SOFE 3770U Assignment 1

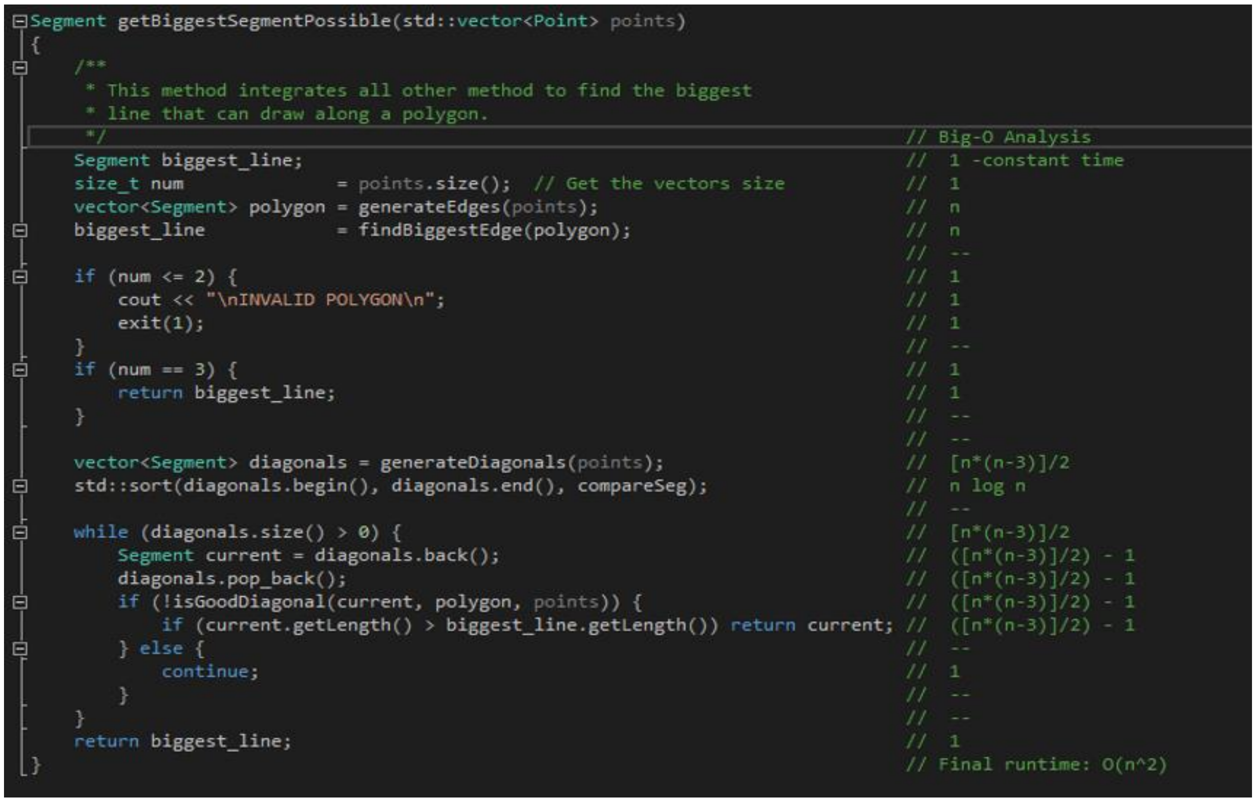
Pseudo code description

The program begins by taking user input x,y positions and assigning them to a vector of integers, this vector is then used to create a vector of Point objects.

Algorithm begins by taking the vector of Point objects and creating a vector of Segment (line) objects making up the polygon. Next, it finds the biggest line among the edges of the polygon, this is especially useful because it eliminates the need to compare the first and last Segment (line) object that can be drawn from each vertex. This biggest edge is used to compare any new diagonals against it to determine the biggest line possible. If the number of sides of the polygon is 3, this is the biggest line possible inside and along the polygon.

The algorithm now calls a function that generates all possible diagonals, without making any duplicates (n^2 time complexity). After all diagonals have been created the algorithm adds the list of lines segments that make up the list of edges of the polygon and the diagonals into one single list and then sort them. Sorting takes n log n time. Now the biggest element is popped from the top of the list and checked for validity. If the line is valid, then that is the biggest line, if not continue with the next line and so on until a valid line is found.

Big-O Analysis

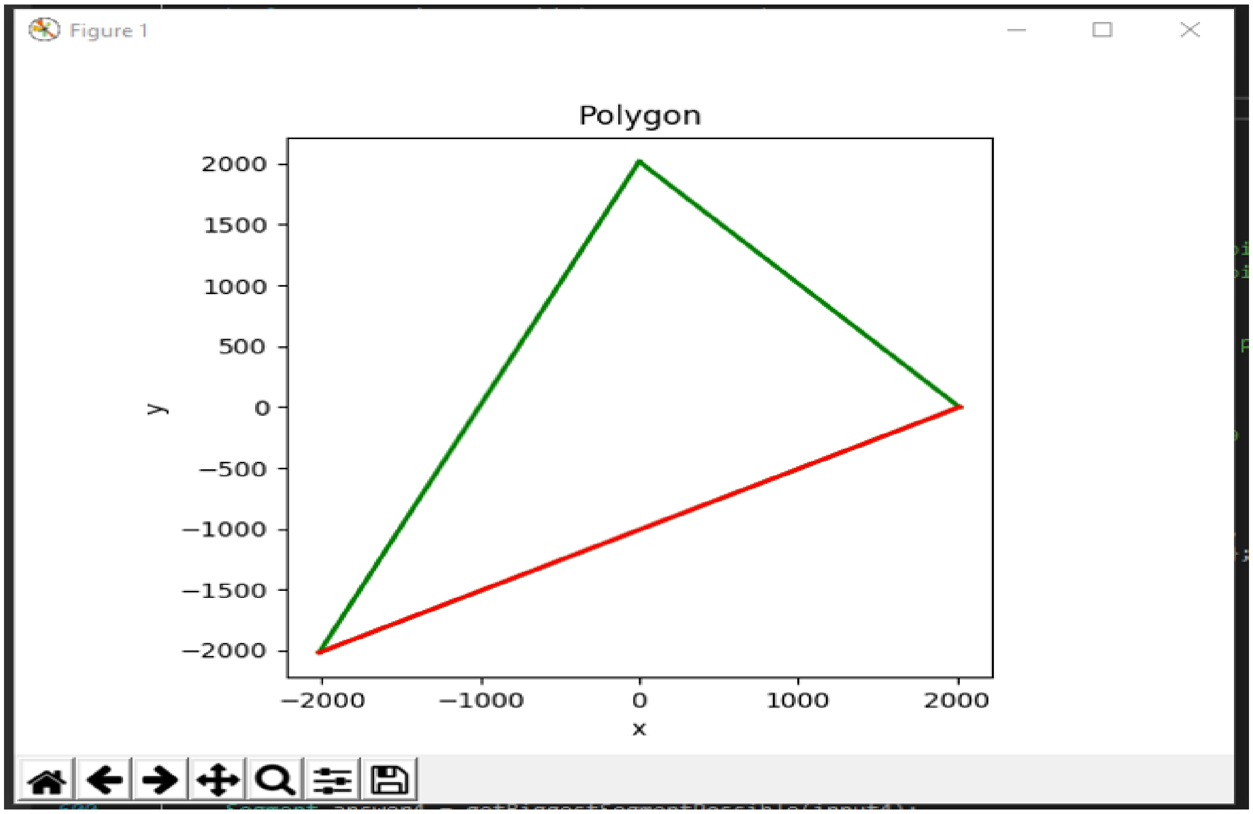


```
Segment getBiggestSegmentPossible(std::vector<Point> points)
{
    /**
     * This method integrates all other method to find the biggest
     * line that can draw along a polygon.
     */                                                              // Big-O Analysis
    Segment biggest_line;                                            //  1 -constant time
    size_t num              = points.size();  // Get the vectors size //  1
    vector<Segment> polygon = generateEdges(points);                 //  n
    biggest_line            = findBiggestEdge(polygon);              //  n
                                                                     //  --
    if (num <= 2) {                                                  //  1
        cout << "\nINVALID POLYGON\n";                               //  1
        exit(1);                                                     //  1
    }                                                                //  --
    if (num == 3) {                                                  //  1
        return biggest_line;                                         //  1
    }                                                                //  --
                                                                     //  --
    vector<Segment> diagonals = generateDiagonals(points);           //  [n*(n-3)]/2
    std::sort(diagonals.begin(), diagonals.end(), compareSeg);       //  n log n
                                                                     //  --
    while (diagonals.size() > 0) {                                   //  [n*(n-3)]/2
        Segment current = diagonals.back();                          //  ([n*(n-3)]/2) - 1
        diagonals.pop_back();                                        //  ([n*(n-3)]/2) - 1
        if (!isGoodDiagonal(current, polygon, points)) {             //  ([n*(n-3)]/2) - 1
            if (current.getLength() > biggest_line.getLength()) return current; //  ([n*(n-3)]/2) - 1
        } else {                                                     //  --
            continue;                                                //  1
        }                                                            //  --
    }                                                                //  --
    return biggest_line;                                             //  1
}                                                                    // Final runtime: O(n^2)
```

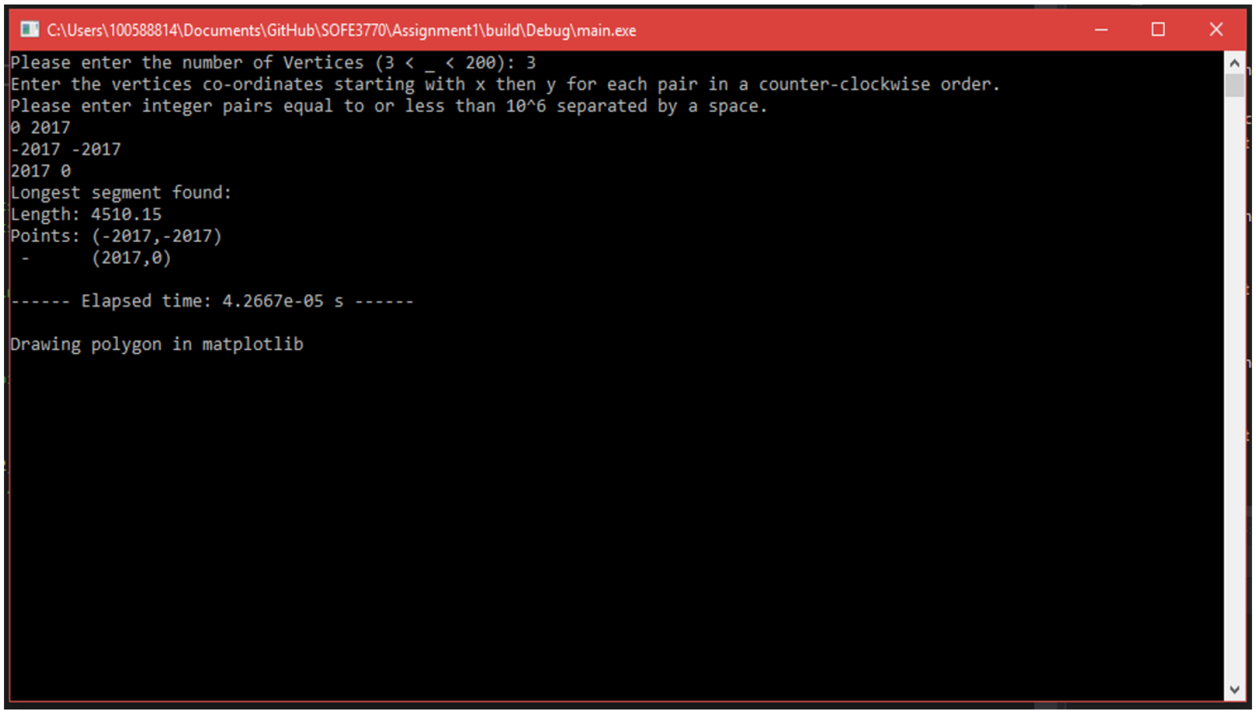
Final Time complexity is O(n^2)

Output results for n = 3/5/7/9/13

N = 3

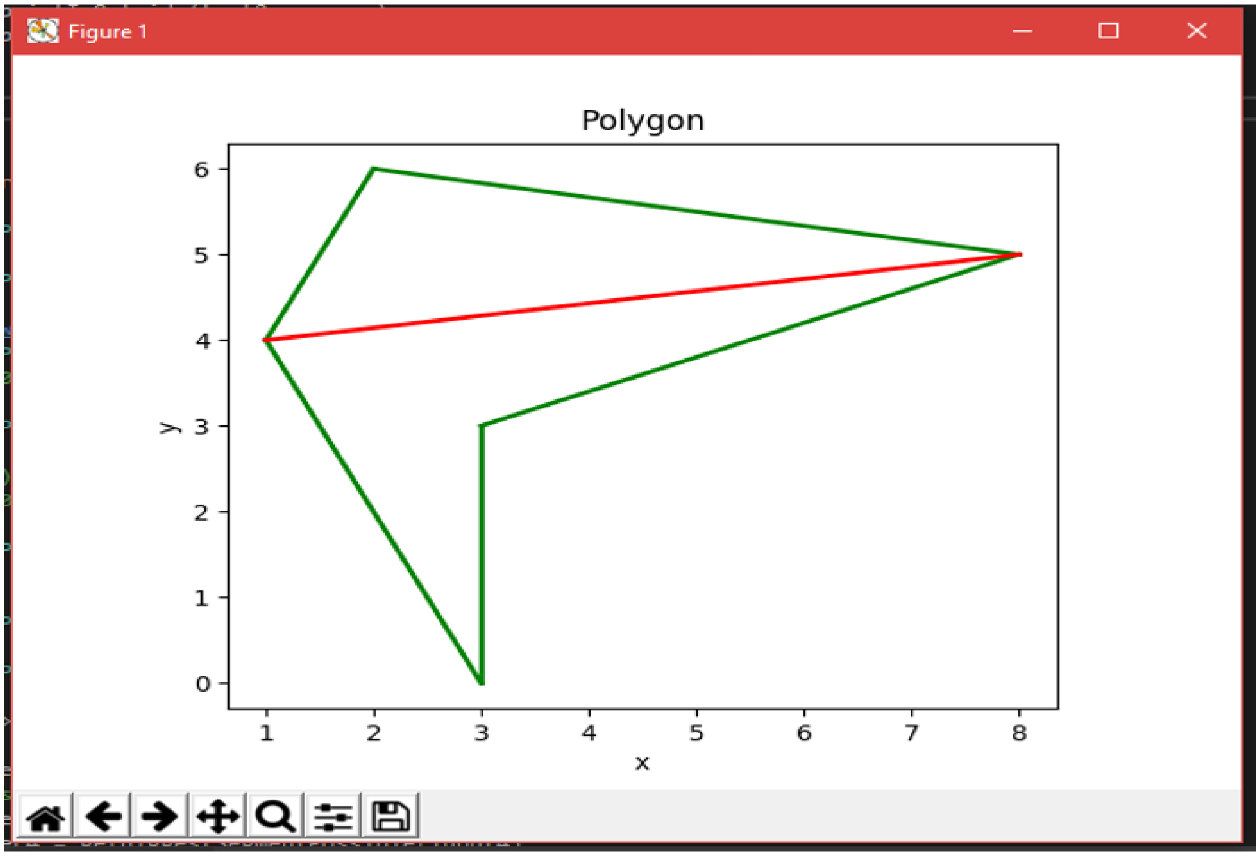


Elapsed Time: 4.266 x 10^{-5} Seconds

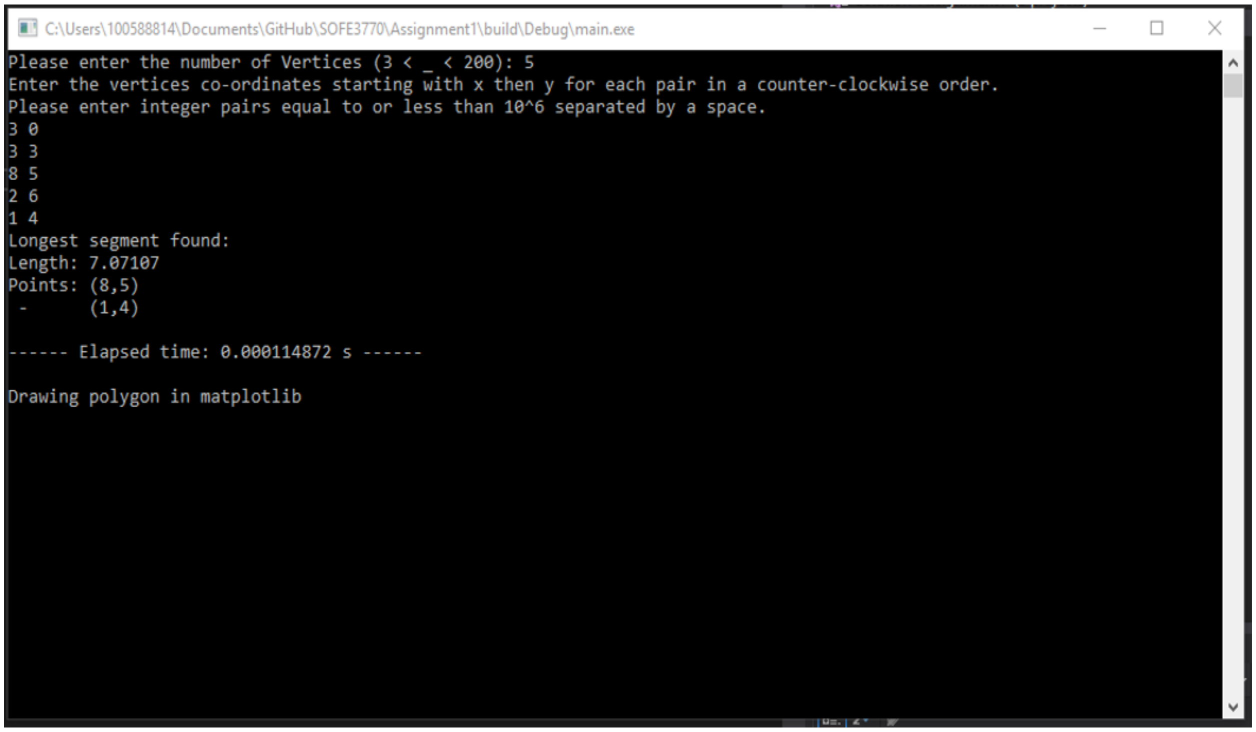

C:\Users\100588814\Documents\GitHub\SOFE3770\Assignment1\build\Debug\main.exe

```
Please enter the number of Vertices (3 < _ < 200): 3
Enter the vertices co-ordinates starting with x then y for each pair in a counter-clockwise order.
Please enter integer pairs equal to or less than 10^6 separated by a space.
0 2017
-2017 -2017
2017 0
Longest segment found:
Length: 4510.15
Points: (-2017,-2017)
 -      (2017,0)

------ Elapsed time: 4.2667e-05 s ------

Drawing polygon in matplotlib
```

N = 5

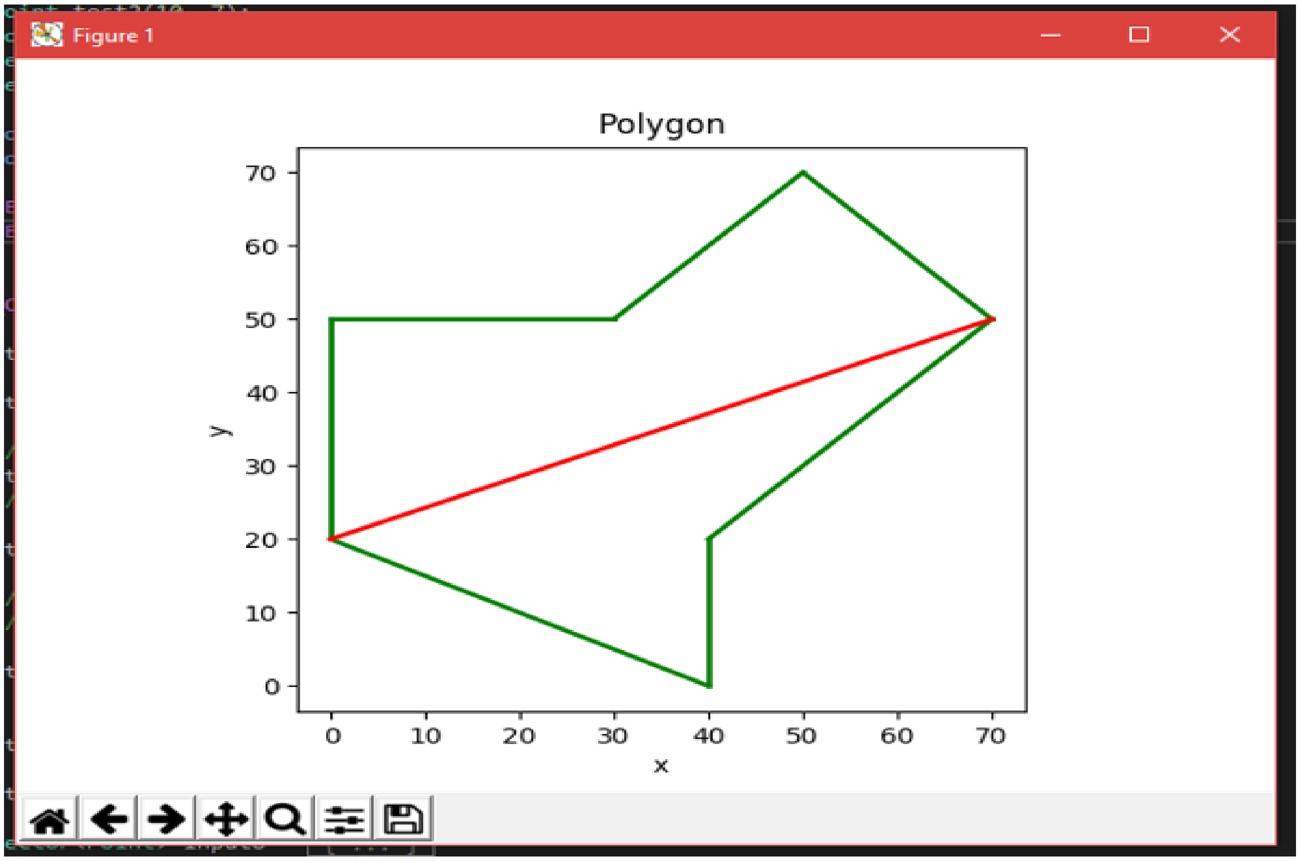


Elapsed Time: 0.0001148 Seconds

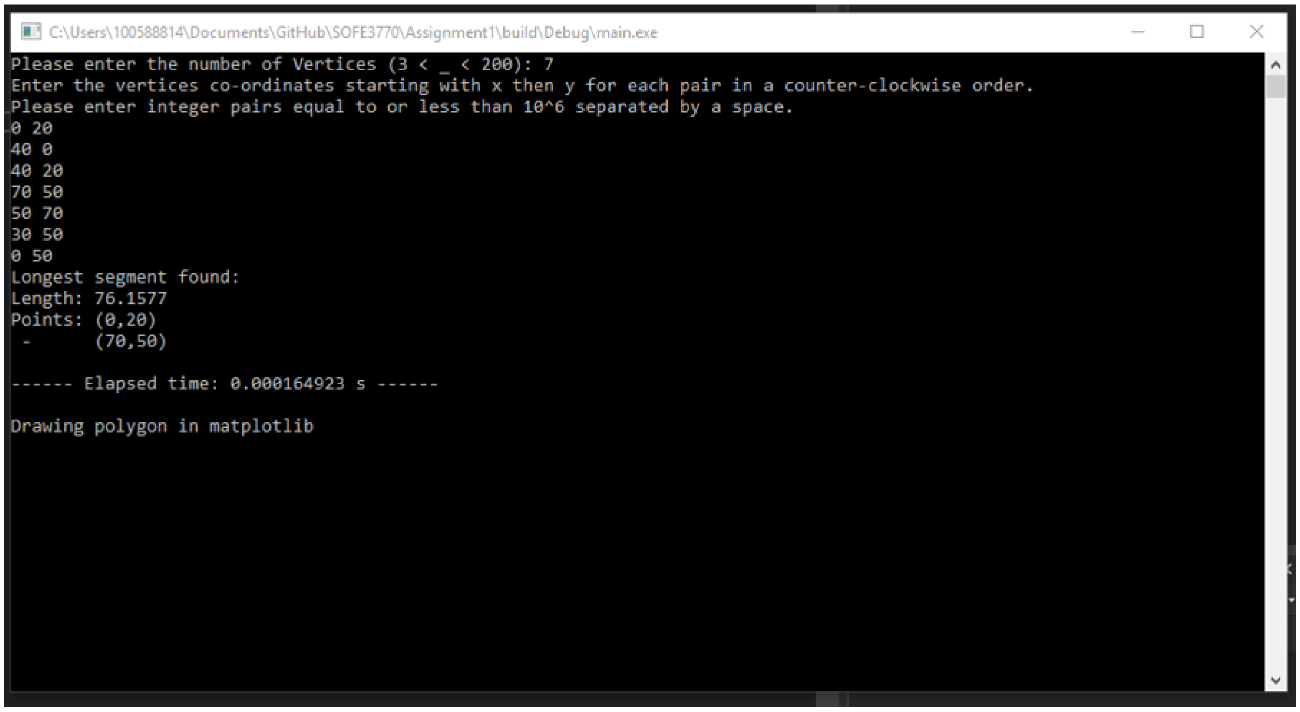

C:\Users\100588814\Documents\GitHub\SOFE3770\Assignment1\build\Debug\main.exe

```
Please enter the number of Vertices (3 < _ < 200): 5
Enter the vertices co-ordinates starting with x then y for each pair in a counter-clockwise order.
Please enter integer pairs equal to or less than 10^6 separated by a space.
3 0
3 3
8 5
2 6
1 4
Longest segment found:
Length: 7.07107
Points: (8,5)
 -      (1,4)

------ Elapsed time: 0.000114872 s ------

Drawing polygon in matplotlib
```

N = 7

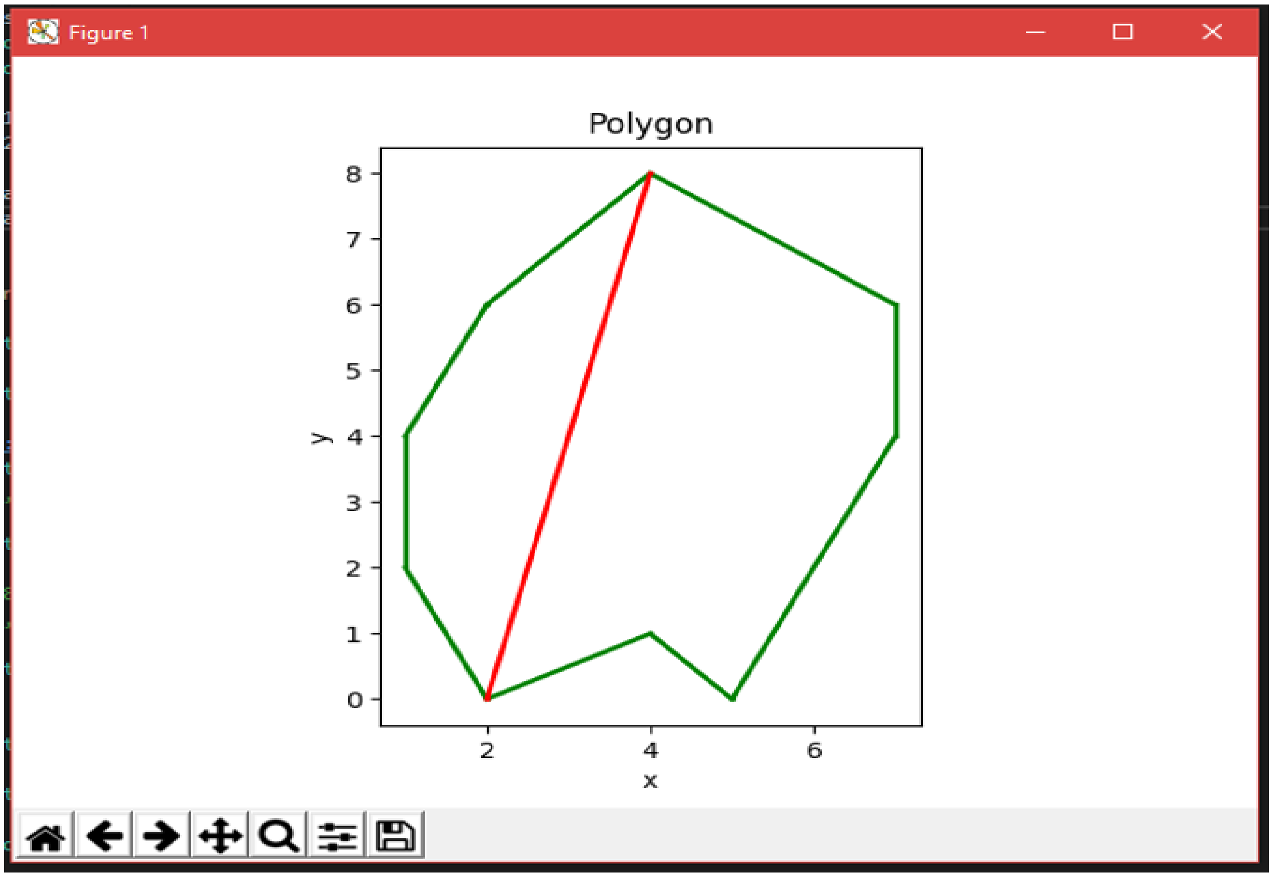


Elapsed Time: 0.000164923 Seconds

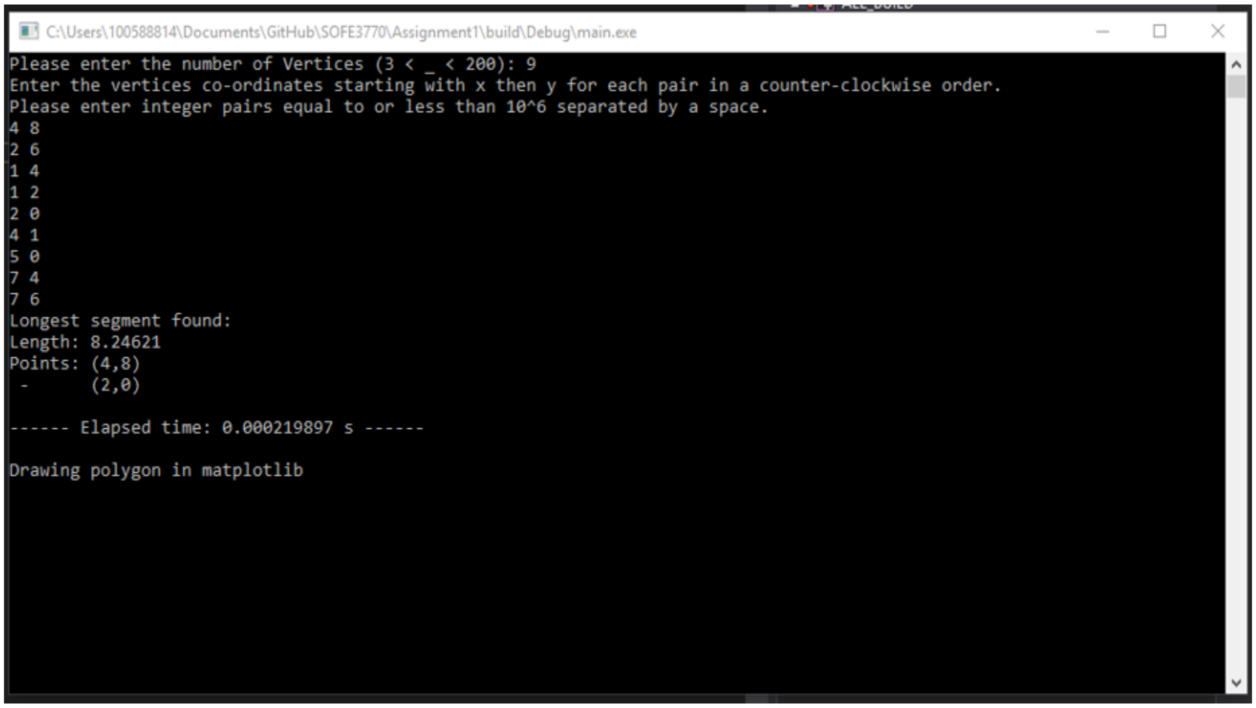

C:\Users\100588814\Documents\GitHub\SOFE3770\Assignment1\build\Debug\main.exe

```
Please enter the number of Vertices (3 < _ < 200): 7
Enter the vertices co-ordinates starting with x then y for each pair in a counter-clockwise order.
Please enter integer pairs equal to or less than 10^6 separated by a space.
0 20
40 0
40 20
70 50
50 70
30 50
0 50
Longest segment found:
Length: 76.1577
Points: (0,20)
 -      (70,50)

------ Elapsed time: 0.000164923 s ------

Drawing polygon in matplotlib
```

N = 9

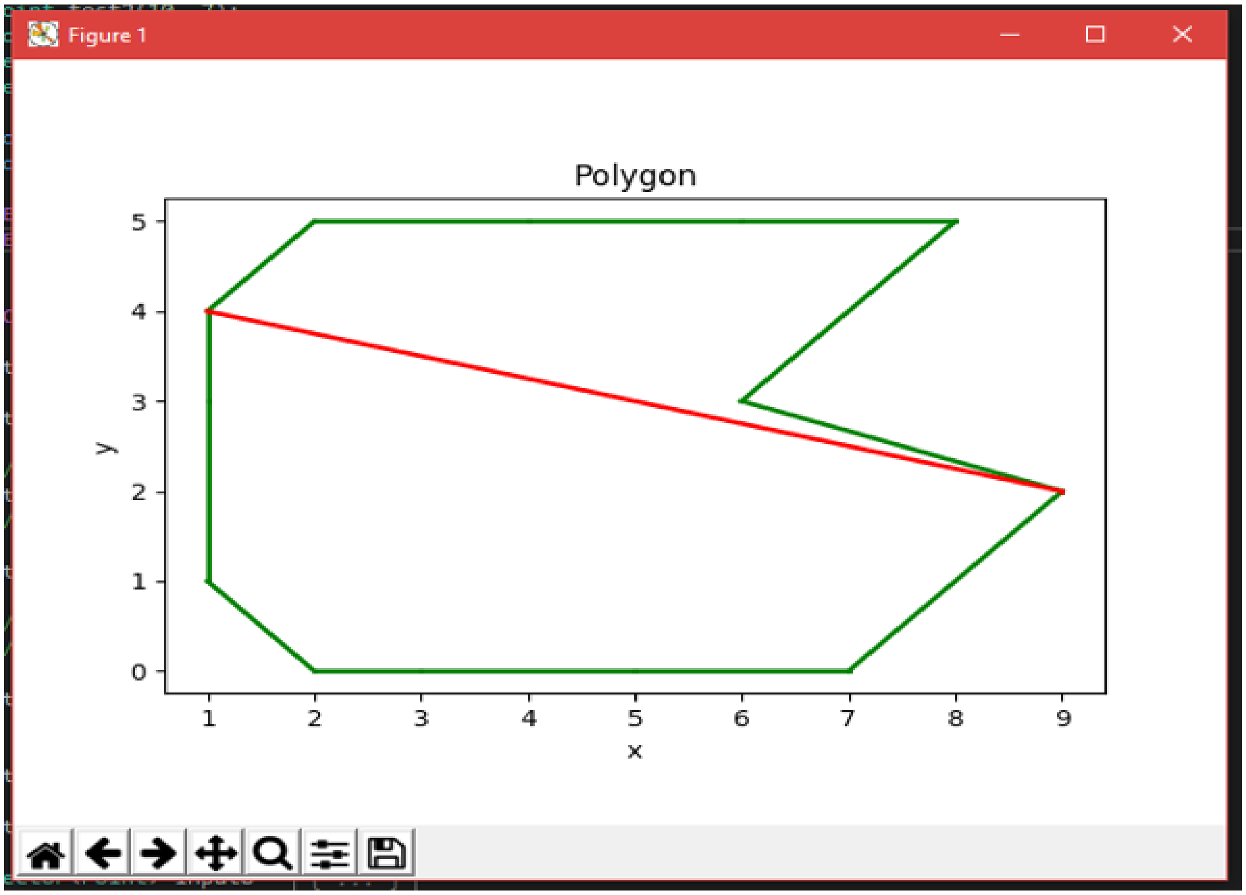


Elapsed Time: 0.000219897 Seconds

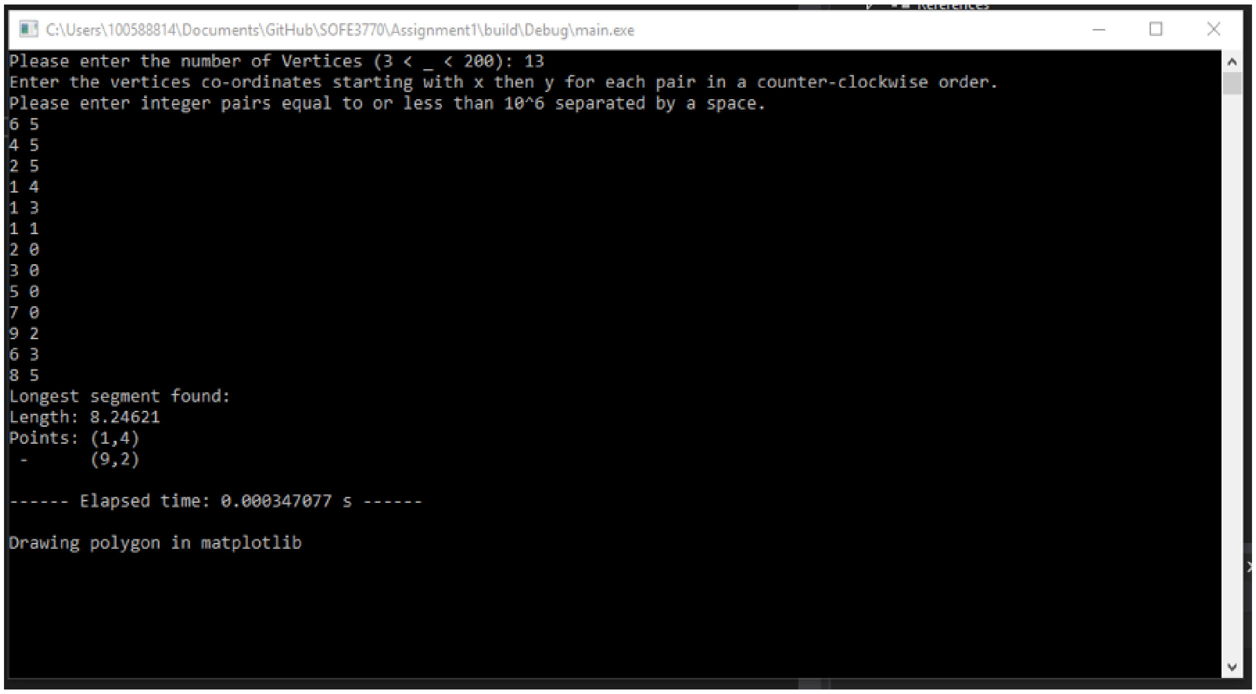

C:\Users\100588814\Documents\GitHub\SOFE3770\Assignment1\build\Debug\main.exe

```
Please enter the number of Vertices (3 < _ < 200): 9
Enter the vertices co-ordinates starting with x then y for each pair in a counter-clockwise order.
Please enter integer pairs equal to or less than 10^6 separated by a space.
4 8
2 6
1 4
1 2
2 0
4 1
5 0
7 4
7 6
Longest segment found:
Length: 8.24621
Points: (4,8)
 -      (2,0)

------ Elapsed time: 0.000219897 s ------

Drawing polygon in matplotlib
```

N = 13



Elapsed Time: 0.000347077 Seconds


C:\Users\100588814\Documents\GitHub\SOFE3770\Assignment1\build\Debug\main.exe

```
Please enter the number of Vertices (3 < _ < 200): 13
Enter the vertices co-ordinates starting with x then y for each pair in a counter-clockwise order.
Please enter integer pairs equal to or less than 10^6 separated by a space.
6 5
4 5
2 5
1 4
1 3
1 1
2 0
3 0
5 0
7 0
9 2
6 3
8 5
Longest segment found:
Length: 8.24621
Points: (1,4)
 -      (9,2)

------ Elapsed time: 0.000347077 s ------

Drawing polygon in matplotlib
```

Source Code (Written in C++)

Main.cpp

```cpp
#include <chrono>  // This is for measuring thread running time
#include <iostream>
#include <vector>
#include "geometry.hpp"

using namespace std;

int main()
{
    // Declaring and Initializng variables
    int polygon_Vertices;
    vector<int> x_Points;
    vector<int> y_Points;
    vector<Point> given_points;

    cout << "Please enter the number of Vertices (3 < _ < 200): ";
    cin >> polygon_Vertices;
    // Error checking for incorrect input
    while (polygon_Vertices > 200 || polygon_Vertices < 3) {
        cout << "Please enter a valid number: ";
        cin >> polygon_Vertices;
    }

    cout << "Enter the vertices co-ordinates starting with x then y ";
    cout << "for each pair in a counter-clockwise order. \n";
    cout << "Please enter integer pairs equal to or less than 10^6 separated by a 
space. \n";
    for (int count = 0; count < polygon_Vertices; count++) {
        int x_Temp;
        cin >> x_Temp;
        x_Points.push_back(x_Temp);
        if (x_Temp > 1000000) {
            cout << "Error";
            cin >> x_Temp;
        }
        int y_Temp;
        cin >> y_Temp;
        y_Points.push_back(y_Temp);
        if (y_Temp > 1000000) {
            cout << "Error";
            cin >> y_Temp;
        }
        // Populates vector with point objects using user defined x and y coordinates
        given_points.push_back(Point(x_Points[count], y_Points[count]));
    }

    auto start = chrono::high_resolution_clock::now();  // Start clock

    // Identify largest segment in polygon
    Segment result = getBiggestSegmentPossible(given_points);

    auto finish                      = chrono::high_resolution_clock::now();  // Stop
clock
    chrono::duration<double> elapsed = finish - start;                        //
Mesure time elapsed

    //  Output length and start/end points of largest segment
    cout << "Longest segment found: \n";
    cout << "Length: " << result.getLength() << "\n";
    cout << "Points: " << result.start << " -      " << result.end;
    cout << "\n------ Elapsed time: " << elapsed.count() << " s ------\n";

    return 0;
}
```

Geometry.hpp

```
#ifndef GEOMETRY_HPP  // These are file guards
#define GEOMETRY_HPP

#include <cmath>
#include <iostream>
#include <vector>

using namespace std;

class Point
{
   public:
    int x;
    int y;
    friend ostream& operator<<(ostream& os, const Point&);  // define os << operator
for point
    friend bool operator==(const Point&, const Point&);     // Defines == for Point
    Point()
    {
        /**
         * This is the default constructor
         */
        this->x = 0;
        this->y = 0;
    }
    Point(int x_cord, int y_cord)
    {
        /**
         * This constructor creates objects with the values passed in
         */
        this->x = x_cord;
        this->y = y_cord;
    }
};

// Make Line class here.
class Segment
{
   private:
    double length;

   public:
    Point start;
    Point end;
    double getLength();  // Returns the length of the line
    Point getMidpoint();
    friend ostream& operator<<(ostream& os, const Segment&);  // define os <<
operator for Segment
    friend bool operator==(const Segment&, const Segment&);   // Defines == for
Segment
    Segment()
    {
        Point default_set = {0, 0};
        this->start       = default_set;
        this->end         = default_set;
        this->length      = 0;
    }
    Segment(Point a, Point b)
    {
        this->start  = a;
        this->end    = b;
        double diff1 = a.x - b.x;
        double diff2 = a.y - b.y;
        double sum   = pow(diff1, 2) + pow(diff2, 2);
        this->length = sqrt(sum);
    }
};

Segment compareSegments(Segment, Segment);
std::vector<Segment> generateEdges(std::vector<Point>);
Segment findBiggestEdge(std::vector<Segment>);
double getEuclideanDistance(Point, Point);
bool compareSeg(Segment, Segment);
std::vector<Segment> generateDiagonals(std::vector<Point>);
Segment findBiggestSegment(std::vector<Segment>);
bool onSegment(Point, Point, Point);
int orientation(Point, Point, Point);
int isLeft(Point, Point, Point);
bool pointIsOutside(Point, std::vector<Point>);
bool doIntersect(Segment, Segment);
bool isGoodDiagonal(Segment, std::vector<Segment>, std::vector<Point>);
Segment getBiggestSegmentPossible(std::vector<Point>);  // Everything integrated

#endif /* GEOMETRY_HPP */
```

Geometry.cpp

```
#include "geometry.hpp"
#include <algorithm>

using namespace std;

double getEuclideanDistance(Point a, Point b)
{
    /**
     * This method returns the distance between two points.
     */
    double diff1    = a.x - b.x;
    double diff2    = a.y - b.y;
    double sum      = pow(diff1, 2) + pow(diff2, 2);
    double distance = sqrt(sum);
    return distance;
}

std::ostream& operator<<(std::ostream& os, const Point& point)
{
    /**
```

```
     * Implements the output operator for the Point class.
     */
    os << "(" << point.x << ',' << point.y << ")" << endl;
    return os;
}

std::ostream& operator<<(std::ostream& os, const Segment& segment)
{
    /**
     * Implements the output operator for the Segment class.
     */
    os << "(" << segment.start.x << ',' << segment.start.y << ")";
    os << " ------ ";
    os << "(" << segment.end.x << ',' << segment.end.y << ")" << endl;
    return os;
}

bool operator==(const Point& one, const Point& two)
{
    if (one.x == two.x && one.y == two.y) return true;
    return false;
}

bool operator==(const Segment& one, const Segment& two)
{
    if (one.start == two.start && one.end == two.end)
        return true;
    else if (one.start == two.end && one.end == two.start)
        return true;
    return false;
}

Segment compareSegments(Segment a, Segment b)
{
    /*
     * Returns the Segment object with the largest length.
     */
    if (a.getLength() >= b.getLength())
        return a;
    else
        return b;
}

bool compareSeg(Segment a, Segment b)
{
    /**
     * This was written for vector::sort
     */
    return (a.getLength() < b.getLength());
}

double Segment::getLength()
{
    /**
     * Returns the length of the line given
     */
    return this->length;
}

Point Segment::getMidpoint()
{
    /**
     * Returns the midpoint of this segment
     */
    int mx, my;
    mx = (int)(double)(this->start.x + this->end.x) / 2;  // Calculate the mid-point
of the line
    my = (int)(double)(this->start.y + this->end.y) / 2;

    Point point(mx, my);
    return point;
}

Segment findBiggestEdge(std::vector<Segment> all_edges)
{
    /**
     * Returns the biggest edge of the polygon as a segment object.
     */
    Segment current_biggest = all_edges[1];
    for (auto& segment : all_edges)  // access vector by reference to avoid copying?
    {
        current_biggest = (compareSegments(current_biggest, segment));
    }
    return current_biggest;
}

std::vector<Segment> generateDiagonals(std::vector<Point> polygon)
{
    /**
     * Generate and return a list segments that represents all the
     * possible diagonals that can be generated from the given list of
     * points which make up the polygon. The points are expected to be
     * sorted counterclockwise and more than 3 in size(No diagonals
     * otherwise).
     */

    vector<Segment> diagonals;  // This will contain all the diagonals generated

    {
        // Generate diagonals from the first point of the polygon
        vector<Point>::iterator it1 = polygon.begin();
        Point point_one             = *it1;
        std::advance(it1, 2);  // Skip a point
        for (vector<Point>::iterator it2 = it1; it2 != polygon.end(); it2++) {
            Segment diagonal_line = {point_one, *it2};
            diagonals.push_back(diagonal_line);
        }
        diagonals.pop_back();  // Remove the last line it is an edge
    }
    {
        // This is for rest of the points in the polygon
        vector<Point>::iterator it1 = polygon.begin();
        for (it1 = ++it1; it1 != polygon.end(); it1++) {
            Point point_one                 = *it1;
            vector<Point>::iterator temp_it = it1;  // Temporarily store iterator
            if (++it1 != polygon.end())
                it1++;  // Skip a point
            else
                break;  // We are at the last point, all diagonals are made
            for (vector<Point>::iterator it2 = it1; it2 != polygon.end(); it2++) {
                Segment diagonal_line = {point_one, *it2};
                diagonals.push_back(diagonal_line);
            }
            it1 = temp_it;  // Reset it11 to its begining position
        }
    }

    return diagonals;
}

std::vector<Segment> generateEdges(std::vector<Point> all_points)
{
    vector<Segment> polygon;
    Point current = all_points[0];
    Point next;
    // Generates edges except for the last edge of the polygon
    for (int count = 1; count < all_points.size(); count++) {
        next = all_points[count];
        polygon.push_back(Segment(current, next));
        current = all_points[count];
    }
    // Creates the last segment (connects to starting point)
    next = all_points[0];
    polygon.push_back(Segment(current, next));
    return polygon;
}

//===================================================================
// a Point is defined by its coordinates {int x, y;}

int isLeft(Point P0, Point P1, Point P2)
{
    /**
     *isLeft(): tests if a point is Left|On|Right of an infinite line.
     */
    return ((P1.x - P0.x) * (P2.y - P0.y) - (P2.x - P0.x) * (P1.y - P0.y));
}

bool pointIsOutside(Point P, std::vector<Point> polygon)
{
    /**
     * Crossing number test to see if the given point lines inside the polygon.
     * This code is patterned after [Franklin, 2000]
     */
    int cn = 0;  // the  crossing number counter

    for (auto V = polygon.begin(); V < polygon.end() - 1; ++V) {
        if (((V[0].y <= P.y) && (V[1].y > P.y))        // an upward cross0ng
            || ((V[0].y > P.y) && (V[1].y <= P.y))) {  // a downward cross0ng
            // compute  the actual edge-ray intersect x-coordinate
            float vt = (float)(P.y - V[0].y) / (V[1].y - V[0].y);
            if (P.x < V[0].x + vt * (V[1].x - V[0].x))  // P.x < 0ntersect
                ++cn;                                   // a valid crossing of y=P.y
right of P.x
        }
    }

    // 0 if even (out), and 1 if  odd (in)
    if ((cn & 1) == 0) return true;
    return false;
```

```
}

bool onSegment(Point p, Point q, Point r)
{
    /**
     * Check if point r is on the line segment made by point p and q.
     */
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y <= max(p.y, r.y) &&
        q.y >= min(p.y, r.y))
        return true;
    return false;
}

int orientation(Point p, Point q, Point r)
{
    /**
     * To find orientation of ordered triplet (p, q, r).
     * The function returns following values
     * 0 --> p, q and r are colinear
     * 1 --> Clockwise
     * 2 --> Counterclockwise
     */
    int val = (int)(double)(q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    // Cast explicitly without warnings
    if (val == 0) return 0;    // colinear
    return (val > 0) ? 1 : 2;  // clock or counterclock wise
}

bool doIntersect(Segment a, Segment b)
{
    /**
     * Find the four orientations needed for general and special
     * cases, and returns true if the line segments intersect.
     */
    Point p1 = a.start;
    Point q1 = a.end;
    Point p2 = b.start;
    Point q2 = b.end;

    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4) return true;
    // Special Cases
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;
    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;
    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;
    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;
    return false;  // Doesn't fall in any of the above cases
}

bool isGoodDiagonal(Segment segment, std::vector<Segment> checkable,
std::vector<Point> polygon)
{
    /**
     * Check if the given segment intersects with any of the given lines
     */
    for (auto& checking : checkable) {
        if (pointIsOutside(segment.getMidpoint(), polygon))
            return true;
        else if (checking.start == segment.start || checking.end == segment.end ||
                 checking.start == segment.end || checking.end == segment.start)
            return false;  // Special case where the intersection is happening at the
end/start
        else if (doIntersect(checking, segment))
            return true;
        else
            return false;  // No intersections
    }
    return false;  // No intersections
}

Segment getBiggestSegmentPossible(std::vector<Point> points)
{
    /**
     * This method integrates all other method to find the biggest
     * line that can draw along a polygon.
     */
    Segment biggest_line;
    size_t num               = points.size();  // Get the vectors size
    vector<Segment> polygon = generateEdges(points);
    biggest_line            = findBiggestEdge(polygon);
    if (num <= 2) {
        cout << "\nINVALID POLYGON\n";
        exit(1);
    }
    if (num == 3) {
        return biggest_line;
    }

    vector<Segment> diagonals = generateDiagonals(points);
    std::sort(diagonals.begin(), diagonals.end(), compareSeg);

    while (diagonals.size() > 0) {
        Segment current = diagonals.back();
        diagonals.pop_back();
        if (!isGoodDiagonal(current, polygon, points)) {
            if (current.getLength() > biggest_line.getLength()) return current;
        } else {
            continue;
        }
    }
    return biggest_line;
}
```

System Specifications:

Processor Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2501 Mhz, 4 Core(s), 4 Logical Processor(s)

OS Microsoft Windows 10 Enterprise, Version 10.0.15063 Build 15063

GPU Name **NVIDIA Quadro M620**

Installed Physical Memory (RAM) 8.00 GB

Total Physical Memory 7.64 GB

Available Physical Memory 2.55 GB

System Type x64-based PC

Compiler MVSC15 for Windows C++