_limits<T>::max() / 2;
  BellmanFord(int n) : adj(n), n(n) {}
  void add_edge(int u, int v, T weight) { adj[u].emplace_back(weight, v); }
  pair<vector<T>, vi> shortest_paths(int src) {
    vector<T> dist(n, INF);  dist[src] = 0;  vi pred(n, -1);  last = 0;
    for (int k = 0; k < n && last != -1; k++) {  last = -1;
      for (int u = 0; u < n; u++) if (dist[u] < INF) for (auto &e : adj[u]) {
```

```
        int v = e.Y; T len = dist[u] + e.X;
        if (len < dist[v]) dist[v] = len, pred[v] = u, last = v;
      }
    }
    if (last == -1) return {dist, pred};  // there were no negative cycles
    for (int k = 0, upd = 1; k < n && upd; k++) {  upd = 0;
      for (int u = 0; u < n; u++) if (dist[u] < INF) for (auto &e : adj[u]) {
        int v = e.Y; T len = dist[u] + e.X;
        if (len < dist[v]) dist[v] = -INF, upd = 1;
      }
    }
    return {dist, pred};  // there was a negative cycle
  }  // Returns true if the most recent invocation encountered a negative cycle
  bool had_negative_cycle() { return last != -1; }
  // OPTIONAL: Find a negative cycle in the graph
  vi find_negative_cycle() {
    n++; adj.resize(n);  // add a new temp vertex
    for (int v = 0; v < n - 1; v++) add_edge(n-1, v, 0);
    vi C, pred = shortest_paths(n-1).Y;
    n--; adj.resize(n);  // delete the temp vertex
    if (!had_negative_cycle()) return C; // no negative cycle found
    for (int i = 0; i < n; i++) last = pred[last];
    for (int u = last; u != last || C.empty(); u = pred[u]) C.push_back(u);
    reverse(C.begin(), C.end());
    return C;
  }
};

// Reconstruct the path corresponding to pred from Dijkstra and Bellman-Ford
vi get_path(int v, vi& pred) {
  vi p = {v};
  while (pred[v] != -1) p.push_back(v = pred[v]);
  reverse(p.begin(), p.end());
  return p;
}

// Find articulation points, bridges, biconnected components and bridge-connected
// components. cut_point[v] = true if v is an articulation point. e.bridge = true
// if e is a bridge. n_vcomps is the number of biconnected components, n_bcomps
// is the number of bridge-connected components. bccs contains biconnected
// components specified by edge indices. bcomp[v] is the index of the
// bridge-connected component containing vertex v.  Complexity: O(V + E)
struct Biconnectivity {
  struct edge {
    int u, v, vcomp;  bool used, bridge;
    edge(int a, int b) : u(a), v(b) { }
    int other(int w) { return w == u ? v : u; }
  };
  int n, m, n_bcomps, n_vcomps, dfs_root, dfs_count, root_children;
  vi dfs_num, dfs_low, cut_point, vcur, bcur, bcomp; vvi bccs, adj; vector<edge> edges;
  void make_vcomp(int i) {  // omit if biconnected components are not required
    bccs.emplace_back(vcur.rbegin(), find(vcur.rbegin(), vcur.rend(), i) + 1);
    vcur.resize(vcur.size() - bccs.back().size());
    for (auto j : bccs.back()) edges[j].vcomp = n_vcomps; n_vcomps++;
  }
  void make_bcomp(int v) {  // omit if bridge-connected components are not required
    int u = -1; n_bcomps++;
    while (u != v) { u = bcur.back(); bcur.pop_back(); bcomp[u] = n_bcomps - 1; }
  }
  void dfs(int u) {
    dfs_low[u] = dfs_num[u] = dfs_count++;
    for (auto i : adj[u]) if (!edges[i].used) {
      auto& e = edges[i]; int v = e.other(u);  e.used = true;
      if (dfs_num[v] == -1) {
        if (u == dfs_root) root_children++;
        vcur.push_back(i), bcur.push_back(v), dfs(v);
        if (dfs_low[v] > dfs_num[u]) { e.bridge = true; make_bcomp(v); }
        if (dfs_low[v] >= dfs_num[u]) { cut_point[u] = true; make_vcomp(i); }
        dfs_low[u] = min(dfs_low[u], dfs_low[v]);
      } else {
        dfs_low[u] = min(dfs_low[u], dfs_num[v]);
        if (dfs_num[v] < dfs_num[u]) vcur.push_back(i);
      }
    }
  }
  Biconnectivity(int n) : n(n), m(0), adj(n) { }
  edge& get_edge(int i) { return edges[i]; }
  int add_edge(int u, int v) {
    adj[u].push_back(m), adj[v].push_back(m), edges.emplace_back(u, v);
    return m++;
  }
  void find_components() {
    dfs_num.assign(n, -1); dfs_low.assign(n, 0); dfs_count = 0;
    vcur.clear(); bcur.clear(); bccs.clear(); cut_point.assign(n, 0);
    bcomp.assign(n, -1); n_vcomps = 0, n_bcomps = 0;
    for (auto& e : edges) e.used = false, e.bridge = false;
    for (int v = 0; v < n; v++) if (dfs_num[v] == -1) {
      bcur = {v}; dfs_root = v;  root_children = 0;  dfs(v);
      cut_point[v] = (root_children > 1); make_bcomp(v);
    }
  }
};

// Find an Eulerian path or tour in a given graph if one exists. For a connected,
// undirected graph, an Euler tour exists if every vertex has an even degree.
// An Euler path exists if all but two vertices have an even degree, these will
// be the endpoints. A connected directed graph has an Euler tour if all
// vertices have indegree == outdegree, or an Euler path if one vertex has
// outdegree - indegree = 1 and one vertex has indegree - outdegree = 1, these
// will be the start and endpoints respectively. You must check existence yourself.
// Call find(start) where start is the first vertex of the path / tour.
// NOTE: Both the start and end-point are included in a tour. Complexity: O(V + E)
struct Eulerian {
  struct edge { int u, v; bool used; int opp(int x) { return x == u ? v : u; } };
  int n, m; vector<edge> edges;  vvi adj;  vi cnt, tour;
  void dfs(int u) {
    while (cnt[u] < (int)adj[u].size()) {
      auto& e = edges[adj[u][cnt[u]++]];
      if (!e.used) e.used = 1, dfs(e.opp(u)), tour.push_back(u);
    }
    if (tour.empty()) tour.push_back(u);
  }
  Eulerian(int n) : n(n), m(0), adj(n) { }
  void add_edge(int u, int v, bool dir) {  // dir = true if the edge is directed
    edges.push_back({u,v,0}), adj[u].push_back(m++);        // or false otherwise
    if (!dir) adj[v].push_back(m-1);
  }
  vi find(int start=0) {
    tour.clear(); cnt.assign(n, 0);  for (auto& e : edges) e.used = 0;
    dfs(start), reverse(tour.begin(), tour.end());
    return tour;
  }
};

// Lowest common ancestor and tree distances using binary lifting.
// Complexity: O(V log(V)) to build, O(log(V)) to query.
template<typename T = int> struct LCA {
  const int LOGN = 20;  // works for n <= 10^6. Change appropriately.
  int n;  vi par, lvl;  vvi anc;  vector<T> len;  vector<vector<pair<int,T>>> adj;
  void dfs(int u, int p, int l, T d) {
    par[u] = p, lvl[u] = l, len[u] = d;
    for (auto v : adj[u]) if (v.X != p) dfs(v.X, u, l+1, d+v.Y);
  }
  // Create a tree with n nodes. Add edges then call build(root).
```

```
  LCA(int n) : n(n), par(n), lvl(n), len(n), adj(n) { }
  void add_edge(int u, int v, T w = 1) {
    adj[u].emplace_back(v, w), adj[v].emplace_back(u, w);
  }
  void build(int root = 0) {  // Call this before making queries
    dfs(root,-1,0,0), anc.assign(n, vi(LOGN, -1));
    for (int i = 0; i < n; i++) anc[i][0] = par[i];
    for (int k = 1; k < LOGN; k++) for (int i = 0; i < n; i++)
      if (anc[i][k-1] != -1) anc[i][k] = anc[anc[i][k-1]][k-1];
  }
  int query(int u, int v) {  // LCA with respect to original root
    if (lvl[u] > lvl[v]) swap(u, v);
    for (int k = LOGN - 1; k >= 0; k--)
      if (lvl[v] - (1 << k) >= lvl[u]) v = anc[v][k];
    if (u == v) return u;
    for (int k = LOGN - 1; k >= 0; k--) {
      if (anc[u][k] == anc[v][k]) continue;
      u = anc[u][k]; v = anc[v][k];
    }
    return par[u];
  }
  int query(int u, int v, int root) {  // OPTIONAL: LCA with respect to any root
    int a = query(u, v), b = query(u, root), c = query(v, root);
    if (a == c && c != b) return b;
    else if (a == b && c != b) return c;
    else return a;
  }
  T dist(int u, int v) { return len[u] + len[v] - 2 * len[query(u,v)]; }
};

// Dinic's algorithm for maximum flow. add_edge returns the id of an edge which can be
// used to inspect the final flow value using get_edge(i).flow. Complexity: O(V^2 E)
template<typename T> struct Dinics {
  struct edge { int to; T flow, cap; };   T INF = numeric_limits<T>::max();
  int n, m;  vi dist, work;  queue<int> q;  vector<edge> edges;  vvi adj;
  bool bfs(int s, int t) {
    dist.assign(n, -1);  dist[s] = 0;  q.push(s);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      for (auto& i : adj[u]) {
        edge& e = edges[i];  int v = e.to;
        if (dist[v] < 0 && e.flow < e.cap) dist[v] = dist[u] + 1, q.push(v);
      }
    }
    return dist[t] >= 0;
  }
  T dfs(int u, int t, T f) {
    if (u == t) return f;
    for (int& i = work[u]; i < (int)adj[u].size(); i++) {
      auto& e = edges[adj[u][i]], &rev = edges[adj[u][i]^1];
      if (e.flow < e.cap && dist[e.to] == dist[u] + 1) {
        T df = dfs(e.to, t, min(f, e.cap - e.flow));
        if (df > 0) { e.flow += df; rev.flow -= df; return df; }
      }
    }
    return 0;
  }
  // Create a flow network with n nodes -- add edges with add_edge(u,v,cap)
  Dinics(int n) : n(n), m(0), adj(n) { }
  int add_edge(int from, int to, T cap) {        // add an edge (from -> to) with
    adj[from].push_back(m++), adj[to].push_back(m++);   // capacity of cap units.
    edges.push_back({to, 0, cap}), edges.push_back({from, 0, 0});
    return m - 2;    // Change {from,0,0} to {from,0,cap} for bidirectional edges
  }
  edge& get_edge(int i) { return edges[i]; }  // get a reference to the i'th edge
  T max_flow(int s, int t) {                  // find the maximum flow from s to t
    T res = 0;  for (auto& e : edges) e.flow = 0;
    while (work.assign(n, 0), bfs(s, t))
      while (T delta = dfs(s, t, INF)) res += delta;
    return res;
  }
};

// Push relabel for maximum flow. add_edge returns the id of an edge which can be
// used to inspect the final flow value using get_edge(i).flow. Complexity: O(V^3)
template<typename T> struct PushRelabel {
  struct edge { int to; T flow, cap; };    T INF = numeric_limits<T>::max();
  int n, m, s, t, max_bkt;    vi h, inq, num_h, cur_e;    vvi g, bkt;
  vector<edge> edges;    vector<T> ex;
  void gap_heuristic(int k) {
    for (int u = 0; u < n; u++) if (u != s && h[u] > k && h[u] <= n) {
      num_h[h[u]]--;  cur_e[u] = 0;
      if (inq[u]) bkt[h[u]].clear(); bkt[n+1].push_back(u);
      h[u] = n+1; num_h[h[u]]++;
      if (h[u] > max_bkt) max_bkt = h[u];
    }
  }
  void push(int u, int v, int id) {
    T tmp = min(ex[u], edges[id].cap - edges[id].flow);
    ex[u] -= tmp, ex[v] += tmp, edges[id].flow += tmp, edges[id^1].flow -= tmp;
  }
  int relabel(int u) {
    int minH = 2 * n;
    for (int id : g[u]) if (edges[id].flow < edges[id].cap)
      minH = min(minH, h[edges[id].to]);
    return 1 + minH;
  }
  void discharge(int u) {
    inq[u] = 0;
    while (ex[u] > 0) {
      for (; cur_e[u] < (int) g[u].size(); cur_e[u]++) {
        int id = g[u][cur_e[u]], v = edges[id].to;
        if (edges[id].cap == edges[id].flow) continue;
        if (h[u] == h[v]+1) {
          push(u, v, id);
          if (inq[v] == 0 && v != s && v != t) {
            bkt[h[v]].push_back(v);  inq[v] = 1;
            if (h[v] > max_bkt) max_bkt = h[v];
          }
        }
        if (ex[u] == 0) break; // remain at cur_e
      }
      if (ex[u] > 0) {
        int prev_h = h[u]; num_h[h[u]]--; h[u] = relabel(u);
        num_h[h[u]]++; cur_e[u] = 0;
        if (num_h[prev_h] == 0 && prev_h <= n - 1) gap_heuristic(prev_h);
      }
    }
  }
  PushRelabel(int n) : n(n), m(0), max_bkt(0), inq(n), num_h(2*n),
      cur_e(n), g(n), bkt(2*n), ex(n) {
    num_h[0] = n - 1; num_h[n] = 1;
  }
  int add_edge(int u, int v, int cap) {
    g[u].push_back(m++); g[v].push_back(m++);
    edges.push_back({v, 0, cap}); edges.push_back({u, 0, 0});
    return m-2;
  }
  edge& get_edge(int i) { return edges[i]; }  // get a reference to the i'th edge
  T max_flow(int _s, int _t) {
    s = _s; t = _t; h.assign(n, 0); h[s] = n;
    for (int id : g[s]) {
      int u = edges[id].to; ex[u] += edges[id].cap;
      if (inq[u] == 0 && u != s && u != t) bkt[0].push_back(u), inq[u] = 1;
```

```
      edges[id].flow += edges[id].cap; edges[id^1].flow -= edges[id].cap;
    }                  // if (max_bkt < n) [change if edge flow not needed]
    while (max_bkt >= 0) if (!bkt[max_bkt].empty()) {
      int u = bkt[max_bkt].back(); bkt[max_bkt].pop_back(); discharge(u);
    } else max_bkt--;
    return ex[t];
  }
};

// Maximum unweighted bipartite matching using the Hopcroft-Karp algorithm.
// Returns the number of matches and a vector of each left node's match,
// or -1 if the node had no match.  Complexity: O(Sqrt(V) E)
struct BipartiteMatching {
  int L, R, p;  vi m, used, d; vvi adj; queue<int> q;
  bool bfs() {
    for (int v=0; v<R; v++) if (!used[v]) d[v] = p, q.push(v);
    while (!q.empty()) {
      int v = q.front(); q.pop();
      if (d[v] != d[R]) for (int u : adj[v]) if (d[m[u]] < p)
        d[m[u]] = d[v] + 1, q.push(m[u]);
    }
    return d[R] >= p;
  }
  int dfs(int v) {
    if (v == R) return 1;
    for (int u : adj[v]) if (d[m[u]] == d[v] + 1 && dfs(m[u])) return m[u] = v, 1;
    d[v] = d[R];  return 0;
  }
  // Create a Bipartite graph with L and R vertices in the left and right part
  BipartiteMatching(int L, int R) : L(L), R(R), d(R+1), adj(R) { }
  void add_edge(int u, int v) { adj[v].push_back(u); } // Add edge left(u) -> right(v)
  pair<int, vi> match() {   // Returns the number of matches and the matches for each
    int res = 0;  m.assign(L, R), used.assign(R+1, 0);       // node in the left part
    for (p=0; bfs(); p = d[R]+1) for (int v=0; v<R; v++)
      if (!used[v] && dfs(v)) used[v] = 1, res++;
    replace(m.begin(), m.end(), R, -1); return {res, m};
  }
};

// Maximum matching in a general, unweighted graph. Returns the number of matches
// and a vector containing each node's match, or -1 if no match.  Complexity: O(V^3)
struct GraphMatching {
  int n, m;  vi match, p, base;  vvi adj;
  int lca(int a, int b) {
    vi used(n);
    while (1) { a=base[a], used[a]=1; if (match[a] == -1) break; a = p[match[a]]; }
    while (1) { b = base[b]; if (used[b]) return b; b = p[match[b]]; }
  }
  void mark_path(vi& blossom, int v, int b, int c) {
    for (; base[v] != b; v = p[match[v]])
      blossom[base[v]] = blossom[base[match[v]]] = 1,  p[v] = c,  c = match[v];
  }
  int find_path(int root) {
    vi used(n); iota(base.begin(), base.end(), 0);  p.assign(n, -1);
    used[root] = 1;  queue<int> q;  q.push(root);
    while (!q.empty()) {
      int v = q.front(); q.pop();
      for (int u : adj[v]) {
        if (base[v] == base[u] || match[v] == u) continue;
        if (u == root || (match[u] != -1 && p[match[u]] != -1)) {
          int cb = lca(v, u);  vi blossom(n);
          mark_path(blossom, u, cb, v), mark_path(blossom, v, cb, u);
          for (int i=0; i<n; i++) if (blossom[base[i]]) {
            base[i] = cb; if (!used[i]) used[i] = 1, q.push(i);
          }
        } else if (p[u] == -1) {
          p[u] = v; if (match[u] == -1) return u;
          u = match[u], used[u] = 1, q.push(u);
        }
      }
    }
    return -1;
  }
  // Create a graph on n vertices
  GraphMatching(int n) : n(n), m(0), base(n), adj(n) { }
  void add_edge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
  pair<int,vi> max_matching() {  // Returns the number of matches and each node's match
    p.assign(n, -1), match.assign(n, -1);
    for (int i=0; i<n; i++) {
      if (match[i] != -1) continue;
      int v = find_path(i), ppv = -1;
      while (v != -1) ppv = match[p[v]], match[v] = p[v], match[p[v]] = v, v = ppv;
    }
    return {(n - count(match.begin(), match.end(), -1)) / 2, match};
  }
};

// Finds a stable matching with the given preferences. mpref[i] lists male i's
// preferred matches in order (highest first). fpref lists female preferences
// in the same format. Returns a list of each male's match. Complexity: O(n^2)
vi stable_matching(const vvi& mpref, const vvi& fpref) {
  int n = (int)mpref.size(); vi mpair(n,-1), fpair(n,-1), p(n); vvi forder(n, vi(n));
  for (int i=0;i<n; i++) for (int j=0; j<n; j++) forder[i][fpref[i][j]] = j;
  for (int i=0; i<n; i++) {
    while (mpair[i] < 0) {
      int w = mpref[i][p[i]++], m = fpair[w];
      if (m == -1) mpair[i] = w, fpair[w] = i;
      else if (forder[w][i] < forder[w][m])
        mpair[m] = -1, mpair[i] = w, fpair[w] = i, i = m;
    }
  }
  return mpair;
};

// Minimum weight assignment (minimum weight perfect bipartite matching) in O(n^2 m)
// where n = #people, m = #tasks. Must have n <= m. A[i][j] is the cost of assigning
// person i to task j. Returns the weight and a vector listing each persons task.
template<typename T> pair<T, vi> hungarian(const vector<vector<T>>& A) {
  int n = (int) A.size(), m = (int) A[0].size();  T inf = numeric_limits<T>::max() / 2;
  vi way(m + 1), p(m + 1), used(m + 1), ans(n); vector<T> u(n+1), v(m+1), minv(m+1);
  for (int i = 1; i <= n; i++) {
    int j0 = 0, j1 = 0;  p[0] = i;  minv.assign(m + 1, inf),  used.assign(m + 1, 0);
    do {
      int i0 = p[j0];  j1 = 0; T delta = inf;  used[j0] = true;
      for (int j = 1; j <= m; j++) if (!used[j]) {
        T cur = A[i0 - 1][j - 1] - u[i0] - v[j];
        if (cur < minv[j]) minv[j] = cur,  way[j] = j0;
        if (minv[j] < delta) delta = minv[j],  j1 = j;
      }
      for (int j = 0; j <= m; j++)
        if (used[j]) u[p[j]] += delta,  v[j] -= delta;
        else minv[j] -= delta;
    } while (j0 = j1, p[j0]);
    do { int j1 = way[j0];  p[j0] = p[j1]; j0 = j1; } while (j0);
  }
  for (int i = 1; i <= m; i++) if (p[i] > 0) ans[p[i] - 1] = i - 1;
  return {-v[0], ans};
}

// Minimum cost flow using successive shortest paths. Finds the minimum cost
// to send cap units of flow from s to t. If you want max flow, use cap = INF.
// F = Flow type, C = Cost type.  Complexity: O(VE + E log(V) * FLOW)
template <typename F, typename C> struct MinCostFlow {
  struct edge { int from, to;  F flow, cap;  C cost; };
```

```cpp
  const C INF = numeric_limits<C>::max();  vector<C> pi, dist;
  int n, m;  vi pred, pe;  vvi g;  vector<edge> edges;
  typedef pair<C, int> pci;  priority_queue<pci, vector<pci>, greater<pci>> q;
  void bellman_ford() {  // Omit bellman_ford if using Leviticus instead of Dijkstra
    pi.assign(n, 0);
    for (int i = 0; i < n - 1; i++) for (auto &e : edges) if (e.flow < e.cap)
        pi[e.to] = min(pi[e.to], pi[e.from] + e.cost);
  }
  bool dijkstra(int s, int t) {  // Swap this for levit(s, t) for random data
    dist.assign(n, INF); pred.assign(n, -1); dist[s] = 0; q.emplace(0, s);
    while (!q.empty()) {
      C d; int u;  tie(d, u) = q.top(); q.pop();
      if (dist[u] == d) for (int i : g[u]) {
        auto &e = edges[i], v = e.to; C rcost = e.cost + pi[u] - pi[v];
        if (e.flow < e.cap && dist[u] + rcost < dist[v])
          pred[v]=u, pe[v]=i, dist[v]=dist[u]+rcost, q.emplace(dist[u]+rcost, v);
      }
    }
    for (int v = 0; v < n; v++) if (pred[v] != -1) pi[v] += dist[v];
    return dist[t] < INF;
  }
  pair<F, C> augment(int s, int t, F cap) {
    F flow = cap;  C cost = 0;
    for (int v = t; v != s; v = pred[v])
      flow = min(flow, edges[pe[v]].cap - edges[pe[v]].flow);
    for (int v = t; v != s; v = pred[v])
      edges[pe[v]].flow += flow, edges[pe[v]^1].flow -= flow,
        cost += edges[pe[v]].cost * flow;
    return {flow, cost};
  }
  // Create a flow network on n vertices
  MinCostFlow(int n) : n(n), m(0), pred(n), pe(n), g(n) {}
  int add_edge(int u, int v, F cap, C cost) {
    edges.push_back({u, v, 0, cap, cost}), g[u].push_back(m++);
    edges.push_back({v, u, 0, 0, -cost}), g[v].push_back(m++);
    return m - 2;
  }
  edge &get_edge(int i) { return edges[i]; }
  pair<F, C> flow(int s, int t, F cap) {
    for (auto &e : edges) e.flow = 0;
    F flow = 0; C cost = 0;  bellman_ford();
    while (flow < cap && dijkstra(s, t)) {
      auto res = augment(s, t, cap - flow); flow += res.X, cost += res.Y;
    }
    return {flow, cost};
  }
};

// If Dijkstra's is too slow for min-cost flow, it can be substituted with
// the Leviticus algorithm. This has average complexity O(E * flow) on random
// graphs which is better than Dijkstra but is O(VE * flow) in the worst case.
bool levit(int s, int t) {
  vi id(n, 0);  dist.assign(n, INF);  dist[s] = 0; deque<int> q;  q.push_back(s);
  while (!q.empty()) {
    int v = q.front();  q.pop_front();   id[v] = 2;
    for (auto i : g[v]) {
      auto& e = edges[i];
      if (e.flow < e.cap && dist[v] + e.cost < dist[e.to]) {
        dist[e.to] = dist[v] + e.cost;
        if (id[e.to] == 0) q.push_back(e.to);
        else if (id[e.to] == 2) q.push_front(e.to);
        id[e.to] = 1, pred[e.to] = v, pe[e.to] = i;
      }
    }
  }
  return dist[t] < INF;
}
```

```cpp
// Find the minimum cut in a weighted, undirected graph. Complexity: O(V^3)
template<typename T> struct MinCut {
  int n; vector<vector<T>> adj; const T INF = numeric_limits<T>::max();
  MinCut(int N) : n(N), adj(n, vector<T>(n)) { }
  void add_edge(int u, int v, T w) { adj[u][v] = adj[v][u] += w; }
  pair<T,vi> cut() {  // Returns the weight and the contents of one side of the cut
    T best = INF;  vi used(n), cut, best_cut;  auto weights = adj;
    for (int p=n-1; p >= 0; p--) {
      int prev, last = 0;  vi add = used, w = weights[0];
      for (int i=0; i<p; i++) {
        prev = last, last = -1;
        for (int j=1; j<n; j++) if (!add[j] && (last==-1 || w[j]>w[last])) last = j;
        if (i == p-1) {
          for (int j=0; j<n; j++) weights[prev][j] += weights[last][j];
          for (int j=0; j<n; j++) weights[j][prev] = weights[prev][j];
          used[last] = 1, cut.push_back(last);
          if (w[last] < best) best = w[last], best_cut = cut;
        } else {
          for (int j=0; j<n; j++) w[j] += weights[last][j];
          add[last] = 1;
        }
      }
    }
    return {best, best_cut};
  }
};

// Find strongly connected components in O(V + E). Optional:
// construct the DAG of SCCs in O(E log(E))
struct SCC {
  int n, comp;  vvi g, gt;  vi seq, vis;
  void dfs(int u, const vvi &adj) {
    for (int v : adj[u]) if (vis[v] == -1) { vis[v] = comp; dfs(v, adj); }
    seq.push_back(u);
  }
  // Create a graph on n vertices
  SCC(int n) : n(n), g(n), gt(n) { }
  void add_edge(int u, int v) { g[u].push_back(v); gt[v].push_back(u); }
  pair<int, vi> find_SCC() {
    vis.assign(n, -1); comp = 0;
    for (int i = 0; i < n; i++) if (vis[i] == -1) { vis[i] = comp; dfs(i, g); }
    vis.assign(n, -1); comp = 0;
    for (int i = n-1; i >= 0; i--) {
      int u = seq[i];
      if (vis[u] == -1) { vis[u] = comp; dfs(u, gt); comp++; }
    }
    return {comp, vis};
  }
  vvi get_dag() { // OPTIONAL: find_SCC() must be called first
    map<pii, int> mmap;  vvi dag(comp, vi());
    for (int u = 0; u < n; u++) for (int v : g[u]) {
      if (vis[u] == vis[v]) continue;
      if (!mmap.count(pii(vis[u], vis[v]))){
        dag[vis[u]].push_back(vis[v]);
        mmap[pii(vis[u], vis[v])] = 1;
      }
    }
    return dag;
  }
};

// Find the mean edge weight of the minimum mean cycle in a directed graph.
// If the graph contains no directed cycle, returns INF.  Complexity: O(VE)
struct MinimumMeanCycle {
  int n; vector<vector<pair<int,double>>> adj; const double INF = DBL_MAX / 2.0;
  MinimumMeanCycle(int N) : n(N), adj(n) { }
  void add_edge(int u, int v, double w) { adj[u].emplace_back(v,w); }
```

```
  double find_weight() {
    vector<vector<double>> DP(n+1, vector<double>(n, INF));
    fill(DP[0].begin(), DP[0].end(), 0);
    for (int i=0; i<n; i++) for (int u=0; u<n; u++) for (auto& e : adj[u])
      DP[i+1][e.X] = min(DP[i+1][e.X], DP[i][u] + e.Y);
    double res = INF;
    for (int i=0; i<n; i++) if (DP[n][i] < INF) {
      double hi = -INF;
      for (int j=0; j<n; j++) hi = max(hi, (DP[n][i]-DP[j][i]) / (n-j));
      res = min(res, hi);
    }
    return res;
  }
};

// Computes the minimum cost arborescence (directed minimum spanning tree) from root
// in a directed graph. Returns INF if no arborescence exists. Complexity: O(VE)
template<typename T> struct MinCostArborescence {
  typedef vector<vector<pair<int,T>>> Graph;  Graph adj;
  int n; const T INF = numeric_limits<T>::max() / 2;
  MinCostArborescence(int N) : adj(N), n(N) { }
  void add_edge(int u, int v, T w) { adj[u].emplace_back(v, w); }
  T find(int root) { return find(root, adj); }
  T find(int root, const Graph& G) {
    int nv = (int)G.size();  T res = 0;  vector<T> mins(nv, INF);
    for (int v=0; v<nv; v++) for (auto& e : G[v]) mins[e.X]=min(mins[e.X],e.Y);
    for (int v=0; v<nv; v++) if (v != root) {
      if (mins[v] == INF) return INF; else res += mins[v];
    }
    SCC scc(nv);  // Include Strongly-connected components code
    for (int v=0; v<nv; v++) for (auto& e : G[v]) if (e.X != root)
      if (e.Y - mins[e.X] == 0) scc.add_edge(v, e.X);
    int m; vi comp;  tie(m, comp) = scc.find_SCC();  Graph G2(m);
    if (m == nv) return res;
    for (int v=0; v<nv; v++) for (auto& e : G[v]) if (comp[v] != comp[e.X])
      G2[comp[v]].emplace_back(comp[e.X], e.Y - mins[e.X]);
    return min(INF, res + find(comp[root], G2));
  }
};
```

6 Tree Decomposition Techniques

```
// Heavy-Light Decomposition. Facilitates ranged queries on trees in O(log^2(n))
// time. decompose_tree(root) returns a vector of values that should be
// initialised in a segment tree for queries. ranged_query accepts a path {u..v}
// and a lambda that takes two values i,j that are indices in the segment tree that
// should be processed in the query. point_query accepts an edge (u, v) and a
// lambda taking the index i corresponding to that edge in the segment tree.
// T is the type of the edge weights.  Complexity: O(n) to build.
template<typename T> struct HeavyLightDecomposition {
  int n; vi heavy, head, par, pos, level; vector<T> cost;
  vector<vector<pair<int,T>>> adj;
  int dfs(int u, int p, int d) {
    int size = 1, max_child = 0, max_child_id = -1;
    par[u] = p, level[u] = d;
    for (auto& child : adj[u]) if (child.X != p) {
      cost[child.X] = child.Y;
      int child_size = dfs(child.X, u, d + 1);
      if (child_size > max_child) max_child = child_size, max_child_id = child.X;
      size += child_size;
    }
    if (max_child * 2 >= size) heavy[u] = max_child_id;
    return size;
  }
  // Create a tree on n vertices -- add edges using add_edge(u, v, cost)
  HeavyLightDecomposition(int n) :
    n(n), heavy(n), head(n), par(n), pos(n), level(n), cost(n), adj(n) { }
  void add_edge(int u, int v, T cost) {
    adj[u].emplace_back(v, cost), adj[v].emplace_back(u, cost);
  }
  vector<T> decompose_tree(int root = 0) {  // Perform HLD.
    vector<T> val(n);  heavy.assign(n, -1);  dfs(root, -1, 0);  int curPos = 0;
    for (int i=0, cur=0; i<n; cur=++i)
      if (par[i] == -1 || heavy[par[i]] != i) while (cur != -1)
        val[curPos] = cost[cur], pos[cur] = curPos++, head[cur] = i, cur = heavy[cur];
    return val;
  }
  template<typename F> void ranged_query(int u, int v, F query) {
    while (head[u] != head[v]) {
      if (level[head[u]] > level[head[v]]) swap(u, v);
      query(pos[head[v]], pos[v]);  v = par[head[v]];
    }
    if (u != v) query(min(pos[u],pos[v])+1, max(pos[u],pos[v]));
  }
  template<typename F> void point_query(int u, int v, F query) {
    query(level[v] > level[u] ? pos[v] : pos[u]);
  }
};

// Centroid Decomposition. Constructs a valid centroid tree of the given tree.
// croot -- the root of the centroid tree
// cadj -- downward adjacency list of the centroid tree
// par -- parent in the centroid tree (-1 for the root)
struct CentroidDecomposition {
  int n, cnt = 0, croot;  vvi adj, cadj;  vi par, mark, size;
  int dfs(int u, int p) {
    size[u] = 1;
    for (int v : adj[u]) if (v != p && !mark[v]) dfs(v, u), size[u] += size[v];
    return size[u];
  }
  int find_centroid(int u, int p, int sz) {
    for (int v : adj[u]) if (v != p && !mark[v])
      if (size[v] * 2 > sz) return find_centroid(v, u, sz);
    return u;
  }
  int find_centroid(int src) { return find_centroid(src, -1, dfs(src, -1)); }
  // Create a tree on n vertices -- add edges using add_edge(u, v)
  CentroidDecomposition(int n) : n(n), adj(n), cadj(n), par(n,-1), mark(n), size(n) { }
  void add_edge(int u, int v) { adj[u].push_back(v), adj[v].push_back(u); }
  int decompose_tree(int src = 0) {
    int c = find_centroid(src);  mark[c] = 1;
    for (int u : adj[c]) if (!mark[u]) {
      int v = decompose_tree(u);
      cadj[c].push_back(v), par[v] = c;
    }
    return croot = c;
  }
};
```

7 Linear Algebra

```
// Reduces the given matrix to reduced row-echelon form using Gaussian Elimination.
// Returns the rank of A. T must be a floating-point type. Complexity: O(n^3).
const double EPS = 1e-10;

template<typename T> int rref(vector<vector<T>>& A) {
  int n = (int)A.size(), m = (int)A[0].size(), r = 0;
  for (int c=0; c<m && r<n; c++) {
    int j = r;
    for (int i=r+1; i<n; i++) if (abs(A[i][c]) > abs(A[j][c])) j = i;
    if (abs(A[j][c]) < EPS) continue;
    swap(A[j], A[r]);  T s = 1.0 / A[r][c];
    for (int j=0; j<m; j++) A[r][j] *= s;
    for (int i=0; i<n; i++) if (i != r) {
```

```
      T t = A[i][c];
      for (int j=0; j<m; j++) A[i][j] -= t * A[r][j];
    }
    r++;
  }
  return r;
}

// Integral matrix triangulation. Used by linear diophantine solver below.
template<typename T> int triangulate(vector<vector<T>>& A, int m, int n, int cols) {
  lldiv_t d;  int ri = 0, ci = 0;
  while (ri < m && ci < cols) {
    int pi = -1;
    for (int i = ri; i < m; i++)
      if (A[i][ci] && (pi == -1 || abs(A[i][ci]) < abs(A[pi][ci]))) pi = i;
    if (pi == -1) ci++;
    else {
      int k = 0;
      for (int i = ri; i < m; i++) if (i != pi) {
        d = lldiv(A[i][ci], A[pi][ci]);
        if (d.quot) { for (int j = ci; j < n; j++) A[i][j] -= d.quot * A[pi][j]; k++; }
      }
      if (!k) { for (int i=ci; i<n && ri!=pi; i++) swap(A[ri][i],A[pi][i]); ri++,ci++; }
    }
  }
  return ri;
}

// System of linear diophantine equations A*x = b. T must be an integral type.
// Returns dim(null space), or -1 if there is no solution, or -2 if inconsistent.
// xp: a particular solution
// basis: an n x n matrix whose first dim columns form a basis of the nullspace.
// All solutions are obtained by adding integer multiples the basis elements to xp.
// Complexity: O(n^3)
template<typename T> tuple<int,vector<T>,vector<vector<T>>> diophantine_linsolve(
    vector<vector<T>>& A, vector<T>& b) {
  int m = (int)A.size(), n = (int)A[0].size(), i, j, rank;  T d;
  vector<vector<T>> mat(n + 1, vector<T>(m + n + 1));
  for (i = 0; i < m; i++) mat[0][i] = -b[i];
  for (i = 0; i < m; i++) for (j = 0; j < n; j++) mat[j + 1][i] = A[i][j];
  for (i = 0; i < n + 1; i++) for (j = 0; j < n + 1; j++) mat[i][j + m] = (i == j);
  rank = triangulate(mat, n + 1, m + n + 1, m + 1), d = mat[rank - 1][m];
  vector<vector<T>> basis(n, vector<T>(n));  vector<T> xp(n);
  if (d != 1 && d != -1) return make_tuple(-1, xp, basis);
  for (i = 0; i < m; i++) if (mat[rank - 1][i]) return make_tuple(-2, xp, basis);
  for (i = 0; i < n; i++) {
    xp[i] = d * mat[rank - 1][m + 1 + i];
    for (j = 0; j < n + 1 - rank; j++) basis[i][j] = mat[rank + j][m + 1 + i];
  }
  return make_tuple(n + 1 - rank, xp, basis);
}

// solves Ax = b exactly. Returns {det, x_star}, solution is x_star[i] / det.
// T must be an integral data type (int, long long, etc.) Complexity: O(n^3)
template<typename T> pair<T,vector<T>> fflinsolve(vector<vector<T>> A, vector<T> b) {
  int k_c, k_r, pivot, sign = 1, n = (int)A.size(); T d = 1;
  for (k_c = k_r = 0; k_c < n; k_c++) {
    for (pivot = k_r; pivot < n && !A[pivot][k_r]; pivot++);
    if (pivot < n) {
      if (pivot != k_r) {
        for (int j = k_c; j < n; j++) swap(A[pivot][j], A[k_r][j]);
        swap(b[pivot], b[k_r]), sign *= -1;
      }
      for (int i = k_r + 1; i < n; i++) {
        for (int j = k_c + 1; j < n; j++)
          A[i][j] = (A[k_r][k_c] * A[i][j] - A[i][k_c] * A[k_r][j]) / d;
        b[i] = (A[k_r][k_c] * b[i] - A[i][k_c] * b[k_r]) / d,  A[i][k_c] = 0;
      }
      if (d) d = A[k_r][k_c];
      k_r++;
    } else d = 0;
  }
  if (!d) {
    for (int k = k_r; k < n; k++) if (b[k]) return {0,{}}; // inconsistent system
    return {0,{}};      // multiple solutions
  }
  vector<T> x_star(n);
  for (int k = n - 1; k >= 0; k--) {
    x_star[k] = sign * d * b[k];
    for (int j = k + 1; j < n; j++) x_star[k] -= A[k][j] * x_star[j];
    x_star[k] /= A[k][k];
  }
  return {sign * d, x_star};
}

// LU-Decomposition. Can be used to solve Ax = b in floating-point
// Returns {determinant, pivot, LU}. Complexity: O(n^3)
// - Call LU_solve(LU, pivot, b) to solve linear system Ax = b
const double EPS = 1e-9;

template<typename T> tuple<T, vi, vector<vector<T>>> LU_decomp(vector<vector<T>> A) {
  int n = (int)A.size(); vi pivot(n); vector<T> s(n); T c, t, det = 1.0;
  for (int i = 0; i < n; i++) {
    s[i] = 0.0;
    for (int j = 0; j < n; j++) s[i] = max(s[i], fabs(A[i][j]));
    if (s[i] < EPS) return make_tuple(0, pivot, A);  // Singular
  }
  for (int k = 0; k < n; k++) {
    c = fabs(A[k][k] / s[k]), pivot[k] = k;
    for (int i = k + 1; i < n; i++) if ((t = fabs(A[i][k] / s[i])) > c)
      c = t, pivot[k] = i;
    if (c < EPS) return make_tuple(0, pivot, A);  // Singular
    if (k != pivot[k]) {
      det *= -1.0;  swap(s[k], s[pivot[k]]);
      swap_ranges(A[k].begin() + k, A[k].end(), A[pivot[k]].begin() + k);
    }
    for (int i = k + 1; i < n; i++) {
      A[i][k] /= A[k][k];
      for (int j = k + 1; j < n; j++) A[i][j] -= A[i][k] * A[k][j];
    }
    det *= A[k][k];
  }
  return make_tuple(det, pivot, A);
}

// Solve Ax = b in floating-point using the LU-decomposition of A.
// T must be a floating-point type (double, long double). Complexity: O(n^2)
template<typename T> vector<T> LU_solve(vector<vector<T>>& LU, vi& piv, vector<T>& b) {
  int n = (int)LU.size();  vector<T> x = b;
  for (int k = 0; k < n - 1; k++) {
    if (k != piv[k]) swap(x[k], x[piv[k]]);
    for (int i = k + 1; i < n; i++) x[i] -= LU[i][k] * x[k];
  }
  for (int i = n - 1; i >= 0; i--) {
    for (int j = i + 1; j < n; j++) x[i] -= LU[i][j] * x[j];
    x[i] /= LU[i][i];
  }
  return x;
}
```

8 Data Structures

```
// Fenwick tree with ranged updates and point queries. Complexity: O(log(n))
template<typename T> struct FenwickTree {
```

```
  int N;  vector<T> A;
  FenwickTree(int n): N(n+1), A(N) {}                         // Create tree with n elements
  void adjust(int b, T v) { for (;b;b-=b&-b) A[b]+=v; }                // Add v to A[0,b)
  void adjust(int a,int b, T v) { adjust(b,v), adjust(a,-v); }         // Add v to A[a,b)
  T pq(int i) { T r=0; for (i++;i<N;i+=i&-i) r+=A[i]; return r; }             // Get A[i]
};

// Fenwick Tree with ranged queries and point updates. Complexity: O(log(n))
template<typename T> struct FenwickTree {
  int N;  vector<T> A;
  FenwickTree(int n): N(n+1), A(N) {}                         // Create tree with n elements
  T rq(int b) { int r=0; for (;b;b-=b&-b) r+=A[b]; return r; }         // Get sum A[0,b)
  T rq(int a,int b) { return rq(b)-rq(a); }                            // Get sum A[a,b)
  void adjust(int i, T v) { for (i++;i<N;i+=i&-i) A[i]+=v; }              // A[i] += v
  int lower_bound(T sum) {                          // find min i such that sum(A[0..i]) >= sum
    int i = 0;                                              // Returns n if there is no such i
    for (int b = 1 << (31-__builtin_clz(N)); b; b /= 2)     // (Only works if A[i] >= 0
      if (i+b < N && sum > A[i+b]) sum -= A[i+b], i+=b;                       // for all i)
    return i;
  }
};

// Sparse table implementing static range minimum query. Can change operation
// to max, gcd, etc. Complexity: O(n log(n)) to build, O(1) to query.
template<typename T> struct SparseTable {
  int n;  vector<vector<pair<T, int>>> sptable;  vi lg;
  SparseTable(const vector<T> &A) : n(A.size()), lg(n+1, 0) {
    for (int i = 2; i <= n; i++) lg[i] = lg[i/2] + 1;
    sptable.assign(lg[n] + 1, vector<pair<T, int>>(n));
    for (int i = 0; i < n; i++) sptable[0][i] = {A[i], i};
    for (int i = 1; i <= lg[n]; i++) for (int j = 0; j + (1 << i) - 1 < n; j++)
      sptable[i][j] = min(sptable[i-1][j], sptable[i-1][j + (1 << (i-1))]);
  }
  pair<T, int> query(int L, int R) { // Find {min A[L..R], i}
    int k = lg[R - L + 1];
    return min(sptable[k][L], sptable[k][R - (1 << k) + 1]);
  }
};

// Segment tree for dynamic range minimum query. For maximum query, change min
// to max, and use min() as the identity value. Can also use gcd, lcm, sum etc
// with appropriate identity. Complexity: O(n) to build, O(log(n)) to query.
template<typename T> struct SegmentTree {
  int n; vector<pair<T,int>> st; const pair<T,int> I = {numeric_limits<T>::max(),-1};
  SegmentTree(const vector<T>& A) : n(A.size()), st(2*n, I) {
    for (int i=0;i<n;i++) st[n+i] = {A[i],i};
    for (int i=n-1; i; --i) st[i] = min(st[2*i], st[2*i+1]);
  }
  void update(int i, int val) {           // Set A[i] = val
    for (st[i+=n] = {val,i}; i > 1; i/= 2) st[i/2] = min(st[i], st[i^1]);
  }
  pair<T,int> query(int l, int r) {      // Find min A[l..r]
    pair<T,int> res = I;
    for (l += n, r += n; l <= r; l /= 2, r /= 2) {
      if (l&1) res = min(res, st[l++]);
      if (~r&1) res = min(res, st[r--]);
    }
    return res;
  }
};

// Performs range updates and queries on an array. Accepts a custom segment class T
// (see example) which contains both the value of the segment and any updates which
// need to be propagated to children. Intervals are 0-based and half-open. Updates
// are of type U. Complexity: O(N) to build, O(log N) to update and query.
template <typename T, typename U> struct SegmentTree {
  T I,t[4]; int N,h; vector<T> A;  // I is the identity value for segments
  SegmentTree(const vector<T>& data, T I=T()): I(I), N(data.size()),
    h(sizeof(int)*8-__builtin_clz(N)), A(2*N,I) {
      copy(data.begin(),data.end(),A.begin()+N);
      for (int i=N-1;i;i--) op(i);
  }
  void op(int i) { A[i].op(A[2*i],A[2*i+1]); }
  void prop(int i) { A[2*i].us(A[i].U); A[2*i+1].us(A[i].U); A[i].NU(); }
  void push(int i) {for (int j=h;j;j--) prop(i>>j);}
  void update(int l, int r, U v) {  // Update is on the half-open interval [l, r)
    push(l+=N); push((r+=N)-1); bool cl=0,cr=0;
    for (;l<r;l/=2,r/=2) {
      if (cl) op(l-1); if (cr) op(r);
      if (l&1) A[l++].us(v), cl=1;
      if (r&1) A[--r].us(v), cr=1;
    }
    if (l==1 && cr) op(1);
    else for (l--;r>0;l/=2,r/=2) {
      if (cl && l) op(l);
      if (cr && (!cl || (l!=r && r!=1))) op(r);
    }
  }
  T query(int l, int r) {  // Query is on the half-open interval [l, r)
    push(l+=N); push((r+=N)-1);
    t[0]=t[2]=I; int i=0,j=2;
    for (;l<r;l/=2,r/=2) {
      if (l&1) t[i^1].op(t[i],A[l++]), i^=1;
      if (r&1) t[j^1].op(A[--r],t[j]), j^=1;
    }
    t[i^1].op(t[i],t[j]);
    return t[i^1];
  }  // OPTIONAL: Find the largest x such that Segment([l,x)).b(...) returns true
  template<class...Ts> pair<int,T> partitionPointRight(int l,Ts...args) {
    int r=l,w=1,p=0; t[0]=I;
    if (r<N) for (push(l+=N);r+2*w<=N && (t[1-p].op(t[p],A[l]),t[1-p].b(args...));)
      if (l&1) branchr(++l),r+=w,p^=1; else l/=2,w*=2;
    for (;w;l*=2,w/=2) if (r+w<=N && (prop(l/2),t[1-p].op(t[p],A[l]),t[1-p].b(args...)))
      l++,r+=w,p^=1;
    return {r,t[p]};
  }  // OPTIONAL: Find the smallest x such that Segment([x, r)).b(...) returns true
  template<class...Ts> pair<int,T> partitionPointLeft(int r,Ts...args) {
    int l=r,w=1,p=0; t[0]=I;
    for (push(r+=N-1);l>=2*w && (t[1-p].op(A[r],t[p]),t[1-p].b(args...));)
      if (~r&1) branchl(r--),l-=w,p^=1; else r/=2,w*=2;
    for (;w;r=2*r+1,w/=2) if(l>=w && (prop(r/2),t[1-p].op(A[r],t[p]),t[1-p].b(args...)))
      r--,l-=w,p^=1;
    return {l,t[p]};
  }
  void branchr(int i) {for (int j=__builtin_ctz(i);j;j--) prop(i>>j);}
  void branchl(int i) {for (int j=__builtin_ctz(i--);j;j--) prop(i>>j);}
};

// Range minimum query example for SegmentTree. Your segment class must implement:
// op: merge two child segments, us: apply a lazy update, NU: clear any pending update
// You must also provide a public field U = the current pending update. You may either
// provide a suitable identity value to SegmentTree or the default constructor is used.
struct RangeMin {
  int a = INT_MAX, U = INT_MIN;                          // U is the current pending update
  void op(RangeMin& b, RangeMin& c) { a=min(b.a,c.a); }           // Merge two segments
  void us(int v) { if (v!=INT_MIN) a=U=v; }                        // Apply a lazy update
  void NU() { U = INT_MIN; }                                    // Node requires no update
  bool b(int v) { return a >= v; }    // OPTIONAL: Partition criteria: Must be monotone
};
SegmentTree<RangeMin, int> st(vector<RangeMin>(20));    // Create a RangeMin SegmentTree

// Multidimensional vector required for multidimensional segment tree
template<class T, int D> struct Vec {typedef vector<typename Vec<T,D-1>::type> type;};
template<typename T> struct Vec<T,0> { typedef T type; };
```

```
template<typename T, int D> using MDV = typename Vec<T,D>::type;

// Multidimensional segment tree supporting point updates and ranged queries.
// The operation and an identity element must be provided by the template traits Op.
// Build initial data with st.build(A) where A is a D-dimensional vector of type T.
// Query ranges are closed hyperrectangles, updates are on single D-dimensional points.
// Complexity: O(2^D N1 N2..ND) to build, O(log(N1)log(N2)..log(ND)) to query.
template<typename T, typename Op, int D> struct SegmentTree {
  template<int _D> using ST = SegmentTree<T,Op,_D>;
  int N; vector<ST<D-1>> A;
  void build(const MDV<T,D>& data) {
    for (int c=0;c<N;c++) A[c+N].build(data[c]);
    for (int c=N-1;c;c--) A[c].merge(A[2*c],A[2*c+1]);
  }
  void merge(const ST<D>& L, const ST<D>& R) {
    for (int c=1;c<2*N;c++) A[c].merge(L.A[c],R.A[c]);
  }
  template<typename...Ts> void merge(const ST<D>& L, const ST<D>& R, int i, Ts...is) {
    for (i+=N;i;i/=2) A[i].merge(L.A[i],R.A[i],is...);
  }  // Create a segment tree with dimensions N1 * N2 * N3 * ...
  template<typename...Ts> SegmentTree(int N,Ts...Ns): N(N), A(2*N,ST<D-1>(Ns...)) {}
  // Set the value at (i1, i2, i3, ...) to v
  template<typename...Ts> void update(const T& v, int i, Ts...is) {
    for (A[i+=N].update(v,is...);i/=2;) A[i].merge(A[2*i],A[2*i+1],is...);
  }  // Perform a ranged query on the range ([i1,j1] * [i2,j2] * ...)
  template<typename...Ts> T query(int i,int j,Ts...limits) {
    T r = Op::I;
    for (i+=N,j+=N;i<=j;i/=2,j/=2) {
      if (i&1) r = Op::op(r,A[i++].query(limits...));
      if (~j&1) r = Op::op(r,A[j--].query(limits...));
    }
    return r;
  }
};
template<typename T, typename Op> struct SegmentTree<T,Op,0> {
  typedef SegmentTree<T,Op,0> ST;  T a;
  SegmentTree() { a = Op::I; }
  void build(const T& data) {a=data;}
  void merge(const ST& L,const ST& R) { a = Op::op(L.a,R.a); }
  void update(const T& v) { a=v; }
  T query() { return a; }
};

// Example: Op for a multidimensional segment tree for ranged sums
struct RangeSumOp {
  static const int I = 0;
  static int op(const int& x, const int& y) { return x+y; }
};

// Union-Find with union-by-rank, path compression, component size and count
// number of connected components. Complexity: O(log*(N)) amortized per query.
struct UnionFind {
  int n;  vi A, s, rank;
  UnionFind(int n) : n(n), A(n), s(n, 1), rank(n) { iota(A.begin(), A.end(), 0); }
  int find(int x) { return A[x]==x ? x : A[x]=find(A[x]); }
  bool merge(int x, int y) {  // Connect x and y. Returns false if x and y were
    x = find(x); y = find(y);                 // already connected, true otherwise
    if (x == y) return false;
    if (rank[x] < rank[y]) swap(x, y);
    A[y] = x; s[x] += s[y]; n--;
    if (rank[x] == rank[y]) rank[x]++;
    return true;
  }
  bool connected(int x, int y) { return (find(x) == find(y)); }
  int size(int x) { return s[find(x)]; }  // Returns the size of the set representing x
  int num_sets() { return n; }            // Returns the number of connected components
};
```

```
// Link-Cut Tree for dynamic connectivity on a forest of trees, dynamic lowest common
// ancestor queries and dynamic aggregate statistics for root-to-node paths. Default
// aggregate is node depths, can be customised. Complexity: O(log(N)) amortized queries
struct LinkCutTree {
  struct Node { int sz,i,f; Node *p,*pp,*l,*r; Node() : f(0),p(0),pp(0),l(0),r(0) {} };
  // Initialise: Create an initially disconnected forest of n isolated vertices --------
  LinkCutTree(int n) : V(n) { for(int i=0; i<n; i++) V[i].i = i, update(&V[i]); }
  // Update operations ------------------------------------------------------------------
  void link(int u, int v) { _link(&V[u], &V[v]); }                // Make u a subtree of v
  void cut(int u) { _cut(&V[u]); }                           // Disconnect u from its parent
  void make_root(int u) {                          // Make u the root of its connected component
    Node* x = &V[u]; access(x);
    if (x->l) x->l->p = 0, x->l->f ^= 1, x->l->pp = x, x->l = 0, update(x);
  }
  // Query operations -------------------------------------------------------------------
  int parent(int u) { access(&V[u]); return V[u].l ? V[u].l->i : -1; }     // Parent of u
  int root(int u) { return _root(&V[u])->i; }         // The root of the tree containing u
  bool connected(int u, int v) { return root(u) == root(v); }  // Are u and v connected?
  int lca(int u, int v) { return _lca(&V[u], &V[v])->i; }      // Find the LCA of u and v
  int query(int u) { return _query(&V[u]); }    // Aggregate path statistic query (depth)
  // OPTIONAL: Customise the aggregate path query below (default is node depth) ------
  void update(Node* x) { x->sz = 1 + (x->l ? x->l->sz : 0) + (x->r ? x->r->sz : 0); }
  int _query(Node* x) { access(x); return x->sz-1; }
  // Internal node operations (probably don't modify below here) -----------------------
  vector<Node> V;
  Node* _root(Node* x) { access(x); while(x->l) { x=x->l;push(x); } splay(x); return x;}
  void _cut(Node* x) { access(x); x->l->p = 0; x->l = 0; update(x); }
  void _link(Node* x, Node* y) { access(x); access(y); x->l = y; y->p = x; update(x); }
  Node* _lca(Node* x, Node* y) { access(x); return access(y); }
  void push(Node* x) {  // Push lazy subtree flipping down the auxillary tree
    if (x->f == 0) return;  x->f = 0;  swap(x->l, x->r);
    if (x->l) x->l->f ^= 1; if (x->r) x->r->f ^= 1; update(x);
  }  // Splay tree right rotation for the auxillary trees
  void rotr(Node* x) {
    Node* y = x->p; Node* z = y->p;
    if((y->l = x->r)) y->l->p = y;
    x->r = y, y->p = x;
    if((x->p = z)) { if(y == z->l) z->l = x; else z->r = x; }
    x->pp = y->pp, y->pp = 0, update(y);
  }  // Splay tree left rotation for the auxillary trees
  void rotl(Node* x) {
    Node* y = x->p; Node* z = y->p;
    if((y->r = x->l)) y->r->p = y;
    x->l = y, y->p = x;
    if((x->p = z)) { if(y == z->l) z->l = x; else z->r = x; }
    x->pp = y->pp, y->pp = 0, update(y);
  }
  void splay(Node* x) {  // Rotates x to become the root of its auxillary tree
    for (Node* y = x->p; y; y = x->p) {
      if (x->p->p) push(x->p->p);  push(x->p); push(x);  // Push flips down the tree
      if(y->p == 0) { if (x == y->l) rotr(x); else rotl(x); }
      else {
        if(y == y->p->l) { if(x == y->l) rotr(y), rotr(x); else rotl(x), rotr(x); }
        else { if(x == y->r) rotl(y), rotl(x); else rotr(x), rotl(x); }
      }
    }
    push(x), update(x);
  }  // Makes the root-to-v path preferred and makes v the root of its auxillary tree.
  Node* access(Node* x) {      // Returns the lowest ancestor of x in the root auxillary
    Node* last = x;  splay(x);          // tree (LCA with the most recently accessed node)
    if(x->r) x->r->pp = x, x->r->p = 0, x->r = 0, update(x);
    while(x->pp) {
      Node* y = x->pp; last = y; splay(y);
      if(y->r) y->r->pp = y, y->r->p = 0;
      y->r = x, x->p = y, x->pp = 0, update(y), splay(x);
    }
    return last;
```

```cpp
  }
};

// Customisable Treap data structure. Complexity: expected O(log(N)) per query.
template<typename K, typename V> struct Node {
  typedef unique_ptr<Node<K,V>> node_p;
  K key; V val;  int p, size=1;  node_p l=0, r=0;
  Node(K key, V val) : key(key), val(val), p(rand()) { update(); }
  node_p left() { auto t = move(l); update(); return t; }
  node_p right() { auto t = move(r); update(); return t; }
  void left(node_p t) { l = move(t); update(); }
  void right(node_p t) { r = move(t); update(); }
  void update() { size = 1 + (l ? l->size : 0) + (r ? r->size : 0); }
};
template<typename K, typename V> struct Treap {
  typedef Node<K,V> node; typedef Treap<K,V> treap; typedef unique_ptr<node> node_p;
  node_p root;  Treap() { }  // Construct an empty Treap
  // Constructs a Treap by merging the Treaps t1 and t2 where t1 < t2
  Treap(treap& t1, treap& t2) : root(merge(move(t1.root), move(t2.root))) { }
  void insert(K key, V val) {  // Insert the (key,value) pair into the Treap
    if (!root) root = make_unique<node>(key, val);
    else root = insert(move(root), key, val);
  } // Remove the item with the given key from the Treap if it exists
  void remove(K key) { if (root) root = remove(move(root), key); }
  // Split the Treap into two Treaps, containing all keys < key and >= key respectively
  pair<treap, treap> split(K key) {
    node_p left, right; tie(left, right) = split(move(root), key);
    return {treap(move(left)), treap(move(right))};
  }  // Create a Treap owning the given root
  Treap(node_p root) : root(move(root)) { }
  pair<node_p, node_p> split(node_p t, K key) {  // Split the subtree t at key
    if (!t) return {nullptr, nullptr};
    if (t->key < key) {  // Change < to <= if you want a {<=, >} split
      node_p left,right,tmp = t->right(); tie(left, right) = split(move(tmp), key);
      return {merge(move(t), move(left)), move(right)};
    } else {
      node_p left,right,tmp = t->left(); tie(left, right)=split(move(tmp), key);
      return {move(left), merge(move(right), move(t))};
    }
  }
  node_p merge(node_p a, node_p b) {  // Merge the subtrees a and b where a < b
    if (!a) return b; if (!b) return a;
    if (a->p < b->p) { a->right(merge(a->right(), move(b))); return a; }
    else { b->left(merge(move(a), b->left())); return b; }
  }
  node_p insert(node_p t, K key, V val) {  // Insert(key,val) into the given subtree
    if (!t) return make_unique<node>(key, val);
    if (key < t->key) t->left(insert(t->left(), key, val));
    else if (key > t->key) t->right(insert(t->right(), key, val));
    else t->val = val;
    return normalise(move(t));
  }
  node_p remove(node_p t, K key) {  // Remove key from the given subtree
    if (!t) return t;
    if (key < t->key) { t->left(remove(t->left(), key)); return t; }
    if (key > t->key) { t->right(remove(t->right(), key)); return t; }
    return merge(t->left(), t->right());
  }
  node_p normalise(node_p t) {  // Ensure that the heap-ordering of the p's is correct
    if (t->l && t->l->p < t->p && (!t->r || t->l->p < t->r->p)) {
      auto tmp = t->left(); t->left(tmp->right());
      tmp->right(move(t)); return tmp;
    } else if (t->r && t->r->p < t->p) {
      auto tmp = t->right(); t->right(tmp->left());
      tmp->left(move(t)); return tmp;
    } else return t;
  }
};

// Implicit Treap data structure. Supports ranged substring, erase, insert, reverse.
// Complexity: expected O(log(N)) per query, O(N) to build from or convert to vector
template<typename T> struct Node {
  typedef unique_ptr<Node> node_p; T val; node_p l, r; int size; bool rev;
  Node(T val) : val(val), rev(0) { update(); }
  node_p left() { auto t = move(l); update(); return t; }
  node_p right() { auto t = move(r); update(); return t; }
  void left(node_p t) { l = move(t); update(); }
  void right(node_p t) { r = move(t); update(); }
  void update() {
    size = 1 + (l ? l->size : 0) + (r ? r->size : 0);
    if (rev) { rev=0; swap(l, r); if (l) l->rev ^= 1; if (r) r->rev ^= 1; }
  }
};
template<typename T> struct ImplicitTreap {
  typedef Node<T> node; typedef ImplicitTreap<T> treap;
  typedef unique_ptr<Node<T>> node_p;  node_p root;
  ImplicitTreap(const vector<T>& A) {  // Build an ImplicitTreap containing A
    function<node_p(int,int)> build = [&](int l, int r) {
      node_p v;  int m = (l+r)/2; if (l >= r) return v; v = make_unique<node>(A[m]);
      v->left(build(l, m)), v->right(build(m+1,r)); return v;
    }; root = build(0, A.size());
  }
  int size() { return root ? root->size : 0; }
  T& operator[](int i) { return lookup(root, i); } // Return the element at position i
  T& lookup(node_p& t, int key) {
    t->update(); int cur = (t->l ? t->l->size : 0); if (cur==key) return t->val;
    if (cur > key) return lookup(t->l, key); return lookup(t->r, key-cur-1);
  }
  ImplicitTreap(node_p root) : root(move(root)) { }  // Create a tree rooted at root
  treap cut(int l, int r) {  // Cut out and return the substring [l, r]
    node_p t1,t2,t3; tie(t1,t2)=split(move(root),l), tie(t2,t3)=split(move(t2),r-l+1);
    root = merge(move(t1), move(t3)); return treap(move(t2));
  }
  void insert(int i, treap&& other) {  // Insert the contents of 'other' at position i
    node_p t1, t2; tie(t1, t2) = split(move(root), i);
    root = merge(move(t1), move(other.root)); root = merge(move(root), move(t2));
  }
  void insert(int i, treap& other) { insert(i, move(other)); }
  void reverse(int l, int r) {  // Reverse the contents of [l, r]
    node_p t1,t2,t3; tie(t1,t2)=split(move(root),l); tie(t2,t3)=split(move(t2),r-l+1);
    t2->rev ^= 1; root = merge(move(t1), move(t2)), root = merge(move(root),move(t3));
  }
  pair<node_p, node_p> split(node_p t, int key, int add=0) {
    if (!t) return {nullptr, nullptr};
    t->update(); int cur = add + (t->l ? t->l->size : 0);
    if (key <= cur) {  // Recursively split the left subtree
      node_p left,right,tmp = t->left(); tie(left,right)=split(move(tmp),key,add);
      return {move(left), merge(move(right), move(t))};
    } else {  // Recursively split the right subtree
      node_p left,right,tmp = t->right(); tie(left,right) = split(move(tmp),key,cur+1);
      return {merge(move(t), move(left)), move(right)};
    }
  }
  node_p merge(node_p l, node_p r) {  // Merge the trees rooted at l and r
    if (l) l->update(); if (r) r->update(); if (!l || !r) return l ? move(l) : move(r);
    bool left = (1.0*rand()/RAND_MAX) < (1.0 * l->size) / (l->size + r->size);
    if (left) { l->right(merge(l->right(), move(r))); return l; }   // Merge randomly to
    else { r->left(merge(move(l), r->left())); return r; }  // maintain expected balance
  }
  vector<T> to_vector() {  // Convert the contents of the tree into a vector<T>
    vector<T> res; res.reserve(size()); function<void(node_p&)> go = [&](node_p& v) {
      if (!v) return; v->update(); go(v->l); res.push_back(v->val); go(v->r);
    }; go(root); return res;
  }
```

```cpp
};

// GNU Policy-Based Data Structures ----------------------------------------------------
// prefix_trie:: A Patricia (compact) trie that implements fast prefix searches.
// Insertion syntax matches std::set::insert, returns pair{iterator, success}
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>
#include <ext/pb_ds/tag_and_trait.hpp>
using namespace __gnu_pbds;

typedef trie<string, null_type, trie_string_access_traits<>, pat_trie_tag,
       trie_prefix_search_node_update> prefix_trie;

// Usage example for Patricia trie
prefix_trie t;                              // Create an empty prefix trie
t.insert("Banana");                         // Insert an element
auto match_range = t.prefix_range("Ban");   // Get all strings matching "Ban*"

for (auto it = match_range.first; it != match_range.second; ++it) cout << *it << ' ';

// GNU Policy-Based Data Structures ----------------------------------------------------
// rope:: An Implicit Cartesian Tree; a data structure that allows for
// fast [O(log(n)] insertion and deletion of arbitrarily long blocks of data.
// Uses most of the same syntax as vector. See examples.
#include <ext/rope>
using namespace __gnu_cxx;

// Usage example for rope
rope<int> v;                                // create an empty rope.
for (int i=0; i<n; ++i) v.push_back(i);     // insert into rope
rope<int> cur = v.substr(pos, length);      // get substring from [pos, pos+length)
v.erase(pos, length);                       // erase substring from [pos, pos+length)
v.insert(v.mutable_begin(), cur);           // use mutable_begin for non-const iterator
for (const auto& x : v) cout << x << ' ';   // iterate over the contents of the rope

// GNU Policy-Based Data Structures ----------------------------------------------------
// ordered_set:: A red-black tree maintaining node order-statistics, allowing for
// fast [O(log(n)] order-statistics queries. Uses the same syntax as std::set.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;

// Usage example for ordered_set
ordered_set s;                          // Create an empty ordered set
for (int i=0; i<n; i++) s.insert(i);    // Insert into the set
cout << *s.find_by_order(3) << endl;    // Find the 3rd element
cout << s.order_of_key(5) << endl;      // Find the order-statistic of 5
```

9 String Processing

```cpp
// Compute the prefix array for the pattern pat. The prefix array is for
// each index i, the length of the longest proper suffix of pat[0...i] that
// is also a proper prefix of pat[0...i]. Complexity: O(m)
template<typename T> vi prefix(const T& pat) {
  int m = (int)pat.size();  vi pre(m, 0);
  for (int j=0, i=1; i<m; ) {
    if (pat[i] == pat[j]) pre[i++] = ++j;
    else if (j>0) j = pre[j-1];
    else i++;
  }
  return pre;
}

// Knuth-Morris-Pratt pattern matching. Complexity: O(n)
// Find all occurrences of the pattern pat in the string str using
// the prefix array pre computed by prefix(pat).
template<typename T> vi find_pattern(const T& str, const T& pat, const vi& pre) {
  int n = (int)str.size(), m = (int)pat.size();  vi res;
  for (int i=0, j=0; i<n; i++) {
    while (j > 0 && str[i] != pat[j]) j = pre[j-1];
    if (str[i] == pat[j]) j++;
    if (j == m) res.push_back(i - m + 1), j = pre[j-1];
  }
  return res;
}

// The z-array of the sequence s is for each index i, the length
// of the longest substring beginning at i that is also a prefix
// of s. Complexity: O(n)
template<typename T> vi z_array(const T& s) {
  int n = (int)s.size(), L = 0, R = 0;  vi z(n, n - 1);
  for (int i = 1, j; i < n; i++) {
    j = max(min(z[i-L],R-i),0);
    for (; i + j < n && s[i+j] == s[j]; j++);
    z[i] = j;
    if (i + z[i] > R) R = i + z[i], L = i;
  }
  return z;
}

// Find all occurrences of pat in str using the z-algorithm in O(n + m)
template<typename T> vi find_pattern(const T& str, T pat) {
  int n = (int)str.size(), m = (int)pat.size();
  pat.insert(pat.end(), str.begin(), str.end());
  vi z = z_array(pat), res;
  for (int i = 0; i < n; i++) if (z[i + m] >= m) res.push_back(i);
  return res;
}

// Linear time online suffix tree. Complexity: O(n) to build.
// Each node in the tree is indexed by an integer, starting from 0 as the root.
// to[u][c] is the node pointed to by node u along an edge beginning with char c.
// len[u] is the length of the parent edge of u (NOTE: len[u] may be greater than the
// length of the string if u is a leaf, ie. true length is min(len[u],n-fpos[u]))
// fpos[u] is an index of s containing the substring on the parent edge of u.
struct SuffixTree {
  const int INF = 1e9;  int node = 0, pos = 0, cap, sz = 1, n = 0;
  string s;  vi len, fpos, link;  vector<map<int, int>> to;
  int make_node(int _pos, int _len) { fpos[sz] = _pos, len [sz] = _len; return sz++; }
  void add_letter(int c) {
    int last = 0;  s += c; n++; pos++;
    while(pos > 0) {
      while(pos > len[to[node][s[n - pos]]]) node=to[node][s[n-pos]], pos-=len[node];
      int edge = s[n - pos], &v = to[node][edge], t = s[fpos[v] + pos - 1];
      if (v == 0) v = make_node(n - pos, INF), link[last] = node, last = 0;
      else if (t == c) { link[last] = node; return; }
      else {
        int u = make_node(fpos[v], pos - 1);
        to[u][c] = make_node(n - 1, INF), to[u][t] = v;
        fpos[v] += pos - 1, len [v] -= pos - 1;
        v = u, link[last] = u, last = u;
      }
      if(node == 0) pos--;
      else node = link[node];
    }
  }
  SuffixTree(const string& S) : cap(2*S.size()), len(cap), fpos(cap), link(cap),
    to(cap) { len[0] = INF; s.reserve(S.size()); for (char c : S) add_letter(c); }
  SuffixTree(int N) : cap(2*N), len(cap), fpos(cap), // Create an empty suffix tree with
    link(cap), to(cap) { len[0] = INF; s.reserve(N); }      // capacity for N characters
  // Find the longest substring of the given pattern beginning at idx that matches a
```

```
  // substring in the tree. Returns {position, length} of the match. Complexity: O(m)
  pii longest_match(const string& pat, int idx) {
    int node = 0, jump = 0, ans = 0, m = (int)pat.size();
    if (to[node][pat[idx]] == 0) return {-1, 0};
    while (to[node][pat[idx]] > 0) {
      jump = 0;  node = to[node][pat[idx]];
      for (int i = fpos[node]; i < n && idx + jump < m
        && jump < len[node] && pat[idx + jump] == s[i]; i++, jump++, ans++);
      if (jump < len[node]) break;
      idx += jump;
    }
    return {fpos[node] + jump - ans, ans};
  }
};

// Linear time online suffix automaton. node stores the states of the automaton.
// node[1] is the root. tail is the value of run(S).  Complexity: O(n) to build.
// State: par -- parent suffix link (edges of the suffix tree of the reverse of S)
//        pos -- length of prefix of S such that run(S[0..pos)) = node
//        edge[x] -- index of node following edge with character x (0 if none)
// Terminal states are all suffix link ancestors of tail (including tail).
// Useful facts: Each node corresponds to an equivalence class of strings w such
// that run(w) = node. Every string in this equivalence class is a suffix of the
// longest string W in the equivalence class. The suffix link leads to the state
// corresponding to the equivalence class of the longest suffix of W that is not
// in the same equivalence class.
struct SuffixAutomaton{
  struct State{
    int par, pos; map<char,int> edge;
    State (int v) : par(0), pos(v) { }
  };
  vector<State> node; int root, tail;
  SuffixAutomaton(const string& S) : root(1), tail(1) { // Create an automaton from S
    node.assign(2, State(0));  for (char c : S) extend(c);
  }
  void extend(char w) {  // Add a character to the string and extend the automaton
    int p = tail, np = node.size(); node.emplace_back(node[p].pos+1);
    for (; p && node[p].edge[w]==0; p=node[p].par)  node[p].edge[w] = np;
    if (p == 0) node[np].par = root;
    else {
      if (node[node[p].edge[w]].pos == node[p].pos+1) node[np].par = node[p].edge[w];
      else {
        int q = node[p].edge[w], r = node.size(); node.push_back(node[q]);
        node[r].pos = node[p].pos+1, node[q].par = node[np].par = r;
        for (; p && node[p].edge[w] == q; p=node[p].par) node[p].edge[w] = r;
      }
    }
    tail = np;
  }
  int run(const string& pat) {  // Return the node reached by running the machine
    int n = root;               // on the input pat, or 0 if pat is not a substring
    for (char c : pat) if ((n = node[n].edge[c]) == 0) return 0;
    return n;
  }
};

// Suffix array construction with LCP. The suffix array is built into sarray and the
// LCP into lcp. NOTE: sarray does not include the empty suffix. lcp[i] is the longest
// common prefix between the strings at sarray[i-1] and sarray[i], lcp[0] = 0.
// Complexity: O(N) or O(N log(N)) for suffix array. O(N) for LCP.
struct suffix_array {
  int n; string str; vi sarray, lcp;
  void bucket(vi& a, vi& b, vi& r, int n, int K, int off=0) {
    vi c(K+1, 0);
    for (int i=0; i<n; i++) c[r[a[i]+off]]++;
    for (int i=0, sum=0; i<=K; i++) { int t = c[i]; c[i] = sum; sum += t; }
    for (int i=0; i<n; i++) b[c[r[a[i]+off]]++] = a[i];
  }
  // Create the suffix array and LCP array of the string s. (LCP is optional)
  suffix_array(string s) : n(s.size()), str(move(s)) { build_sarray(); build_lcp(); }
  // ------------------ OPTION 1: Linear time suffix array ----------------------------
  #define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
  typedef tuple<int,int,int> tiii;
  void sarray_int(vi &s, vi &SA, int n, int K) {
    int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2, name=0, c0=-1, c1=-1, c2=-1;
    vi s12(n02 + 3, 0), SA12(n02 + 3, 0), s0(n0), SA0(n0);
    for (int i=0, j=0; i < n+(n0-n1); i++) if (i%3 != 0) s12[j++] = i;
    bucket(s12, SA12, s, n02, K, 2), bucket(SA12, s12, s, n02, K, 1);
    bucket(s12, SA12, s, n02, K, 0);
    for (int i = 0; i < n02; i++) {
      if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2)
        name++, c0 = s[SA12[i]], c1 = s[SA12[i]+1], c2 = s[SA12[i]+2];
      if (SA12[i] % 3 == 1) s12[SA12[i]/3] = name;
      else s12[SA12[i]/3 + n0] = name;
    }
    if (name < n02) {
      sarray_int(s12, SA12, n02, name);
      for (int i = 0; i < n02; i++) s12[SA12[i]] = i + 1;
    } else for (int i = 0; i < n02; i++) SA12[s12[i] - 1] = i;
    for (int i=0, j=0; i < n02; i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
    bucket(s0, SA0, s, n0, K);
    for (int p=0, t=n0-n1, k=0; k < n; k++) {
      int i = GetI(), j = SA0[p];
      if (SA12[t] < n0 ?
          (pii(s[i], s12[SA12[t] + n0]) < pii(s[j], s12[j/3])) :
          (tiii(s[i],s[i+1],s12[SA12[t]-n0+1]) < tiii(s[j],s[j+1],s12[j/3+n0]))) {
        SA[k] = i; t++;
        if (t == n02) for (k++; p < n0; p++, k++) SA[k] = SA0[p];
      } else {
        SA[k] = j; p++;
        if (p == n0) for (k++; t < n02; t++, k++) SA[k] = GetI();
      }
    }
  }
  void build_sarray() {
    if (n <= 1) { sarray.assign(n, 0); return; }
    vi s(n+3, 0); sarray.assign(n+3, 0);
    for (int i=0; i<n; i++) s[i] = (int)str[i] - CHAR_MIN + 1;
    sarray_int(s, sarray, n, 256), sarray.resize(n);
  }
  // ------------------ OPTION 2: O(N log(N)) time suffix array -----------------------
  void build_sarray() {
    sarray.assign(n, 0); vi r(2*n, 0), sa(2*n), tmp(2*n); if (n <= 1) return;
    for (int i=0; i<n; i++) r[i] = (int)str[i] - CHAR_MIN + 1, sa[i] = i;
    for (int k=1; k<n; k *= 2) {
      bucket(sa,tmp,r,n,max(n,256),k), bucket(tmp,sa,r,n,max(n,256),0);
      tmp[sa[0]] = 1;
      for (int i=1; i<n; i++) {
        tmp[sa[i]] = tmp[sa[i-1]];
        if ((r[sa[i]] != r[sa[i-1]]) || (r[sa[i]+k] != r[sa[i-1]+k])) tmp[sa[i]]++;
      }
      copy(tmp.begin(), tmp.begin()+n, r.begin());
    }
    copy(sa.begin(), sa.begin()+n, sarray.begin());
  }
  // -------------------- OPTIONAL: If you need LCP array ----------------------------
  void build_lcp() {
    int h = 0; vi rank(n); lcp.assign(n, 0);
    for (int i = 0; i < n; i++) rank[sarray[i]] = i;
    for (int i = 0; i < n; i++) {
      if (rank[i] > 0) {
        int j = sarray[rank[i]-1];
        while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
        lcp[rank[i]] = h;
```

```
      }
      if (h > 0) h--;
    }
  }
  // OPTIONAL: Pattern matching -- Find all occurrences of pat[j..] in O(m log(n))
  // Returns an iterator pair of the matching locations in the suffix array
  struct Comp {
    const string& s; int m, j;
    Comp(const string& str,int m, int j) : s(str), m(m), j(j) { }
    bool operator()(int i, const string& p) const { return s.compare(i,m,p,j,m) < 0; }
    bool operator()(const string& p, int i) const { return s.compare(i,m,p,j,m) > 0; }
  };
  auto find(const string& pat, int j=0) {
    return equal_range(sarray.begin(), sarray.end(), pat, Comp(str,pat.size(),j));
  }
};

// Aho-Corasick dictionary matching automaton. Add dictionary words with add_key(key)
// then build_links(). To report every match in the text, report the contents of
// node[u].output for all suffix link ancestors u of v, for each node v on the path
// taken through the automaton by the text.
// Complexity: to build - O(M), to count matches - O(N), to report all matches - O(kN)
// k = no. of keys, M = total key length, N = text length.
struct AhoCorasick {
  struct State { map<char,int> edge; int link, cnt, tot; vi output; };
  int n, k; vector<State> node; vi len;
  int make_node() { node.emplace_back(); return n++; }
  void add_key(const string& y) {  // Add key y to the dictionary
    int v = 0;
    for (char c : y) {
      if(!node[v].edge[c]) node[v].edge[c] = make_node();
      v = node[v].edge[c];
    }
     node[v].cnt++, node[v].output.push_back(k++), len.push_back((int)y.size());
  }
  void build_links() {  // Call this once all keys have been inserted
    node[0].link = -1, node[0].tot = 0;  queue<int> q; q.push(0);
    while (!q.empty()) {
      int v = q.front(); q.pop(); node[v].tot = node[v].cnt;
      if (node[v].link != -1) node[v].tot += node[node[v].link].tot;
      for (auto it: node[v].edge) {
        int c = it.first, u = it.second, j = node[v].link;
        while (j != -1 && !node[j].edge[c]) j = node[j].link;
        if (j != -1) node[u].link=node[j].edge[c];
        q.push(u);
      }
    }
  }  // Create an empty Aho-Corasick automaton
  AhoCorasick() : n(1), k(0), node(1) { }
  ll count_matches(const string& x) {  // Count the number of substrings of the given
    ll ans = 0;  int v = 0;                // text that match a key: Complexity: O(N)
    for (int i=0; i<(int)x.size(); i++) {
      while (v && !node[v].edge[x[i]]) v = node[v].link;
      v = node[v].edge[x[i]]; ans += node[v].tot;
    }
    return ans;
  }
};
```

10 Miscellaneous

```
// Date manipulation -- Conversion from Gregorian dates to Julian days. The Julian
// day is the number of days since November 24th 4714 BC. Note that there is no year
// zero in the AD calendar, so 4714 BC corresponds to year -4713. Gregorian dates
// are expressed as {year,month,day}.

// Determine the day of the week for the given Julian date. 0 = Monday ... 6 = Sunday
int day_of_week(int jd) { return jd % 7; }

// Converts the given Gregorian date into the corresponding Julian day
int to_julian(int y, int m, int d) {
    return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
        367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
        3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 + d - 32075;
}

// Converts the given Julian day into the corresponding Gregorian date
tuple<int,int,int> to_gregorian(int jd) {
    int x, n, i, j, y, m, d;
    x = jd + 68569, n = 4 * x / 146097, x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001, x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447, d = x - 2447 * j / 80, x = j / 11;
    m = j + 2 - 12 * x, y = 100 * (n - 49) + i + x;
    return make_tuple(y,m,d);
}

// Returns true if the given year is a leap year in the Gregorian calendar
bool leap_year(int y) { return (y % 400 == 0 || (y % 4 == 0 && y % 100 != 0)); }

// Returns the number of days in the given month/year in the Gregorian calendar.
int days_in(int y, int m) { return m == 2 ? 28 + leap_year(y) : 31 - (m-1) % 7 % 2; }

// 2-SAT solver. Include SCC code from graph algorithms. VAR(x) is variable x,
// NOT(VAR(x)) is the negation of variable x. Complexity: O(n + m)
int VAR(int x) { return 2*x; }
int NOT(int x) { return x^1; }

struct TwoSAT {
  int n;  SCC scc;
  // Create a 2-SAT equation with n variables
  TwoSAT(int n) : n(n), scc(2 * n) { }
  void add_or(int u, int v) {
    if (u == NOT(v)) return;
    scc.add_edge(NOT(u), v);  scc.add_edge(NOT(v), u);
  }
  void add_true(int u){ add_or(u, u); }
  void add_false(int u) { add_or(NOT(u), NOT(u)); }
  void add_xor(int u, int v) { add_or(u, v); add_or(NOT(u), NOT(v)); }
  pair<bool, vector<bool>> solve() {
    vi comp = scc.find_SCC().Y;  vector<bool> val(n);
    for (int i = 0; i < 2 * n; i += 2){
      if (comp[i] == comp[i + 1]) return {false, val};
      val[i/2] = (comp[i] > comp[i + 1]);
    }
    return {true, val};
  }
};

// Cubic equation solver. Solves ax^3 + bx^2 + cx + d = 0.
// a must be non-zero, does NOT work well when a is NEAR 0.
const double EPSILON = 1e-8, PI = acos(-1);

template<typename T> vector<T> cubic(T a, T b, T c, T d) {
  b /= a, c /= a, d /= a;  // Make sure T is non-integral (double or long double)!
  T q = (b*b - 3*c)/9, r = (2*b*b*b - 9*b*c + 27*d) / 54, z = r*r - q*q*q;
  if (z <= EPSILON) {
    vector<T> sol;  T theta = acos(r/pow(q,1.5));
    for (int i=0; i<3; i++) sol.push_back(-2*sqrt(q)*cos((theta+i*2*PI)/3) - b/3);
    return sol;
  }
  T s = cbrt(sqrt(z)+abs(r)); s = (s + q/s) * (r < 0 ? 1 : -1) - b/3;
  return {s};
}
```

```
// Custom type hashing example. Your type must implement equality. The hash
// function must be consistent with ==, that is (a==b) => (hash(a)==hash(b))
struct MyType {
  int a; string b;
  bool operator==(const MyType& r) const { return a == r.a && b == r.b; }
};
namespace std {
  template<> struct hash<MyType> {
    size_t operator()(const MyType& x) const {
      return hash<int>()(x.a) ^ hash<string>()(x.b);
    }
  };
}

unordered_map<MyType,string> my_map;

// Arabic / Roman numeral conversion for 0 < x < 4000. Just be greedy from high to low.
const string R[13] = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
const int A[13] = {1000,900,500,400,100,90,50,40,10,9,5,4,1};

// Josephus Problem (0-based): k=2 special case. Complexity: O(1)
ll survivor(ll n) { return (n - (1LL << (63 - __builtin_clzll(n)))) * 2; }

// Josephus Problem (0-based): Determine the survivor. Complexity: O(n)
int survivor(int n, int k) {
  vi A(n+1);  // A[i] is the survivor with i people, killing every k'th
  for (int i=2; i<=n; i++) A[i] = (A[i-1]+(k%i))%i;
  return A[n];  // OPTIONAL: Return entire array if multiple values needed
}

// Fast convolution using Fast Fourier Transform. Complexity: O(n log(n))
typedef complex<double> comp;
const double PI=acos(-1.0);

void fft(vector<comp> &a, int invert=0) {    // Compute the FFT of the polynomial
  int n=a.size(), i, j, len; comp w, u, v;   // whose coefficients are given by
  for(i=1, j=0;i<n;i++) {                    // the elements of a.
    int bit = n/2; for(; j >= bit; bit /= 2) j-=bit;
    j += bit; if(i < j) swap(a[i], a[j]);
  }
  for(len=2;len<=n;len<<=1) {
    double ang=2*PI/len*(invert?-1:1); comp wlen = polar(1.0, ang);
    for(i=0; i<n; i+=len) for(j=0, w=1; j < len/2; j++)
      u=a[i+j], v=a[i+j+len/2]*w, a[i+j]=u+v, a[i+j+len/2]=u-v, w*=wlen;
  }
  if(invert) for(i=0;i<n;i++) a[i]/=n;
}

// Compute the convolution a * b
template<typename T> vector<T> multiply(const vector<T>& a, const vector<T>& b) {
  int i, n;  vector<comp> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  for(n=1;n<2*(int)max(a.size(), b.size());n*=2);
  fa.resize(n), fb.resize(n), fft(fa), fft(fb);
  for(i=0;i<n;i++) fa[i]*=fb[i];
  fft(fa, 1); vector<T> res(n);   // Remove rounding below if T is non-integral
  for(i=0;i<n;i++) res[i]=(T)(fa[i].real()+0.5);
  return res;
}

// Numerical integration. Integrate f(x) for x in [a,b]. n is the number
// of intervals (it must be even). If K is an upper bound on the 4th derivative
// of f for all x in [a,b], then the error is bounded by (K h^5) / (180 n^4)
template<typename F> double integrate(F f, double a, double b, int n) {
  double ans = f(a) + f(b), h = (b-a)/n;
  for (int i=1; i<n; i++) ans += f(a+i*h) * (i%2 ? 4 : 2);
  return ans * h / 3;
}
```

```
// Numerical differentiation. h is the step size. Error is O(h^4)
template<typename F> double differentiate(F f, double x, double h) {
  return (-f(x+2*h) + 8*(f(x+h) - f(x-h)) + f(x-2*h)) / (12*h);
}

// Simplex algorithm for solving linear programs of the form
//      maximize     c^T x
//      subject to   Ax <= b
//                   x >= 0
// solve() returns INF if unbounded, NaN if infeasible.
// T must be a floating-point type. Complexity is unbounded in general.
const double EPS = 1e-9;

template<typename T> struct LPSolver {
  const T INF = numeric_limits<T>::infinity(), NaN = numeric_limits<T>::quiet_NaN();
  int m, n;  vi N, B;  vector<vector<T>> D;
  void pivot(int r, int s) {
    T inv = 1.0 / D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r)
      for (int j = 0; j < n + 2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] * inv;
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;  swap(B[r], N[s]);
  }
  bool simplex(int phase) {
    int x = phase == 1 ? m + 1 : m, s = -1, r = -1;
    for (; ; s=-1,r=-1) {
      for (int j = 0; j <= n; j++) if (!(phase == 2 && N[j] == -1))
        if (s == -1 || D[x][j] < D[x][s] || (D[x][j] == D[x][s] && N[j] < N[s])) s = j;
      if (D[x][s] > -EPS) return true;
      for (int i = 0; i < m; i++) if (!(D[i][s] < EPS))
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
          ((D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r])) r = i;
      if (r == -1) return false;
      pivot(r, s);
    }
  }
  // Create a solver for max(c^T x) st. Ax <= b, x >= 0.
  LPSolver(const vector<vector<T>>& A, const vector<T>& b, const vector<T>& c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, vector<T>(n + 2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m + 1][n] = 1;
  }
  pair<T, vector<T>> solve() {  // Returns {objective_value, optimial_solution}
    int r = 0; vector<T> x = vector<T>(n);
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
      pivot(r, n);
      if (!simplex(1) || D[m + 1][n + 1] < -EPS) return {NaN, x};
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++) if (s == -1 || D[i][j] < D[i][s]
          || (D[i][j] == D[i][s] && N[j] < N[s])) s = j;
        pivot(i, s);
      }
    }
    if (!simplex(2)) return {INF, x};
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return {D[m][n + 1], x};
  }
};
```

11 Formulas and Theorems

Arithmetic series and powers: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

Geometric and arithmetic-geometric series: $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}$, $\sum_{i=0}^{n} ic^i = \frac{nc^{n+2}-(n+1)c^{n+1}+c}{(c-1)^2}$

Binomial sums: $\sum_{k=0}^{n} \binom{r+k}{k} = \binom{r+n+1}{n}$, $\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$, $\sum_{k=0}^{n} \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$,

Binomial identities: $\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$, $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Catalan numbers: Dyck words of length $2n$. $C_n = \frac{1}{n+1}\binom{2n}{n}$, $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$, $C_0 = 1$

Derangements: Permutations without fixed points. $!0 = !1 = 0$, $!n = (n-1)(!(n-1) + !(n-2))$

Stirling numbers of the first kind: The number of permutations on n elements with k cycles.

$$\begin{bmatrix}0\\0\end{bmatrix} = 1, \quad \begin{bmatrix}n\\0\end{bmatrix} = \begin{bmatrix}0\\n\end{bmatrix} = 0, \quad \begin{bmatrix}n\\1\end{bmatrix} = (n-1)!, \quad \begin{bmatrix}n\\n\end{bmatrix} = 1, \quad \begin{bmatrix}n\\k\end{bmatrix} = (n-1)\begin{bmatrix}n-1\\k\end{bmatrix} + \begin{bmatrix}n-1\\k-1\end{bmatrix}$$

Stirling numbers of second kind: The number of partitions of n elements into k (non-empty) subsets.

$$\left\{{0 \atop 0}\right\} = 1, \quad \left\{{n \atop 0}\right\} = \left\{{0 \atop n}\right\} = 0, \quad \left\{{n \atop k}\right\} = k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}, \quad \sum_{k=0}^{n} \left\{{n \atop k}\right\} = B_n$$

Bell numbers: The number of set partitions of n elements. $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$, $B_0 = 1$

1st order Eulerian numbers: The number of permutations on n elements with k ascents.

$$\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1, \left\langle {n \atop k} \right\rangle = \left\langle {n \atop n-1-k} \right\rangle, \left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle, \left\langle {n \atop m} \right\rangle = \sum_{k=0}^{m} \binom{n+1}{k}(m+1-k)^n(-1)^k,$$

2nd order: Permutations on $\{1, 1, ...n, n\}$ with $a_j > a_i, a_k$ if $i < j < k$ and $a_i = a_k$ with m ascents.

$$\left\langle\!\left\langle {n \atop 0} \right\rangle\!\right\rangle = 1, \quad \left\langle\!\left\langle {n \atop n} \right\rangle\!\right\rangle = 0 \text{ for } n \neq 0, \quad \left\langle\!\left\langle {n \atop m} \right\rangle\!\right\rangle = (m+1)\left\langle\!\left\langle {n-1 \atop m} \right\rangle\!\right\rangle + (2n-1-m)\left\langle\!\left\langle {n-1 \atop m-1} \right\rangle\!\right\rangle$$

Integer partitions: $P(x) = \prod_{k=1}^{\infty} \left(\frac{1}{1-x^k}\right)$, $p(n) = \sum_{k \geq 1} (-1)^{k-1} \left(p\left(n - \frac{k(3k+1)}{2}\right) + p\left(n - \frac{k(3k-1)}{2}\right)\right)$

Restricted partitions: $p(n,k) = p(n-k,k) + p(n-1,k-1)$, $p(0,0) = 1$, $p(n,k) = 0, n \leq 0$ or $k \leq 0$

Balls in bins: The number of ways to place n balls into k bins.

		Identical balls	**Distinguishable balls**
Identical bins	Empty bins ok	$\sum_{i=1}^{k} p(n,i)$	$\sum_{i=1}^{k} \left\{{n \atop i}\right\}$
	No empty bins	$p(n,k)$	$\left\{{n \atop k}\right\}$
Distinguishable bins	Empty bins ok	$\binom{n+k-1}{k-1} = \binom{n+k-1}{n}$	k^n
	No empty bins	$\binom{n-1}{k-1}$	$\sum_{i=0}^{k} (-1)^i \binom{k}{i}(k-i)^n = \left\{{n \atop k}\right\} k!$

Trigonometry: Sin rule: $\frac{\sin(\alpha)}{a} = \frac{\sin(\beta)}{b} = \frac{\sin(\gamma)}{c}$, Cosine rule: $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$

Circle inscribed in triangle: radius $= \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$, centre $= \frac{a\vec{v}_a + b\vec{v}_b + c\vec{v}_c}{a+b+c}$, $s = \frac{a+b+c}{2}$

Circumcircle: radius $= \frac{abc}{4A}$, $A =$ area of triangle, centre $=$ intersection of perpendicular bisectors

Trig Identities: $\sin^2(u) = \frac{1}{2}(1 - \cos(2u))$, $\cos^2(u) = \frac{1}{2}(1 + \cos(2u))$

$\sin(u) + \sin(v) = 2\sin(\frac{u+v}{2})\cos(\frac{u-v}{2})$, $\sin(u) - \sin(v) = 2\sin(\frac{u-v}{2})\cos(\frac{u-v}{2})$

$\cos(u) + \cos(v) = 2\cos(\frac{u+v}{2})\cos(\frac{u-v}{2})$, $\cos(u) - \cos(v) = -2\sin(\frac{u+v}{2})\sin(\frac{u-v}{2})$

$\sin(u)\sin(v) = \frac{1}{2}(\cos(u-v) - \cos(u+v))$, $\cos(u)\cos(v) = \frac{1}{2}(\cos(u-v) + \cos(u+v))$

Dot and cross product: $\vec{u} \cdot \vec{v} = \|\vec{u}\|\|\vec{v}\|\cos(\theta)$, $\vec{u} \times \vec{v} = \|\vec{u}\|\|\vec{v}\|\sin(\theta)\mathbf{n}$

Rotation matrix: $\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$ (counter-clockwise by θ)

Number and sum of divisors: multiplicative, $\tau(p^k) = k+1$, $\sigma(p^k) = \frac{p^{k+1}-1}{p-1}$

Linear Diophantine equations: $a \cdot s + b \cdot t = c$ iff $\gcd(a,b)|c$, $(s,t) = (s_0, t_0) + k \cdot \left(\frac{a}{\gcd(a,b)}, -\frac{a}{\gcd(a,b)}\right)$

Euler's Theorem: If a and b are relatively prime, $a^{\phi(b)} \equiv 1 \mod b$, $a^{p-1} \equiv 1 \mod p$ for prime p

Wilson's Theorem: p is a prime iff $(p-1)! \equiv -1 \mod p$

Lucas' Theorem: $\binom{n}{m} = \prod_{i=0}^{k} \binom{m_i}{n_i} \mod p$ where m_i, n_i are the base p coefficients of m and n

---

Pick's Theorem: $A = i + \frac{b}{2} - 1$, $A =$ area, $i =$ interior lattice points, $b =$ boundary lattice points.

Euler's Formula: $V - E + F - C = 1$, $V =$ vertices, $E =$ edges, $F =$ faces, $C =$ connected components.

Cayley's Formula: A complete graph on n labelled vertices has n^{n-2} spanning trees.

Erdös Gallai: $\{d_n\}$ is a degree sequence iff $\sum_{i=1}^{k} d_i$ is even and $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k), \forall k$

Moser's Circle: A circle is divided into $\binom{n}{4} + \binom{n}{2} + 1$ pieces by chords connecting n points

Burnside's Lemma: $|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$

Möbius Inversion Formula: If $g(n) = \sum_{d|n} f(d)$ then $f(n) = \sum_{d|n} \mu(d) g\left(\frac{n}{d}\right)$

Möbius Function: $\mu(n) = \begin{cases} 1 & n \text{ is square-free with an even number of prime factors} \\ -1 & n \text{ is square-free with an odd number of prime factors} \\ 0 & n \text{ has a squared prime factor} \end{cases}$

Usable Chooses: $\binom{n}{k}$ is safe assuming $50,000,000$ is not TLE. $\binom{28}{k}$ is okay for all $k \leq n$.

n	29	30 − 31	32 − 33	34 − 38	39 − 45	46 − 59	60 − 92	93 − 187	188 − 670
k	11	10	9	8	7	6	5	4	3

Combinatorial bounds: $B_{13} = 27,644,437$, $C_{15} = 9,694,845$, $p(80) = 15,796,476$

Some primes for modding: $10^9 + 103$, $10^9 + 321$, $10^9 + 447$, $10^9 + 637$, $10^9 + 891$

Konig's theorem: On a bipartite graph:

1. The size of the minimum vertex cover is equal to the size of the maximum matching
2. The size of the minimum edge cover plus the size of maximum matching equals the number of vertices
3. The size of the maximum independent set equals the size of the minimum edge cover

Spanning Trees in Complete Bipartite Graphs: $K_{n,m}$ has $m^{n-1} \times n^{m-1}$ spanning trees