Isai Eliseo Mercado Oliveros
Misaie
Section 1
Time spent: 9 hours.

PSEUDOCODE

Function solve. On input = list of points

-Based on the X coordidate, grab most left point and most right point

Time = O(n);

- Call divideList. On imput list of points, most left point, most right point

Divide points-cointainer into two containers. One container with upper points and another container with lower points.

Time = O(n);

- call solving. On input upper-points container, most left point, most right point
- call solving. On input lower-points container, most right point, most left point

Time = O(nlogn)

Function solving. On input = list of points, most left point, most right point

-If list of points has a size zero, then return

-Else if list of points has a size of 1, then save that point into global container of perimeter points, then return

-Else call findFurthestPoint. On input list of points

-take that furthest point and add it to global container of perimeter points

- Call divideList. On input = list of points , most left point, furthest point

Divide current list of points into south points and north point according to the tabgent created by most left and furthest point

- Call solving. On input = upper list of points, most left pint, furthest point
- Call solving. On input = lower list of points, furthest point, most right point

Function findFurthestPoint. On input = list of points, most left point, most right point

- For every point point in the list
- Call findDistance. On input = most left point, most right point, point. Return distance
- if this distance is greater than last distance, then this distance is saved, else do nothing
- after passing all points return greatest distance

Function findDistance. On input = most left point, most right point, point

- in order to avoid using divition, distance is computed by comparing magnitude of the cross product of the three points rather than computing tangents. In addition we only nee a relative distance not the exact distance.
- cross product = ((most left point X - most right X) * (most left point Y - point Y)) * ((most left point Y - most right point Y) * (most left point X - point X))
- Return absolute value of cross product

Function divideList. On input = list of points, most left point, most right point

- For every point in the list of points call signCrossProduct On input = most left point, most right point, point. Return positive one or negative one
- If sign of cross product is positive one, then add point to upper points list

Else if sign of cross product is negative one, then add point to lower points list

Function signCrossProduct. On input = most left point, most right point, point

- it does the same as crossProduct function but this function does not use absolute value, so it returns the sign by returning positive one or negative one

WORST CASE THEORETICAL ANALISYS

It starts as a list of size n

- The list is divided in two by splitting points to north or south points sub-lists

Time of operation is $O(n)$ because we read every point in the list.

- On every half we call a recursive function that will split the lists in two again

-All the other operations are size n. There are no n^2 functions because no point has to do operations with every single point of the list again.

The function is $T(n) = 2\,T(n/2) + n$

The general solution is $C1n + C2n\log_2(n)$

By master theorem worst case is $O(n\log(n))$

Specific solution

$T(2^k)=2T(2^k/2) + 2^k$

$T(2^k)=2T(2^{k-1}) + 2^k$

$T(n)=2T(n-1) + 2^k$

$T(n) - 2T(n-1) = 2^k$

$x^2 - 2x = 2^k n^0$

$x(x - 2) (x - 2)^{0+1}$

$x = 2, x = 2$

$T(n) = C1\,2^k + C2\,k2^k$

$T(2^k) = C1\,2^{\log_2(n)} + C2\,\log_2(n)2^{\log_2(n)}$

$T(n) = C1\,n + C2\,n\log(n)$

$n = 10, T(10) = 0.0754461$

$n = 20, T(20) = 2 * 0.0754461 + 20 = 20.1508922$

$n = 10, C1\,10 + C2\,10 = 0.0754461$

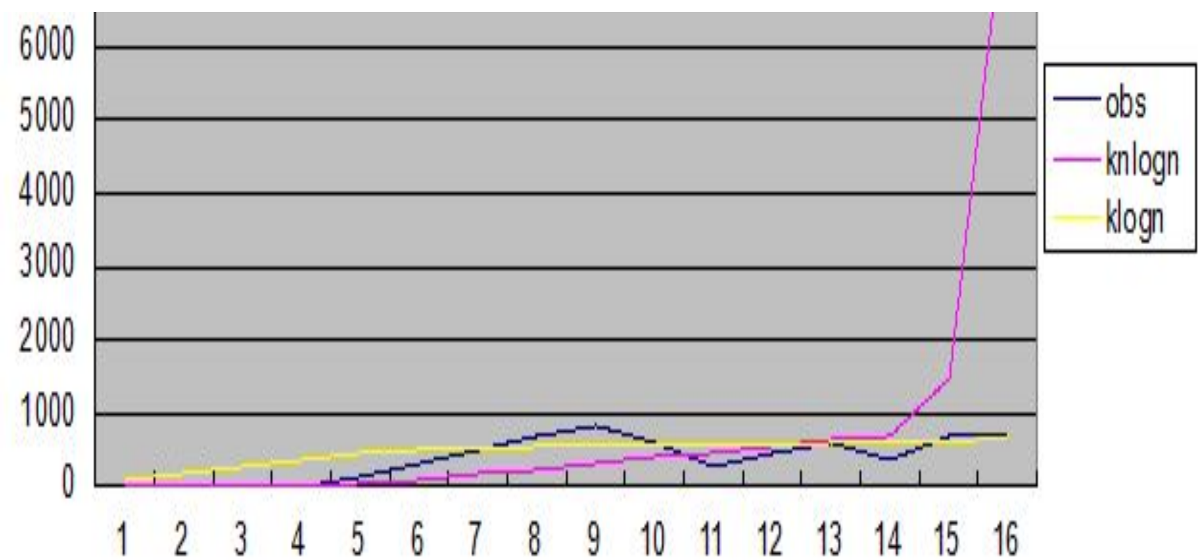
$n = 20, C1\,20+ C2\,26.0206 = 20.150892$

$C1 = -3.31438, C2 = 3.32193$

$T(n)=-3.31438n + 3.32193\,n\log(n)$

EXPERIMENTAL RESULTS

	A	B	C	D	E	F	G	H	I
1	n	obs	nlogn	obs-nlogn	knlogn	obs-k*nlogn	logn	klogn	obs - klogn
2	10	7	10	3	0.0012	6.9988	1	100	-93
3	100	7	200	193	0.024	6.976	2	200	-193
4	1000	10	3000	2990	0.36	9.64	3	300	-290
5	10000	23	40000	39977	4.8	18.2	4	400	-377
6	100000	170	500000	499830	60	110	5	500	-330
7	200000	350	1060205.999	1059855.999	127.22472	222.7752801	5.301029996	530.1029996	-180.1029996
8	300000	530	1643136.376	1642606.376	197.176365	332.8236348	5.477121255	547.7121255	-17.71212547
9	400000	700	2240823.997	2240123.997	268.90	431.1011204	5.602059991	560.2059991	139.7940009
10	500000	835	2849485.002	2848650.002	341.9382	493.0617997	5.698970004	569.8970004	265.1029996
11	600000	600	3466890.75	3466290.75	416.02689	183.97311	5.77815125	577.815125	22.18487496
12	700000	290	4091568.628	4091278.628	490.988235	-200.9882354	5.84509804	584.509804	-294.509804
13	800000	500	4722471.99	4721971.99	566.696639	-66.69663875	5.903089987	590.3089987	-90.3089987
14	900000	600	5358818.258	5358218.258	643.058191	-43.05819102	5.954242509	595.4242509	4.575749056
15	1000000	400	6000000	5999600	720	-320	6	600	-200
16	2000000	725	12602059.99	12601334.99	1512.2472	-787.247199	6.301029996	630.1029996	94.89700043
17	10000000	700	70000000	69999300	8400	-7700	7	700	0
18									
19					k = 0.00012			k=100	
20									



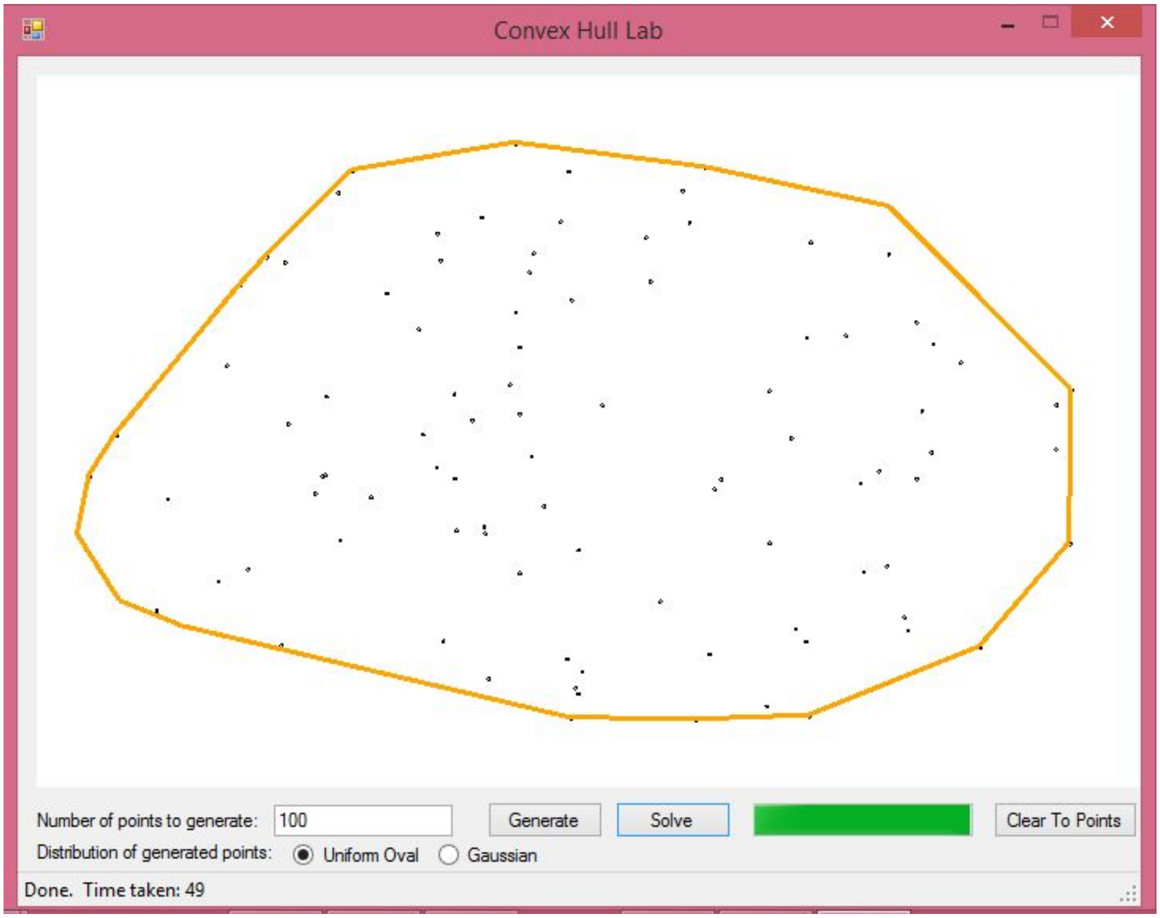
EMPIRICAL ANALYSIS

The theoretical analysis and the experimental results do not match. According to the analysis to my pseudo code and my research on the internet, the upper boundary should be n*log(n).However, I found out that a plot log(n) fits better in comparison to the shape of my results' plot. I could not explain why my quick hull algorithm displays a log(n) behavior. Therefore my calculations seem somewhat out of place. I was thinking on redoing my my theoretical analysis by changing the recurrence relation to a form T(n) = T(n-1) + n. However, my algorithm splits the list in two halves and for each half it calls a recursive function as the following pseudo code shows

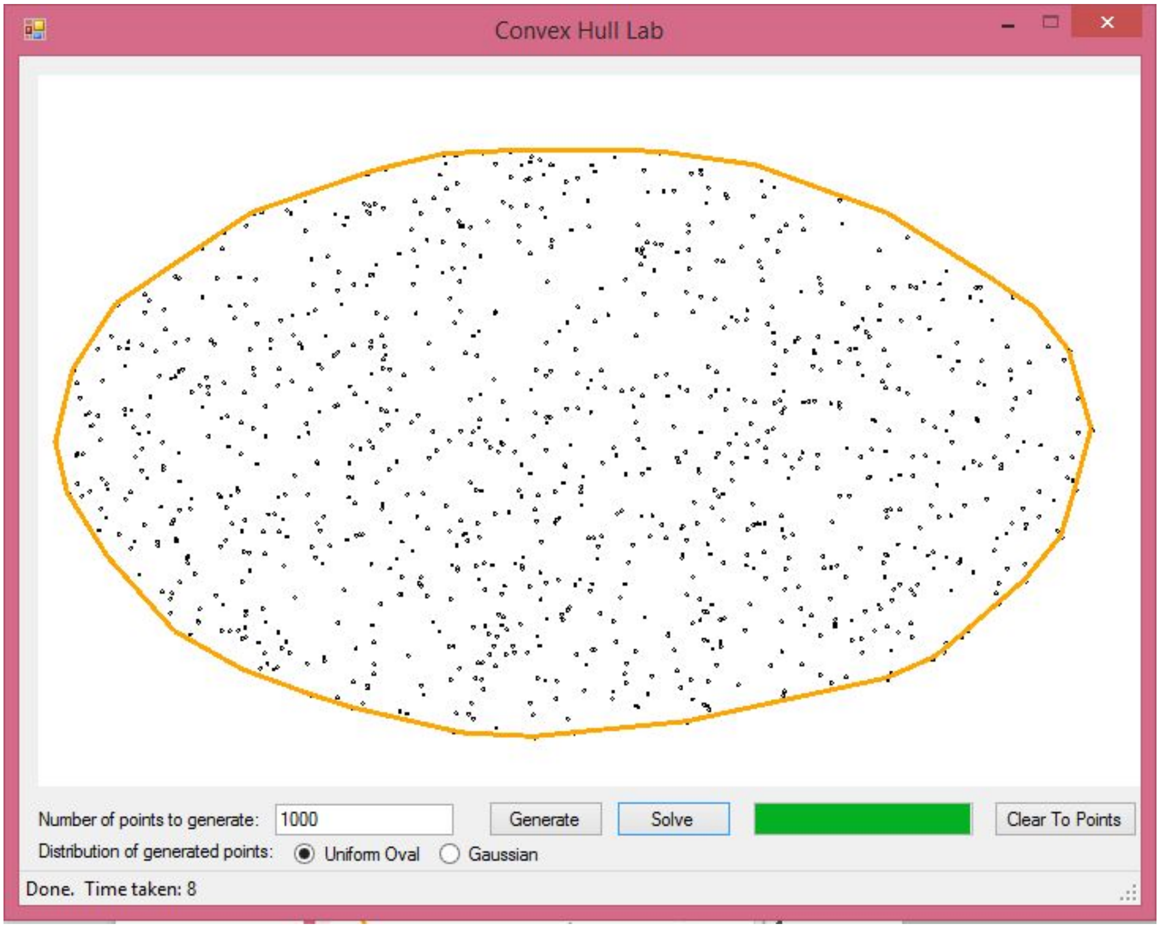
```
Function main (list of points){      //T(n)
-split list in two halves            // O(n)
-call recursion on first half        //T(n/2)
-call recursion on second half       //T(n/2)
}
```

Thus, the recurrence relation I used was T(n) = 2 T(n/2) + O(n), and as mentioned before, while researching on the internet about the recurrence relation used for quick hull, I found out that
T(n) = 2 T(n/2) + O(n) is correct
(http://stackoverflow.com/questions/13524344/complexity-of-the-quickhull-algorithm)
Finally, I could not figure out why my algorithm follows log(n) and not n*log(n).

SCREEN SHOT WITH 100 POINTS



SCREEN SHOT WITH 1000 POINTS



CODE

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using System.Windows.Forms;
using System.Data.Linq;
using System.Linq;

namespace _2_convex_hull
{
    22 references
    class ConvexHullSolver
    {
        private static List<OutterPoint> upperPerimeter; // north half of perimeter-points
        private static List<OutterPoint> lowerPerimeter; // south half of perimeter-points

        private static PointF lowestXPoint; // this points are saved because I calculate the angle of the added-points
        private static PointF highestXPoint; // to perimer lists when the point is inserted

        private static int UP = 1; // flag to tell function is that it is solving upper points
        private static int DOWN = 0; // flag to tell function that it is solving lower points

        1 reference
        public void Solve(PictureBox box, Graphics g, List<PointF> pointList)
        {
            ConvexHullSolver.upperPerimeter = new List<OutterPoint>();
            ConvexHullSolver.lowerPerimeter = new List<OutterPoint>();

            ExtremeXPoints points = new ExtremeXPoints();
            points.setXExtremePoints(pointList); // get points with lowest x value and greatest x value

            SplitedLists lists = new SplitedLists();
            lists.setSouthAndNorthLists(pointList, points.Left, points.Right); // splits list in north and south lists

            ConvexHullSolver.lowestXPoint = points.Left;
```

```
    ConvexHullSolver.lowestXPoint = points.Left;
    ConvexHullSolver.highestXPoint = points.Right;

    ConvexHullSolver.upperPerimeter.Add(new OutterPoint(ConvexHullSolver.highestXPoint, points.Right)); //saving lowest x point and highest x point to
                                                                                                          //upper perimeter list
    ConvexHullSolver.upperPerimeter.Add(new OutterPoint(ConvexHullSolver.highestXPoint, points.Left));
    ConvexHullSolver.lowerPerimeter.Add(new OutterPoint(ConvexHullSolver.lowestXPoint, points.Left));  //saving lowest x point and highest x point to
                                                                                                          //lower perimeter list
    ConvexHullSolver.lowerPerimeter.Add(new OutterPoint(ConvexHullSolver.lowestXPoint, points.Right));

    solving(box, g, lists.NorthList, points.Left, points.Right, UP); // call recursive function for first half
    solving(box, g, lists.SouthList, points.Right, points.Left, DOWN); // call recursive function for second half


    paintHull(box, g, points.Left, points.Right);
}

1 reference
private void paintHull(PictureBox box, Graphics g, PointF left, PointF right)
{
    upperPerimeter = upperPerimeter.OrderBy(x => x.Angle).ToList();
    lowerPerimeter = lowerPerimeter.OrderBy(x => x.Angle).ToList();

    for (int a = 0; a < upperPerimeter.Count - 1; a++)
    {
        g.DrawLine(new Pen(Color.Orange, 3), upperPerimeter[a].Point, upperPerimeter[a + 1].Point);
    }

    for (int a = 0; a < lowerPerimeter.Count - 1; a++)
    {
        g.DrawLine(new Pen(Color.Orange, 3), lowerPerimeter[a].Point, lowerPerimeter[a + 1].Point);
    }

    box.Refresh();
```

```
}

4 references
public void solving(PictureBox box, Graphics g, List<System.Drawing.PointF> pointList, PointF leftPoint, PointF rightPoint, int cuadrant)
{
    int indexOfInsertion = pointList.IndexOf(rightPoint);
    if (pointList.Count == 0)
    {
        return; // id list id zero size it means that there are no more points above it so it returns
    }
    else if (pointList.Count == 1)
    {
        if (cuadrant == UP)
            ConvexHullSolver.upperPerimeter.Add(new OutterPoint(ConvexHullSolver.highestXPoint, pointList[0]));
        if (cuadrant == DOWN)
            ConvexHullSolver.lowerPerimeter.Add(new OutterPoint(ConvexHullSolver.lowestXPoint, pointList[0]));
        return; // if list is size 1, it means that the point is an edge. Therefore it is saved into the perimeter list
    }
    else
    {
        PointF furthestPoint = findFurthestPoint(leftPoint, rightPoint, pointList); // by comparing cross vector distances we get relative distance
                                                                                    //that avoid divition

        if (cuadrant == UP)
            ConvexHullSolver.upperPerimeter.Add(new OutterPoint(ConvexHullSolver.highestXPoint, furthestPoint)); // the furthest point is added to the
                                                                                                                 //perimeter list
        if (cuadrant == DOWN)
            ConvexHullSolver.lowerPerimeter.Add(new OutterPoint(ConvexHullSolver.lowestXPoint, furthestPoint));

        SplitedLists leftLists = new SplitedLists();
        leftLists.setSouthAndNorthLists(pointList, leftPoint, furthestPoint); // from this split we only care about the upper points of the list,
                                                                              //the lower points are inside the hull

        SplitedLists rightLists = new SplitedLists(); // from this split we only care about the upper points of the list, the lower points are inside
                                                      //the hull
        rightLists.setSouthAndNorthLists(pointList, furthestPoint, rightPoint);

        solving(box, g, leftLists.NorthList, leftPoint, furthestPoint,cuadrant); // re call function to solve left smaller subproblem
        solving(box, g, rightLists.NorthList, furthestPoint, rightPoint, cuadrant); // re call function to solve right smaller subproblem
    }
}

1 reference
public PointF findFurthestPoint(PointF left, PointF right, List<PointF> pointList)
{
    double farthestDistance = 0.0;
    PointF farthestPoint = new PointF();

    foreach (PointF point in pointList) // for each point in the list the one with the greates distance is saved
    {
        double pointDistance = distance(left, right, point);
        if (pointDistance > farthestDistance)
        {
            farthestDistance = pointDistance;
            farthestPoint = point;
        }
    }
    return farthestPoint;
}

1 reference
public double distance(PointF left, PointF right, PointF point) // distance is calculated by cross product of the tangents created by most left and
                                                                //most right point compared to the point in cuestion
{
    double valueX = right.X - left.X;
    double valueY = right.Y - left.Y;
```

```csharp
        double num = valueX * (left.Y - point.Y) - valueY * (left.X - point.X);
        return Math.Abs(num);
    }

}

#region temporal classes

7 references
class SplitedLists
{
    5 references
    public List<PointF> NorthList { get; set; }
    3 references
    public List<PointF> SouthList { get; set; }

    3 references
    public SplitedLists()
    {
        NorthList = new List<PointF>();
        SouthList = new List<PointF>();
    }
    3 references
    public void setSouthAndNorthLists(List<PointF> pointList, PointF leftX, PointF rightX)
    {
        foreach (PointF point in pointList) // for each point location is calculated and it is sent to northlist or south list
        {
            if (getLocation(leftX, rightX, point) == 1)
            {
                NorthList.Add(point);
            }
            else
            {
                SouthList.Add(point);
```

```csharp
            }
        }
    }

    1 reference
    private int getLocation(PointF leftX, PointF rightX, PointF point) // location is calculated by seeing at the cross product sign
    {
        double location = (rightX.X - leftX.X) * (point.Y - leftX.Y) - (rightX.Y - leftX.Y) * (point.X - leftX.X);
        return (location > 0) ? 1 : -1;
    }

}

3 references
class ExtremeXPoints
{
    9 references
    public PointF Left { get; set; }
    9 references
    public PointF Right { get; set; }
    1 reference
    public ExtremeXPoints()
    {
        Left = new PointF();
        Right = new PointF();
    }

    1 reference
    public void setXExtremePoints(List<PointF> pointList)
    {
        PointF mostLeft = pointList[0];
        PointF mostRight = pointList[0];
        foreach (PointF point in pointList) // for each point the x coordinate is compared and only the extremes are saved
        {
```

```
            if (point.X < mostLeft.X)
            {
                mostLeft = point;
            }
            if (point.X > mostRight.X)
            {
                mostRight = point;
            }
        }
        Left = mostLeft;
        Right = mostRight;
    }
}

13 references
class OutterPoint
{
    3 references
    public double Angle { get; set; }
    5 references
    public PointF Point { get; set; }

    8 references
    public OutterPoint(PointF left, PointF point)
    {
        Angle = (point.Y - left.Y) / (point.X - left.X); // when ordering perimeter points clock wise, the angle is used to sort them. How ever no trig
                                                         //functions are used since we only care about the ratio
        Point = point;
    }
}


 #endregion
```