**DAA LAB 11 EXERCISE**

**PRATEEK MOHANTY**

**20BCE1482**

* **CLOSEST PAIR PROBLEM**

**CODE**

#include<iostream>

#include<cmath>

using namespace std;

class Points

{

    public:

    int x;

    int y;

};

float distance(int x1,int y1,int x2,int y2)

{

    int xdis=x1-x2;

    int ydis=y1-y2;

    float res=pow(xdis\*xdis + ydis\*ydis,0.5);

    return res;

}

void closestPairPoints(struct Points point1[],int n)

{

    int min=INT16\_MAX;

    int xindex1=-1,yindex1=-1;

    int xindex2=-1,yindex2=-1;

    for(int i=0;i<n;i++)

    {

        int xpoint=point1[i].x;

        int ypoint=point1[i].y;

        for(int j=i+1;j<n;j++)

        {

            int xpoint1=point1[j].x;

            int ypoint1=point1[j].y;

            float t=distance(xpoint,ypoint,xpoint1,ypoint1);

            if(t<min)

            {

                min=t;

                xindex1=point1[i].x;

                yindex1=point1[i].y;

                xindex2=point1[j].x;

                yindex2=point1[j].y;

            }

        }

    }

    cout<<"Closest pair of points are: "<<"("<<xindex1<<","<<yindex1<<")";

    cout<<" "<<"("<<xindex2<<","<<yindex2<<")"<<endl;

    cout<<"The distance between the points are: "<<distance(xindex1,yindex1,xindex2,yindex2);

}

int main()

{

    cout<<"Enter the number of points: ";

    int n;

    cin>>n;

    struct Points arr[n];

    for(int i=0;i<n;i++)

    {

        int x,y;

        cin>>x>>y;

        arr[i].x=x;

        arr[i].y=y;

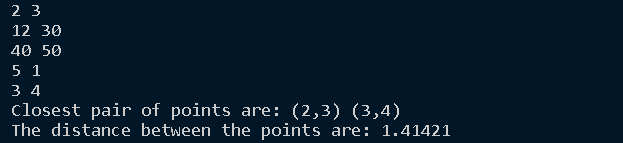
    }

      closestPairPoints(arr,n);

    return 0;

}

**OUTPUT**

****

* **GRAHAM SCAN CONVEX HULL ALGORITHM**

**CODE**

#include<iostream>

#include<stack>

#include<stdlib.h>

using namespace std;

// Graham scan algorithm for convex hull

// Time complexity : O(nlog(n))

struct Point

{

    public:

     int x;

     int y;

};

// BOTTOM MOST POINT

Point lowesty;

// checking if the points are clockwise or counterclock wise

int ccw(Point a,Point b,Point c)

{

    float area =(b.x-a.x)\*(c.y-a.y)-(b.y-a.y)\*(c.x-a.x);

    if(area<0)return 2;   // anticlockwise

    if(area>0) return 1 ;  // clockwise

    return 0;    // collinear points

}

// Point next to the top of stack

Point stackNext(stack<Point>&s)

{

    Point p=s.top();

    s.pop();

    Point res=s.top();

    s.push(p);                // putting back the first point in the stack

    return res;

}

// function to swap two points

void swap(Point &p1,Point &p2)

{

    Point temp=p1;

    p1=p2;

    p2=temp;

}

// calculating the distance between two points for sorting in case they are collinear

int dist(Point p1,Point p2)

{

    return (p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y);

}

int compare(const void \*vp1,const void \*vp2)

{

      Point \*p1=(Point \*)vp1;

      Point \*p2=(Point \*)vp2;

      // Find orientation

      int r=ccw(lowesty,\*p1,\*p2);

      if(r==0)

      {

          // points are collinear

          // then sort based on closets point first

          return (dist(lowesty,\*p2)>=dist(lowesty,\*p1))?-1:1;

      }

      if(r==2)return -1;

      else return 1;

}

void convexHull(Point points[],int n)

{

    // finding the bottom most point

    int ymin=points[0].y,min=0;

    for(int i=1;i<n;i++)

    {

        int y=points[i].y;

        // choose the bottom most or the left most point

        // if two points are equal choose the left most point

        if(y<ymin || (ymin==y && points[i].x<points[min].x))

        {

            ymin=points[i].y;

            min=i;

        }

    }

    // swap the minimum point to the first position

    swap(points[0],points[min]);

    // sort n-1 points with respect to the first point.

    // A point p1 comes before p2 in sorted output if p2

    // has larger polar angle (in counterclockwise direction) than p1

    lowesty=points[0];

    qsort(&points[1],n-1,sizeof(Point),compare);

    // removing all the collinear points and keeping only the last one

    int m=1;

    for(int i=1;i<n ;i++)

    {

        // keep moving i while the angle is same

        // with respect to p0

        while(i<n-1 && ccw(lowesty,points[i],points[i+1])==0)

        i++;

        points[m]=points[i];

        m++;

    }

    // If modified array of points has less than 3 points

    // convex hull is not possible

    if(m<3) return ;     // less than 3 points cannot form convex hull

   // create an empty stack

   stack<Point >s;

   s.push(points[0]);

   s.push(points[1]);

   s.push(points[2]);

   // process for remaining n-3 points

   for(int i=3;i<m;i++)

   {

          while(s.size()>1 && ccw(stackNext(s),s.top(),points[i])!=2)

          s.pop();

          s.push(points[i]);

   }

   // finally stack has the output points

   while(!s.empty())

   {

       Point p=s.top();

       cout<<"("<<p.x<<","<<p.y<<")"<<endl;

       s.pop();

   }

}

// sorting the points based on polar angle with the left most point

int main()

{

    cout<<"Enter the number of points: ";

    int n;

    cin>>n;

    Point points[n];

    cout<<"Enter the x and y coordinates of the points: ";

    for(int i=0;i<n;i++)

    {

           int x,y;

           cin>>x>>y;

           points[i].x=x;

           points[i].y=y;

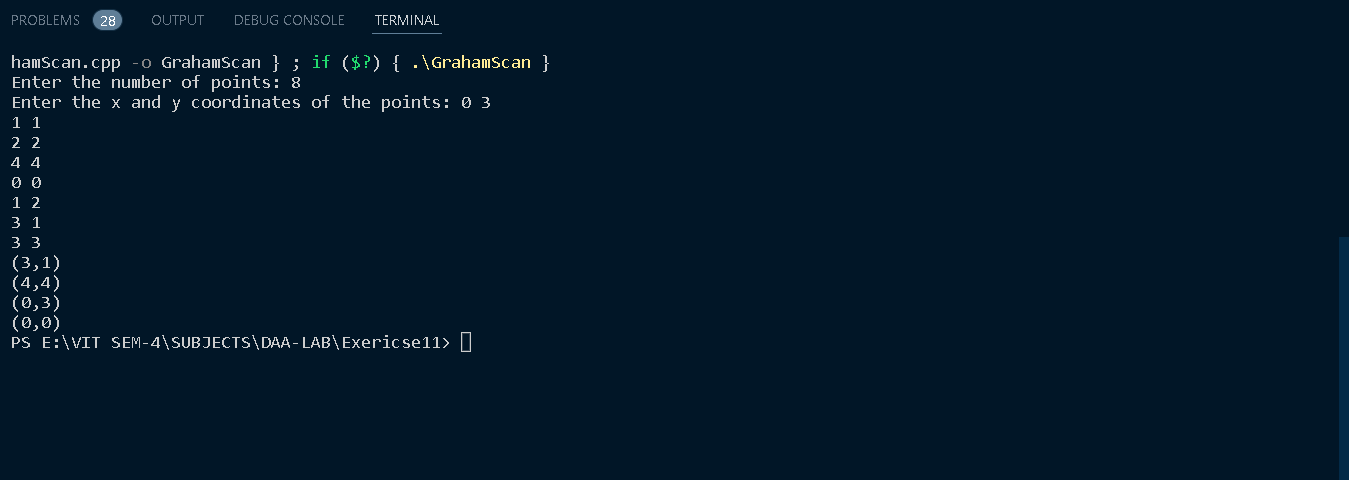
    }

    convexHull(points,n);

    return 0;

}

**OUTPUT**

****

* **Fulkerson algorithm for finding maximum flow**

**CODE**

#include<iostream>

#include<limits.h>

#include<queue>

#include<string.h>

using namespace std;

// Number of vertices in given graph

#define V 6

// function returns true , if it is a path from source to sink

bool sourceTosink(int rGraph[V][V],int s,int t,int parent[])

{

    bool visited[V];

    // initialize the visited array to 0

    memset(visited,0,sizeof(visited));

     // Create a queue, enqueue source vertex and mark source

    // vertex as visited

    queue<int>q;

    q.push(s);

    visited[s]=true;

    parent[s]=-1;

    // standard BFS Loop

    while(!q.empty())

    {

        int u=q.front();

        q.pop();

        for(int v=0;v<V;v++)

        {

            if(visited[v]==false && rGraph[u][v]>0){

                if(v==t){

                    parent[v]=u;

                    return true;

                }

                q.push(v);

                parent[v]=u;

                visited[v]=true;

            }

        }

    }

}

int fordFulkerson(int graph[V][V],int s,int t)

{

    int u,v;

    int rgraph[V][V];

    int max\_flow=0;

    for(int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        rgraph[i][j]=graph[i][j];

    }

    int parent[V];

    while(sourceTosink(rgraph,s,t,parent)){

        int path\_flow=INT16\_MAX;

        for(int i=t;i!=s;i=parent[i]){

            u=parent[i];

            path\_flow=min(path\_flow,rgraph[u][i]);

        }

        for(int i=t;i!=s;i=parent[i])

        {

            u=parent[i];

            rgraph[u][i]-=path\_flow;

            rgraph[u][i]+=path\_flow;

        }

        // Add path flow to overall flow

        max\_flow+=path\_flow;

    }

    return max\_flow;

}

int main()

{

    int graph[V][V]

        = { { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },

            { 0, 4, 0, 0, 14, 0 },  { 0, 0, 9, 0, 0, 20 },

            { 0, 0, 0, 7, 0, 4 },   { 0, 0, 0, 0, 0, 0 } };

    cout << "The maximum possible flow is "

         << fordFulkerson(graph, 0, 5);

    return 0;

}

**OUTPUT**

****