Computer Science Fundamental Algorithms

Ali Sharifi and Behdad Esfahbod
Sharif University of Technology
Tehran, Iran

2001

Contents

1 Graph Algorithms

1.1 Minimum Weight Spanning Tree - Prim Algorithm

```
{
 Minimum Spanning Tree

 Prim Algorithm  O(N3)

 Input:
  G: UnDirected weighted graph (Infinity = No Edge)
  N: Number of vertices
 Output:
  Parent: Parent of each vertex in the MST, Parent[1] = 0
  NoAnswer: Has no spanning trees (== Not connected)

 Reference:
  CLR, p509

 By Ali
}
program
  MinimumSpanningTree;

const
  MaxN = 100 + 1;
  Infinity = 10000;

var
  N: Integer;
  G: array[1 .. MaxN, 1 .. MaxN] of Integer;
  Parent: array[1 .. MaxN] of Integer;
  NoAnswer: Boolean;

  Key: array[1 .. MaxN] of Integer;
  Mark: array[1 .. MaxN] of Boolean;

procedure MST;
var
  I, U, Step: Integer;
  Min: Integer;
begin
  for I := 1 to N do
    Key[I] := Infinity;
  FillChar(Mark, SizeOf(Mark), 0);
  Key[1] := 0;
  Parent[1] := 0;
  for Step := 1 to N do
  begin
    Min := Infinity;
    for I := 1 to N do
      if not Mark[i] and (Key[I] < Min) then
      begin
        U := I;
        Min := Key[I];
      end;
    if Min = Infinity then
    begin
      NoAnswer := True;
      Exit;
    end;
    Mark[U] := True;
    for I := 1 to N do
      if not Mark[I] and (Key[I] > G[U, I]) then
      begin
        Key[I] := G[U, I];
        Parent[I] := U;
      end;
  end;
  NoAnswer := False;
end;

begin
  MST;
end.
```

1.2 Single Source Shortest Path - Dijkstra Algorithm

```
{
 Single Source Shortest Paths - Without Negative Weight Edges

 Dijkstra Algorthm  O(N2)

 Input:
  G: Directed weighted graph (No Edge = Infinity)
  N: Number of vertices
  S: The source vertex
 Output:
  D[I]: Length of minimum path from S to I (No Path = Infinity)
  P[I]: Parent of vertex I in its path from S, P[S] = 0, (No Path->P[I]=0)

 Note:
  Infinity should be less than the max value of its type
  No negative edge

 Reference:
  CLR, p527

 By Ali
}

program
  Dijkstra;

const
  MaxN = 100 + 1;
  Infinity = 10000;

var
  N, S: Integer;
  G: array [1 .. MaxN, 1 .. MaxN] of Integer;
  D, P: Array[1..MaxN] of Integer;
  NoAnswer: Boolean;

  Mark: array[1 .. MaxN] of Boolean;

procedure Relax(V, U: Integer);
begin
  if D[U] > D[V] + G[V, U] then
  begin
    D[U] := D[V] + G[V, U];
    P[U] := V;
  end;
end;

procedure SSSP;
var
  I, U, Step: Integer;
  Min: Integer;
begin
  FillChar(Mark, SizeOf(Mark), 0);
  FillChar(P, SizeOf(P), 0);
  for I := 1 to N do
    D[I] := Infinity;
  D[S] := 0;
  for Step := 1 to N do
  begin
    Min := Infinity;
    for I := 1 to N do
      if not Mark[I] and (D[I] < Min) then
      begin
        U := I;
        Min := D[I];
      end;
    if Min = Infinity then
      Break;
    Mark[U] := True;
    for I := 1 to N do
      Relax(U, I);
  end;
end;

begin
  SSSP;
end.
```

1.3 Single Source Shortest Path - BellmanFord Algorithm

```
{
 Single Source Shortest Paths - With Negative Weight Edges

 BellmanFord Algorthm  O(N3)

 Input:
  G: Directed weighted graph (No Edge = Infinity)
  N: Number of vertices
  S: The source vertex
 Output:
  D[I]: Length of minimum path from S to I
  P[I]: Parent of vertex I in its path from S, P[S] = 0
  NoAnswer: Graph has cycle with negative length

 Note:
  Infinity should be less than the max value of its type

 Reference:
  CLR, p532

 By Ali
}
program
  BellmanFord;

const
  MaxN = 100 + 2;
  Infinity = 10000;

var
  N, S: Integer;
  G: array [1 .. MaxN, 1 .. MaxN] of Integer;
  D, P: Array[1..MaxN] of Integer;
  NoAnswer: Boolean;

procedure Relax(V, U: Integer);
begin
  if D[U] > D[V] + G[V, U] then
  begin
    D[U] := D[V] + G[V, U];
    P[U] := V;
  end;
end;

procedure SSSPNeg;
var
  I, J, K: Integer;
begin
  FillChar(P, SizeOf(P), 0);
  for i := 1 to N do
    D[I] := Infinity;
  D[S] := 0;
  for K := 1 to N - 1 do
    for I := 1 to N do
      for J := 1 to N do
        Relax(I, J);
  NoAnswer := False;
  for I := 1 to N do
    for J := 1 to N do
      if D[J] > D[I] + G[I, J] then
      begin
        NoAnswer := True;
        Exit;
      end;
end;

begin
  SSSPNeg;
end.
```

1.4 Cutting Edges - DFS Method

```
{
 Cut Edge

 DFS Method  O(N2)

 Input:
  G: Undirected Simple Graph
  N: Number of vertices,
 Output:
  EdgeNum: Nunmber of CutEdges
  EdgeList[i]: CutEdge I

 Reference:
  Creative, p224

 By Ali
}
program
  CutEdge;

const
  MaxN = 100 + 2;

var
  N: Integer;
  G: array[1 .. MaxN, 1 .. MaxN] of Integer;
  EdgeNum: Integer;
  EdgeList: array[1 .. MaxN * MaxN, 1 .. 2] of Integer;

  DfsNum: array[1 .. Maxn] of Integer;
  DfsN: Integer;

function Dfs(V: Integer; Parent: Integer) : Integer;
var
  I, J, Hi: Integer;
begin
  DfsNum[V] := DfsN;
  Dec(DfsN);
  Hi := DfsNum[V];
  for I := 1 to N do
    if (G[V, I] <> 0) and (I <> Parent) then
      if DfsNum[I] = 0 then
      begin
        J := Dfs(I, V);
        if J <= DfsNum[I] then
        begin
          Inc(EdgeNum);
          EdgeList[EdgeNum, 1] := V;
          EdgeList[EdgeNum, 2] := I;
        end;
        if Hi < J then
          Hi := J;
      end
      else
        if Hi < DfsNum[I] then
          Hi := DfsNum[I];
  Dfs := Hi;
end;

procedure CutEdges;
var
  I: Integer;
begin
  FillChar(DfsNum, SizeOf(DfsNum), 0);
  DfsN := N;
  EdgeNum := 0;
  for I := 1 to N do
    if DfsNum[I] = 0 then
      Dfs(I, 0); {I == Root of tree}
end;

begin
  CutEdges;
end.
```

1.5 Cutting Vertices - DFS Method

```
{
 Cut Vertex

 DFS Method  O(N2)

 Input:
  G: Undirected Simple Graph
  N: Number of vertices,
 Output:
  IsCut[i]: Vertex i is a CutVertex.

 Reference:
  Creative, p224

 By Ali
}
program
  CutVertex;

const
  MaxN = 100 + 2;

var
  N: Integer;
  G: array[1 .. MaxN, 1 .. MaxN] of Integer;
  IsCut: array[1 .. MaxN] of Boolean;

var
  DfsNum: array[1..Maxn] of Integer;
  DfsN: Integer;

function BC(V: Integer; Parent: Integer) : Integer;
var
  I, J, ChNum, Hi: Integer;
begin
  DfsNum[V] := DfsN;
  Dec(DfsN);
  ChNum := 0;
  Hi := DFSNum[v];
  for I := 1 to N do
    if (G[V, I] <> 0) and (I <> Parent) then
      if DFSNum[I] = 0 then
      begin
        Inc(ChNum);
        J := BC(I, V);
        if J <= DfsNum[V] then
          if (Parent <> 0) or (ChNum > 1) then
            IsCut[V] := True;
        if Hi < J then
          Hi := J;
      end
      else
        if Hi < DfsNum[I] then
          Hi := DfsNum[I];
  BC := Hi;
end;

procedure CutVertices;
var
  I: Integer;
begin
  FillChar(DfsNum, SizeOf(DfsNum), 0);
  FillChar(IsCut, SizeOf(IsCut), 0);
  DfsN := N;
  for I := 1 to N do
    if DfsNum[I] = 0 then
      BC(I, 0); {I == The root of the DFS tree}
end;

begin
  CutVertices;
end.
```

1.6 BiConnected Components - DFS Method

```
{
 BiConnected Components

 DFS Method  O(N2)

 Input:
  G: UnDirected simple graph
  N: Number of vertices
 Output:
  IsCut[I]: Vertex I is CutVertex
  CompNum: Number of components
  Comp[I]: Vertices in component I
  CompLen[I]: Size of component I

 Reference:
  Creative, p224

 By Ali
}
program
  BiConnectedComponents;

const
  MaxN = 100 + 2;

var
  N: Integer;
  G: array[1 .. MaxN, 1 .. MaxN] of Integer;
  IsCut: array[1 .. MaxN] of Boolean;
  CompNum: Integer;
  Comp: array[1 .. MaxN, 1 .. MaxN] of Integer;
  CompLen: array[1 .. MaxN] of Integer;

  DfsNum: array[1 .. Maxn] of Integer;
  DfsN: Integer;
  Stack: array[1 .. MaxN] of Integer;{** Must be changed to 2dim if we want
                                         to have the edges of a comp.}
  SN: Integer; {Size of stack}

procedure Push(V: Integer);
begin
  Inc(SN);
  Stack[SN] := V;
end;

function Pop: Integer;
begin
  if SN = 0 then
    Pop := -1
  else
  begin
    Pop := Stack[SN];
    Dec(SN);
  end;
end;

function BC(V: Integer; Parent: Integer) : Integer;
var
  I, J, Hi, ChNum: Integer;
begin
  DfsNum[V] := DfsN;
  Dec(DfsN);
  Push(V);
  ChNum := 0;
  Hi := DfsNum[v];
  for I := 1 to N do
    {** insert (v, i) into Stack, each edge will be inserted twice.}
    if (G[V, I] <> 0) and (I <> Parent) then
      if DfsNum[I] = 0 then
      begin
        Inc(ChNum);
        J := BC(I, V);
        if J <= DFSNum[V] then
        begin
          if (Parent <> 0) or (ChNum > 1) then
            IsCut[V] := True;
          Inc(CompNum);
          CompLen[CompNum] := 0;
          repeat
            Inc(CompLen[CompNum]);
            Comp[CompNum, CompLen[CompNum]] := Pop;
```

```
            {** and pop all edges }
          until Comp[CompNum, CompLen[CompNum]] = V;
          Push(V);
        end;
        if Hi < J then
          Hi := J;
      end
      else
        if Hi < DFSNum[I] then
          Hi := DFSNum[I];
  BC := Hi;
end;

procedure BiConnected;
var
  I: Integer;
begin
  FillChar(DfsNum, SizeOf(DfsNum), 0);
  FillChar(IsCut, SizeOf(IsCut), 0);
  SN := 0;
  CompNum := 0;
  DfsN := N;
  for I := 1 to N do
    if DfsNum[I] = 0 then
      BC(I, 0);  {I == The root of the DFS tree}
end;

begin
  BiConnected;
end.
```

1.7 Strongly Connected Components - DFS Method

```
{
 Strongly Connected Components

 DFS Method  O(N2)

 Input:
  G: Directed simple graph
  N: Number of vertices
 Output:
  CompNum: Number of components.
  Comp[I]: Component number of vertex I.

 Reference:
  CLR, p489

 By Ali
}
program
  StronglyConnectedComponents;

const
  MaxN = 100 + 2;

var
  N: Integer;
  G: array[1 .. MaxN, 1 .. MaxN] of Integer;
  CompNum: Integer;
  Comp: array[1 .. MaxN] of Integer;

var
  Fin: array[1 .. MaxN] of Integer;
  DfsN: Integer;

procedure Dfs(V: Integer);
var
  I: Integer;
begin
  Comp[V] := 1;
  for I := 1 to N do
    if (Comp[I] = 0) and (G[V, I] <> 0) then
      Dfs(I);
  Fin[DfsN] := V;
  Dec(DfsN);
end;

procedure Dfs2(V: Integer);
var
  I: Integer;
begin
  Comp[V] := CompNum;
  for I := 1 to N do
    if (Comp[I] = 0) and (G[I, V] <> 0) then
      Dfs2(I);
end;

procedure StronglyConnected;
var
  I: Integer;
begin
  FillChar(Comp, SizeOf(Comp), 0);
  DfsN := N;
  for I := 1 to N do
    if Comp[I] = 0 then
      Dfs(I);
  FillChar(Comp, SizeOf(Comp), 0);
  CompNum := 0;
  for I := 1 to N do
    if Comp[Fin[I]] = 0 then
    begin
      Inc(CompNum);
      Dfs2(Fin[I]);
    end;
end;

begin
  StronglyConnected;
end.
```

1.8 Eulerian Tour - Flory Algorithm

```
{
 Eulerian Tour

 Flory Algorithm  O(N2)

 Input:
  G: Directed (not nessecerily simple) connected eulerian graph
  N: Number of vertices
 Output:
  CLength: Length of tour
  C: Eulerian tour

 Reference:
  West

 By Ali
}
program
  EulerianTour;

const
  MaxN = 50 + 2;

var
  N: Integer;
  G: array[1 .. MaxN, 1 .. MaxN] of Integer;
  CLength: Integer;
  C: array[1 .. MaxN * MaxN] of Integer;

  Lcl: Integer;
  Lc: array[1 .. MaxN] of Integer;
  Tb: array[1 .. MaxN * MaxN, 1 .. 2] of Integer;
  Mark, MMark: array[1 .. MaxN] of Boolean;
  MainV: Integer;

function DFS(v: Integer): boolean;
var
  i: Integer;
begin
  if Mark[v] and (v <> MainV) then
  begin
    DFS := false;
    exit;
  end;
  Mark[v] := true;
  Inc(Lcl);
  Lc[Lcl] := v;
  DFS := true;
  if (v = Mainv) and (Lcl > 1) then
    exit;
  for i := 1 to N do
    if G[v, i] > 0 then
    begin
      Dec(G[v, i]);
      { Dec(G[j, i]); // if graph is undirected!!!}
      if DFS(i) then
        exit;
      Inc(G[v, i]);
      { Inc(G[j, i]); // if graph is undirected!!!}
    end;
  Dec(Lcl);
  DFS := false;
end;

function FindACycle(v: Integer): Boolean;
var
  i, j: Integer;
begin
  FindACycle := false;
  if MMark[v] then
    exit;
  FillChar(Mark, SizeOf(Mark), 0);
  Lcl := 0;
  MainV := v;
  DFS(v);
  if Lcl < 2  then
  begin
    MMark[v] := true;
    exit;
  end;
  FindACycle := true;
```

```
end;


procedure Euler(v: Integer);
var
  i, j, k, u: Integer;
begin
  Tb[1, 1] := v;
  Tb[1, 2] := 0;
  FillChar(MMark, SizeOf(MMark), 0);
  if not FindACycle(v) then
  begin
    CLength := 0;
    exit;
  end;
  for i := 1 to Lcl do  begin
    Tb[i, 1] := Lc[i];
    Tb[i, 2] := i + 1;
  end;
  Tb[Lcl, 2] := 0;
  k := Lcl;
  u := 1;
  repeat
    while FindACycle(Tb[u, 1]) do  begin
      j := Tb[u, 2];
      Tb[u, 2] := k + 1;
      for i := 2 to Lcl do  begin
        Inc(k);
        Tb[k, 1] := Lc[i];
        Tb[k, 2] := k + 1;
      end;
      Tb[k, 2] := j;
    end;
    u := Tb[u, 2];
  until u = 0;
  u := 1;
  k := 0;
  repeat
    Inc(k);
    C[k] := Tb[u, 1];
    u := Tb[u, 2];
  until u = 0;
  CLength := k;
end;

begin
  Euler(1); {Starting vertex}
end.
```

1.9 Minimum Average Weigth Cycle - Karp Algorithm

```
{
 Minimum Average Weight Cycle

 Karp Algorithm  O(N3)

 Input:
  G: Directed weighted simple connected graph (No Edge = Infinity)
  N: Number of vertices
 Output:
  MAW: Average weight of minimum cycle
  CycleLen: Length of cycle
  Cycle: Vertices of cycle
  NoAnswer: Graph does not have directed cycle (NoAnswer->MAW = Infinity)

 Note:
  G should be connected

 Reference:
  CLR

 By Behdad
}
program
  MinimumAverageWeightCycle;

const
MaxN = 100 + 2;
  Infinity = 10000;

var
N: Integer;
  G, P, Ans: array [0 .. MaxN, 0 .. MaxN] of Integer;
  MAW : Extended;
  CycleLen: Integer;
  Cycle: array [1 .. MaxN] of Integer;
  NoAnswer : Boolean;


procedure MAWC;
var
  I, J, K, Q, L : Integer;
  S: Integer;
  T, T2: Extended;
  Flag : Boolean;
begin
  for I := 0 to N do
    for J := 0 to N do
      P[I, J] := Infinity;
  S := 1;
  P[S, 0] := 0;
  Ans[S, 0] := S;
  L := 0;
  repeat
    Inc(L);
    Flag := True;
    for I := 1 to N do
      for J := 1 to N do
        if (G[I, J] < Infinity) and (G[I, J] + P[I, L - 1] < P[J, L]) then
        begin
          P[J, L] := G[I, J] + P[I, L - 1];
          Ans[J, L] := I;
          Flag := False;
        end;
  until (L = N) or Flag;

  MAW := Infinity;
  for I := 1 to N do
    if (P[I, N] < Infinity) then
    begin
      T2 := (P[I, N] - P[I, 0]) / N;
      if P[I, 0] >= Infinity then
        T2 := 0;
      L := 0;
      for J := 1 to N - 1 do
        if P[I, J] < Infinity then
        begin
          T := (P[I, N] - P[I, J]) / (N - J);
          if T > T2 then
          begin
            T2 := T;
            L := J;
```

```
          end;
        end;
      if T2 < MAW then
      begin
    MAW := T2;
        Q := I;
      end;
    end;

  FillChar(G[0], SizeOf(G[0]), 0);
  K := Q;
  I := 0;
  L := N;
  J := N;
  while J >= 0 do
  begin
    if G[0, K] = 1 then
    begin
      I := K;
      Break;
    end;
    G[0, K] := 1;
    K := Ans[K, J];
    Dec(J);
  end;
  if I <> 0 then
  begin
    K := Q;
    while K <> I do
    begin
      K := Ans[K, L];
      Dec(L);
    end;
  end;

  CycleLen := 0;
  NoAnswer := MAW >= Infinity;
  if not NoAnswer and (I <> 0) then
  begin
    J := 1;
    T := 0;
    repeat
      G[J, 0] := K;
      Inc(J);
      K := Ans[K, L];
      Dec(L);
    until K = I;
    G[J, 0] := G[1, 0];
    for I := J downto 2 do
    begin
      Inc(CycleLen);
      Cycle[CycleLen] := G[I, 0];
    end;
  end;
end;

begin
  MAWC;
end.
```

1.10 Bipartite Maximum Matching - Augmenting Path Method

```
{
 Maximum Bipartite Matching

 Augmenting Path Alg.  O(N2.E) Implementation O(N4)
 but very near to O(N.E) Implementation O(N3)

 Input:
  G: UnDirected Simple Bipartite Graph
  M, N: Number of vertices
 Output:
  Mt: Match of Each Vertex (0 if not matched)
  Matched: size of matching (number of matched edges)

 Reference:
  West

 By Behdad
}
program
  BipartiteMaximumMatching;

const
  MaxNum = 100 + 2;

var
  M, N : Integer;
  G : array [1 .. MaxNum, 1 .. MaxNum] of Integer;
  Mt : array [1 .. 2, 1 .. MaxNum] of Integer;
  Mark : array [0 .. MaxNum] of Boolean;
  Matched : Integer;

function  MDfs (V : Integer) : Boolean;
var
  I : Integer;
begin
  if V = 0 then begin MDfs := True; Exit; end;
  Mark[V] := True;
  for I := 1 to N do
    if (G[V, I] <> 0) and not Mark[Mt[2, I]] and MDfs(Mt[2, I]) then
    begin
      Mt[1, V] := I;
      Mt[2, I] := V;
      MDfs := True;
      Exit;
    end;
  MDfs := False;
end;

procedure AugmentingPath;
var
  I: Integer;
begin
  FillChar(Mark, SizeOf(Mark), 0);
  FillChar(Mt, SizeOf(Mark), 0);
  for I := 1 to M do
    if (Mt[1, I] = 0) and MDfs(I) then
    begin
      Inc(Matched);
      FillChar(Mark, SizeOf(Mark), 0);
      I := 0;
    end;
end;

begin
  AugmentingPath;
end.
```

1.11 Bipartite Maximum Independent Set - Matching Method

```
{
 Bipartite Maximum Independent Set

 Matching Method  O(N3)

 Input:
  G: UnDirected Simple Bipartite Graph
  M, N: Number of vertices
 Output:
  I1[I]: Vertex I from first part is in the set
  I2[I]: Vertex I from second part is in the set
  IndSize: Size of independent set

 Reference:
  West

 By Behdad
}
program
  BipartiteMaximumIndependentSet;

const
  MaxNum = 100 + 2;

var
  M, N: Integer;
  G: array [1 .. MaxNum, 1 .. MaxNum] of Integer;
  Mark: array [1 .. MaxNum] of Boolean;
  M1, M2, I1, I2: array [1 .. MaxNum] of Integer;
  IndSize: Integer;

function  ADfs (V : Integer) : Boolean;
var
  I : Integer;
begin
  Mark[V] := True;
  for I := 1 to N do
    if (G[V, I] <> 0)
       and ((M2[I] = 0) or not Mark[M2[I]] and ADfs(M2[I])) then
    begin
      M2[I] := V;
      M1[V] := I;
      ADfs := True;
      Exit;
    end;
  ADfs := False;
end;

procedure BDfs (V : Integer);
var
  I : Integer;
begin
  Mark[V] := True;
  for I := 1 to N do
    if (G[V, I] = 1) and (I2[I] = 1) then
    begin
      I2[I] := 0; I1[M2[I]] := 1;
      BDfs(M2[I]);
    end;
end;

procedure BipIndependent;
var
  I: Integer;
  Fl: Boolean;
begin
  IndSize := M + N;
  FillChar(Mark, SizeOf(Mark), 0);
  repeat
    Fl := True;
    FillChar(Mark, SizeOf(Mark), 0);
    for I := 1 to M do
      if not Mark[I] and (M1[I] = 0) and ADfs(I) then
      begin
        Dec(IndSize);
        Fl := False;
      end;
  until Fl;
  FillChar(I1, SizeOf(I1), 0);
  FillChar(I2, SizeOf(I2), 0);
  FillChar(Mark, SizeOf(Mark), 0);
```

```
  for I := 1 to M do if M1[I] = 0 then I1[I] := 1;
  for I := 1 to N do I2[I] := 1;
  for I := 1 to M do
    if M1[I] = 0 then
      BDfs(I);
end;

begin
  BipIndependent;
end.
```

1.12 Maximum Weighted Bipartite Matching - Hungarian Algorithm

```
{
 Maximum Weighted Bipartite Matching

 Hungarian Algorithm  O(N4) but acts like O(N3)

 Input:
  G: UnDirected Simple Bipartite Graph (No Edge = Infinity)
  N: Number of vertices of each part
 Output:
  Mt: Match of Each Vertex (Infinity = Not Matched)
  NoAnswer: Graph does not have complete matching

 Reference:
  West

 By Behdad
}
program
WeightedBipartiteMatching;

const
MaxN = 100 + 2;
  Infinity = 10000;

var
N: Integer;
G: array [1 .. MaxN, 1 .. MaxN] of Integer;
  Mt : array [0 .. 1, 0 .. MaxN] of Integer;
  NoAnswer: Boolean;

  Color, P, Cover, Q : array [0 .. 1, 0 .. MaxN] of Integer;
  I, J, K, S, T : Integer;

procedure BFS (V : Integer);
var
  QL, QR: Integer;
begin
  QL := 1;
  QR := 1;
  Q[0, 1] :=  0;
  Q[1, 1] :=  V;
  Color[0, V] := 1;
while QL <= QR do
  begin
    K := 1 - Q[0, QL];
   J := Q[1, QL];
    Inc(QL);
    for I := 1 to N do
    begin
      if K = 1 then S := G[J, I] else S := G[I, J];
      if K = 1 then T := Cover[0, J] + Cover[1, I] else T := Cover[1, J] + Cover[0, I];
  if (Color[K, I] = 0) and (S = T) and ((K = 1) or (Mt[0, I] = J)) then
      begin
        Color[K, I] := 1;
        P[K, I] := J;
       Inc(QR);
        Q[0, QR] := K;
        Q[1, QR] := I;
      end;
    end;
  end;
end;

procedure Assignment;
var
  Sum : Longint;
  Count : Integer;
  B : Boolean;
begin
  FillChar(Mt, SizeOf(Mt), 0);
  FillChar(Cover, SizeOf(Cover), 0);
  for I := 1 to N do
  for J := 1 to N do
     if G[I, J] > Cover[0, I] then
       Cover[0, I] := G[I, J];
  repeat
    repeat
      FillChar(Color, SizeOf(Color), 0);
      FillChar(P, SizeOf(P), 0);
      B := False;
      for I := 1 to N do
```

```
      if (Mt[0, I] = 0) and (Color[0, I] = 0) then
         BFS(I);
       for J := 1 to N do
         if (Mt[1, J] = 0) and (Color[1, J] = 1) then
         begin
           B := True;
           Break;
         end;
       if B then
       begin
         Dec(Count);
         K := 1;
         while True do
         begin
           if K = 1 then
           begin
  Mt[1, J] := P[1, J];
             S := J;
           end
           else
             Mt[0, J] := S;
           if P[K, J] = 0 then
             Break;
           J := P[K, J];
           K := 1 - K;
         end;
       end;
     until not B;
     J :=  Infinity;
     for S := 1 to N do
     begin
      if Color[0, S] = 0 then
         Continue;
      for T := 1 to N do
         if (Color[1, T] = 0) and (Cover[0, S] + Cover[1, T] - G[S, T] < J) then
          J := Cover[0, S] + Cover[1, T] - G[S, T];
     end;
     if J < Infinity then
     begin
      for I := 1 to N do
       begin
        if Color[0, I] = 1 then
           Dec(Cover[0, I], J);
        if Color[1, I] = 1 then
           Inc(Cover[1, I], J);
       end;
     end;
   until Count = 0;
   NoAnswer := False;
   for I := 1 to N do
     if G[I, Mt[0, I]] >= Infinity then
     begin
       NoAnswer := True;
       Break;
     end;
end;

begin
  Assignment;
end.
```

1.13 Maximum Network Flow - FordFulkerson Algorithm

```
{
 Maximum Netword Flow

 Ford Fulkerson Alg.  O(N.E2) Implemantation O(N5)

 Input:
  N: Number of vertices
  C: Capacities (No restrictions)
  S, T: Source, Target(Sink)
 Output:
  F: Flow (SkewSymmetric: F[i, j] = - F[j, i])
  Flow: Maximum Flow

 Reference:
  CLR

 By Behdad
}
program
  MaximumFlow;

const
  MaxN = 100 + 2;

var
  N, S, T : Integer;
  C, F : array [1 .. MaxN, 1 .. MaxN] of Integer;
  Mark : array [1 .. MaxN] of Boolean;
  Flow : Longint;

function  Min (A, B : Integer) : Integer;
begin
  if A <= B then
    Min := A
  else
    Min := B;
end;

function  FDfs (V, LEpsilon : Integer) : Integer;
var
  I, Te : Integer;
begin
  if V = T then
  begin
    FDfs := LEpsilon;
    Exit;
  end;
  Mark[V] := True;
  for I := 1 to N do
    if (C[V, I] > F[V, I]) and not Mark[I] then
    begin
      Te := FDfs(I, Min(LEpsilon, C[V, I] - F[V, I]));
      if Te > 0 then
      begin
        F[V, I] := F[V, I] + Te;
        F[I, V] := - F[V, I];
        FDfs := Te;
        Exit;
      end;
    end;
  FDfs := 0;
end;

procedure FordFulkerson;
var
  Flow2 : Longint;
begin
  repeat
    FillChar(Mark, SizeOf(Mark), 0);
    Flow2 := Flow;
    Inc(Flow, FDfs(S, MaxInt));
  until Flow = Flow2;
end;

begin
  FordFulkerson;
end.
```

1.14 Maximum Network Flow - LiftToFront Algorithm

```
{
 Maximum Network Flow

 LiftToFront Alg.  O(N3)

 Input:
  N: Number of vertices
  C: Capacities (No restrictions)
  S, T: Source, Target(Sink)
 Output:
  F: Flow (SkewSymmetric: F[i, j] = - F[j, i])
  Flow: Maximum Flow

 Reference:
  CLR

 By Behdad
}
program
  MaximumFlow;

const
  MaxN = 100 + 2;

var
  N, S, T : Integer;
  C, F : array [1 .. MaxN, 1 .. MaxN] of Integer;
  Flow : Longint;

  H : array [1 .. MaxN] of Integer;
  E : array [1 .. MaxN] of Longint;
  LNext, LLast : array [0 .. MaxN] of Integer;

function  Min (A, B : Longint) : Longint;
begin
  if A <= B then Min := A else Min := B;
end;

function  CanPush (A, B : Integer) : Boolean;
begin
  CanPush := (C[A, B] > F[A, B]) and (H[A] > H[B]);
end;

procedure Push (A, B : Integer);
var
  Eps : Integer;
begin
  Eps := Min(E[A], C[A, B] - F[A, B]);
  Dec(E[A], Eps);
  Inc(F[A, B], Eps);
  F[B, A] := - F[A, B];
  Inc(E[B], Eps);
end;

procedure Lift(A : Integer);
var
  I : Integer;
begin
  if A in [S, T] then Exit;
  H[A] := 2 * N;
  for I := 1 to N do
    if (C[A, I] > F[A, I]) and (H[A] > H[I] + 1) then
      H[A] := H[I] + 1;
end;

procedure DisCharge (A : Integer);
var
  I : Integer;
begin
  while E[A] > 0 do
  begin
    Lift(A);
    for I := 1 to N do
      if CanPush(A, I) then
        Push(A, I);
  end;
end;

procedure InitializePreFlow;
var
  I, L : Integer;
```

```
begin
  H[S] := N;
  E[S] := MaxLongInt;
  L := 0;
  for I := 1 to N do
  begin
    if I <> S then
      Push(S, I);
    if not (I in [S, T]) then
    begin
      LLast[I] := L;
      LNext[L] := I;
      L := I;
    end;
  end;
end;

procedure MoveToFront (V : Integer);
begin
  LNext[LLast[V]] := LNext[V];
  LLast[LNext[LLast[V]]] := LLast[V];
  LNext[V] := LNext[0];
  LLast[LNext[V]] := V;
  LNext[0] := V;
  LLast[V] := 0;
end;

procedure LiftToFront;
var
  V, BH, I : Integer;
begin
  InitializePreFlow;
  V := LNext[0];
  while V <> 0 do
  begin
    BH := H[V];
    DisCharge(V);
    if BH <> H[V] then
      MoveToFront(V);
    V := LNext[V];
  end;
  Flow := 0;
  for I := 1 to N do
    Inc(Flow, F[S, I]);
end;

begin
  LiftToFront;
end.
```

1.15 Perfect Matching - Augmenting Path Method

```
{
 Maximum Perfect Matching

 My Augmenting Path Alg.  O(N2.E) Implementation O(N4)
 but very near to O(N.E) Implementation O(N3)

 Input:
  G: UnDirected Simple Graph
  N: Number of vertices
 Output:
  Mt: Match of Each Vertex (0 if not matched)
  Matched: size of matching (number of matched edges)

 By Behdad
}

program
  PerfectMaximumMatching;

const
  MaxN = 100 + 2;

var
  N : Integer;
  G : array [1 .. MaxN, 1 .. MaxN] of Boolean;
  Mt : array [1 .. MaxN] of Integer;
  Mark : array [-1 .. MaxN] of Byte;
  Matched : Integer;

function  Max (A, B : Integer) : Integer;
begin
  if A >= B then Max := A else Max := B;
end;

function  MDfs (V : Integer) : Boolean;
var
  I : Integer;
begin
  if V = 0 then begin MDfs := True; Exit; end;
  Mark[V] := 1;
  for I := 1 to N do
    if G[V, I] and (Mark[Mt[I]] = 0) then
    begin
      Inc(Mark[I]);
      if MDfs(Mt[I]) then
      begin
        Dec(Mark[I]);
        Mt[V] := I;
        Mt[I] := V;
        MDfs := True;
        Exit;
      end;
      Dec(Mark[I]);
    end;
  MDfs := False;
end;

procedure AugmentingPath;
var
  I : Integer;
begin
  FillChar(Mark, SizeOf(Mark), 0);
  FillChar(Mt, SizeOf(Mark), 0);
  Mark[-1] := 1;
  for I := 1 to N do
  begin
    if Mt[I] = 0 then
    begin
      Mt[I] := -1;
      if MDfs(I) then
      begin
        Inc(Matched);
        FillChar(Mark, SizeOf(Mark), 0);
        Mark[-1] := 1;
        I := 0;
        Continue;
      end;
      Mt[I] := 0;
    end;
  end;
end;
```

```
begin
  AugmentingPath;
end.
```

2 Planar Graph Algorithms

2.1 Graph Planarity Check - Democron-Malgrange Algorithm

```
{
 Planarity Check

 O(NE) Democron-Malgrange Alg.  Implementation O(N4)

 Input:
  G: UnDirected Simple Graph
  N: Number of vertices
 Output:

 Reference:
  West

 By Behdad
}
program
  PlanarityCheck;

const
  MaxN = 50 + 2;

type
  TSet = set of 0 .. MaxN;
  TBridge = record
    V, A, F : TSet; {Vertices, Adj. Vertices, Faces}
    D : Integer;    {Number of Faces}
  end;
  TVertex = record
    F : TSet;    {Faces}
    E : Boolean; {Embedded}
    B : Integer; {Bridge Number}
  end;

var
  N, BN, FN : Integer; {Number of Vertices, Bridges, Faces}
  G, H : array [1 .. MaxN, 1 .. MaxN] of Boolean;
  E : array [1 .. MaxN, 1 .. MaxN, 1 .. 2] of Integer;
  Br : array [1 .. MaxN] of TBridge;
  Vr : array [1 .. MaxN] of TVertex;

procedure NoSolution;
begin
  Writeln('Graph is not planar');
  Halt;
end;

procedure Found;
begin
  Writeln('Graph is planar');
end;

procedure EmbedEdge (I, J, Aa, Bb : Integer; Fl : Boolean);
begin
  G[I, J] := False; G[J, I] := G[I, J];
  H[I, J] := True ; H[J, I] := H[I, J];
  if Fl then
  begin E[I, J, 1] := Aa; E[I, J, 2] := Bb; E[J, I] := E[I, J]; end;
  with Vr[I] do begin E := True; F := F + [Aa, Bb]; B := 0; end;
  with Vr[J] do begin E := True; F := F + [Aa, Bb]; B := 0; end;
end;

procedure ChangeEdge (I, J, Aa, Bb : Integer);
begin
  if E[I, J, 1] = Aa then E[I, J, 1] := Bb else
  if E[I, J, 2] = Aa then E[I, J, 2] := Bb;
  E[J, I] := E[I, J];
  with Vr[I] do F := F - [Aa] + [Bb];
  with Vr[J] do F := F - [Aa] + [Bb];
end;

procedure UpdateBridges; forward;

procedure Initialize;
var
  I, J : Integer;
  Mark : array [1 .. MaxN] of Boolean;
  function Dfs (V, P : Integer) : Boolean;
  var
```

```
    I : Integer;
    Fl : Boolean;
  begin
    Mark[V] := True;
    for I := 1 to N do
      if G[V, I] then
      begin
        Fl := Mark[I];
        if (Mark[I] and (I <> P)) or (not Mark[I] and Dfs(I, V)) then
        begin
          if Fl then J := I;
          if J <> 0 then EmbedEdge(V, I, 1, 2, True);
          if not Fl and (I = J) then J := 0;
          Dfs := True;
          Exit;
        end;
      end;
    Dfs := False;
  end;

begin
  BN := 0;
  FillChar(Mark, SizeOf(Mark), 0);
  Dfs(1, 0);
  FN := 2;
  UpdateBridges;
end;

procedure UpdateBridgeFaces (B : Integer);
var
  I : Integer;
begin with Br[B] do begin
  F := [1 .. N];
  for I := 1 to N do if I in A then F := F * Vr[I].F;
  D := 0; for I := 1 to N do if I in F then Inc(D);
end; end;

procedure UpdateBridges;
var
  Mark : array [1 .. MaxN] of Boolean;
  procedure FindBridgeVertices (V : Integer);
  var
    I : Integer;
  begin
    Include(Br[BN].V, V);
    Vr[V].B := BN;
    Mark[V] := True;
    for I := 1 to N do
      if G[V, I] then
        if not Vr[I].E and not Mark[I] then
          FindBridgeVertices(I)
        else
          if Vr[I].E then
            Include(Br[BN].A, I);
  end;
var
  I, J : Integer;
begin
  FillChar(Mark, SizeOf(Mark), 0);
  for I := 1 to N do with Vr[I] do
    if not E and (B = 0) then
    begin
      Inc(BN);
      FillChar(Br[BN], SizeOf(Br[BN]), 0);
      FindBridgeVertices(I);
      UpdateBridgeFaces(BN);
    end;
end;

procedure RelaxBridge (B : Integer);
var
  Mark : array [1 .. MaxN] of Boolean;
  X, Y, J : Integer;
  function  FindPath (V : Integer) : Boolean;
  var
    I : Integer;
  begin
    Mark[V] := True;
    for I := 1 to N do
      if not Mark[I] and G[V, I] and ((I in Br[B].A) or
        ((I in Br[B].V) and FindPath(I))) then
      begin
        if not Mark[I] then Y := I;
```

```
        EmbedEdge(V, I, J, FN, True); FindPath := True;
        Exit;
      end;
    FindPath := False;
  end;
var
  I, X2 : Integer;
  S : TSet;
begin
  FillChar(Mark, SizeOf(Mark), 0);
  Inc(FN);
  for I := 1 to N do if I in Br[B].F then begin J := I; Break; end;
  for I := 1 to N do if I in Br[B].A then begin X := I; FindPath(X); Break; end;
  for I := 1 to N do if I in Br[B].V then Vr[I].B := 0;
  Br[B] := Br[BN]; Dec(BN);
  X2 := X;
  repeat
    for I := 1 to N do
      if H[X, I] and ((E[X, I, 1] = J) or (E[X, I, 2] = J)) then
        Break;
    ChangeEdge(X, I, J, FN);
    X := I;
  until X = Y;
  EmbedEdge(X2, Y, J, FN, False);
  for I := 1 to BN do if J in Br[I].F then UpdateBridgeFaces(I);
  UpdateBridges;
end;

procedure RelaxEdge (X, Y : Integer);
var
  I, J, X2 : Integer;
begin
  for J := FN downto 0 do
    if J in Vr[X].F * Vr[Y].F then
      Break;
  if J = 0 then NoSolution;
  Inc(FN);
  X2 := X;
  repeat
    for I := 1 to N do
      if H[X, I] and ((E[X, I, 1] = J) or (E[X, I, 2] = J)) then
        Break;
    ChangeEdge(X, I, J, FN);
    X := I;
  until X = Y;
  EmbedEdge(X2, Y, J, FN, True);
  for I := 1 to BN do if J in Br[I].F then UpdateBridgeFaces(I);
end;

procedure Planar;
var
  I, J : Integer;
  procedure FindMin (var I : Integer);
  var
    J : Integer;
  begin
    I := 1;
    for J := 1 to BN do
      if Br[I].D > Br[J].D then
        I := J;
    if BN = 0 then
      I := 0;
  end;

begin
  BN := 0;
  Initialize;
  repeat
    FindMin(I);
    if I <> 0 then
    begin
      if Br[I].D = 0 then NoSolution;
      RelaxBridge(I);
    end;
    for I := 1 to N do
      for J := 1 to I - 1 do
        if G[I, J] and Vr[I].E and Vr[J].E then
          RelaxEdge(I, J);
  until BN = 0;
end;

begin
  Planar;
```

```
  Found;
end.
```

2.2 Faces Of Planar Graph - Greedy Algorithm

```
{
 Find Faces Of A Planar Graph

 Greedy Alg.  O(N3)

 Input:
  N: Number of vertices
  G[I]: List of vertices adjacent to I in counter-clockwise order
  D[I]: Degree of vertex I
  P[I]: Position of Vertex P
 Output:
  Edge[I][J]: Number of the face that lies to the left of edge (I,J)
  FaceNum: Number of faces, including the outer one
  FaceDeg[I]: Number of vertices on face I
  Face[I]: Vertices of face I

 Notes:
   G should represent a valid embedding of a planar connected graph
   A, B that FindFaces accepts represent these:
     A = Index of point with minimum X (and with minimum Y within them)
     B = The rightmost point that A is connected to
     These are used to set the face number of outer face of graph to 1
     Pass 0, 0 to ignore these
   Edge[I][J] <> Edge[J][I]
   FindOuterFaceEdge finds A and B for FindFaces

 By Behdad
}
program
  Faces;

const
  MaxN = 30 + 1;

var
  N: Integer;
  G, Edge: array [1 .. MaxN, 0 .. MaxN] of Integer;
  D, FaceDeg: array [1 .. MaxN] of Integer;
  Face: array [1 .. MaxN * 3, 0 .. MaxN * 6] of Integer;
  FaceNum: Integer;
  P: array [1 .. MaxN] of record X, Y: Integer; end;

procedure FindFace (A, B: Integer);
var
  I, S, T: Integer;
begin
  Inc(FaceNum);
  S := A;
  T := B;
  FaceDeg[FaceNum] := 0;
  Face[FaceNum, 0] := A;
  repeat
    Inc(FaceDeg[FaceNum]);
    Face[FaceNum, FaceDeg[FaceNum]] := B;
    Edge[A, B] := FaceNum;
    for I := 1 to D[b] do
      if A = G[B, I] then
        Break;
    A := B;
    B := G[B, I - 1];
  until (A = S) and (B = T);
end;

procedure FindFaces (A, B: Integer);
var
  I, J: Integer;
begin
  for I := 1 to N do
    G[I, 0] := G[I, D[I]];
  FillChar(Edge, SizeOf(Edge), 0);
  for I := 1 to N do
    for J := 1 to D[I] do
      Edge[I, G[I, J]] := -1;
  FaceNum := 0;
  if (A > 0) and (B > 0) then
    FindFace(B, A);
  for I := 1 to N do
    for J := 1 to N do
      if Edge[I, J] = -1 then
        FindFace(I, J);
end;
```

```
procedure FindOuterFaceEdge (var A, B: Integer);
var
  I, J: Integer;
begin
  A := 1;
  for I := 2 to N do
    if (P[I].X < P[A].X) or ((P[I].X = P[A].X) and (P[I].Y <= P[A].Y)) then
      A := I;
  B := G[A, 1];
  for I := 2 to D[A] do
  begin
    J := G[A, I];
    if (P[J].X-P[A].X) * (P[B].Y-P[A].Y) > (P[J].Y-P[A].Y) * (P[B].X-P[A].X) then
      B := J;
  end;
end;

var
  A, B: Integer;

begin
  FindOuterFaceEdge(A, B);
  FindFaces(A, B);
end.
```

2.3 Sort Edges Of Planar Graph - Quick Sort

```
{
 Sort edges of a planar graph

 Quick Sort  O(E.LgN)

 Input:
  N: Number of vertices
  G[I]: List of vertices adjacent to I
  D[I]: Degree of vertex I
  P[I]: Position of Vertex P
 Output:
  G[I]: List of vertices adjacent to I in counter-clockwise order

 Notes:
   G should be planar with representation P

 By Ali
}
program
  Faces;

type
  Point = record
    x, y: Integer;
  end;

const
  MaxN = 100 + 1;

var
  N: Integer;
  G: array [1 .. MaxN, 0 .. MaxN] of Integer;
  D: array [1 .. MaxN] of Integer;
  P: array [1 .. MaxN] of Point;

  Tab: Array[1..MaxN] of Extended;
  Pair: Array[1..MaxN] of Integer;

function Comp(a, b: Extended): Integer;
begin
  if abs(a - b) < 1e-8 then Comp := 0
  else
    if a > b then  Comp := 1 else Comp := -1;
end;

function Angle(P, Q: Point): Extended;
var
  a: Extended;
begin
  Q.x := Q.x - P.x;  Q.y := Q.y - P.y;
  if Comp(Q.x, 0) = 0 then
    if Q.y > 0 then
      Angle := Pi / 2
    else
      Angle := 3 * Pi / 2
  else
  begin
    a := ArcTan(Q.y / Q.x);
    if Q.x < 0 then
      a := a + Pi;
    if a < 0 then
      a := a + 2 * Pi;
    Angle := a;
  end;
end;

procedure Swap(var a, b: Integer);
var
  c: Integer;
begin
  c := a; a := b; b := c;
end;

procedure Sort(l, r: Integer);
var
  i, j: integer;
  x, y: Extended;
begin
  i := l; j := r; x := Tab[(l+r) DIV 2];
  repeat
    while Tab[i] < x do i := i + 1;
```

```
    while x < Tab[j] do j := j - 1;
    if i <= j then
    begin
      y := Tab[i]; Tab[i] := Tab[j]; Tab[j] := y;
      Swap(Pair[i], Pair[j]);
      i := i + 1; j := j - 1;
    end;
  until i > j;
  if l < j then Sort(l, j);
  if i < r then Sort(i, r);
end;

procedure SortEdges;
var
  i, j: Integer;
begin
  for i := 1 to N do  begin
    for j := 1 to D[i] do  begin
      Pair[j] := G[i, j];
      Tab[j] := Angle(P[i], P[Pair[j]]);
    end;
    if D[i] > 1 then
      Sort(1, D[i]);
    for j := 1 to D[i] do  G[i, j] := Pair[j];
  end;
end;

begin
  SortEdges;
end.
```

3 Linear Equations

3.1 2Satisfiability Problem - Dfs Method

```
{
 2Satisfiability Problem

 DFS Method  O(N3)

 Input:
  M: Number of clauses
  N: Number of Xs
  Pairs[I]: 1<=I<=M Literals of clause I, -x means ~x (not x)
 Output:
  Value[I]: Value of x[i] in a satisfiability condition
  NoAnswer: System is not satisfiable

 By Behdad
}
program
  TwoSat;

const
  MaxN = 100 + 2;
  MaxM = 100 + 2;

var
  M, N : Integer;
  Pairs : array [1 .. MaxM, 1 .. 2] of Integer;
  Value : array [1 .. MaxN] of Boolean;
  NoAnswer : Boolean;

  Mark, MarkBak: array [1 .. MaxN] of Boolean;

function  DFS (V : Integer) : Boolean;
var
  I, J : Integer;
begin
  if Mark[Abs(V)] then
  begin
    DFS := Value[Abs(V)] xor (V < 0);
    Exit;
  end;
  Mark [Abs(V)] := True;
  Value[Abs(V)] := (V > 0);
  for I := 1 to 2 do
    for J := 1 to M do
      if (Pairs[J, I] = -V) and not DFS(Pairs[J, 3 - I]) then
      begin
        DFS := False;
        Exit;
      end;
  DFS :=  True;
end;

procedure TwoSatisfy;
var
  I: Integer;
begin
  FillChar(Mark,SizeOf(Mark),0);
  MarkBak := Mark;
  NoAnswer := False;
  for I := 1 to N do
    if not Mark[I] then
      if not DFS(-I) then
      begin
        Mark := MarkBak;
        if not DFS(I) then
        begin
          NoAnswer := True;
          Break;
        end;
      end;
end;

begin
  TwoSatisfy;
end.
```

3.2 System of Linear Equations - Deletion Method

```
{
 System of Linear Equations

 Deletion Method  O(N3)

 Input:
   M: Number of Equations
   N: Number of Xs
   Dij: 1<=i<=M 1<=j<=N Coefficient of Xj in Equation i
   Di(n+1): Equation i's constant value
 Output:
   NoAnswer: System does not have unique answer
   XFound[i]: Xi is unique
   X[i]: Xi (has mean iff XFound[i])
   XFounds: Number of unique Xs (== N => System has unique solution)

 By Behdad
}
program
  EquationsSystem;

const
  MaxM = 100 + 2;
  MaxN = 100 + 2;
  Epsilon = 1E-6;

var
  M, N : Integer;
  D : array [1 .. MaxM, 1 .. MaxN + 1] of Real;
  X : array [1 .. MaxN + 1] of Extended;
  XFound : array [1 .. MaxN] of Boolean;
  XFounds : Integer;
  NoAnswer : Boolean;

function CheckZero (X : Real) : Real;
begin
  if Abs(X) <= Epsilon then
    CheckZero := 0
  else
    CheckZero := X;
end;

procedure IncreaseZeroCoefficients;
var
  I, J, P, Q: Integer;
  R: Real;
begin
  for I := 1 to M do
  begin
    for J := 1 to N + 1 do
      if not XFound[J] and (D[I, J] <> 0) then
        Break;
    if J <= N then
    begin
      for P := 1 to M do
        if (P <> I) and (D[P, J] <> 0) then
        begin
          R := CheckZero(D[P, J] / D[I, J]);
          if R <> 0 then
            for Q := 1 to N + 1 do
              if Q <> J then
                D[P, Q] := CheckZero(D[P, Q] - (D[I, Q] * R));
          D[P, J] := 0;
        end;
      XFound[J] := True;
    end;
  end;
end;

procedure ExtractUniques;
var
  I, J, P, Q : Integer;
begin
  for I := 1 to M do
  begin
    P := 0;
    for J := 1 to N do
      if (D[I, J] <> 0) then
      begin
        Inc(P);
        Q := J;
```

```
      end;
    if (P = 0) and (D[I, N + 1] <> 0) then
    begin
      NoAnswer := True;
      Exit;
    end
    else
    if P = 1 then
    begin
      X[Q] := D[I, N + 1] / D[I, Q];
      XFound[Q] := True;
      Inc(XFounds);
    end;
  end;
end;

procedure SolveSystem;
begin
  NoAnswer := False;
  XFounds := 0;
  FillChar(XFound, Sizeof(XFound), 0);
  IncreaseZeroCoefficients;
  FillChar(XFound, Sizeof(XFound), 0);
  ExtractUniques;
end;

begin
  SolveSystem;
end.
```

4 String Functions

4.1 SubString Matching - First Match - KMP Algorithm

```
{
 String Matching - First Match

 K.M.P. Algorithm  O(N)

 Input:
  S: Haystack string
  Q: Needle string
  SL, QL: The length of two strings above
 Output:
  Return Value: Position of first match of Q in S, 0 = Not Found

 Reference:
  Creative, p154

 By Ali
}
program
  StringMatch;

const
  MaxL = 1000 + 1;

var
  S, Q: array[1 .. MaxL] of Char;
  SL, QL: Integer;

  Next: array[1 .. MaxL] of Integer;

procedure ComputeNext;
var
  i, j: Integer;
begin
  Next[1] := -1;
  Next[2] := 0;
  for i := 3 to QL do
  begin
    j := Next[i - 1] + 1;
    while (j > 0) and (Q[i - 1] <> Q[j]) do
      j := Next[j] + 1;
    Next[i] := j;
  end;
end;

function KMP: Integer;
var
  i, j, Start: Integer;
begin
  ComputeNext;
  j := 1;  i := 1; Start := 0;
  while (i <= SL) and (Start = 0) do
  begin
    if S[i] = Q[j] then  begin
      Inc(i);
      Inc(j);
    end
    else  begin
      j := Next[j] + 1;
      if j = 0 then  begin
        j := 1;
        Inc(i);
      end;
    end;
    if j = QL + 1 then  Start := i - QL;
  end;
  KMP := Start;
end;

begin
  Writeln(KMP);
end.
```

4.2 SubString Matching - All Matches - KMP Algorithm

```
{
 String Matching - All Matches

 K.M.P. Algorithm  O(N)

 Input:
  S: Haystack string
  Q: Needle string
  SL, QL: The length of two strings above
 Output:
  Pos: Index of all occurences of Q in S
  PosNum: Number of occurences of Q in S

 Reference:
  Creative, p154

 By Ali
}
program
  StringAllMatch;

const
  MaxL = 1000 + 1;

var
  S, Q: array[1 .. MaxL] of Char;
  SL, QL: Integer;
  Pos: array[1 .. MaxL] of Integer;
  PosNum: Integer;

  Next: array[1 .. MaxL] of Integer;

procedure ComputeNext;
var
  i, j: Integer;
begin
  Next[1] := -1;
  Next[2] := 0;
  for i := 3 to QL + 1 do
  begin
    j := Next[i - 1] + 1;
    while (j > 0) and (Q[i - 1] <> Q[j]) do
      j := Next[j] + 1;
    Next[i] := j;
  end;
end;

procedure AllKMP;
var
  i, j: Integer;
begin
  ComputeNext;
  PosNum := 0;
  j := 1;  i := 1;
  while (i <= SL) do
  begin
    if S[i] = Q[j] then  begin
      Inc(i);
      Inc(j);
    end
    else  begin
      j := Next[j] + 1;
      if j = 0 then  begin
        j := 1;
        Inc(i);
      end;
    end;
    if j = QL + 1 then
    begin
      Inc(PosNum);
      Pos[PosNum] :=  i - QL;
      j := Next[j] + 1;
      if j = 0 then  begin
        j := 1;
        Inc(i);
      end;
    end;
  end;
end;

begin
```

```
  AllKMP;
end.
```

5 Sorting And Searching

5.1 Quick Sort - Static Median

```
{
 Quick Sort Algorithm

 O(NLgN)

 Input:
  A: Array of integer
  L, R: The range to be sorted
 Output:
  Ascending sorted list
 Notes:
  L must be <= R

 Reference:
  TAOCP

 By Knuth
}

procedure Sort(l, r: Integer);
var
  i, j, x, y: integer;
begin
  i := l; j := r; x := a[(l+r) DIV 2];
  repeat
    while a[i] < x do i := i + 1;
    while x < a[j] do j := j - 1;
    if i <= j then
    begin
      y := a[i]; a[i] := a[j]; a[j] := y;
      i := i + 1; j := j - 1;
    end;
  until i > j;
  if l < j then Sort(l, j);
  if i < r then Sort(i, r);
end;
```

5.2 Heap Sort (ADT) - NonRecursive Methods

```
{
 Heap Sort Algorithm

 O(NLgN)
 Input:
  A: array of integer
  N: number of integers
 Output:
  Ascending Sorted list
 Notes:
  Heap is MaxTop

 Reference:
  FCS

 By Behdad
}
program
  HeapSort;

const
  MaxN = 32000;

var
  N : Integer;
  A : array [1 .. MaxN] of Integer;
  HSize : Integer;

function  BubbleUp (V : Integer) : Integer;
var
  Te : Integer;
begin
  while (V > 1) and (A[V] > A[V div 2]) do
  begin
    Te := A[V]; A[V] := A[V div 2]; A[V div 2] := Te;
    V := V div 2;
  end;
  BubbleUp := V;
end;

function  BubbleDown (V : Integer) : Integer;
var
  Te : Integer;
  C : Integer;
begin
  while 2 * V <= HSize do
  begin
    C := 2 * V;
    if (C < HSize) and (A[C] < A[C + 1]) then
      Inc(C);
    if A[V] < A[C] then
    begin
      Te := A[V]; A[V] := A[C]; A[C] := Te;
      V := C;
    end
    else
      Break;
  end;
  BubbleDown := V;
end;

function  Insert (K : Integer) : Integer;
begin
  Inc(HSize);
  A[HSize] := K;
  Insert := BubbleUp(HSize);
end;

function  Delete (V : Integer) : Integer;
begin
  Delete := A[V];
  A[V] := A[HSize];
  Dec(HSize);
  if BubbleUp(V) = V then
    BubbleDown(V);
end;

function  DeleteMax : Integer;
var
  Te : Integer;
begin
```

```
  DeleteMax := A[1];
  Te := A[1]; A[1] := A[HSize]; A[HSize] := Te;
  Dec(HSize);
  BubbleDown(1);
end;

function ChangeKey (V, K : Integer) : Integer;
begin
  A[V] := K;
  ChangeKey := BubbleDown(BubbleUp(V));
end;

procedure Heapify (Count : Integer);
var
  I : Integer;
begin
  HSize := Count;
  for I := N div 2 downto 1 do
    BubbleDown(I);
end;

procedure Sort (Count : Integer);
begin
  Heapify(Count);
  while HSize > 0 do
    DeleteMax;
end;

begin
  Sort(N);
end.
```

5.3 Binary Search - NonRecursive

```
{
 Binary Search

 O(LgN)

 Input:
  X: Array of elements in ascending sorted order
  L: 1
  R: Number of elements
  Z: Key to find
 Output:
  Return Value: Index of Z in X (-1 = Not Found)

 Reference:
  Creative, p121

 By Ali
}
program
  BinarySearch;

const
  MaxN = 10000 + 2;

var
  X: array [1 .. MaxN] of Integer;

function BSearch(L, R: Integer; Z: Integer): Integer;
var
  Mid: Integer;
begin
  while L < R do
  begin
    Mid := (L + R) div 2;
    if Z > X[Mid] then  L := Mid + 1
    else
      R := Mid;
  end;
  if X[L] = Z then
    BSearch := L
  else
    BSearch := -1;
end;

begin
  Writeln(BSearch(1, 3, 7682));
end.
```

6 Data Structors

6.1 Union-Find ADT - Simple Unions

```
{
 Union-Find Data Structure

 Operations:
  Init(N): Initialize list for use with N records
  Union(X, Y): Merge groups of X and Y
  Find(X): Return the group of a X

 Reference:
  Creative, p80-83

 By Ali
}
program
  UnionFind;

const
  MaxN = 10000 + 2;

var
  List: array[1 .. MaxN] of record
    P: Integer; {Parent (= 0 for roots)}
    S: Integer; {Size of group (= 0 for non-roots)}
  end;

procedure Init(N: Integer);
var
  i: Integer;
begin
  FillChar(List, SizeOf(List), 0);
  for i := 1 to N do
    List[i].S := 1;
end;

function Find(a: Integer): Integer;
var
  i, j: Integer;
begin
  i := a;
  while List[i].P <> 0 do
    i := List[i].p;
  while (a <> i) do
  begin
    j := List[a].P;
    List[a].P := i;
    a := j;
  end;
  Find := i;
end;

procedure Union(a, b: Integer);
var
  i, j: Integer;
begin
  a := Find(a);
  b := Find(b);
  if (a = b) then
    Exit;
  if List[b].S > List[a].S then
  begin
    i := a;  a := b;  b := i;
  end;
  Inc(List[a].S, List[b].S);
  List[b].S := 0;
  List[b].P := a;
end;

begin
  Init(2);
  Union(1, 2);
  Writeln(Find(2));
end.
```

6.2 Huge Integer Numbers - Library

```
{
 Huge Integer Numbers

 By Mehran
}
unit numunit;
interface
const
  CBase = 10;
  MaxN = 100;
type
  TInt = longint;
  TDigit = integer;
  TWorkDigit = longint;
  TNumber = record
    n:integer;
    a:array [0..MaxN] of TDigit;
  end;

function compareNumber(const a, b:TNumber):integer; (* approved *)
procedure assignInt(var a:TNumber; n:TInt); (* approved *)
function zeroNumber(const a:TNumber):boolean; (* approved *)
function toIntNumber(const a:TNumber):TInt; (* approved *)

function addNumber(const a,b:TNumber; var res:TNumber):boolean; (* approved *)
function mulNumber(const a,b:TNumber; var res:TNumber):boolean; (* approved *)
function subNumber(const a,b:TNumber; var res:TNumber):boolean; (* approved *)
function divNumber(const a,b:TNumber; var d, m:TNumber):boolean;

function addIntNumber(const a:TNumber; b:TInt; var res:TNumber):boolean; (* approved *)
function mulIntNumber(const a:TNumber; b:TInt; var res:TNumber):boolean; (* approved *)
function subIntNumber(const a:TNumber; b:TInt; var res:TNumber):boolean; (* approved *)

function shlNumber(const a:TNumber; b:integer; var res:TNumber):boolean; (* approved *)
procedure shrNumber(const a:TNumber; b:integer; var res:TNumber); (* approved *)

function powNumber(const a:TNumber; pow:integer; var res:TNumber):boolean; (* approved *)
procedure sqrtNumber(const a:TNumber; var res:TNumber);

implementation
function maxInteger(a,b:integer):integer;
begin
  if a > b then maxInteger := a else maxInteger := b;
end;
function minInteger(a,b:integer):integer;
begin
  if a < b then minInteger := a else minInteger := b;
end;

function compareNumber(const a, b:TNumber):integer;
var
  i:integer;
begin
  if a.n <> b.n then
    compareNumber := a.n - b.n
  else begin
    i := a.n - 1;
    while i >= 0 do begin
      if a.a[i] <> b.a[i] then
        break;
      dec(i);
    end;
    compareNumber := a.a[i] - b.a[i];
  end;
end;

procedure assignInt(var a:TNumber; n:TInt);
begin
   a.n := 0;
   while n > 0 do begin
     a.a[a.n] := n mod CBase;
     n := n div CBase;
     inc(a.n);
   end;
   a.a[a.n] := 0;
end;

function zeroNumber(const a:TNumber):boolean;
begin
  zeroNumber := a.n = 0;
end;
```

```
function toIntNumber(const a:TNumber):TInt;
var
  return:TInt;
  i:integer;
begin
  return := 0;
  i := a.n;
  while i > 0 do begin
    dec(i);
    return := return * CBase + a.a[i];
  end;
  toIntNumber := return;
end;

function addNumber(const a,b:TNumber; var res:TNumber):boolean;
var
  i:integer;
  c:TDigit;
begin
  res.n := maxInteger(a.n,b.n);
  c := 0;
  i := 0;
  while i < res.n do begin
    inc(c,a.a[minInteger(i,a.n)] + b.a[minInteger(i,b.n)]);
    res.a[i] := c mod CBase;
    c := c div CBase;
    inc(i);
  end;
  if c <> 0 then begin
    res.a[res.n] := 1;
    inc(res.n);
  end;
  if res.n = maxN then
    addNumber := false
  else begin
    res.a[res.n] := 0;
    addNumber := true;
  end;
end;

function mulNumber(const a,b:TNumber; var res:TNumber):boolean;
var
  c:TWorkDigit; (* can be integer *)
  i,j,max:integer;
begin
  res.n := a.n + b.n - 1;
  if res.n >= maxN then begin
    mulNumber := false;
    exit;
  end;
  c := 0;
  for i := 0 to res.n-1 do begin
    for j := 0 to i do
      inc(c,TWorkDigit(a.a[minInteger(j,a.n)]) * b.a[minInteger(i-j,b.n)]);
    res.a[i] := c mod CBase;
    c := c div CBase;
  end;
  if c > 0 then begin
    res.a[res.n] := c;
    inc(res.n);
  end;
  if res.n = maxN then
    mulNumber := false
  else begin
    res.a[res.n] := 0;
    mulNumber := true;
  end;
end;

function subNumber(const a,b:TNumber; var res:TNumber):boolean;
var
  i:integer;
  c:TDigit;
begin
  res.n := maxInteger(a.n,b.n);
  c := 0;
  i := 0;
  while i < res.n do begin
    inc(c,a.a[minInteger(i,a.n)] - b.a[minInteger(i,b.n)]);
    if c < 0 then begin
      res.a[i] := c + CBase;
      c := -1;
```

```
    end else begin
      res.a[i] := c;
      c := 0;
    end;
    inc(i);
  end;
  if c = -1 then begin
    c := 1;
    i := 0;
    while i < res.n do begin
      res.a[i] := (CBase - 1) - res.a[i] + c;
      if res.a[i] = CBase then begin
        res.a[i] := 0;
        c := 1;
      end else
        c := 0;
      inc(i);
    end;
    subNumber := false
  end else begin
    subNumber := true;
  end;
  res.a[res.n] := 0;
  while (res.n > 0) and (res.a[res.n-1] = 0) do
    dec(res.n);
end;

function divNumber(const a,b:TNumber; var d, m:TNumber):boolean;
var
  i,j:integer;
  c:TWorkDigit;
  diff2,diff:TDigit;
begin
  if zeroNumber(b) then begin
    divNumber := false;
    exit;
  end;
  if a.n < b.n then begin
    assignInt(d,0);
    m := a;
  end;
  m := a;
  i := a.n - b.n + 1;
  d.n := i;
  d.a[i] := 0;
  while i > 0 do begin
    dec(i);
    d.a[i] := 0;
    for j := i+b.n downto i do begin
      diff := m.a[j] - b.a[j-i];
      if diff <> 0 then
        break;
    end;
    while diff >= 0 do begin
      diff := 0;
      c := 0;
      inc(d.a[i]);
      for j := i to i+b.n do begin
        inc(c,b.a[j-i]-m.a[j]);
        if c < 0 then begin
          m.a[j] := -c;
          c := 0;
        end else if c mod CBase = 0 then begin
          m.a[j] := 0;
          c := c div CBase;
        end else begin
          m.a[j] := CBase - c mod CBase;
          c := c div CBase + 1;
        end;
        diff2 := m.a[j] - b.a[j-i];
        if diff2 <> 0 then
          diff := diff2;
      end;
    end;
    while (m.n > 0) and (m.a[m.n-1] = 0) do
      dec(m.n);
  end;
  while (d.n > 0) and (d.a[d.n-1] = 0) do
    dec(d.n);
  divNumber := true;
end;

function addIntNumber(const a:TNumber; b:TInt; var res:TNumber):boolean;
```

```
var
  i:integer;
begin
  res.n := a.n;
  i := 0;
  while i < res.n do begin
    inc(b,a.a[i]);
    res.a[i] := b mod CBase;
    b := b div CBase;
    inc(i);
  end;
  while b > 0 do  begin
    res.a[res.n] := b mod CBase;
    b := b div CBase;
    inc(res.n);
    if res.n = maxN then begin
      addIntNumber := false;
      exit;
    end;
  end;
  res.a[res.n] := 0;
  addIntNumber := true;
end;

function mulIntNumber(const a:TNumber; b:TInt; var res:TNumber):boolean;
var
  i:integer;
  c:TInt;
begin
  res.n := a.n;
  c := 0;
  i := 0;
  while i < res.n do begin
    inc(c,b*a.a[i]);
    res.a[i] := c mod CBase;
    c := c div CBase;
    inc(i);
  end;
  while c > 0 do  begin
    res.a[res.n] := c mod CBase;
    c := c div CBase;
    inc(res.n);
    if res.n = maxN then begin
      mulIntNumber := false;
      exit;
    end;
  end;
  res.a[res.n] := 0;
  mulIntNumber := true;
end;

function subIntNumber(const a:TNumber; b:TInt; var res:TNumber):boolean;
var
  i:integer;
begin
  res.n := a.n;
  i := 0;
  while i < res.n do begin
    dec(b,a.a[i]);
    if b < 0 then begin
      res.a[i] := -b;
      b := 0;
    end else if b mod CBase = 0 then begin
      res.a[i] := 0;
      b := b div CBase;
    end else  begin
      res.a[i] := CBase - b mod CBase;
      b := b div CBase + 1;
    end;
    inc(i);
  end;
  if b > 0 then begin
    subIntNumber := false;
    exit;
  end;
  res.a[res.n] := 0;
  subIntNumber := true;
  while  (res.n > 0) and (res.a[res.n-1] = 0) do
    dec(res.n);
end;

function shlNumber(const a:TNumber; b:integer; var res:TNumber):boolean;
var
```

```
  i:integer;
begin
  res.n := a.n + b;
  if res.n >= Maxn then
    shlNumber := false
  else begin
    for i := 0 to a.n do
      res.a[i+b] := a.a[i];
    for i := 0 to b-1 do
      res.a[i] := 0;
    shlNumber := true;
  end;
end;

procedure shrNumber(const a:TNumber; b:integer; var res:TNumber);
var
  i:integer;
begin
  if b > a.n then
    b := a.n;
  res.n := a.n - b;
  for i := a.n downto b do
    res.a[i-b] := a.a[i];
end;

function powNumber(const a:TNumber; pow:integer; var res:TNumber):boolean;
var
  temp:TNumber;
  i:integer;
begin
  powNumber := true;
  i := $4000;
  assignInt(res,1);
  while i > 0 do begin
    if not mulNumber(res,res,temp) then begin
      powNumber := false;
      exit;
    end;
    res := temp;
    if i and pow > 0 then begin
      if not mulNumber(res,a,temp) then begin
        powNumber := false;
        exit;
      end;
      res := temp;
    end;
    i := i shr 1;
  end;
end;

procedure sqrtNumber(const a:TNumber; var res:TNumber);
var
  temp1,temp2,temp3:TNumber;
  i:integer;
begin
  if zeroNumber(a) then begin
    res := a;
    exit;
  end;
  res.n := a.n div 2 + 1;
  for i := 0 to res.n-1 do
    res.a[i] := CBase - 1;
  res.a[res.n] := 0;
  while true do begin
    mulNumber(res,res,temp1);
    if compareNumber(temp1,a) <= 0 then
      break;
    addNumber(temp1,a,temp2);
    mulIntNumber(res,2,temp1);
    divNumber(temp2,temp1,res,temp3);
  end;
end;

end.
```

7 Computational Geometry

7.1 Convex Hull - Jordan Gift Wrapping Algorithm

```
{
 Convex Hull - Shortest Polygon

 Jordan Gift Wrapping Algorithm  O(N.K)

 Input:
  N: Number of points
  P[I]: Coordinates of point I

 Output:
  K: Number of points on ConvexHull
  C: Index of points in ConvexHull

 Note:
  It finds the shortest ConvexHull (In case of many points on a line)

 Reference:
  Creative

 By Behdad
}
program
  ConvexHullJordan;

const
  MaxN = 1000 + 2;

type
  Point = record
    X, Y : Integer;
  end;

var
  N, K : Integer;
  P : array [1 .. MaxN] of Point;
  C : array [1 .. MaxN] of Integer;

  Mark : array [1 .. MaxN] of Boolean;

function Left (var A, B, C : Point) : Longint;
begin
  Left := (Longint(A.X)-B.X)*(Longint(C.Y)-B.Y) -
          (Longint(A.Y)-B.Y)*(Longint(C.X)-B.X);
end;

function D (var A, B : Point) : Extended;
begin
  D := Sqrt(Sqr(Longint(A.X) - B.X) + Sqr(Longint(A.Y) - B.Y));
end;

procedure ConvexHull;
var
  I, J: Integer;
begin
  FillChar(Mark, SizeOf(Mark), 0);
  C[1] := 1;
  for I := 1 to N do
    if P[I].Y < P[C[1]].Y then
      C[1] := I
    else
    if (P[I].Y = P[C[1]].Y) and (P[I].X < P[C[1]].X) then
      C[1] := I;
  Mark[C[1]] := True;
  K := 1;
  repeat
    J := C[1];
    for I := 1 to N do then
      if (not Mark[I]) then
        if Left(P[J], P[C[K]], P[I]) < 0 then
          J := I
        else
          if (Left(P[J], P[C[K]], P[I]) = 0) and (D(P[C[K]], P[I]) > D(P[C[K]], P[J])) then
            J := I;
    Inc(K);
    C[K] := J;
  until C[K] = C[1];
end;
```

```
begin
  ConvexHull;
end.
```

7.2 Computational Geometry - Library

```
unit geomunit;
interface
const
  MaxN = 100;
  Epsilon = 1e-6;
type
  TNumber = real;
  TAngle = TNumber;
  TPoint = record
    x,y:TNumber;
  end;
  TLine = record
    a,b,c:TNumber;
  end;
  TCircle = record
    o:TPoint;
    r2:TNumber;
  end;
  TPoly = record
    n:integer;
    p:array [1..MaxN] of TPoint;
  end;

  procedure addPoint(const o:TPoint;var p:TPoint); (* confirmed *)
  procedure subPoint(const o:TPoint;var p:TPoint); (* confirmed *)
  function lineValue(const l:TLine; const p:TPoint):TNumber; (* confirmed *)
  function circleValue(const c:TCircle; const p:TPoint):TNumber; (* confirmed *)
  function comp(const n1,n2:TNumber):integer; (* confirmed *)
  function normal(const l:TLine; var res:TLine):boolean; (* confirmed *)
  function sameLine(l1,l2:TLine):boolean; (* confirmed *)
  function getLine(const p1,p2:TPoint;var l:TLine):boolean; (* confirmed *)
  function intersection(const l1,l2:TLine;var p:TPoint):integer; (* confirmed *)
  function polygonArea(const p:TPoly):TNumber; (* confirmed *)
  function pointDist2(const p1,p2:TPoint):TNumber; (* confirmed *)
  function amoodMonasef(const p1,p2:TPoint;var l:TLine):boolean; (* confirmed *)
  procedure amoodBar(const l:TLine;const p:TPoint;var res:TLine); (* confirmed *)
  procedure movaziBa(const l:TLine;const p:TPoint;var res:TLine); (* confirmed *)
  procedure Rotate(const o, p: TPoint; alpha: TAngle; var res: TPoint); (* confirmed *)
  function lineAng(l: TLine): TAngle; (* confirmed *)
  function angle(const l1, l2: TLine): TAngle; (* confirmed *)
  function solve(a,b,c:TNumber;var x1,x2:TNumber):integer; (* confirmed *)
  function solvePrim(a,b,c:TNumber;var x1,x2:TNumber):integer; (* confirmed *)
  function circleCircle(c1,c2:TCircle; var p1, p2:TPoint):integer; (* confirmed *)
  function lineCircle(const l:TLine; const c:TCircle; var p1, p2:TPoint):integer; (* confirmed *)
  function momasCircle(const p:TPoint; const c:TCircle; var p1, p2:TPoint):integer; (* confirmed *)

implementation

procedure swapNumber(var a,b:TNumber);
var
  c:TNumber;
begin
  c := a;
  a := b;
  b := c;
end;

procedure addPoint(const o:TPoint;var p:TPoint);
begin
  p.x := p.x + o.x;
  p.y := p.y + o.y;
end;

procedure subPoint(const o:TPoint;var p:TPoint);
begin
  p.x := p.x - o.x;
  p.y := p.y - o.y;
end;

function lineValue(const l:TLine; const p:TPoint):TNumber;
begin
  lineValue := l.a*p.x+l.b*p.y+l.c;
end;

function circleValue(const c:TCircle; const p:TPoint):TNumber;
begin
  circleValue := sqr(p.x - c.o.x) + sqr(p.y - c.o.y) - c.r2;
end;

function comp(const n1,n2:TNumber):integer;
var
```

```
  diff:TNumber;
begin
  diff := n1 - n2;
  if abs(diff) < Epsilon then
    comp := 0
  else if diff < 0 then
    comp := -1
  else
    comp := 1;
end;

function samePoint(const p1,p2:TPoint):boolean;
begin
  samePoint := (comp(p1.x, p2.x) = 0) and (comp(p1.y, p2.y) = 0);
end;

function normal(const l:TLine; var res:TLine):boolean;
var
  denom:TNumber;
begin
  denom := sqrt(sqr(l.a) + sqr(l.b));
  if comp(denom,0) = 0 then
    normal := false
  else begin
    res.a := l.a / denom;
    res.b := l.b / denom;
    res.c := l.c / denom;
    normal := true;
  end;
end;

function sameLine(l1,l2:TLine):boolean;
begin
  if normal(l1,l1) and normal(l2,l2) then begin
    sameLine := (comp(l1.a,l2.a) = 0) and
                (comp(l1.b,l2.b) = 0) and
                (comp(l1.c,l2.c) = 0) or
                (comp(l1.a,-l2.a) = 0) and
                (comp(l1.b,-l2.b) = 0) and
                (comp(l1.c,-l2.c) = 0);
  end else begin
    sameLine := false;
  end;
end;

function getLine(const p1,p2:TPoint;var l:TLine):boolean;
begin
  if samePoint(p1,p2) then
    getLine := false
  else begin
    l.a := p1.y - p2.y;
    l.b := p2.x - p1.x;
    l.c := p1.x * p2.y - p2.x * p1.y;
    getLine := true;
  end;
end;

function intersection(const l1,l2:TLine;var p:TPoint):integer;
var
  denom:TNumber;
begin
  denom := l1.a * l2.b - l2.a * l1.b;
  if comp(denom,0) = 0 then begin
    if sameLine(l1,l2) then
      intersection := 2
    else
      intersection := 0;
  end else begin
    intersection := 1;
    p.x := (l1.b * l2.c - l2.b * l1.c) / denom;
    p.y := (l1.c * l2.a - l2.c * l1.a) / denom;
  end;
end;

function polygonArea(const p:TPoly):TNumber;
var
  x1,y1,x2,y2,x3,y3:TNumber;
  i:integer;
  return:TNumber;
begin
  with p do begin
    return := 0;
    x1 := p[1].x;
```

```
    y1 := p[1].y;
    x2 := p[2].x;
    y2 := p[2].y;
    for i := 3 to n do begin
      with p[i] do begin
        x3 := x;
        y3 := y;
      end;
      return := return + (y3-y1)*(x2-x1)-(y2-y1)*(x3-x1);
      x2 := x3;
      y2 := y3;
    end;
    return := abs(return / 2);
  end;
  polygonArea := return;
end;

function pointDist2(const p1,p2:TPoint):TNumber;
begin
  pointDist2 := sqr(p1.x-p2.x)+sqr(p1.y-p2.y);
end;
function amoodMonasef(const p1,p2:TPoint;var l:TLine):boolean;
begin
  l.a := p1.x - p2.x;
  l.b := p1.y - p2.y;
  l.c := (sqr(p2.x) + sqr(p2.y) - sqr(p1.x) - sqr(p1.y)) / 2;
  amoodMonasef := (comp(l.a,0) = 0) or (comp(l.b,0) = 0);
end;

procedure amoodBar(const l:TLine;const p:TPoint;var res:TLine);
begin
  res.a := -l.b;
  res.b := l.a;
  res.c := l.b * p.x - l.a * p.y;
end;

procedure movaziBa(const l:TLine;const p:TPoint;var res:TLine);
begin
  res.a := l.a;
  res.b := l.b;
  res.c := -l.a * p.x - l.b * p.y;
end;

procedure Rotate(const o, p: TPoint; alpha: TAngle; var res: TPoint);
var
  t: TPoint;
begin
  t.x := p.x - o.x;
  t.y := p.y - o.y;
  res.x := t.x * cos(alpha) - t.y * sin(alpha) + o.x;
  res.y := t.y * cos(alpha) + t.x * sin(alpha) + o.y;
end;

function lineAng(l: TLine): TAngle;
var
  A: TAngle;
begin
  if comp(l.b,0) = 0 then
    if l.a < 0 then  lineAng := Pi / 2
    else                 lineAng := 3 * Pi / 2
  else
  begin
    A := ArcTan(-l.a/l.b);
    if l.b < 0 then  A := A + Pi;
    if A < 0 then  A := A + 2 * Pi;
    lineAng := A;
  end;
end;

function angle(const l1, l2: TLine): TAngle;
var
  a: TAngle;
begin
  a := lineAng(l2) - lineAng(l1);
  if a < 0 then
    a := a + 2 * pi;
  angle := A;
end;

function solve(a,b,c:TNumber;var x1,x2:TNumber):integer;
var
  delta :TNumber;
begin
```

```
  delta := sqr(b) - 4 * a * c;
  case comp(delta,0) of
    -1:solve := 0;
    0: begin
      solve := 1;
      x1 := -b/(2*a);
      x2 := x1;
    end;
    1: begin
      solve := 2;
      delta := sqrt(delta);
      x1 := (-b+delta)/(2*a);
      x2 := (-b-delta)/(2*a);
    end;
  end;
end;

function solvePrim(a,b,c:TNumber;var x1,x2:TNumber):integer;
var
  delta :TNumber;
begin
  delta := sqr(b) - a * c;
  case comp(delta,0) of
    -1:solvePrim := 0;
    0: begin
      solvePrim := 1;
      x1 := -b/a;
      x2 := x1;
    end;
    1: begin
      solvePrim := 2;
      delta := sqrt(delta);
      x1 := (-b+delta)/a;
      x2 := (-b-delta)/a;
    end;
  end;
end;

function circleCircle(c1,c2:TCircle; var p1, p2:TPoint):integer;
var
  d2,v:TNumber;
  o:TPoint;
  return:integer;
begin
  o := c1.o;
  subPoint(o,c2.o);
  subPoint(o,c1.o);
  d2 := pointDist2(c1.o,c2.o);
  v := c1.r2 - c2.r2 + d2;
  return :=
  solvePrim(4*d2,-2*c2.o.x*v,sqr(v)-4*sqr(c2.o.y)*c1.r2,p1.x,p2.x);
  solvePrim(4*d2,-2*c2.o.y*v,sqr(v)-4*sqr(c2.o.x)*c1.r2,p2.y,p1.y);
  if (return > 1) and
     ((comp(circleValue(c1,p1),0)<>0) or (comp(circleValue(c2,p1),0)<>0)) then
    swapNumber(p1.x,p2.x);
  addPoint(o,p1);
  addPoint(o,p2);
  circleCircle := return;
end;

function lineCircle(const l:TLine; const c:TCircle; var p1, p2:TPoint):integer;
var
  x1,x2,y1,y2,v:TNumber;
  n,return:integer;
begin
  v := lineValue(l,c.o);
  return :=
  solvePrim(sqr(l.a)+sqr(l.b),l.a*v,sqr(v)-c.r2*sqr(l.b),p1.x,p2.x);
  solvePrim(sqr(l.a)+sqr(l.b),l.b*v,sqr(v)-c.r2*sqr(l.a),p2.y,p1.y);
  addPoint(c.o,p1);
  addPoint(c.o,p2);
  if (return > 1) and
     ((comp(linevalue(l,p1),0)<>0) or (comp(circlevalue(c,p1),0)<>0)) then
    swapNumber(p1.x,p2.x);
  lineCircle := return;
end;

function momasCircle(const p:TPoint; const c:TCircle; var p1, p2:TPoint):integer;
var
  c2:TCircle;
begin
  c2.o := p;
  c2.r2 := pointDist2(c.o,p)-c.r2;
```

```
  case comp(c2.r2,0) of
    -1:momasCircle := 0;
    0:begin
      momasCircle := 1;
      p1 := p;
    end;
    1:begin
      momasCircle := 2;
      circleCircle(c,c2,p1,p2);
    end;
  end;
end;

end.
```