Essential Algorithms Coursework (CS2AO17)

Jounaid Ruhomaun (bg016931)

In the following report I will document the development of my 'Robot Path-planning' program (in Java) which implements the D&C convex hull algorithm in order to traverse a robot from *point A* to *point B* whilst avoiding the *polygon P* in-between.

Set Up

Cartesian Grid Class:

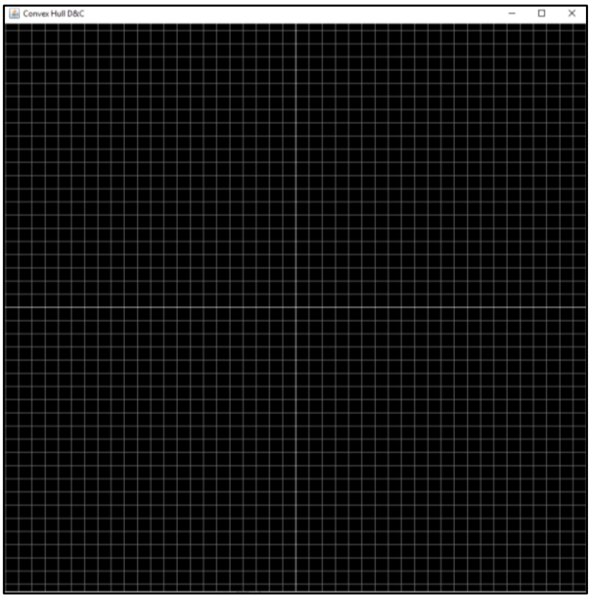
I initially needed a normalised Cartesian grid where I could draw points, lines and shapes as per the requirements of this coursework. This also made implementing the D&C class easier as I could just feed it an array of normal points and expect the same output.

```java
public class CartesianPlane extends JPanel {

    public final double spacing = 20;

    double width;
    double height;
    double xaxis;
    double yaxis;

    static Polygon shapes[] = new Polygon[2];

    public static void initialiseShapes() {

    public CartesianPlane() {
        setBackground(Color.BLACK);
    }

    public void lines(Graphics2D g, double x1, double y1, double x2, double y2) {
        x1 = xaxis + x1 * spacing;
        y1 = yaxis - y1 * spacing;
        x2 = xaxis + x2 * spacing;
        y2 = yaxis - y2 * spacing;

        g.setStroke(new BasicStroke(2));
        g.draw(new Line2D.Double(x1, y1, x2, y2));
    }

    public void point(Graphics2D g, double x, double y) {
        x = xaxis + x * spacing;
        y = yaxis - y * spacing;

        g.draw(new Line2D.Double(x, y, x, y));
    }

    public void shape(Graphics2D g, Polygon p) {
        g.setStroke(new BasicStroke(1));

        for (int i = 0; i < p.returnCornerLength() - 2; i++) {

            lines((Graphics2D) g, p.Corners[i].x, p.Corners[i].y, p.Corners[i + 1].x, p.Corners[i + 1].y);
        }
        lines((Graphics2D) g, p.Corners[p.returnCornerLength() - 2].x, p.Corners[p.returnCornerLength() - 2].y, p.Corners[0].x, p.Corners[0].y);

    }

    public void pointSet(Graphics2D g, Polygon p) {
        g.setStroke(new BasicStroke(4));

        for (int i = 0; i < p.returnCornerLength(); i++) {
            point(g,p.Corners[i].x,p.Corners[i].y);
        }
    }
```

As you can see above, I used the variable *spacing* to scale the JPanel so every 20 pixels = 1 space for my grid (this makes the grid easy to plot/see points and lines). The four methods shown allow me to: draw lines between points, draw a point, draw a shape (set of points joined by lines) and draw a set of points. These will be used later. The background is set to black.

```java
    @Override
    public void paint(Graphics g) {

        super.paint(g);

        width = getWidth();
        height = getHeight();

        xaxis = width / 2.0;
        yaxis = height / 2.0;
        double x1 = 0;
        double y1 = 0;
        double x2 = width;
        double y2 = height;

        Graphics2D g2 = (Graphics2D) g;

        g2.setColor(Color.GRAY);
        g2.setStroke(new BasicStroke(1));

        for (double x = spacing; x < width; x += spacing) {

            g2.draw(new Line2D.Double(xaxis + x, y1, xaxis + x, y2));
            g2.draw(new Line2D.Double(xaxis - x, y1, xaxis - x, y2));
        }

        for (double y = spacing; y < height; y += spacing) {

            g2.draw(new Line2D.Double(x1, yaxis + y, x2, yaxis + y));
            g2.draw(new Line2D.Double(x1, yaxis - y, x2, yaxis - y));
        }

        g2.setColor(Color.WHITE);
        g2.draw(new Line2D.Double(x1, yaxis, x2, yaxis));
        g2.draw(new Line2D.Double(xaxis, y1, xaxis, y2));
```

The screenshot to the left shows how the grid is drawn onto the frame using two sets of for loops to draw the grid in grey, and the X/Y axis in white (frame shown in the screenshot below).



Polygon Class:

The next class to be implemented was the *Polygon* class which would allow me to store a set of points representing the corners of a polygon. This class will also be used to represent the convex hulls. The constructor sets the amount of corners for the polygon and also initializes each point in the *Corners[]* array. The methods include general getters and setters with a method to display each point in the polygon to the console (used primarily in debugging).

```java
package reading.ac.uk.bg016931.jounaid.ruhomaun.ConvexHullPathFinding;

import java.awt.BasicStroke;

public class Polygon extends JPanel {
    public Point Corners[];

    public Polygon(int points) {
        Corners = new Point[points];
        for (int i = 0; i < Corners.length; i++)
            Corners[i] = new Point();
    }

    public int getX(int x) {
        x = Corners[x].x;
        return x;
    }

    public int getY(int y) {
        y = Corners[y].y;
        return y;
    }

    public int returnCornerLength() {
        return Corners.length;
    }

    public void setPoint(int pointNo, int x, int y) {
        Corners[pointNo].setLocation(x, y);
    }

    public void displayPoints() {
        for (int i = 0; i < Corners.length; i++)
            System.out.println(Corners[i].toString());
        System.out.println("\n");
    }
```

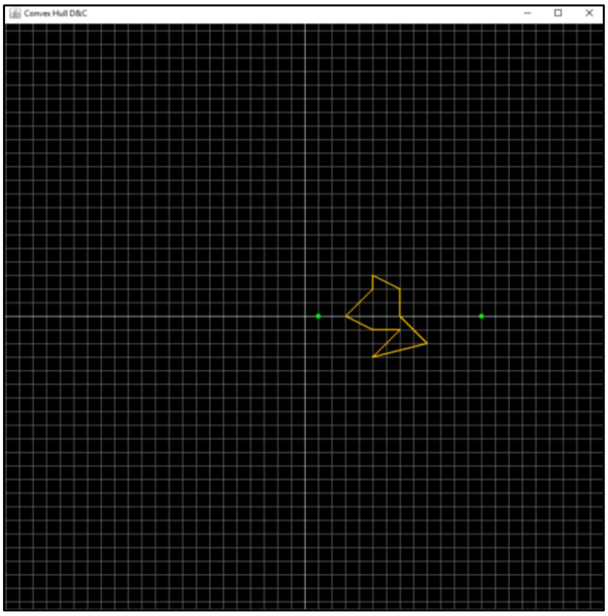
```java
static Polygon shapes[] = new Polygon[2];

public static void initialiseShapes() {
    shapes[0] = new Polygon(10);
    shapes[0].setPoint(0, 3, 0);
    shapes[0].setPoint(1, 5, 2);
    shapes[0].setPoint(2, 5, 3);
    shapes[0].setPoint(3, 7, 2);
    shapes[0].setPoint(4, 7, 0);
    shapes[0].setPoint(5, 9, -2);
    shapes[0].setPoint(6, 5, -3);
    shapes[0].setPoint(7, 7, -1);
    shapes[0].setPoint(8, 5, -1);
}
```

The above screenshot (taken from the *CartisianPlane* class) shows how the obstacle polygons are initialised in an array called *shapes[]*. The method *initialiseShapes()* sets the points for each corner, for each shape (this is done manually).

Start Class:

The final class required for the setup is JPanel driver class (shown in the top left screenshot). The title is set to *'Convex Hull D&C'* and the window size is set to 900x900 giving us a 45x45 grid when the spacing is normalised.

```java
package reading.ac.uk.bg016931.jounaid.ruhomaun.ConvexHullPathFinding;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class start {

    static CartesianPlane grid = new CartesianPlane();

    public static void main(String args[]) {

        JFrame window = new JFrame("Convex Hull D&C");
        JPanel panel = new JPanel();

        window.add(grid);
        window.setSize(900, 900);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
        window.add(panel);
    }
}
```



```java
        g2.setColor(Color.GREEN);
        g2.setStroke(new BasicStroke(6));
        Point a = new Point(1,0);
        Point b = new Point(13,0);

        point(g2, a.x, a.y);
        point(g2, b.x, b.y);

        initialiseShapes();

        g2.setColor(Color.ORANGE);
        shape(g2, shapes[0]);

```

The above screenshot shows how the methods I created before are used to draw the initial set up (as per the coursework requirements). *Point a* is set so 1,0 and *point b* is set to 13,0 and the shape created earlier is drawn in orange in-between the points (shown in the screenshot to the left).

Divide and Conquer class:

In order to create a convex hull for a set of points, another class must be made. This allows me to instantiate multiple D&C instances for different sets of points, making task 2 much easier.

```java
    public DivideAndConquer(Polygon p, Point a, Point b) {
        Polygon initialPoints = new Polygon(p.returnCornerLength() + 1);

        initialPoints.setPoint(0, a.x, a.y);

        for (int i = 0; i < p.returnCornerLength(); i++) {
            initialPoints.setPoint(i + 1, p.Corners[i].x, p.Corners[i].y);
        }

        initialPoints.setPoint(p.returnCornerLength(), b.x, b.y);

        initialPoints.displayPoints();

        Arrays.sort(initialPoints.Corners, new Comparator<Point>() {
            public int compare(Point a, Point b) {
                int xComp = Integer.compare(a.x, b.x);
                return xComp;
            }
        });

        initialPoints.displayPoints();
        ans = divide(initialPoints);
    }

    public Polygon calcDandC() {
        return ans;
    }

    public int returnAnsLength() {
        return ans.returnCornerLength();
    }
}
```

The constructor takes in the obstacle polygon and both the start and end points. All these points are then put into a single polygon called *initialPoints.* The polygon's points are then ordered by the x value. A globally initialised polygon called *ans* is used store the convex hull for *initalPoints* and can be accessed using the method shown below the constructor.

The *initialPoints* are passed to the *divide()* method which carries out the D&C algorithm.

```java
    private Polygon divide(Polygon a) {

        if (a.returnCornerLength() <= 6)
            return bruteHull(a);

        Polygon left = new Polygon(a.returnCornerLength() / 2);
        Polygon right = new Polygon(a.returnCornerLength() / 2);

        for (int i = 0; i < a.returnCornerLength() / 2; i++)
            left.Corners[i] = a.Corners[i];
        for (int i = a.returnCornerLength() / 2; i < a.returnCornerLength(); i++)
            right.Corners[i - a.returnCornerLength() / 2] = a.Corners[i];

        Polygon left_hull = divide(left);
        Polygon right_hull = divide(right);

        left_hull.displayPoints();
        right_hull.displayPoints();

        return merger(left_hull, right_hull);
    }
```

To the left you can see the general driver method for the D&C class. It takes in a *Polygon* object and checks whether the amount of points is considerd 'small' (<=6 for my implementation, this can be adjusted). If so, it will proceed to brute force calculate the convex hull and return that. Polygons with a point amount above the threshold are split into two parts, *left* and *right.* Both halves are then passed into the divide class again. The two halves are then merged to form one convex hull. You can see how this recursion will split large polygons into chunks of 6 point polygons that will be calculated separately and then joined together to form the overall convex hull.

```java
    private Polygon bruteHull(Polygon a) {

        Set<Point> s = new HashSet<Point>();

        for (int i = 0; i < a.returnCornerLength(); i++) {
            for (int j = i + 1; j < a.returnCornerLength(); j++) {

                int x1 = a.Corners[i].x, x2 = a.Corners[j].x;
                int y1 = a.Corners[i].y, y2 = a.Corners[j].y;

                int a1 = y1 - y2;
                int b1 = x2 - x1;
                int c1 = (x1 * y2) - (y1 * x2);
                int pos = 0, neg = 0;
                for (int k = 0; k < a.returnCornerLength(); k++) {
                    if (a1 * a.Corners[k].x + b1 * a.Corners[k].y + c1 <= 0)
                        neg++;
                    if (a1 * a.Corners[k].x + b1 * a.Corners[k].y + c1 >= 0)
                        pos++;
                }
                if ((pos == a.returnCornerLength()) || (neg == a.returnCornerLength())) {
                    s.add(a.Corners[i]);
                    s.add(a.Corners[j]);
                }
            }
        }

        Polygon ret = new Polygon(s.size());
        ret.Corners = s.toArray(new Point[s.size()]);

        ret.displayPoints();

        mid.setLocation(0, 0);
        int n = ret.returnCornerLength();
        for (int i = 0; i < n; i++) {
            mid.x += ret.Corners[i].x;
            mid.y += ret.Corners[i].y;
            ret.Corners[i].x *= n;
            ret.Corners[i].y *= n;
        }

        Arrays.sort(ret.Corners, new Comparator<Point>() {
            public int compare(Point p1, Point q1) {
                if (comp(p1,q1) == true)
                    return 1;
                if (comp(p1,q1) == false)
                    return -1;
                else
                    return -1;
            }
        });

        for (int i=0; i<n; i++) {
            ret.Corners[i].setLocation(ret.Corners[i].x / n, ret.Corners[i].y / n);
        }

        return ret;
    }
```

The screenshot to the left shows my brute force method to calculate the convex hull. A set of points is initialized which stores the points considered to be on the convex hull. Each pair of points is then consecutively checked for whether it is in the bounds of the convex hull. If all the remaining points are on the same side of the line created by the initial pair of points, then that line will be an edge of the convex hull. The *Polygon ret* stores the points of the convex hull. This polygon is then sorted and oriented before being returned.

```java
    private Polygon merger(Polygon a, Polygon b) {
        int n1 = a.returnCornerLength(), n2 = b.returnCornerLength();

        int ia = 0, ib = 0;
        for (int i = 1; i < n1; i++)
            if (a.Corners[i].x > a.Corners[ia].x)
                ia = i;

        for (int i = 1; i < n2; i++)
            if (b.Corners[i].x < b.Corners[ib].x)
                ib = i;

        int inda = ia, indb = ib;
        boolean done = false;
        while (done = false) {
            done = true;
            while (orientation(b.Corners[indb], a.Corners[inda], a.Corners[(inda + 1) % n1]) >= 0)
                inda = (inda + 1) % n1;
            while (orientation(a.Corners[inda], b.Corners[indb], b.Corners[(n2 + indb - 1) % n2]) <= 0) {
                indb = (n2 + indb - 1) % n2;
                done = false;
            }
        }
        int uppera = inda, upperb = indb;
        inda = ia;
        indb = ib;
        done = false;
        int g = 0;
        while (done = false) {
            done = true;
            while (orientation(a.Corners[inda], b.Corners[indb], b.Corners[(indb + 1) % n2]) >= 0)
                indb = (indb + 1) % n2;
            while (orientation(b.Corners[indb], a.Corners[inda], a.Corners[(n1 + inda - 1) % n1]) <= 0) {
                inda = (n1 + inda - 1) % n1;
                done = false;
            }
        }

        int lowera = inda, lowerb = indb;
        Polygon ret = new Polygon(a.returnCornerLength() + b.returnCornerLength());
        int i = 0;

        int ind = uppera;
        ret.setPoint(i, a.Corners[uppera].x, a.Corners[uppera].y);
        i++;
        while (ind != lowera) {
            ind = (ind + 1) % n1;
            ret.setPoint(i, a.Corners[ind].x, a.Corners[ind].y);
            i++;
        }
        ind = lowerb;
        ret.setPoint(i, b.Corners[lowerb].x, b.Corners[lowerb].y);
        i++;
        while (ind != upperb) {
            ind = (ind + 1) % n2;
            ret.setPoint(i, b.Corners[ind].x, b.Corners[ind].y);
            i++;
        }
        return ret;
    }
```

In order to merge the two convex hulls, we must first find the upper tangents of the two polygons (*a* and *b*). The variable *ib* stores the leftmost point of *b* and vice versa the variable *ia* stores the rightmost point of a, calculated through the above for loops. The upper and lower tangents are then calculated and the complete merged convex hull is stored in the *Polygon ret*. This object is returned.

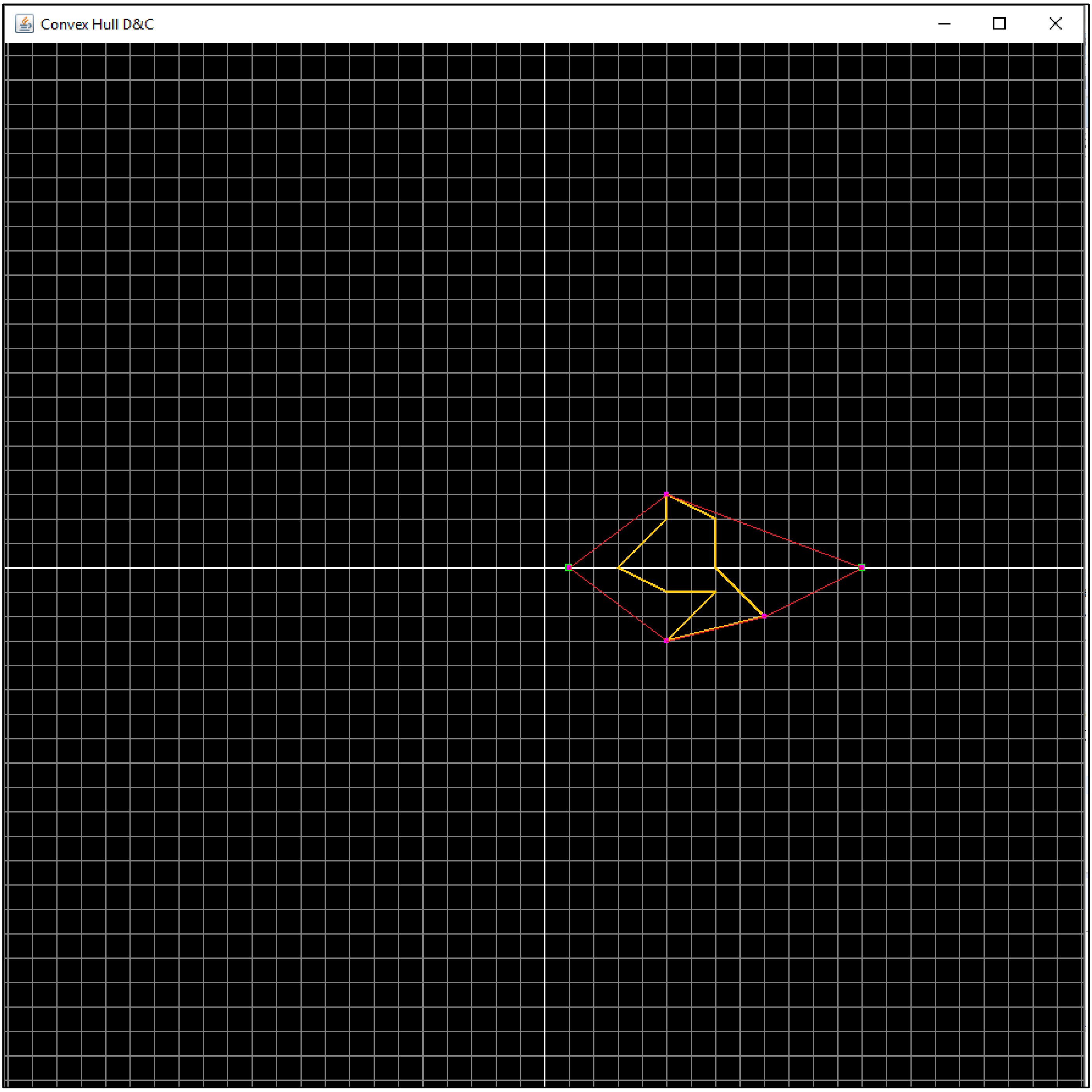
```java
package reading.ac.uk.bg016931.jounaid.ruhomaun.ConvexHullPathFinding;

import java.awt.Point;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.Set;

public class DivideAndConquer {
    Polygon ans;
    Point mid = new Point();

    private int quad(Point p) {
        if (p.x >= 0 && p.y >= 0)
            return 1;
        if (p.x <= 0 && p.y >= 0)
            return 2;
        if (p.x <= 0 && p.y <= 0)
            return 3;
        return 4;
    }

    private boolean comp(Point p1, Point q1) {
        Point p = new Point(p1.x - mid.x, p1.y - mid.x);
        Point q = new Point(q1.x - mid.x, q1.y - mid.x);

        int one = quad(p);
        int two = quad(q);

        if (one != two)
            return (one < two);
        return (p.y * q.x < q.y * p.x);
    }

    private int orientation(Point a, Point b, Point c) {

        int res = (b.y - a.y) * (c.x - b.x) - (c.y - b.y) * (b.x - a.x);
        if (res == 0)
            return 0;
        if (res > 0)
            return 1;
        return -1;
    }

```

To the left you can see the remaining three methods of the *DivideAndConquer* class. These classes are used for polygon orientation and are not particularly relevant to the D&C convex hull algorithm.

Task One:

```
        DivideAndConquer aTob = new DivideAndConquer(shapes[0],a,b);

        g2.setColor(Color.MAGENTA);
        aTob.calcDandC().displayPoints();
        pointSet(g2, aTob.calcDandC());

```

My grid is already set for task one, all that's left to do is implement a few lines of code to pass the obstacle polygon and the points to the D&C algorithm (shown in the screenshot to the left)



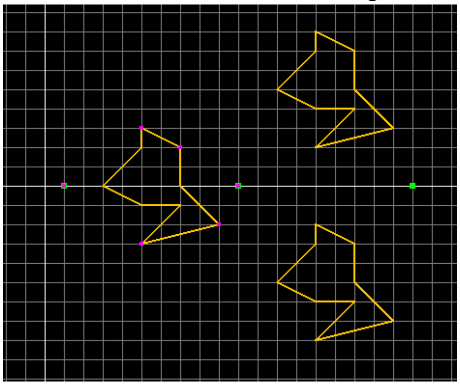
As you can see in the screenshot, the magenta points represent each point on the convex hull. The red line (drawn on after) helps to visualize what this convex hull shape will look like, and the possible paths the robot can take around the obstacle. For further proof of functionality see the screenshot below, which shows points of the convex hull printed to the console.

```
Problems  @ Javadoc  Declaration  Console
<terminated> start [Java Application] C:\Program Files (x86)\Java\jre1.8.0_151\bin\javaw.exe (21 Jan 2019, 04:33:07)


java.awt.Point[x=13,y=0]
java.awt.Point[x=9,y=-2]
java.awt.Point[x=5,y=-3]
java.awt.Point[x=5,y=3]
java.awt.Point[x=1,y=0]
```

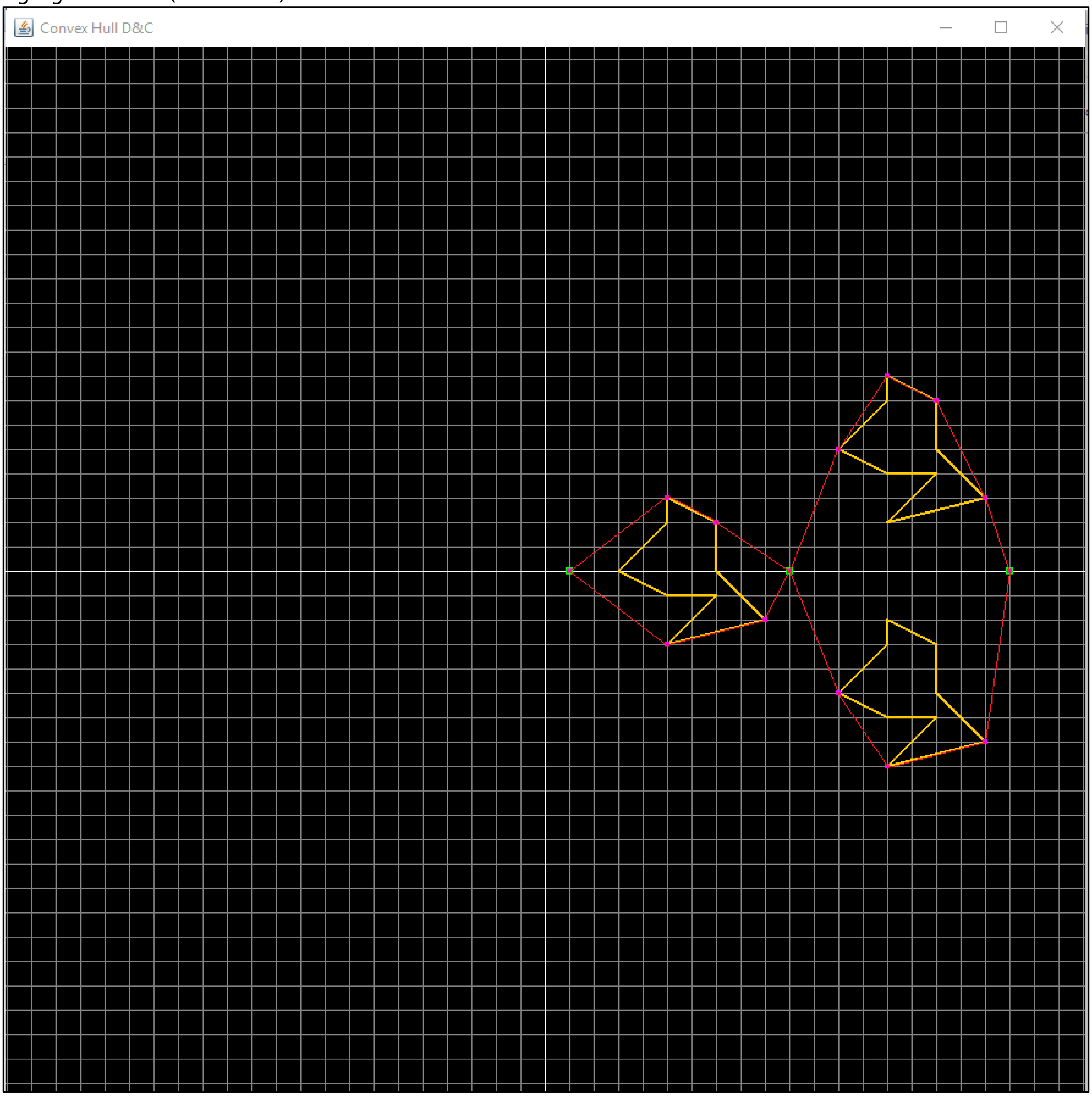
Task Two:

```java
        g2.setColor(Color.GREEN);
        g2.setStroke(new BasicStroke(6));
        Point a = new Point(1,0);
        Point b = new Point(10,0);
        Point c = new Point(19,0);

        point(g2, a.x, a.y);
        point(g2, b.x, b.y);
        point(g2, c.x, c.y);

        initialiseShapes();

        g2.setColor(Color.ORANGE);
        shape(g2, shapes[0]);
        shape(g2, shapes[1]);
        shape(g2, shapes[2]);

        DivideAndConquer aTob = new DivideAndConquer(shapes[0],a,b);

        g2.setColor(Color.MAGENTA);
        pointSet(g2, aTob.calcDandC());

        DivideAndConquer bToc = new DivideAndConquer(shapes[3],b,c);
        aTob.calcDandC().displayPoints();
        bToc.calcDandC().displayPoints();
        pointSet(g2, bToc.calcDandC());

    }

}
```

Only a few modifications have to be made to the *Paint* method for it to be suitable for task two. A new point *(Point c)* is added to the grid 9 squares right of point b (b now changed to x = 10, y = 0). Two more obstacles (initialised as having the same shape as the previous obstacle, but being translated (+9,+5), (+9,-5) respectively (as per the coursework requirements)) are added in-between points *b* and *c*. This can be seen in the image below.

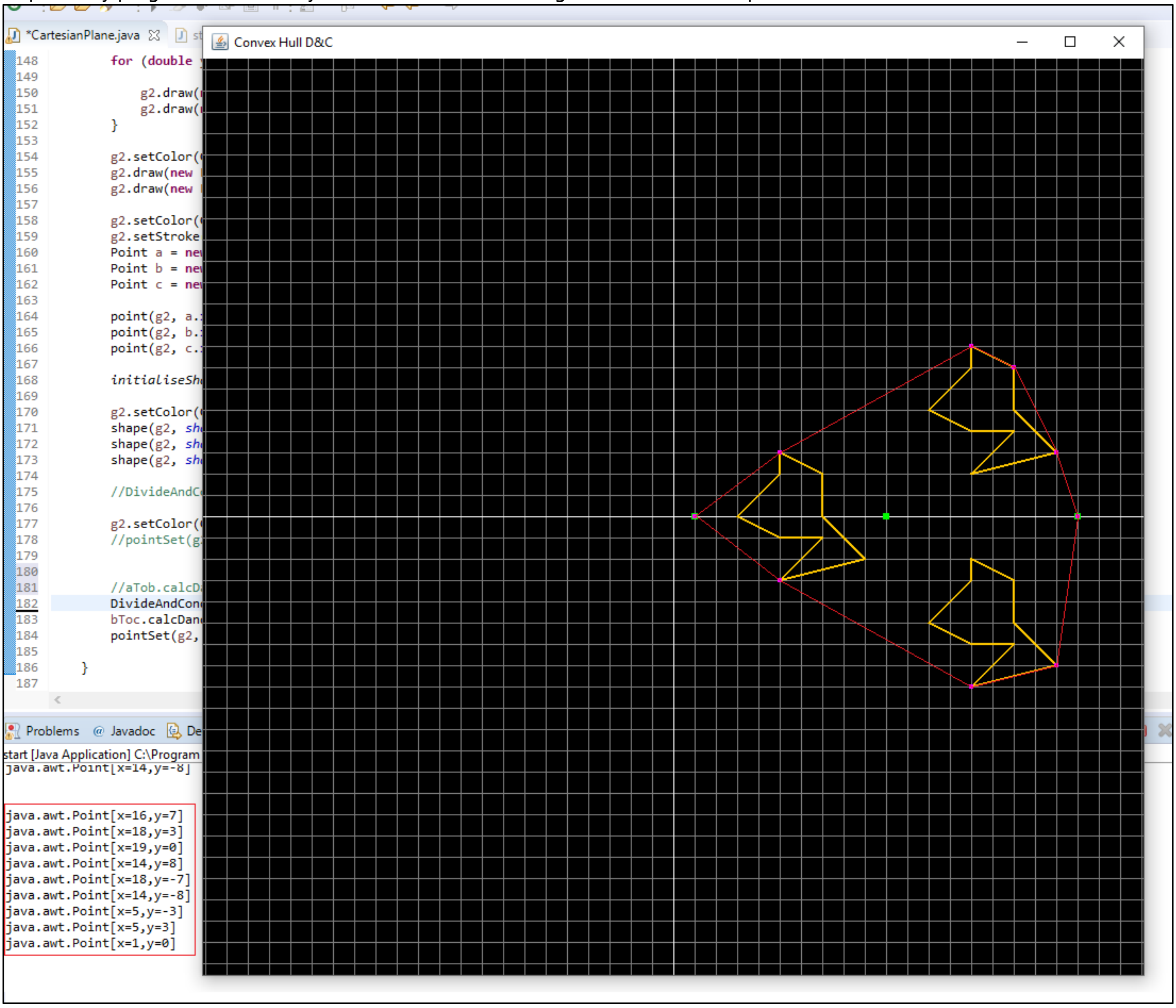


A new *DivideAndConquer* object is initiated with *shapes[3]* (the points of the shapes one and two added consecutively), *Point b* and *Point c* being passed to it. The image below shows the complete convex hull pathing points (in magenta) from *a* to *b* to *c*, highlighted in red (drawn after).



Further Proof of Functionality:

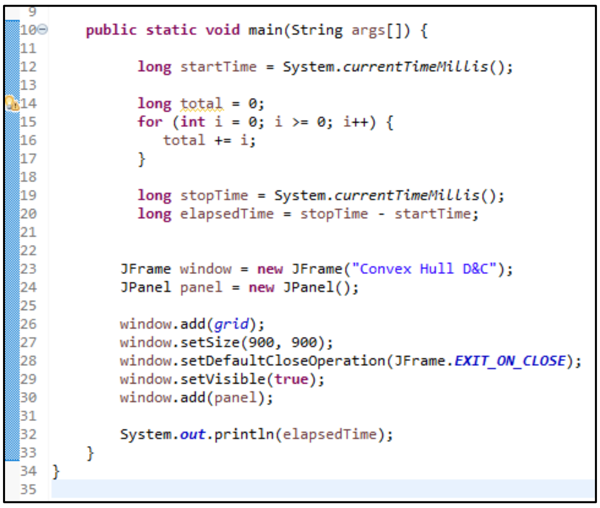
To prove my programs functionality, I will now instantiate a single D&C class for the path *a* to *c.*



As you can see with only a few simple modifications I can calculate the convex hull for the every point on the grid (from *a* to *c).* Highlighted in the red box in the bottom right corner of the screenshot is the console output of the convex hull points (for further proof).

Conclusion and Analysis:

To conclude, my program is suitable for both tasks one and two, and any other combination of polygons and points. I have constructed a D&C algorithm to calculate the convex hulls of sets of points on the grid. I have displayed my results through a JPanel GUI with colour coded components making the project easier to visualize.



```
    public static void main(String args[]) {

        long startTime = System.currentTimeMillis();

        long total = 0;
        for (int i = 0; i >= 0; i++) {
            total += i;
        }

        long stopTime = System.currentTimeMillis();
        long elapsedTime = stopTime - startTime;


        JFrame window = new JFrame("Convex Hull D&C");
        JPanel panel = new JPanel();

        window.add(grid);
        window.setSize(900, 900);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
        window.add(panel);

        System.out.println(elapsedTime);
    }
}
```

Upon analysis of my program, it seems to be running faster when I simply brute force the points by bypassing the D&C methods. I tested my program using all three shapes, going from points *a* to *c* (a total of 28 points, every single point on the grid). The system timer I used can be seen in the screenshot to the right.

When brute forcing the *elapsedTime* came to an average of 3.430 seconds whereas going through the D&C methods resulted in a average time of 3.842 seconds. This is unexpected as brute forcing has a time complexity of O(n^3) where as my D&C algorithm has a time complexity of O(n * log n).

I believe this is due to the fact that I have not tested the program with a 'large' (x>1000) number of points. I could extend my program to calculate the convex hull of a random set of points, this would allow me to test for a large number of points.