Code for Common Algorithms

Table of Contents

GRAPH ALGORITHMS

Dijkstra's Shortest Path Algorithm

Explained by MoA and VLing

**Goal:
Finding shortest path and path lengths from node Start to all other nodes.

==> The input:
(for this program using files, but can be changed to a procedure)

The program uses a file.
** The file should read:

```
0       1 1000 1000 1000    5
1000    0    7 1000 1000   10
1000 1000    0    5    1 1000
1000 1000 1000    0    6 1000
1000 1000 1000 1000    0    7
1000 1000    5    2 1000    0
```

It doesn't have to be tabbed -- It just looks good here.
Basically, If the shortest you have so far (Short[min]) + the distance from where you are to where you are going (Graph[min,j]) is less than what is there right now (Short[j]) then you want to change it. We have highlighted the line that implements this pseudo-code.

==> The output to the output file:

```
          1      2      3      4      5      6   (To Destination nodes)

1-->      0      3     10      7     11      5   (Shortest Distance)
1-->   1000      1      2      6      3      1   (Previous Node)

2-->   1000      0      7     12      8     10
2-->   1000   1000      2      3      3      2

3-->   1000   1000      0      5      1      8
3-->   1000   1000   1000      3      3      5

4-->   1000   1000     21      0      6     13
4-->   1000   1000      6   1000      4      5

5-->   1000   1000     15      8      0      7
5-->   1000   1000      6      6   1000      5

6-->   1000   1000      8      2      5      0
6-->   1000   1000      6      6      4   1000
```

(From Originating nodes)

==> So what's all this crap? 2 Things.

1. First, to find shortest distance needed to get from one node, START, to any other one.

   How you do this is to run the program/procedure with START as node you want as the to be the first.

   Afterwards, short[i] will give shortest distance from START to i.

2. To find the shortest path from one node, START, to any other.

   PREV[i] will give the previous node to get to i node with the shortest distance.
   i.e. when start = 1, means we are finding the shortest paths and distances from the first node,

```
                1  2   3  4   5  6 (To Destination Nodes)

   Short =      0  3  10  7  11  5
   Prev  =   1000  1   2  6   3  1

   So the shortest path from node start to node 4 is 7
   The path there:
     look at prev for node 4, it is 6, so the previous node was 6
     look at prev for node 6, it is 1, so the previous node was 1

   Therefore the shortest path backward is 4, 6, 1
   Therefore the shortest path in order is 1, 6, 4

   When to stop backtracking?  When prev is the start node you are done

   The only thing you need to do here is to change this into procedure only,
   so long live plagiarism (you have our permission) and, SMILE =)

                                                                              }
Uses Crt;

Const Max=6;                      {Size of matrix. if 6, then matrix 6x6}
      NoConnection=1000;          {Number that indicates no connection}

Var
   Used:Array[1..Max] of Boolean;
   Short:Array[0..Max] of Integer;  {Array with shortest path distance stored}
   Prev:Array[1..Max] of Integer;   {Array with previous node}
   Graph:Array[1..Max,1..Max] of Integer;
   i,j,k,min,start:Integer;
   f, f1:Text;

Begin
          {=================================================}
Output business, don't sweat this
     ClrScr;
     Assign(f1,'Dijkstra.out');
     Rewrite(f1);
     write(f1,'    ');
     for i:=1 to max do
       write(f1,i:7);
     write(f1,'   (Destination nodes)');
     writeln(f1);
     writeln(f1);
          {=================================================}

{=============================================================================}{
This reads input into matrix.  To change it into a procedure, make this
matrix universal or pass it in, as most of you would already know             }
     Assign(f,'Dijkstra.in');
     Reset(f);
     For i:=1 to Max do
         Begin
              For j:=1 to Max do
                  Read(f,Graph[i,j]);
              ReadLn(f);
              Used[i]:=False;
         End;
     Close(f);
{=============================================================================}

          {=================================================}                 {
This program shows shortest path distance and previous node from all.
To do just one, pass in the node needed and set start = node                  }
     For Start:=1 to Max do
         Begin
```

```
        {=================================================}
Main part of Dijkstra's algorithm.
Read Sean Lie's sheet to figure out what this part is if you don't know
              For i:=1 to Max do
                Prev[i]:=NoConnection;
              For i:=1 to Max do
                  Begin
                       Short[i]:=Graph[Start,i];
                       Used[i]:=False;
                       if (short[i] <> noconnection) and (i<>start) then
                         Prev[i]:=Start;
                  End;
              Short[0]:=MaxInt;
              Short[Start]:=0;
              Used[Start]:=True;
              For i:=2 to Max do
                  Begin
                       Min:=0;
                       For j:=1 to Max do
                           If (Short[j]<Short[min]) And Not Used[j] then
                              min:=j;
                       Used[min]:=true;
                     {==============================}
                       For j:=1 to Max do
                           If (Short[min]+Graph[min,j])<Short[j] then
                           begin
                             Short[j]:=Short[min]+Graph[min,j];
                             Prev[j]:=min;
                           end;

                  End;
        {=================================================}
Rest is just output, you can ignore.
              Write(f1,Start,'-->');
              For i:=1 to Max do
                  Write(f1,Short[i]:7);
              Writeln(f1);
              Write(f1,Start,'-->');
              For i:=1 to Max do
                  Write(f1,prev[i]:7);
              WriteLn(f1);
              WriteLn(f1);
        End;
     Close(f1);
End.
```

```
{FLOYDS ALGORITHM WITH CODE TO STORE THE PATHS}
{note maxint is the equivalent of infinite meaning no path exists}

{global constants declaration}
const matrixsize = 100; {size of connection matrix}

{global variable declaration}
var connection_matrix,shortest_path_matrix : array[1..matrixsize,1..matrixsize] of integer;
    path_storage : array[1..matrixsize,1..matrixsize] of byte; {stores shortest paths}

{calculating shortest paths using floyds}
procedure floyds;

  {local variable declaration}
  var i,j,k : byte;

  begin {floyds procedure}

    {copies connection matrix into shortest path matrix}
    shortest_path_matrix:=connection_matrix;

    {clears shortest paths storage}
    fillchar(path_storage,sizeof(path_storage),0);

    {sets initial paths of where you come from}
    for i:=1 to matrixsize do begin

      for j:=1 to matrixsize do begin

        {if connection exists then set initial connection}
        if shortest_path_matrix[i,j]<maxint then begin

          {current shortest path to go to j from i = i (direct connection)}
          path_storage[i,j]:=i;

        end;

      end;

    end;

    {main floyds loop}
    for j:=1 to matrixsize do begin

      for i:=1 to matrixsize do begin

        for k:=1 to matrixsize do begin

          {checks if going from i to j from j to k is possible (to avoid overflowing integer)}
          if (shortest_path_matrix[i,j]<maxint) and (shortest_path_matrix[j,k]<maxint) then begin

            {if (distance from i to j) + (distance from j to k) less than (distance from i to k)}
            if shortest_path_matrix[i,j]+shortest_path_matrix[j,k] < shortest_path_matrix[i,k] then
begin

              {replace (distance from i to k) with (distance from i to j) + (distance from j to k)}
              shortest_path_matrix[i,k]:=shortest_path_matrix[i,j]+shortest_path_matrix[j,k];

              path_storage[i,k]:=path_storage[j,k];

            end; {if (distance...}

          end; {if path exists...}

        end; {for k:=...}

      end; {for i:=...}

    end; {for j:=...}

  end; {floyds procedure}
```

```
{finding shortest path from 'start' to 'finish' using the paths array}
function find_shortest_path (start,finish : integer) : string;

  var out,temp : string; {string to store the path in correct order}
      templocation : integer;

  begin {find_shortest_path function}

    {if path exists from 'start' to 'finish'}
    if shortest_path_matrix[start,finish] < maxint then begin

      {initializes the output to the ending location, ie 'finish'}
      str(finish,out);

      {temporary location to work from, if you want to you can actually modify 'finish' itself}
      templocation:=finish;

      {loop until you found your path}
      repeat

        {convert your current location into a string}
        str(path_storage[start,templocation],temp);

        {add this to your current paths}
        out:=temp+'->'+out;

        {change your current location}
        templocation:=path_storage[start,templocation];

      until templocation=start; {until your current location is your starting location}

    end else out:='';

    {returns path string}
    find_shortest_path:=out;

  end; {find_shortest_path function}
```

```
{Mininmum Spanning Tree Algorithm}
{Mininmum Spanning Tree is used to change a weighted undirected graph into a tree, using the least
amount of weight}
{The Algorithm spits out the minimum amount needed to make the tree but with some tinkering can be
used to output the tree itself as well}
{There are two Algorithms Prim's and Kruskal's}
const nodes=10;

{Prim's Algorithm}
{Prim's Algorithm starts with one tree (initially one node) and builds onto that tree until it
contains all the nodes}

var
   matrix: array [1..nodes,1..nodes] of integer;
   beenthere: array [1..nodes] of boolean;
   paths_to: array [1..nodes] of integer;
   cost, chosennode, shortest: integer;
   a,b:integer; {Counters}
begin
     Read_the_Matrix;
     cost:=0;
     for a:=1 to nodes do beenthere[a]:=false;
     for a:=1 to nodes do paths_to[a]:=maxint;     {Initialization}

     {You can start Prim's on any node to begin with... I'm choosing the first node because it's
easiest}

     beenthere[1]:=true;
     for a:=1 to nodes do paths_to[a]:=matrix[a,1];
     {Because the tree starts at one node, the paths to the tree are initially the edges to that
node}
     for b:=1 to nodes-1 do begin {We need to make nodes-1 connections}
     {We now greedily find the shortest path in our paths_to array}
     {We must also check that the node does not complete a cycle... we check this by making sure we
haven't been at the node before}
          shortest:=maxint;
          for a:=1 to nodes do if (paths_to[a]<shortest) and not (beenthere[a]) then begin
              shortest:=paths_to[a];
              chosennode:=a;
          end;
          inc(cost,shortest);
          beenthere[chosennode]:=true; {Saying that we've been to this node now}
          for a:=1 to nodes do if matrix[a,chosennode]<paths_to[a] then
paths_to[a]:=matrix[a,chosennode];
          {The for loop updates our paths_to node, with the addition of chosennode, any shorter path
to a node from chosennode must be considered}
     end;
     writeln(cost);
end.

{Kruskal's Algorithm}
{Kruskal's algorithm is faster for graphs who have a low density}
{Kruskal's algroithm starts with as many trees as nodes and merges them together}

type tree= set of 1..nodes;
var
   matrix:array [1..nodes,1..nodes] of integer;
   trees: array [1..nodes] of tree;
   shortest, node1, node2, cost:integer;
   done: tree;
   a,b,c:integer; {Counters}
begin
     Read_The_Matrix;
     cost:=0;
     for a:=1 to nodes do done:=done+[a]; {Make a finished set easier to check}
     for a:=1 to nodes do trees[a]:=[a]; {Set every tree to the node itself}
     repeat
          {We must find the smallest edge in the matrix}
```

```
            shortest:=maxint;
            for a:=1 to nodes-1 do
                {Again we have to check not only if the path is the smallest but whether it makes a
cycle}
                {We check this by making the two nodes do not belong on the same tree already}
                for b:=a+1 to nodes do if (matrix[a,b]<shortest) and (trees[a]<>trees[b]) then begin
                    shortest:=matrix[a,b];
                    node1:=a;
                    node2:=b;
                end;
                trees[node1]:=trees[node1]+trees[node2];
                trees[node2]:=trees[node1];              {Merges the two trees}
                for a:=1 to nodes do if a in trees[node1] then trees[a]:=trees[node1];
                inc(cost,shortest);
        until(trees[1]=done);
        writeln(cost);
end.
```

GEOMETRIC ALGORITHMS

```
(* Intersections of lines, unfinished angle and GCD *)

Const
     InF = 'lines2.in';

Type
    Point      = record
                 X,Y : Real;
                 End;

    Line       = record
                 A,B,C : Real;
                 End;

    Intersect = (One,Infinite,None);

Var
   I    :  Integer; {counter}
   Pts  :  Array[1..4] of point; {1,2 are one line; 3,4 are the second}
   P,Q  :  Line; {the two lines}
   T    :  Text;
   NoInt:  Boolean; {a global variable to check if the lines actually
intersect}
   IX   :  Point;      {point of intersection}
   IXT  :  Intersect;  {type of intersection}

Procedure LineTimesK( K : Real; var X : Line );
          Begin
               X.A := X.A*K; X.B := X.B * K; X.C := X.C*K;
          End;

Function GCD(X,Y:Integer):Integer;
Begin
     If Y=0 then GCD:=X Else GCD:=GCD(Y,X MOD Y);
End;


Function  Angle( P, Q : Line ) : Real;
          Var Dot : Real;
          Begin
               Dot := P.A*Q.A + P.B*Q.B;
               Dot := Dot / Sqrt(Sqr(P.A)+Sqr(P.B));
               Dot := Dot / Sqrt(Sqr(Q.A)+Sqr(Q.B));
               Cos
          End;

Procedure ConvertToLine( A,B : Point; var T : Line);
          Begin
               {Convert two points A and B into an equation (T)
               uses Cartesian equation}

               T.B := A.x - B.x;  {from any math text}
               T.A := -(A.Y - B.Y);
               T.C := -(T.A*A.X+T.B*A.Y);
               {WriteLn ( T.A:0:2,'x + ',T.B:0:2,'y + ',T.C:0:2);}
          End;
```

```
Procedure Intersection( A,B : Line; var T : Point; var X : Intersect );
          Var C : Line;
          Begin
               X := One;  {start with only one intersection}

               If A.A = 0 Then Begin C := A; A := B; B := C; End;
               If B.A = 0 Then
                  Begin
                       {B.B must be zero otherwise it is not a line}
                       T.Y := -B.C/B.B;
                       If A.A <> 0 then {if the first equaiton is normal
just skip the manipulation}
                          T.X := -(a.B*T.Y+a.C)/a.A {substitute into B}
                       Else If ((-A.C/A.B) = T.Y) Then X := Infinite
Else X := None;
                       {otherwise check to see if the equations are the
same}
                       Exit; {don't proceed otherwise we do messy things
with the lines}
                  End;
               {same process but for .B instead of .A}
               If A.B = 0 Then Begin C := A; A := B; B := C; End;
               If B.B = 0 Then
                  Begin
                       {B.A must be zero otherwise it is not a line}
                       T.X := -B.C/B.A;
                       If A.B <> 0 then {if the first equaiton is normal
just skip the manipulation}
                          T.Y := -(a.A*T.X+a.C)/a.B {substitute into B}
                       Else If ((-A.C/A.A) = T.X) Then X := Infinite
Else X := None;
                       {otherwise check to see if the equations are the
same}
                       Exit; {don't proceed otherwise we do messy things
with the lines}
                  End;
               {after this no zeros are present in the original
equations}

               LineTimesK( B.A / A.A, A ); {multiply the first line so
the A values are the same
                                            and then the equations can be
subracted}
               A.A := A.A-B.A; A.B := A.B-B.B; A.C := A.C-B.C;
               {subtract equations}
               If ABS(A.B) < 0.0001 Then
                  If ABS(A.C) < 0.0001 Then X := Infinite
                  Else X := None
               Else
               Begin
                    T.Y := -A.C/A.B;  {A is now in the form 0X + bY + c
= 0}
                    T.X := -(B.B*T.Y+B.C)/B.A; {substitute into B}
               End;
          End;

Begin
```

```
    Assign(T,InF); Reset(T);

    While Not Eof(T) do
          Begin
               {read in four points
               two on the first line
               and two on the second line}
               For I := 1 to 4 Do
                   ReadLn(t, Pts[i].x, Pts[i].y );
               ConvertToLine( Pts[1], Pts[2], P);
               ConvertToLine( Pts[3], Pts[4], Q);
               Intersection(P,Q, IX, IXT);
               If IXT = Infinite Then Write( 'Infinite ');
               If IXT = None Then Write( 'Parallel ');
               WriteLn( IX.X:0:2, ',', IX.Y:0:2 );
          End;

    ReadLn;

    close(T);
End.
```

```
(* Convex Hull Algorithm *)
(* Programmer: Mohammed Ajmal *)
(* PROBLEM: Given N points in the plane, find the points (or # of points)
            of the N that form a convex polygon such that ALL N points lie
            on the boundaries of or inside the polygon.

   IDEA: GIFT-WRAPPING ALGORITHM
         Pick a point that is known to be on the convex hull.
         In this case, I pick the one with the largest x-coordinate,
         i.e. the one that is furthest to the right.
         Next, calculate the "angles" between that point and all other points.
         Pick the point that makes the smallest angle with the current point.
         Why? Prove it to yourselves (I can't explain this thru text, guys!)
         At this point, we have to move the point we have just picked to the
         front of the points array so that we don't keep picking it over
         and over and over and over.....
         Continue process until minimum angle is formed with your sentinel
         value (this is just the first point that was on your hull!!!)
   NOTE: - The theta function DOESN'T RETURN THE ACTUAL ANGLE!
           IT RETURNS SOMETHING OF THE SAME ORDER.
         - The way I have coded this algorithm, is so that it returns all the
           points that lie on the hull, as opposed to simply the vertices of
           the hull.
           I have done the following for your ease. There are two lines in
           the Procedure Wrap marked off with this symbol ( --> ). Change
           the "<" sign to ">" sign and you have now rewritten the program
           to return only the vertices! HAPPY? Try the following input to
           see the difference between the two:
           9
           0 0
           0 1
           0 2
           1 0
           1 1
           1 2
           2 0
           2 1
           2 2
         - Text File Format
           First line = # of points in input
           Lines 2..# of points+1 = x-coord y-coord
*)
Uses Crt;
Const
     Max = 100;           (* Max # of points in input *)
     Path = 'convex.in'; (* Input file path *)
Type                      (* A point has an x and y coordinate - both real *)
    Points=Record
                x,y:Real;
            End;
Var
   i,
   NumPoints:Integer;           (* Stores # of points in input *)
   P:Array[1..Max+1] of Points; (* Stores the actual points *)
   F:Text;                      (* Text file for input *)


Function Theta(p1,p2:Points):Real;
(* This function doesn't return the angle just something of the same order. *)
Var
   dx,dy,ax,ay,temp:Real;

Begin
     dx:=p2.x-p1.x; ax:=Abs(dx);
     dy:=p2.y-p1.y; ay:=Abs(dy);
     If (dx=0) AND (dy=0) then temp:=0 Else
        temp:=dy/(ax+ay);
     If dx<0 then temp:=2-temp Else     (* Correction for quadrant *)
        If dy<0 then temp:=4+temp;
     Theta:=temp*90.0;
End; (* Function Theta *)
```

```
Procedure Wrap;
(* Actual Convex Hull finding procedure! *)
Var
   i,
   Hull,              (* Stores the number of points on the hull so far *)
   Min:Integer;       (* Stores the index of the point we're adding to hull *)
   LastMinAngle,      (* The minimum angle for last point added *)
   MinAngle,          (* The minimum angle for the current point *)
   TempAngle,Angle,Temp:Real;
   T:Points;          (* A temporary points variable to allow for swapping *)

Begin
     Min:=1;          (* Initially, assume first point is farthest right *)
     (* Find the point that is farthest right (i.e. greatest x-value *)
     For i:=2 to NumPoints do
         If p[i].x > p[Min].x then Min:=i;
     p[NumPoints+1]:=p[Min];  (* Set up sentinel for ending condition *)
     Hull:=0; MinAngle:=0; TempAngle:=0;
     Repeat
           LastMinAngle:=TempAngle;
           MinAngle:=360.0;
           WriteLn(p[Min].x:0:2,' ',p[Min].y:0:2);
           Inc(Hull);         (* Increase # of points on hull so far *)
           (* Bring current min point to the front *)
           T:=p[Min];              (* SWAP *)
           p[Min]:=p[Hull];        (* SWAP *)
           p[Hull]:=T;             (* SWAP *)
           (* Now, for all points left, perform the algorithm *)
           For i:=Hull+1 to NumPoints+1 do
               Begin
                    Angle:=Theta(p[Hull],p[i]);
                    Temp:=Angle-LastMinAngle; (* Need smallest difference *)
                    If Temp < 0 then Temp:=Temp+360; (* No negatives!!! *)
                    If (Temp < MinAngle) AND ((p[i].x <> p[Hull].x) OR (p[i].y <> p[Hull].y)) then
                       Begin
                            MinAngle:=Temp;     (* Update *)
                            TempAngle:=Angle;
                            Min:=i;
                       End
                    Else


                    (* This is a check to see if the points are lined up *)
                    (* If they are, pick the one closest to the last point *)
                    If (Temp = MinAngle) AND ((p[i].x <> p[Hull].x) OR (p[i].y <> p[Hull].y)) then
(* --> *)                 If (Abs(p[Hull].x - p[i].x) < Abs(p[Hull].x - p[Min].x)) OR
(* --> *)                    (Abs(p[Hull].y - p[i].y) < Abs(p[Hull].y - p[Min].y)) then
                                Min:=i;
                End; (* For-loop through array of points *)
     Until (Min=NumPoints+1) OR (Hull=NumPoints); (* Ending conditions *)
     WriteLn;
     WriteLn('There are ',Hull,' points on the convex hull.');
End; (* Procedure Wrap *)

Begin
     ClrScr;
     For i:=1 to Max+1 do              (* Initialization of points array *)
         Begin
              p[i].x:=0; p[i].y:=0;
         End;                          (* END Initialization *)
     Assign(F,Path);                   (* Input section *)
     Reset(F);
     ReadLn(F,NumPoints);
     For i:=1 to NumPoints do ReadLn(F,p[i].x,p[i].y);
     Close(F);                         (* End input section *)
     Wrap;                             (* Call to gift-wrapping procedure *)
     ReadLn;
End. (* Main program *)
```

```
{ determines whether a point is inside a convex polygon }

{ for the given point, draw lines to each vertex
  this forms a number of small triangles
  if the area of the small triangles adds up to the area of the large
  polygon, then the point is inside/on the polygon
  else, it is outside the polygon
  note that this only works for convex polygons
}

{ input file "poly.in"
  format :
    # of edges
    the next # lines : for each edge : 4 numbers
                       indicating the x and y co-ordinates of the
                       start and end points of that edge
  note that the edges must be given in order -
  i.e., edges[2].p2 = edges[3].p1, etc
}

type
  point = record
    x, y : real;
  end;

var
  edges : array[1..100] of record { stores the edges of the polygon }
    p1, p2 : point;
  end;
  nedges : byte; { number of edges }

function IsPointInside(p : point) : boolean;
{ the function that does the actual work }
var
  i : byte;
  totalarea : real; { the total area of the poly }
  trianglearea : real; { the summed areas of the little triangles }
begin
  { now find the area - up minus down products div 2 }
  totalarea := 0;
  for i := 1 to nedges do begin
    totalarea := totalarea + edges[i].p1.x * edges[i].p2.y
                           - edges[i].p1.y * edges[i].p2.x;
  end;
  totalarea := abs(totalarea) / 2;
  { now find the triangle areas }
  trianglearea := 0;
  for i := 1 to nedges do begin
    trianglearea := trianglearea +
      abs(
          p.x           * edges[i].p1.y - p.y           * edges[i].p1.x
        + edges[i].p1.x * edges[i].p2.y - edges[i].p1.y * edges[i].p2.x
        + edges[i].p2.x * p.y           - edges[i].p2.y * p.x
      ) / 2;
  end;
  if abs(totalarea - trianglearea) < 0.0001 then
    { if the areas are within reasonable limits, taking into account
      floating point errors }
    ispointinside := true
  else ispointinside := false;
end;

var
  f : text; { input file }
  i : byte;
  p : point;
```

```
begin
  { get the edges from file }
  assign(f, 'poly.in');
  reset(f);
  readln(f, nedges);
  for i := 1 to nedges do readln(f, edges[i].p1.x, edges[i].p1.y,
                                    edges[i].p2.x, edges[i].p2.y);
  close(f);
  { done inputting edges }
  { now prompt a user for a point to test }
  readln(p.x, p.y);
  writeln(ispointinside(p));
  readln;
end.
```

```
{ determines whether a point is inside or outside any bounded region }

{ input file "poly.in" stores the boundary of the region
  format :
    # of edges
    on the following # lines : each edge, given by 4 numbers, the
                               x and y co-ordinates of the start and end
                               points of the edge
}

{ to do this, consider a point you know is outside the region -
  e.g., (100000, 100000) - the "point at infinity"
  then draw the line from the given point to the "point at infinity"
  and count how many times this line intersects the edges of the region
  if it's an even number of times - then the point is outside the region
  if it's odd - then the point is inside the region
}

type
  point = record
    x, y : real;
  end;

var
  edges : array[1..100] of record
    p1, p2 : point;
  end;
  nedges : byte;

{ edges is the array storing the boundary of the region
  the edges need not be given in order, but it's a good idea anyway -
  i.e., edges[2].p2 = edges[3].p1, etc
}

function IsBetween(a, x, b : real) : boolean;
begin
  isbetween := ((a <= x) and (x <= b)) or ((b <= x) and (x <= a));
end;

function IsPointInside(p : point) : boolean;
{ this function does the actual work }
const POI : point = (x : 100000; y : 100000); { the "point at infinity" }
      IncrementBy = 0.5; { number to increment c by, to avoid
                           "clipping" an edge }
var
  TimesIntersected : byte; { stores the number of intersections }
  a, b, c : real; { for the line from P to POI ==> ax + by + c = 0 }
  corner : boolean; { stores whether the line intersects a vertex -
                      "ambiguious case" }
  i : byte;
  ea, eb, ec : real; { stores the line for the edge }
  x, y, temp : real; { other variables }
label 1, 2;
begin
  timesintersected := 0;
  corner := false;
  { get the equation of the line }
  a := poi.y - p.y;
  b := -(poi.x - p.x);
  c := -(a * p.x + b * p.y);
  for i := 1 to nedges do begin
  1 :
    if corner then begin
      c := c - incrementby; { return c to the original value }
      corner := false;
    end;
  2 : { reconsider with new c value }
    { find the line for the current edge }
    ea := (edges[i].p2.y - edges[i].p1.y);
    eb := -(edges[i].p2.x - edges[i].p1.x);
    ec := -(ea * edges[i].p1.x + eb * edges[i].p1.y);
```

```
    if (ea * p.x + eb * p.y + ec = 0) and (
       ( (edges[i].p1.x = edges[i].p2.x)
         and isbetween(edges[i].p1.y, p.y, edges[i].p2.y) )
    or ( (edges[i].p1.x <> edges[i].p2.x)
         and isbetween(edges[i].p1.x, p.x, edges[i].p2.x) ) ) then begin
      { the point is on an edge! }
      ispointinside := true;
      exit;
    end;
    { now find the point of intersection }
    if (a * eb - b * ea = 0) and (c * eb - b * ec = 0) then begin
      { the edge and the line are the same }
      if corner then writeln(1 / (i - i)); { something's wrong }
      { shift the line and reconsider }
      corner := true;
      c := c + incrementby;
      goto 2;
    end else
    if a * eb - b * ea <> 0 then begin { not parallel }
      x := -(c * eb - b * ec) / (a * eb - b * ea);
      y := -(c + a * x) / b;
      if (x >= p.x) then begin { if the range is possibly right }
        if edges[i].p1.x = edges[i].p2.x then begin
          { the edge is vertical }
          if (y = edges[i].p1.y) or (y = edges[i].p2.y) then begin
            if corner then writeln(1 / (i - i));
            corner := true;
            c := c + incrementby;
            goto 2;
          end else
          if isbetween(edges[i].p1.y, y, edges[i].p2.y) then inc(timesintersected);
        end else begin
          if (x = edges[i].p1.x) or (x = edges[i].p2.x) then begin
            if corner then writeln(1 / (i - i));
            corner := true;
            c := c + incrementby;
            goto 2;
          end else
          if isbetween(edges[i].p1.x, x, edges[i].p2.x) then inc(timesintersected);
        end;
      end;
    end;
  end;
  { return result }
  if timesintersected mod 2 = 0 then ispointinside := false
  else ispointinside := true;
end;

var
  i : byte;
  temp : point;
  f : text;

begin
  { get input - the input file stores the boundary of the region }
  assign(f, 'poly.in');
  reset(f);
  readln(f, nedges);
  for i := 1 to nedges do readln(f, edges[i].p1.x, edges[i].p1.y,
                                    edges[i].p2.x, edges[i].p2.y);
  close(f);
  { finished getting input }
  { now ask the user for a point to test }
  readln(temp.x, temp.y);
  writeln(IsPointInside(temp));
  readln;
end.
```

MISCELLANEOUS

ShellSort

```
Procedure Shell_Sort(Var A:Style);
Var
   k,j,temp:Integer;
Begin
     k:=0;
     Repeat k:=3*k+1; Until k > Max-1;
     Repeat
           k:=k Div 3;
           For i:=k to Max do
               Begin
                    Temp:=A[i];
                    j:=i;
                    While (A[j-k]>Temp) and (j>k) Do
                          Begin
                               A[j]:=A[j-k];
                               j:=j-k;
                               Inc(swaps);
                          End;
                    A[j]:=Temp;
               End;
     Until k<=1;
End;
```

```
(* PARSING *)
Const    {43,27,-0.9,42.9,9.0,-9t}
     InF = 'parse.pas';
     N   = 2;  {number of legal input sets 7 9 20.9}

     Letters = 1; {just for the demonstration}
     Numbers = 2; {"}

Var
   Line           : String;     {the line to be parsed}
   LegalInputSets : array[1..N] of set of char;  {sets you really should know what these are, if you
don't look them up}
   T              : Text;
   CurPos         : Byte;       {current position in Line}
   I,J            : Byte;       {temp variables for the demo}


Function  GetFromSetofLegal( StNum : Byte; var Pos : Byte; Data : String ) : String;
{Using LegalInputSets[StNum] find a 'word' within Data starting at Position Pos}
{'word' can mean word, character, integer or whatever}
          Var TempString : String;
          Begin
               While (Not (Data[Pos] In LegalInputSets[StNum])) AND (Pos <= Length(Data)) Do
                     Inc(Pos);   {Move the position until we hit the first useful point}


               If Pos >= Length(Data) Then Begin GetFromSetOfLegal := ''; Exit; End;
               TempString := '';


               While (Data[Pos] In LegalInputSets[StNum]) AND (Pos <= Length(Data)) do
                     Begin   {Move the position while we have useful stuff}
                          TempString := TempString + Data[Pos]; {add legal input to the 'word'}
                          Inc(Pos);
                     End;


               GetFromSetofLegal := TempString; {always remember to do this}
          End;

Begin
     LegalInputSets[1] := ['A'..'Z','a'..'z'];
     LegalInputSets[2] := ['0'..'9','-','.'];

     Assign(T, InF );
     Reset(T);
     I := 0;

     While (Not Eof(t)) and (I < 3) do
           Begin
                Inc(I);
                ReadLn(t, Line);
                WriteLn( 'numbers in line ', I);
                J := 1;
                Repeat Write( GetFromSetofLegal(Numbers,J,Line), ' ' ) Until J > Length(Line);
                WriteLn;
                WriteLn( 'words in line ', I);
                J := 1;
                Repeat Write( GetFromSetofLegal(Letters,J,Line),' ' ) Until J > Length(Line);
                WriteLn;
                WriteLn;
           End;

     ReadLn;
     Close(T);
End.
```

```
{Dynamic solution for golf
Desription: You have a certain number golf clubs, and each can be used
to hit the ball a fixed distance. Program has to find the minimum number
of strokes required to reach the distance.

The first few are lines are reading in the input and initilization.
The first loop, loops through all the clubs,
the second loop, loops from the minimum distance the club can hit, to
the distance required to reach.
Then it checks if the the number of steps currently stored is greater than
using the club to hit from the previous spot, and if it is then the value
gets replaced}

Const
   Infinity = 65278;

Var
   intMeter: Array[0..10000] Of Word; {Array for every point between source, and destination}
   bytClub: Array[1..32] Of Byte; {Array for club sizes available}
   bytClubs: Byte; {# of Clubs}
   i, j, intMeters: Word;
   txtInput: Text;

Begin
   Assign(txtInput, 'Golf.in'); Reset(txtInput);
   Readln(txtInput, intMeters); {Distance needed to hit}
   FillChar(intMeter, SizeOf(intMeter), 254); intMeter[0]:=0; bytClubs:=0; {Initialization}
   Readln(txtInput, bytClubs);
   For i:=1 To bytClubs Do
      Readln(txtInput, bytClub[i]); {Read club sizes into array}

   {======The actual algorithm=====}
   For i:=1 To bytClubs Do {Loop through all clubs}
      For j:=bytClub[i] To intMeters Do {Loop from shortest distance the club can hit, to the
distanced need to reach}
         If intMeter[j] > (intMeter[j - bytClub[i]] + 1) Then {Check if the current # of clubs is
less than previous amount}
            intMeter[j]:= intMeter[j - bytClub[i]] + 1; {Set array element to shorter number, if
shorter path possibe}
   {======End Of Algorithm======}

   If intMeter[intMeters] >= Infinity Then
      Writeln('Distance can not be reached')
   Else
      Writeln('Distance can be reached in ', intMeter[intMeters], ' stroke(s).');
   Readln;
End.
```

```
{Binary Tree
Try strings like * 2 n n + 3 n n 1 n n
This is the equivalent of 2 * (3 + 1), and you can see this in infix}
Uses
   CRT;

Type
   ptrNode = ^Node; {Pointer to node}
   Node = Record {Record for each node in the tree}
      Value: Char; {Value stored in node, can be any data type}
      Left, Right: ptrNode; {Pointers to Left, Right Nodes}
   End;

Var
   nodTop: ptrNode; {The node for the top element in the tree}
   chrLeft, chrRight: Char; {Left and right values for input}

{Recursive function, reads input from keyboard, and builds tree}
Procedure ReadData(nodNode: ptrNode);
Begin
   Write('Input for left node (n = nil): '); Readln(chrLeft); {Read data for left node}
   If chrLeft = 'n' Then
      nodNode^.Left:=Nil {If left  = 0 then set left pointer to null}
   Else Begin
      New(nodNode^.Left); {Allocates memory for left node}
      nodNode^.Left^.Value:=chrLeft; {Assigns input value to node}
      ReadData(nodNode^.Left); {Recurses again, passing the left node, as the parent of the other
two nodes}
   End;

   Write('Input for right node (n = nil): '); Readln(chrRight); {Read data for right node}
   If chrRight = 'n' Then
      nodNode^.Right:=Nil {If left  = 0 then set right pointer to null}
   Else Begin
      New(nodNode^.Right); {Allocates memory for left node}
      nodNode^.Right^.Value:=chrRight;  {Assigns input value to node}
      ReadData(nodNode^.Right); {Recurses again, passing the right node, as the parent of the other
two nodes}
   End;
End;

{Prints the tree in prefix notation}
Procedure PrintTreePrefix(nodNode: ptrNode);
Begin
   Write(nodNode^.Value); {Writes the value of the top node}
   If nodNode^.Left <> Nil Then
      PrintTreePrefix(nodNode^.Left); {Recurses to the left node if not null}
   If nodNode^.Right <> Nil Then
      PrintTreePrefix(nodNode^.Right);{Recurses to the right node if not null}
End;

Procedure PrintTreeInfix(nodNode: ptrNode);
Begin
   If nodNode^.Left <> Nil Then
      PrintTreeInfix(nodNode^.Left);
   Write(nodNode^.Value);
   If nodNode^.Right <> Nil Then
      PrintTreeInfix(nodNode^.Right);
End;

Procedure PrintTreePostfix(nodNode: ptrNode);
Begin
   If nodNode^.Left <> Nil Then
      PrintTreePostfix(nodNode^.Left);
   If nodNode^.Right <> Nil Then
      PrintTreePostfix(nodNode^.Right);
   Write(nodNode^.Value);
End;
```

```
{Garbage Collection Procedure - a feature that's built into almost every other programming language
except pascal}
Procedure CleanUp(nodNode: ptrNode);
Begin
   If nodNode^.Left = Nil Then
      If nodNode^.Right = Nil Then
         Dispose(nodNode)
      Else
         CleanUp(nodNode^.Right)
   Else
      CleanUp(nodNode^.Left);
End;

Begin
   ClrScr;
   New(nodTop); {Allocates memory for top node}
   Write('Enter value for top node: '); Readln(chrLeft);
   nodTop^.Value:=chrLeft;
   ReadData(nodTop); {Calls procedure to read in data - see procedure for more info}

   ClrScr;
   PrintTreePrefix(nodTop); {Prints the tree - see procedure for more info}
   Readln;

   ClrScr;
   PrintTreeInfix(nodTop);
   Readln;

   ClrScr;
   PrintTreePostfix(nodTop);
   Readln;

   CleanUp(nodTop);
   Readln;
End.
```