

Table of Content

Finding articulation points, bridges and biconnected components.....	2
Maximum bipartite matching with augmenting paths.....	2
CohenSutherland algorithm.....	2
Job Scheduling.....	4
Optimal Binary Search Tree.....	5
Convex Hull.....	5
Ford-Fulkerson with BFS.....	6
DioPhantine Equation and Euclid's extended algorithm.....	8
Determining if a point lies on the interior of a polygon.....	10
Solution 2 (2D).....	10
Solution 4 (3D).....	11
Area.....	12
Centroid.....	12
Centroid of a 3D shell described by 3 vertex facets.....	12
Strongly Connected Component.....	12
Disjoint set operation.....	12
Graham Scan(Q).....	13
Extended-Euclid(a,b).....	13
Euler's Phi function.....	13
Modular Linear Equation Solver(a,b,n).....	13
Modular-Exponentiation(a,b,n).....	13
Knuth-Morris-Pratt Algorithm.....	14
Cyrus-Beck line clipping Algorithm.....	14
JAVA Selected material.....	14
Taking Input.....	14
Alternate way of taking input using StreamTokenizer.....	15
BigInteger (Function signatures).....	15
StringTokenizer.....	16
Integer.....	16
Finding symmetry line for a non self-intersecting polygon (convex/concave).....	16
Trigonometric functions.....	17
Vector.....	17
Translation, Scaling & Rotation.....	18
Numerical Methods.....	18
Descartes' rule of sign.....	18
Derivative & factor of a polynomial.....	18
Lagrange's Interpolation Formula.....	18
Assignment Problem.....	18
Map.....	20
Stable Matching.....	21
Eular cycles.....	21
Fleury's Algorithm- constructing Eulerian trails.....	21
Directed Eulerian circuit.....	21

**Finding articulation points, bridges and biconnected components:**

```

blockDFS (v)
    pred(v) = num(v) = i++;
    for all vertices u adjacent to v
        if edge(uv) is not on stack
            push(edge(uv));
        if num(u) is 0
            blockDFS(u);
            if pred(u) ≥ num(v)
                e = pop();
                while e != edge(vu)
                    output e;
                e = pop();
                output e;
            else pred(v) = min(pred(v), pred(u));
        else if u is not the parent of v
            pred(v) = min(pred(v), num(u));

```

**Maximum bipartite matching with augmenting paths:**

```

findMaximumMatching(bipartite graph)
    for all unmatched vertices v
        set level of all vertices to 0;
        set parent of all vertices to null;
        level(v) = 1;
        last = null;
        clear queue;
        enqueue(v);
        while queue is not empty and last is null
            v = dequeue();
            if level(v) is an odd number
                for all vertices u adjacent to v such that level(u) is 0
                    if u is unmatched
                        parent(u) = v;
                        last = u;
                        break;
                    else if u is matched but not with v
                        parent(u) = v;
                        level(u) = level(v) + 1;
                        enqueue(u);
            else
                enqueue(vertex u matched with v);
                parent(u) = v;
                level(u) = level(v) + 1;
        if last is not null
            for u = last ; u is not null ; u = parent(parent(u))
                matchedWith(u) = parent(u);
                matchedWith(parent(u)) = u;

```

**Clipping a subject polygon with a clipper polygon - CohenSutherland algorithm.**  
**Polygons must be convex:**

```

# include <stdio.h>
# include <math.h>

# define SIZE 2001

typedef struct{
    double x,y;
}Point;

enum { LEFT=0,RIGHT,BEHIND,BEYOND,ORIGIN,DESTINATION,BETWEEN};
enum { COLLINEAR=0,PARALLEL,SKEW,SKEW_CROSS,SKEW_NO_CROSS};

```

```

/* p2 on which side of p1-p0 */
int classify(Point p0,Point p1,Point p2)
{ Point a,b;
  double t;
  a.x=p1.x-p0.x;
  a.y=p1.y-p0.y;
  b.x=p2.x-p0.x;
  b.y=p2.y-p0.y;
  t=a.x*b.y-a.y*b.x;
  if(t>0.0) return LEFT;
  if(t<0.0) return RIGHT;
  if((a.x*a.x+a.y*a.y)<(b.x*b.x+b.y*b.y)) return BEYOND;
  if((a.x>0.0 && b.x<0.0) || (a.x<0.0 && b.x>0.0)) return BEHIND;
  if((a.y>0.0 && b.y<0.0) || (a.y<0.0 && b.y>0.0)) return BEHIND;
  if(p0.x==p2.x && p0.y==p2.y) return ORIGIN;
  if(p1.x==p2.x && p1.y==p2.y) return DESTINATION;
  return BETWEEN;
}
double dotProduct(Point a,Point b)
{ return a.x*b.x+a.y*b.y;
}
/* intersection of b-a and d-c : t of b-a */
int intersect(Point a,Point b,Point c,Point d,double *t)
{ double denom,num;
  int aclass;
  Point n,ba,ac;
  n.x=d.y-c.y;
  n.y=c.x-d.x;
  ba.x=b.x-a.x;
  ba.y=b.y-a.y;
  ac.x=a.x-c.x;
  ac.y=a.y-c.y;
  denom=dotProduct(n,ba);
  if(denom==0.0){
    aclass=classify(c,d,a);
    if(aclass==LEFT || aclass==RIGHT)
      return PARALLEL;
    else
      return COLLINEAR;
  }
  num=dotProduct(n,ac);
  *t=-num/denom;
  return SKEW;
}
/* clip polygon p[] using edge b-a */
int lineClip(Point a,Point b,Point p[],int n)
{ Point r[SIZE];
  Point org,dest,crosspt;
  int orgIsInside,destIsInside;
  double t;
  int i,j;
  p[n]=p[0];
  for(i=j=0;i<n;i++){
    org=p[i];
    dest=p[i+1];
    orgIsInside=(classify(a,b,org)!=LEFT);
    destIsInside=(classify(a,b,dest)!=LEFT);
    if(orgIsInside!=destIsInside){
      intersect(a,b,org,dest,&t);
      crosspt.x=a.x+t*(b.x-a.x);
      crosspt.y=a.y+t*(b.y-a.y);
    }
  }
}

```

```

    if(orgIsInside && destIsInside)
        r[j++]=dest;
    else if(orgIsInside && !destIsInside){
        if(org.x!=crosspt.x || org.y!=crosspt.y)
            r[j++]=crosspt;
    }
    else if(!orgIsInside && !destIsInside)
        ;
    else{
        r[j++]=crosspt;
        if(dest.x!=crosspt.x || dest.y!=crosspt.y)
            r[j++]=dest;
    }
}
for(i=0;i<j;i++)
    p[i]=r[i];
return j;
}
int polygonClip(Point subject[],int m,Point clipper[],int n)
{ int tm;
  int i;
  clipper[n]=clipper[0];
  for(i=0;i<n;i++){
      tm=lineClip(clipper[i],clipper[i+1],subject,m);
      m=tm;
  }
  return m;
}

```

**Job Scheduling of n jobs with deadlines  $d_i$  and penalty/profit  $p_i$ . If  $p_i$  is profit late jobs give 0 profit. if  $p_i$  is penalty only late jobs have penalties. Time complexity =  $O(n \cdot \lg n)$ .**

```

#include <stdio.h>
#include <stdlib.h>

#define MAXJOBS 1000
#define min(a,b) (((a)<(b)) ? (a):(b))

typedef struct{
    int d,p,alpha;
    /* job done in [alpha-1,alpha] */
}Job;
int cmp(const void *a,const void *b)
{ Job u,v;
  u=(Job *)a;
  v=(Job *)b;
  return v.p-u.p;
}
/* Jobs are q[1],q[2],...,q[n]
   If job i is late q[i].alpha==0 */
void Schedule(Job q[],int n)
{ int i,j,k,l;
  qsort(q+1,n,sizeof(Job),cmp);
  MakeSet(0);
  a[0]=0;
  for(i=1;i<=n;i++){
      MakeSet(i);
      a[i]=i;
      q[i].alpha=0;
  }
  q[1].alpha=min(q[1].d,n);
  Union(q[1].alpha-1,q[1].alpha);
}

```

```

j=FindSet(q[l].alpha);
a[j]=a[q[l].alpha-1];
for(i=2;i<=n;i++){
    j=FindSet(min(q[i].d,n));
    if(!a[j]) continue;
    q[i].alpha=a[j];
    k=FindSet(a[j]-1);
    Union(j,k);
    l=FindSet(j);
    a[l]=a[k];
}
}

```

#### Optimal Binary Search Tree with Knuth's Optimization:

```

Optimal-BST(p,q,n)
for i=1 to n+1 do
    e[i,i-1] = q[i-1];
    w[i,i-1] = q[i-1];
for k=1 to n do
    for i=1 to n-k+1 do
        j = i+k-1;
        e[i,j] = ∞;
        w[i,j] = w[i,j-1] + p[j] + q[j];
        for r=root[i,j-1] to root[i+1,j] do
            t = e[i,r-1] + e[r+1,j] + w[i,j];
            if t < e[i,j] then
                e[i,j] = t;
                root[i,j] = r;
    return e and root

```

Here,  $p[i]$  = probability of a query ending in  $i$ -th node,  $1 \leq i \leq n$   
 $q[i]$  = probability of a failed query,  $0 \leq i \leq n$

#### Convex Hull Code

```

long p[MAX][2],h[MAX][2];
void Swap(long &x,long &y)
{ long t;
  t = x, x = y, y = t;
}
long Dis(long i,long j)
{ return ((p[i][0]-p[j][0])*(p[i][0]-p[j][0]) +
          (p[i][1]-p[j][1])*(p[i][1]-p[j][1]));
}
long Area(long i,long j,long k)
{ return( p[i][0]*(p[j][1]-p[k][1]) + p[j][0]*(p[k][1]-p[i][1]) +
          p[k][0]*(p[i][1]-p[j][1]));
}
void SetMinimum(long n)
{ long index,i;
  index = 0 ;
  for(i=1;i<n;i++){
    if( p[i][1] > p[index][1] ) continue;
    else if( p[i][1] < p[index][1] )
      index = i;
    else{
      if( p[i][0] < p[index][0] )
        index = i;
    }
  }
  Swap(p[0][0],p[index][0]);
  Swap(p[0][1],p[index][1]);
}

```

```

}
long ConvexHull(long n)
{ long i,j,count,prev,next,area,dis1,dis2,index;
  SetMinimum(n);
  h[0][0] = p[0][0];
  h[0][1] = p[0][1];
  index = 1;
  prev = 0;
  while(1){
    next = (prev+1) % n;
    for( j=next,count=1 ; count<=n-2; count++ ){
      i = (j+count) % n;
      area = Area(prev,next,i);
      if( area > 0 ) continue;
      else if ( area < 0 )
        next = i;
      else{
        dis1 = Dis(prev,next);
        dis2 = Dis(prev,i);
        if( dis2 > dis1 ) next = i;
      }
    }
    if( next==0) break;
    else{
      h[index][0] = p[next][0];
      h[index][1] = p[next][1];
      index++;
      prev = next ;
    }
  }
  return index;
}
void main()
{ long i,j,k,l,n,num,index,Case;
  scanf("%ld",&Case);
  printf("%ld\n",Case);
  for( num=0; num<Case ; ){
    scanf("%ld",&n );
    if( n==-1) continue;
    for( i=0; i<n ; i++)
      scanf("%ld%ld",&p[i][0],&p[i][1]);
    index = ConvexHull(n-1);
    printf("%ld\n",index+1);
    for(i=0; i<index; i++)
      printf("%ld %ld\n",h[i][0],h[i][1]);
    printf("%ld %ld\n",h[0][0],h[0][1]);
    if(num!=Case-1)
      printf("-1\n");
    num++;
  }
}

```

### Ford-Fulkerson with BFS

```

/*root = 0;
 *problem vertices : 1-n;
 *category vertices: (n+1)-(n+m)
 *sink = np+nk+1
 */
#define SIZE 500
#define QSIZE 300

int f[SIZE][SIZE],c[SIZE][SIZE],path[SIZE][SIZE];

```

```

int visit[SIZE],index[SIZE],q[QSIZE],parent[SIZE],minflow[SIZE];
int t,m,n,serial,front,rear;

int min(int a, int b)
{ return a <= b ? a : b;
}
int dequeue()
{ int temp;
  if(front == rear) return -1;
  temp = q[front];
  front = (front+1) % QSIZE;
  return temp;
}
void enqueue(int x)
{ q[rear] = x;
  rear = (rear+1) % QSIZE;
}
bool bfs()
{ int node,nextnode,prevnode,cf,minf,i;
  front = rear = 0;
  enqueue(0);
  visit[0] = serial;
  parent[0] = -1;
  minflow[0] = INT_MAX;
  while(1){
    node = dequeue();
    if(node == t){
      minf = minflow[t];
      while(1){
        prevnode = parent[node];
        if(prevnode < 0) break;
        f[prevnode][node] += minf;
        f[node][prevnode] = -f[prevnode][node];
        node = prevnode;
      }
      return true;
    }
    if(node == -1) return false;
    for(i = 0; i < index[node]; i++){
      nextnode = path[node][i];
      if(visit[nextnode] != serial){
        cf = c[node][nextnode] - f[node][nextnode];
        if(cf <= 0) continue;
        visit[nextnode] = serial;
        parent[nextnode] = node;
        minflow[nextnode] = min(minflow[node],cf);
        enqueue(nextnode);
      }
    }
  }
}
int main()
{ /*t = supersink*/
  int i,j,k,b,d,m,n,cap,src,tgt,sum;
  serial = 10;
  while(scanf("%d",&n) == 1){
    t = 2*n+1;
    index[0] = index[t] = 0;
    for(i = 1; i <= n; i++){
      index[i] = index[n+i] = 0;
      scanf("%d",&cap);
      f[i][n+i] = 0;
    }
  }
}

```

```

    f[n+i][i] = 0;
    /*n+i denotes the outgoing node for regulator i
       i denotes the incoming node for regulator i*/
    c[i][n+i] = cap;
    c[n+i][i] = 0;
    path[n+i][index[n+i]++] = i;
    path[i][index[i]++] = n+i;
}
scanf("%d",&m);
for(i = 1; i <= m; i++){
    scanf("%d %d %d",&src,&tgt,&cap);
    f[src+n][tgt] = 0;
    f[tgt][src+n] = 0;
    c[src+n][tgt] = cap;
    c[tgt][src+n] = 0;
    path[src+n][index[n+src]++] = tgt;
    path[tgt][index[tgt]++] = src+n;
}
scanf("%d %d",&b,&d);
for(i = 1; i <= b; i++){
    scanf("%d",&tgt);
    f[0][tgt] = 0;
    f[tgt][0] = 0;
    c[0][tgt] = INT_MAX;
    c[tgt][0] = 0;
    path[0][index[0]++] = tgt;
    path[tgt][index[tgt]++] = 0;
}
for(i = 1; i <= d; i++){
    scanf("%d",&src);
    f[src+n][t] = 0;
    f[t][src+n] = 0;
    c[src+n][t] = INT_MAX;
    c[t][src+n] = 0;
    path[src+n][index[n+src]++] = t;
    path[t][index[t]++] = src+n;
}
do{
    serial++;
}while(bfs());
sum = 0;
for(i = 0; i < index[t];i++){
    j = path[t][i];
    sum += f[j][t];
}
printf("%d\n",sum);
}
return 0;
}

```

### DioPhantine Equation and Euclid's extended algorithm

```

typedef long int lint;
void Euclid(lint u, lint v, lint &f,lint &g)
{ lint up[4] = {0, 1, 0, 0}, vp[4] = {0,0,1,0};
  lint q, t[4], i;
  double ft;
  up[3] = u;
  vp[3] = v;
  while(vp[3] != 0){
    ft = (double)up[3] / vp[3];
    ft = floor(ft);
    q = (lint)ft;

```



```

        for(i = 1; i <= 3; i++){
            t[i] = up[i] - vp[i]*q;
            up[i] = vp[i];
            vp[i] = t[i];
        }
    }
    f = up[1];
    g = up[2];
    return;
}
lint gcd(lint a,lint b)
{ lint t;
  if(a<b){
    t=a;
    a=b;
    b=t;
  }
  while(b){
    t=a%b;
    a=b;
    b=t;
  }
  return a;
}
int main()
{ lint n,n1,n2,gd,u,v,temp,d,e,f,minx,miny;
  long double total,min,c1,c2,t,x,y,x0,y0,start,end,start1,end1;
  while(scanf("%ld",&n) == 1 && n){
    scanf("%Lf %ld %Lf %ld",&c1,&n1,&c2,&n2);
    gd = gcd(n1,n2);
    if(n % gd != 0 ){
      printf("failed\n");
      continue;
    }
    else{
      d = n1 / gd;
      e = n2 / gd;
      f = n / gd;
      Euclid(d,e,u,v);
      x0 = u * ((long double)f);
      y0 = v * ((long double)f);
      /*x = x0-e*t, y = y0+d*t*/
      start = ceil((-y0)/((long double)d));
      end = floor((x0)/((long double)e));
      min = INT_MAX;
      if(end < start) printf("failed\n");
      else{
        min = INT_MAX;
        t = start;
        x = x0 - e*t;
        y = y0 + d*t;
        total = x*c1 + y*c2;
        minx = x;
        miny = y;
        min = total;
        t = end;
        x = x0 - e*t;
        y = y0 + d*t;
        total = x*c1 + y*c2;
        if(total < min){
          minx = x;
          miny = y;
        }
      }
    }
  }
}

```

```

        min = total;
    }
    printf("%ld %ld\n",minx, miny);
}
}
}
return 0;
}

```

### Determining if a point lies on the interior of a polygon

```

#define MIN(x,y) (x < y ? x : y)
#define MAX(x,y) (x > y ? x : y)
#define INSIDE 0
#define OUTSIDE 1

typedef struct{
    double x,y;
}Point;

int InsidePolygon(Point *polygon,int N,Point p)
{ int counter = 0;
  int i;
  double xinters;
  Point p1,p2;

  p1 = polygon[0];
  for(i=1;i<=N;i++){
    p2 = polygon[i % N];
    if(p.y > MIN(p1.y,p2.y)){
      if(p.y <= MAX(p1.y,p2.y)){
        if(p.x <= MAX(p1.x,p2.x)){
          if (p1.y != p2.y){
            xinters = (p.y-p1.y)*(p2.x-p1.x)/(p2.y-p1.y)+p1.x;
            if (p1.x == p2.x || p.x <= xinters) counter++;
          }
        }
      }
    }
    p1 = p2;
  }
  if(counter % 2 == 0) return(OUTSIDE);
  else return(INSIDE);
}

```

### Solution 2 (2D)

```

typedef struct{
    int h,v;
} Point;
int InsidePolygon(Point *polygon,int n,Point p)
{ int i;
  double angle=0;
  Point p1,p2;
  for (i=0;i<n;i++){
    p1.h = polygon[i].h - p.h;
    p1.v = polygon[i].v - p.v;
    p2.h = polygon[(i+1)%n].h - p.h;
    p2.v = polygon[(i+1)%n].v - p.v;
    angle += Angle2D(p1.h,p1.v,p2.h,p2.v);
  }
  if (ABS(angle) < PI) return(FALSE);
  else return(TRUE);
}

```

```

/* Return the angle between two vectors on a plane
   The angle is from vector 1 to vector 2, positive anticlockwise
   The result is between -pi -> pi
*/
double Angle2D(double x1, double y1, double x2, double y2)
{ double dtheta,theta1,theta2;
  theta1 = atan2(y1,x1);
  theta2 = atan2(y2,x2);
  dtheta = theta2 - theta1;
  while (dtheta > PI)
    dtheta -= TWOPI;
  while (dtheta < -PI)
    dtheta += TWOPI;
  return(dtheta);
}

```

#### **Solution 4 (3D)**

To determine whether a point is on the interior of a convex polygon in 3D one might be tempted to first determine whether the point is on the plane, then determine it's interior status. Both of these can be accomplished at once by computing the sum of the angles between the test point (q below) and every pair of edge points  $p[i] \rightarrow p[i+1]$ . This sum will only be  $2\pi$  if both the point is on the plane of the polygon AND on the interior. The angle sum will tend to 0 the further away from the polygon point q becomes.

The following code snippet returns the angle sum between the test point q and all the vertex pairs. Note that the angle sum is returned in radians.

```

typedef struct{
  double x,y,z;
} XYZ;
#define EPSILON 0.0000001
#define MODULUS(p) (sqrt(p.x*p.x + p.y*p.y + p.z*p.z))
#define TWOPI 6.283185307179586476925287
#define RTOD 57.2957795

double CalcAngleSum(XYZ q,XYZ *p,int n)
{ int i;
  double m1,m2,anglesum=0,costheta;
  XYZ p1,p2;
  for(i=0;i<n;i++){
    p1.x = p[i].x - q.x;
    p1.y = p[i].y - q.y;
    p1.z = p[i].z - q.z;
    p2.x = p[(i+1)%n].x - q.x;
    p2.y = p[(i+1)%n].y - q.y;
    p2.z = p[(i+1)%n].z - q.z;
    m1 = MODULUS(p1);
    m2 = MODULUS(p2);
    if(m1*m2 <= EPSILON)
      return(TWOPI); /* We are on a node, consider this inside */
    else
      costheta = (p1.x*p2.x + p1.y*p2.y + p1.z*p2.z) / (m1*m2);
      anglesum += acos(costheta);
  }
  return(anglesum);
}

```

#### **Note**

For most of the algorithms above there is a pathological case if the point being queried lies exactly on a vertex. The easiest way to cope with this is to test that as a separate process and make your own decision as to whether you want to consider them inside or outside.

**Area**

The area is given by 
$$A = \frac{1}{2} \sum_{i=0}^{N-1} (x_i y_{i+1} - x_{i+1} y_i)$$

Note for polygons with holes. The holes are usually defined by ordering the vertices of the enclosing polygon in the opposite direction to those of the holes. This algorithm still works except that the absolute value should be taken after adding the polygon area to the area of all the holes. That is, the holes areas will be of opposite sign to the bounding polygon area. The sign of the area expression above (without the absolute value) can be used to determine the ordering of the vertices of the polygon. If the sign is positive then the polygon vertices are ordered counter clockwise about the normal, otherwise clockwise.

The only restriction that will be placed on the polygon for this technique to work is that the polygon must not be self intersecting, for example the solution will fail in the following cases.

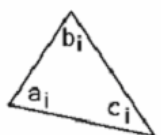
**Centroid**

As in the calculation of the area above,  $x_N$  is assumed to be  $x_0$ , in other words the polygon is closed.

$$c_x = \frac{1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1}) (x_i y_{i+1} - x_{i+1} y_i) \quad c_y = \frac{1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1}) (x_i y_{i+1} - x_{i+1} y_i)$$

**Centroid of a 3D shell described by 3 vertex facets**

The centroid  $C$  of a 3D object made up of a collection of  $N$  triangular faces with vertices  $(a_i, b_i, c_i)$  is given below.  $R_i$  is the average of the vertices of the  $i$ 'th face and  $A_i$  is twice the area of the  $i$ 'th face. Note the faces are assumed to be thin sheets of uniform mass, they need not be connected or form a solid object. This reduces to the equations above for a 2D 3 vertex polygon.

$$C = \frac{\sum_{i=0}^{N-1} A_i R_i}{\sum_{i=0}^{N-1} A_i}$$


$$R_i = (a_i + b_i + c_i) / 3$$

$$A_i = \|(b_i - a_i) \otimes (c_i - a_i)\|$$

**Strongly Connected Component**

1. Call DFS( $G$ ) to compute finishing time  $f[u]$  for each vertex  $u$
2. Compute  $G^T$
3. Call DFS( $G^T$ ), but in the main loop of DFS, consider the vertices in order of decreasing  $f[u]$ .
4. Output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component.

**Disjoint set operation**

**MakeSet**( $x$ )

1.  $p[x] = x$
2.  $\text{rank}[x] = 0$

**Union**( $x, y$ )

1.  $\text{Link}(\text{FindSet}(x), \text{FindSet}(y))$

**Link**( $x, y$ )

1. if  $\text{rank}[x] > \text{rank}[y]$

2. then  $p[y] = x$
3. else  $p[x] = y$
4.     if  $\text{rank}[x] = \text{rank}[y]$
5.         then  $\text{rank}[y] = \text{rank}[y] + 1$

**FindSet(x)**

1. if  $x \neq p[x]$
2.     then  $p[x] = \text{FindSet}(p[x])$
3. return  $p[x]$

**Graham Scan(Q)**

$Q$  is a set of points representing polygon vertices in arbitrary order,  $|Q| \geq 3$ . When the algorithm terminates, stack  $S$  contains exactly the vertices of  $\text{CH}(Q)$ , in counter-clockwise order.

1. Let  $P_0$  the point in  $Q$  with the minimum  $y$ -coordinate, or the leftmost such point in the case of a tie.
2. Let  $\langle P_1, P_2, \dots, P_m \rangle$  be the remaining points in  $Q$ , sorted by polar angle in counterclockwise order around  $P_0$  ( If more than one point has the same angle, remove all but the one that is farthest from  $P_0$ )
3.  $\text{top}[S] = 0$
4. Push( $P_0, S$ )
5. Push( $P_1, S$ )
6. Push( $P_2, S$ )
7. For  $i = 3$  to  $m$
8.   do while the angle formed by points  $\text{Next-To-Top}(S), \text{Top}(S)$  and  $P_i$
9.     makes a nonleft turn
10.         do Pop( $S$ )
11.         Push( $S, P_i$ )
12.     return  $S$

**Extended-Euclid(a,b)**

1. If  $b = 0$
2.     then return  $(a, 1, 0)$
3.  $(d', x', y') = \text{Extended-Euclid}(b, a \bmod b)$
4.  $(d, x, y) = (d', y', x' - \lfloor a/b \rfloor y')$
5. return  $(d, x, y)$

**Euler's Phi function**

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

**Modular Linear Equation Solver(a,b,n)**

Obtains the solutions of  $ax = b \bmod n$

1.  $(d, x', y') = \text{Extended-Euclid}(a, n)$
2. if  $d \mid b$
3.     then  $x_0 = x'(b/d) \bmod n$
4.     for  $i = 0$  to  $d - 1$
5.         do print  $(x_0 + i(n/d)) \bmod n$
6.     else print "no solutions"

**Modular-Exponentiation(a,b,n)**

1.  $c = 0$
2.  $d = 1$
3. let  $\langle b_k, b_{k-1}, \dots, b_0 \rangle$  be the binary representation of  $b$
4. for  $i = k$  downto  $0$
5.     do  $c = 2c$
6.      $d = (d \cdot d) \bmod n$
7.     if  $b_i = 1$
8.         then  $c = c + 1$

```

9.          d = (d · a) mod n
10.         return d

```

### Knuth-Morris-Pratt Algorithm

#### **KMP-Matcher(T,P)**

```

1. n = length[T]
2. m = length[P]
3. p = Compute-Prefix-Function(P)
4. q = 0
5. for i = 1 to n
6.   do while q > 0 and P[q+1] != T[i]
7.     do q = p[q]
8.     if P[q+1] = T[i]
9.       then q = q + 1
10.    if q = m
11.      then print "Pattern occurs with shift" i - m
12.      q = p[q]

```

#### **Compute-Prefix-Function(P)**

```

1. m = length[P]
2. p[1] = 0
3. k = 0
4. for q = 2 to m
5.   do while k > 0 and P[k+1] != P[q]
6.     do k = p[k]
7.     if P[k+1] = P[q]
8.       then k = k + 1
9.     p[q] = k
10.  return p

```

### Cyrus-Beck line clipping Algorithm

Precalculate  $N_i$  and  $P_E$  for each edge

for ( each line segment to be clipped )

```

{   if ( P1 == P0 )
    line is degenerated, so clip as a point;
    else
    { tE = 0; tL = 1;
      for ( each candidate intersection with a clip edge )
      {   if ( Ni · D != 0 ) /* Ignore edges parallel to line */
          { calculate t;
            use sign of Ni · D to categorize as PE or PL;
            if (PE) tE = max(tE , t);
            if (PL) tL = min(tL , t);
          }
      }
      if (tE > tL) return NULL;
      else return P(tE) and P(tL) as true clip intersection;
    }
}

```

#### **Note:**

$$\begin{array}{lll}
 N_i \cdot D < 0 \Rightarrow PE & N_i \cdot D > 0 \Rightarrow PL & t = \frac{N_i \cdot [P_0 - P_E]}{-N_i \cdot D}, \quad D = P_0 - P_1 \\
 \Rightarrow \text{Angle} > 90^\circ & \Rightarrow \text{Angle} < 90^\circ &
 \end{array}$$

$P_0P_1$  is the line segment.  $P_E$  is a point on the clip edge.

### JAVA Selected material

#### Taking Input:

```

static BufferedReader stdin =
new BufferedReader(new InputStreamReader(System.in));

```

```
String s = stdin.readLine();
Stringtokenizer tokens = new StringTokenizer(s);
S = tokens.nextToken();
```

#### Alternate way of taking input using StreamTokenizer:

```
static StreamTokenizer input = null;
static int getInt() throws IOException
{
    if (input == null){
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        input = new StreamTokenizer(br);
    }
    input.nextToken();
    return (int)(input.nval+0.01);
}
```

**nval:** If the current token is a number, this field contains the value of that number

**sval :** If the current token is a word token, this field contains a string giving the characters of the word token.

#### BigInteger (Function signatures)

```
static BigInteger valueOf(long val)
int intValue()
double doubleValue()
Long longValue()
String toString()
String toString(int radix)
int compareTo(BigInteger val)
    Returns -1, 0 or 1 as this BigInteger is numerically less than, equal to, or
    greater than val.
BigInteger abs()
BigInteger add(BigInteger val)
BigInteger subtract(BigInteger val)
BigInteger multiply(BigInteger val)
BigInteger divide(BigInteger val)
BigInteger remainder(BigInteger val)
BigInteger[] divideAndRemainder(BigInteger val)
    Returns an array of two BigIntegers containing (this / val) followed by
    (this%val).
BigInteger mod(BigInteger m)
BigInteger gcd(BigInteger val)
BigInteger max(BigInteger val)
BigInteger min(BigInteger val)
BigInteger negate()
BigInteger pow(int exponent)
BigInteger modPow(BigInteger exponent, BigInteger m)
    Returns a BigInteger whose value is (thisexponent mod m).
BigInteger setBit(int n)
BigInteger shiftLeft(int n)
BigInteger shiftRight(int n)
BigInteger and(BigInteger val)
BigInteger not()
BigInteger or(BigInteger val)
BigInteger xor(BigInteger val)
byte[] toByteArray()
    Returns a byte array containing the two's-complement representation of this
    BigInteger.
BigInteger clearBit(int n)
    Returns a BigInteger whose value is equivalent to this BigInteger with the
    designated bit cleared.
BigInteger flipBit(int n)
```

Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit flipped.

```
int getLowestSetBit()
```

```
boolean testBit(int n)
```

Returns true if and only if the designated bit is set.

### StringTokenizer

```
boolean hasMoreTokens()
```

```
String nextToken()
```

```
String nextToken(String delim)
```

### Integer

```
Integer(int value)
```

```
Integer(String s)
```

```
static int parseInt(String s)
```

```
static int parseInt(String s, int radix)
```

```
String toString()
```

static String **toString**(int i) Returns a new String object representing the specified integer.

```
static String toString(int i, int radix)
```

```
static Integer valueOf(String s)
```

```
static Integer valueOf(String s, int radix)
```

### finding whether symmetry line exists for a non self-intersecting polygon (convex/concave)

```
#include <stdio.h>
```

```
#include <math.h>
```

```
#define MAX 200
```

```
#define PREC 1e-8
```

```
struct Point
```

```
{ int x,y;
```

```
}p[MAX];
```

```
int N;
```

```
int equals(double x,double y,double prec)
```

```
{ return fabs(x-y) < prec;
```

```
}
```

```
bool isSymLine(double x1,double y1,double x2,double y2,int prev,  
int next,Point p[],int N)
```

```
{ double a,b,c,mx,my;
```

```
int j,k,l;
```

```
a = y1-y2; b = x2-x1; c = y1*(x1-x2) - x1*(y1-y2);
```

```
for(k=0;k<N/2;k++)
```

```
{ j = (next+k)%N; l = (prev-k+N)%N;
```

```
// Now find mirror of alpha = p[j].x and beta = p[j].y
```

```
mx = -(p[j].x*(a*a-b*b) + 2*a*(c+b*p[j].y))/(a*a+b*b);
```

```
my = (p[j].y*(a*a-b*b) - 2*b*(c+a*p[j].x))/(a*a+b*b);
```

```
if(!equals(mx,p[l].x,PREC) || !equals(my,p[l].y,PREC))
```

```
return false;
```

```
}
```

```
return true;
```

```
}
```

```
bool oddCase(void)
```

```
{ int i,j,k;
```

```
double x1,y1,x2,y2;
```

```
for(i=0;i<N;i++)
```

```
{ x1 = p[i].x; y1 = p[i].y;
```

```
j = (i+N/2)%N; k = (j+1)%N;
```



```

        x2 = (p[j].x + p[k].x)/2.0; y2 = (p[j].y + p[k].y)/2.0;
        if(isSymLine(x1,y1,x2,y2,(i-1+N)%N,(i+1)%N,p,N)) break;
    }
    return i<N;
}
bool evenCase(void)
{
    int i,j,k,l;
    double x1,y1,x2,y2;
    for(i=0;i<N;i++)
    {
        j = (i+1)%N; k = (i+N/2)%N; l = (k+1)%N;
        x1 = (p[i].x+p[j].x)/2.0; y1 = (p[i].y+p[j].y)/2.0;
        x2 = (p[k].x+p[l].x)/2.0; y2 = (p[k].y+p[l].y)/2.0;
        if(isSymLine(x1,y1,x2,y2,i,j,p,N)) break;
    }
    if(i<N) return true;
    for(i=0;i<N;i++)
    {
        x1 = p[i].x; y1 = p[i].y;
        j = (i+N/2)%N;
        x2 = p[j].x; y2 = p[j].y;
        if(isSymLine(x1,y1,x2,y2,i,i,p,N)) break;
    }
    return i<N;
}
int main(void)
{
    int testCase,dataSet;
    int i;
    bool t;
    scanf("%d",&dataSet);
    for(testCase=0;testCase<dataSet;testCase++)
    {
        scanf("%d",&N);
        for(i=0;i<N;i++) scanf("%d %d",&p[i].x,&p[i].y);
        if(N%2) t = oddCase();
        else t = evenCase();
        if(t) printf("Yes\n");
        else printf("No\n");
    }
    return 0;
}

```

### Trigonometric functions

**asin:** Returns the arcsine of x in the range  $-\pi/2$  to  $\pi/2$  radians.

**acos:** Returns the arccosine of x in the range 0 to  $\pi$  radians.

**atan:** Returns a value in the range  $-\pi/2$  to  $\pi/2$  radians.

**atan2(y,x):** Returns the arctangent of y/x. It returns a value in the range  $-\pi$  to  $\pi$  radians, using the signs of both parameters to determine the quadrant of the return value. It is well defined for every point other than the origin, even if x equals 0 and y does not equal 0.

### Vector

- If 3 points A(**a**),B(**b**) & R(**r**) with common origin are collinear, scalars  $\alpha, \beta$  &  $\gamma$  are such that  $\alpha+\beta+\gamma = 0$  and  $\alpha\mathbf{a}+\beta\mathbf{b}+\gamma\mathbf{r} = 0$ .
- If position vectors **a,b,c,d** of four points A,B,C,D, no three of which are collinear and the non-zero scalars  $\alpha, \beta, \gamma, \delta$  are such that  $\alpha+\beta+\gamma+\delta = 0$  and  $\alpha\mathbf{a}+\beta\mathbf{b}+\gamma\mathbf{c}+\delta\mathbf{d} = 0$ , then the four points are coplanar.
- $[\mathbf{a} \ \mathbf{b} \ \mathbf{c}] = [\mathbf{b} \ \mathbf{c} \ \mathbf{a}] = [\mathbf{c} \ \mathbf{a} \ \mathbf{b}] = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$ . If vectors **a,b,c** represent the 3 sides of a parallelepiped, then its volume  $V = |[\mathbf{a} \ \mathbf{b} \ \mathbf{c}]|$ .
- Let A(**a**),B(**b**),C(**c**),D(**d**) be the 4 vertices of a tetrahedron. Then its volume  $V = \frac{1}{6}[\mathbf{a} - \mathbf{d} \ \mathbf{b} - \mathbf{d} \ \mathbf{c} - \mathbf{d}]$

- Let  $a, b, c, d$  are 4 vectors. Then  $\mathbf{d} = \frac{[\mathbf{d}bc]\mathbf{a} + [\mathbf{d}ca]\mathbf{b} + [\mathbf{dab}]\mathbf{c}}{[\mathbf{abc}]}$ ,  $[\mathbf{a} \ \mathbf{b} \ \mathbf{c}] \neq 0$
- Let 2 non-coplanar straight lines be:  $\mathbf{r} = \mathbf{a} + t\mathbf{b}$ ,  $\mathbf{r} = \mathbf{a}' + s\mathbf{b}'$ . Then, shortest distance between these lines =  $\frac{[\mathbf{b}\mathbf{b}'\mathbf{a}' - \mathbf{a}]}{|\mathbf{b} \times \mathbf{b}'|}$ .
- Equation of a plane:  $p - \mathbf{r} \cdot \mathbf{n} = 0$ .  $p$  is the perpendicular distance from origin.  $\mathbf{n}$  is unit normal of the plane.
- Equation of the bisectors of the angles between 2 planes  $p - \mathbf{r} \cdot \mathbf{n}_1 = 0$ ,  $p - \mathbf{r} \cdot \mathbf{n}_2 = 0$ : If the origin & the points on the bisector lie on the same side of the given planes, the 2 perpendicular distances have the same sign. Then, the equation is  $p - \mathbf{r} \cdot \mathbf{n}_1 = p' - \mathbf{r} \cdot \mathbf{n}_2$  or  $p - p' = \mathbf{r} \cdot (\mathbf{n}_1 - \mathbf{n}_2)$ . When the origin & points on the other bisector are on the opposite sides of one of the given planes, then the perpendicular distances will be of opposite signs. Hence the equation is  $p + p' = \mathbf{r} \cdot (\mathbf{n}_1 + \mathbf{n}_2)$ .

### Translation, Scaling & Rotation

<i>Translation matrix</i>	<i>Scaling Matrix</i>	<i>Rotation about Z</i>
$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

### Numerical Methods

#### Descartes' rule of sign

- In a polynomial equation  $P(x) = 0$ , the terms being written in the order of power of  $x$ , the number of positive roots **cannot exceed** the number of changes of sign (+ to - & vice versa) in the coefficients of  $P(x)$ , and the negative roots **cannot exceed** the number of changes of sign  $P(-x)$ .
- If  $P(x)$  is continuous in  $[a, b]$ , then if  $P(a)$  &  $P(b)$  have opposite signs, there exists an odd number of real roots of the equation  $P(x) = 0$  in  $(a, b)$ . If  $P(a)$  &  $P(b)$  have same sign, then there exists no real root or an even number of real roots in  $(a, b)$ .

#### Derivative & factor of a polynomial

Let  $P(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n$ . Let  $b_0 = a_0$ ,  $b_r = b_{r-1}x + a_r$ ,  $[r=1, 2, \dots, n]$  &  $Q(x) = b_0x^{n-1} + b_1x^{n-2} + \dots + b_{n-2}x + b_{n-1}$ . Then  $P(x) = (x-X)Q(x) + b_n$ . Therefore derivative at  $x = X$ ,  $P'(X) = Q(X)$ . Also if  $x-X$  is a factor of  $P(x)$  then  $P(x) = (x-X)Q(x)$ .

#### Lagrange's Interpolation Formula

$$f(x) = \frac{(x-x_1)(x-x_2)\dots(x-x_n)}{(x_0-x_1)(x_0-x_2)\dots(x_0-x_n)} f(x_0) + \frac{(x-x_0)(x-x_2)\dots(x-x_n)}{(x_1-x_0)(x_1-x_2)\dots(x_1-x_n)} f(x_1) \\ + \dots + \frac{(x-x_0)(x-x_1)\dots(x-x_{n-1})}{(x_n-x_0)(x_n-x_1)\dots(x_n-x_{n-1})} f(x_n), [f(x) \text{ is a polynomial in } n]$$

#### Assignment Problem

```
# include <stdio.h>
# define SIZE 5
void matching(int c[2*SIZE][2*SIZE], int n, int m[2*SIZE])
{ int qsize=n+1, q[2*SIZE+1], rear, front, par[2*SIZE], level[2*SIZE], i, j, k, last;
  for(i=0; i<n; i++) m[i]=-1;
  for(k=0; k<n; k++)
    if(m[k]==-1){
```

```

last=-1;
for(i=0;i<n;i++){
    level[i]=0; par[i]=-1;
}
rear=front=0; level[k]=1; q[rear++]=k;
if(rear==qsize) rear=0;
while(rear!=front){
    if(last!=-1) break;
    i=q[front++];
    if(front==qsize) front=0;
    if(level[i]%2){
        for(j=0;j<n;j++){
            if(c[i][j] && level[j]==0){
                level[j]=level[i]+1; par[j]=i;
                if(m[j]==-1){
                    last=j; break;
                }
            }
            else{
                q[rear++]=j;
                if(rear==qsize) rear=0;
            }
        }
    }
    else{
        j=m[i]; level[j]=level[i]+1;
        par[j]=i; q[rear++]=j;
        if(rear==qsize) rear=0;
    }
}
while(last!=-1){
    m[last]=par[last]; m[par[last]]=last; last=par[par[last]];
}
}
}
int isperfect(int m[],int n)
{ int i;
  for(i=0;i<n;i++)
    if(m[i]==-1) return 0;
  return 1;
}
void dfs(int k,int i,int mark[2][SIZE],int e[SIZE][SIZE],int n,int m[],int depth)
{ int j;
  if(depth) mark[k][i]=1;
  if(k==0){
    for(j=0;j<n;j++){
        if(e[i][j]==0 && m[i]!=j+n){
            mark[k][i]=1;
            if(mark[1][j]==0) dfs(1,j,mark,e,n,m,depth+1);
        }
    }
  }
  else{
    for(j=0;j<n;j++){
        if(e[j][i]==0 && m[i+n]==j){
            mark[k][i]=1;
            if(mark[0][j]==0) dfs(0,j,mark,e,n,m,depth+1);
        }
    }
  }
}
}
void OptimalAssignment(int w[SIZE][SIZE],int n,int m[])
{ int u[SIZE],v[SIZE],e[SIZE][SIZE],c[2*SIZE][2*SIZE];
  int mark[2][SIZE],i,j,k,eta;
  for(i=0;i<n;i++){

```

```

    u[i]=v[i]=0;
    for(j=0;j<n;j++)
        if(w[i][j]>u[i]) u[i]=w[i][j];
}
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        e[i][j]=u[i]+v[j]-w[i][j];
for(i=0;i<2*n;i++)
    for(j=0;j<2*n;j++)
        c[i][j]=0;
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        if(e[i][j]==0) c[i][n+j]=c[n+j][i]=1;
matching(c,2*n,m);
while(!isperfect(m,2*n)){
    for(i=0;i<n;i++)
        mark[0][i]=mark[1][i]=0;
    for(i=0;i<n;i++)
        if(m[i]==-1 && mark[0][i]==0) dfs(0,i,mark,e,n,m,0);
    eta=-1;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(mark[0][i] && mark[1][j]==0)
                if(eta==-1 || eta>u[i]+v[j]-w[i][j]) eta=u[i]+v[j]-w[i][j];
    for(i=0;i<n;i++){
        if(mark[0][i]) u[i]-=eta;
        if(mark[1][i]) v[i]+=eta;
    }
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            e[i][j]=u[i]+v[j]-w[i][j];
    for(i=0;i<2*n;i++)
        for(j=0;j<2*n;j++)
            c[i][j]=0;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(e[i][j]==0) c[i][n+j]=c[n+j][i]=1;
    matching(c,2*n,m);
}
}
int main(void)
{ int n=5,w[SIZE][SIZE]={{4,1,6,2,3},{5,0,3,7,6},{2,3,4,5,8},
                        {3,4,6,3,4},{4,6,5,8,6}};

  int m[2*SIZE],i,total;
  OptimalAssignment(w,n,m);
  total=0;
  for(i=0;i<n;i++)
      total+=w[i][m[i]-n];
  printf("%d\n",total);
  return 0;
}

```

**Map**

```

#include <stdio.h>
#include <map>
#include <malloc.h>

struct species {
    char s[81];
} buf;
bool operator==(const species &a, const species &b){
    return !strcmp(a.s, b.s);
}

```

```

}
bool operator<(const species &a, const species &b){
    return strcmp(a.s, b.s) < 0;
}
map<species,int> cnt;

main()
{ int tot = 0;
  while(gets(buf.s)){
    cnt[buf]++;
    tot++;
  }
  for (map<species,int>::iterator i = cnt.begin(); i!=cnt.end(); *i++){
    printf("%s %0.4lf\n",i->first.s, 100.0*i->second/tot);
  }
}

```

### Stable Matching

Let there be  $n$  men and  $n$  women. A perfect matching is stable if and only if there is no man  $x$  and woman  $a$  such that  $x$  prefers  $a$  to his current partner and  $a$  prefers  $x$  to her current partner. A stable matching always exists.

#### **Algorithm**

**Input:** Preference rankings by each of  $n$  men and  $n$  women

**Iteration:** Each unmatched man proposes to the highest woman on his preference list who has not previously rejected him and is not yet matched. If each woman receives exactly one proposal, stop with these being the remaining confirmed matches.

Otherwise, at least one woman receives at least two proposals. Every woman receiving more than one proposal rejects all but the highest on her list. Every woman receiving a proposal says "maybe" to the most attractive proposal received.

**Note:** The same algorithm can be used with women instead of men proposing. Of all stable matchings, every man is happiest under the male-proposal algorithm, and every woman is happiest under the female-proposal algorithm.

### Eular cycles

#### Fleury's Algorithm- constructing Eulerian trails

**Input:** A graph  $G$  with one nontrivial component and at most two odd vertices.

**Initialization:** Start at a vertex that has odd degree unless  $G$  is even, in which case start at any vertex.

**Iteration:** From the current vertex, traverse any remaining edge whose deletion from the remaining graph does not leave a graph with two non-trivial components. Stop when all edges have been traversed.

#### Directed Eulerian circuit

**Input:** A digraph  $G$  that is an orientation of a connected graph and has  $d^+(u) = d^-(u)$  for all  $u \in V(G)$ .

**Step 1:** Choose a vertex  $v \in V(G)$ . Let  $G'$  be the digraph obtained from  $G$  by reversing direction on each edge. Search  $G'$  to construct a tree  $T'$  consisting of paths from  $v$  to all other vertices (use BFS or DFS).

**Step 2:** Let  $T$  be the reversal of  $T'$ ;  $T$  contains a  $u,v$ -path in  $G$  for each  $u \in V(G)$ . Specify an arbitrary ordering of the edges that leave each vertex  $u$ , except that for  $u \neq v$  the edge leaving  $u$  in  $T$  must come last. (The tree edge is last in order).

**Step 3:** Construct an Eulerian circuit from  $v$  as follows: whenever  $u$  is the current vertex, exit along the next unused edge in the ordering specified for edges leaving  $u$ .