**Convex Hull Report**

**1. Code to solve Convex Hull**

*# Quicksort*

    def sortPoints(self, points):

        if (points == []) :

            return []

        else:

            pivot= points[0].x()

            varia= points[0]

            lesser = *self*.sortPoints([x for x in points[1:] if x.x() < pivot])

            greater = *self*.sortPoints([x for x in points[1:] if x.x() >= pivot])

            return lesser + [varia] + greater

*# Reorders the points in a clockwise order*

*# Method to reorder the list in a clockwise order. This is needed so we can't have the list ordered based on the x value*

*# This would not let us go through the polygon in the right order. [1, 2, 3] if 2 is lower than 1 than it should be 1 -> 3 -> 2 -> 1*

*# Parameters:*

*# points (list): A list of QpointF objects ordered based on their x value*

*# Returns:*

*# list: Return a list of QpointF objects ordered in clockwise order*

    def ord\_clockwise(self, points):

        line1 = QLineF(points[0], points[1])

        line2 = QLineF(points[0], points[2])

*# slope of a like  = rise / run*

        slope1 = (line1.dy() / line1.dx())

        slope2 = (line2.dy() / line2.dx())

        if(slope1 <= slope2): *# alternative swap points[1], points[2] = points[2], points[1]*

            temp = points[1]

            points[1] = points[2]

            points[2] = temp

        return points

*# Method to find the right most point of the left polygon O(n) this n is smaller tho.*

    def find\_right\_most(self, left\_pol):

        right\_most = left\_pol[0]

        x = 0

        for i in range(len(left\_pol)):

            if left\_pol[i].x() > right\_most.x():

                right\_most = left\_pol[i]

                x = i

        return (x, right\_most)

*# Method claculates the slope of the line that two points form*

    def find\_slope(self, l\_point, r\_point):

        line = QLineF(l\_point, r\_point)

        slope = (line.dy() / line.dx())

        return slope

*# Divide-and-conquer. The median of a set of numbers is the middle number in the set*

*# Method divide-and-conquer to split points into groups of three or less, connect them and form convex hulls.*

*# Parameters:*

*# points (list): A list of QpointF objects*

*# Returns:*

*# list: Return a list of QpointF objects*

    def div\_con(self, points):

        if len(points) <= 3:

            if(len(points) == 3):

                points = *self*.ord\_clockwise(points)

*# polygon = [QLineF(points[i],points[(i+1)%len(points)]) for i in range(len(points))]*

*# self.showHull(polygon, BLUE)*

            return points

        else:

            middle = len(points)//2 *# floor*

            left = points[0: middle]

            right = points[middle:]

*#self.printLR(left, right)*

            left\_poly = *self*.div\_con(left)

            right\_poly = *self*.div\_con(right)

        return *self*.merge(left\_poly, right\_poly)

*# Method merge connects two lists of points (polygons) and creates a new convex hull.*

*# We merge them by finding the upper tangent line and lower tanget line.*

*# Parameters:*

*# left\_pol (list): A list of points, QpointF, ordered in clockwise*

*# right\_pol (list): A list of points, QpointF, ordered in clockwise*

*# Returns:*

*# list: Return a list of QpointF objects ordered in clockwise order which represents the new convex hull*

    def merge(self, left\_pol, right\_pol):

*# right polygon left most point. This will always be the first point in the right hand list*

        r\_anchor\_point = right\_pol[0]

        r\_anchor\_i = 0

*# left polygon right most point. Since the list is ordered clockwise we have to iterate through the list to find the right most point of the left polygon*

        answer = *self*.find\_right\_most(left\_pol)

        l\_anchor\_i = answer[0]

        l\_anchor\_point = answer[1]

*#given two list of points ordered in clockwise order*

        slope\_change = True

        u\_tan\_slope = *self*.find\_slope(l\_anchor\_point, r\_anchor\_point) *#initial slope*

        l\_tan\_slope = u\_tan\_slope

        r\_low\_anchor\_point = r\_anchor\_point

        r\_low\_anchor\_i = r\_anchor\_i

        l\_low\_anchor\_point = l\_anchor\_point

        l\_low\_anchor\_i = l\_anchor\_i

        while(slope\_change):

            left\_change = False

            right\_change = False

*# Loop for RIGHT POLYGON*

            index = (r\_anchor\_i + 1) % len(right\_pol)

            while (index < len(right\_pol)):

                slope = *self*.find\_slope(l\_anchor\_point, right\_pol[index])

                if(slope > u\_tan\_slope):

                    r\_anchor\_point = right\_pol[index] *# new anchor point in the right pol*

                    r\_anchor\_i = index *# index of anchor point in right pol*

                    u\_tan\_slope = slope *# we assign the upper tangent line*

                    left\_change = True

                    index += 1

                else:

                    break

*# Loop for LEFT POLYGON*

            ind = (l\_anchor\_i - 1) % len(left\_pol)

            while(ind >= 0):

                slope = *self*.find\_slope(left\_pol[ind], r\_anchor\_point)

                if(slope < u\_tan\_slope):

                    l\_anchor\_point = left\_pol[ind]

                    l\_anchor\_i = ind

                    u\_tan\_slope = slope

                    right\_change = True

                    ind -= 1

                else:

                    break

            if not left\_change and not right\_change: slope\_change = False

        slope\_change = True

        while(slope\_change):

            left\_change = False

            right\_change = False

*# Loop for RIGHT POLYGON*

            i = (r\_low\_anchor\_i - 1) % len(right\_pol)

            while (i >= 0):

                slope = *self*.find\_slope(l\_low\_anchor\_point, right\_pol[i])

                if(slope < l\_tan\_slope):

                    r\_low\_anchor\_point = right\_pol[i] *# new anchor point in the right pol*

                    r\_low\_anchor\_i = i *# index of anchor point in right pol*

                    l\_tan\_slope = slope *# we assign the upper tangent line*

                    left\_change = True

                    i -= 1

                else:

                    break

*# Loop for LEFT POLYGON*

            index\_ = (l\_low\_anchor\_i + 1) % len(left\_pol)

            while(index\_ < len(left\_pol)):

                slope = *self*.find\_slope(left\_pol[index\_], r\_low\_anchor\_point)

                if(slope > l\_tan\_slope):

                    l\_low\_anchor\_point = left\_pol[index\_]

                    l\_low\_anchor\_i = index\_

                    l\_tan\_slope = slope

                    right\_change = True

                    index\_ += 1

                else:

                    break

            if not left\_change and not right\_change: slope\_change = False

*# line = QLineF(l\_low\_anchor\_point,r\_low\_anchor\_point)*

*# self.showTangent([line], PINK)*

        poly = []

        if l\_anchor\_i == l\_low\_anchor\_i:

            poly.append(left\_pol[l\_anchor\_i])

        else:

            here = 0

            notfound = True

            while(notfound):

                if here == l\_anchor\_i:

                    poly.append(left\_pol[here])

                    notfound = False

                else :

                    poly.append(left\_pol[here])

                here += 1

        if r\_anchor\_i == r\_low\_anchor\_i:

            poly.append(right\_pol[r\_anchor\_i])

        else:

            here = r\_anchor\_i

            notfound = True

            while(notfound):

                if here == r\_low\_anchor\_i:

                    poly.append(right\_pol[here])

                    notfound = False

                else:

                    poly.append(right\_pol[here])

                here = (here + 1) % len(right\_pol)

        if not l\_low\_anchor\_i == 0:

            notEnd = True

            here = l\_low\_anchor\_i

            while(notEnd):

                if(here == (len(left\_pol) - 1)):

                    poly.append(left\_pol[here])

                    notEnd = False

                else:

                    poly.append(left\_pol[here])

                here += 1

        return poly

*# This is the method that gets called by the GUI and actually executes*

*# the finding of the hull*

    def compute\_hull(self, points, pause, view):

        print()

        print("-------------------NEW CALL--------------------------")

*self*.pause = pause

*self*.view = view

        assert( type(points) == list and type(points[0]) == QPointF )

        t1 = time.time()

*# SORT THE POINTS BY INCREASING X-*

        points = *self*.sortPoints(points)

        t2 = time.time()

        print('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t2-t1))

        t3 = time.time()

*# DIVIDE-AND-CONQUER CONVEX HULL SOLVER*

        poly = *self*.div\_con(points)

        t4 = time.time()

        polygon = [QLineF(poly[i],poly[(i+1)%len(poly)]) for i in range(len(poly))]

*# when passing lines to the display, pass a list of QLineF objects.  Each QLineF*

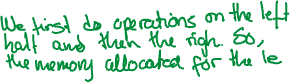
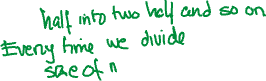
*# object can be created with two QPointF objects corresponding to the endpoints*

*self*.showHull(polygon, BLACK)

*self*.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4-t3))

**2. Theoretical complexity analysis**

Space:



Time:

Theoretically. We half the list until we get a number of three or less. This division of the list into halves gives us a O(log n) after we have these small sets of points we merge them and as it shows below it will take O(n). Thus, the time complexity for this algorithm is O(nlogn).

Q order list of points

def div\_con(points Q): #O(nlog n)

        if Q <= 3:

            order\_list\_in\_clockwise(points) #O(3)

            return Q

        else:

            left = Q[:Q//2]

            right = Q[Q//2:]

            left\_poly = div\_con(left)

            right\_poly = div\_con(right)

        return merge(left\_poly, right\_poly) #**O(n)**

def merge(points left\_pol, points right\_pol): **#O(n) = O(n) + O(n)**

        r\_anchor\_point = r\_low\_anchor\_ point = left most point in right polygon

        l\_anchor\_point = l\_low\_anchor\_point = right most point in left polygon

# Finding the slope of points already in the list is **O(1)**

        u\_tan\_slope = l\_tan\_slope = find\_slope(l\_anchor\_point, r\_anchor\_point)

# Loop to find upper points

        while(slope\_change): # **O(n)** **= O(n) + O(n)**

*# Loop for RIGHT POLYGON*

            while(slope increases): **# O(n) worse case we visit the small set of n**

# calculating the slope is a simple division thus O(1)

                slope = find\_slope(l\_anchor\_point, right\_pol[index])

                if(slope > u\_tan\_slope):

                    r\_anchor\_point = right\_pol[index]

u\_tan\_slop = slope

                index += 1

                else:

                    break

*# Loop for RIGHT POLYGON*

            while(slope decreases): **# O(n) worse case we visit the small set of n**

# calculating the slope is a simple division thus O(1)

                slope = find\_slope(left\_pol[index], r\_anchor\_point)

                if(slope < u\_tan\_slope):

                    l\_anchor\_point = left\_pol[index]

u\_tan\_slop = slope

                    index -= 1

                else:

                    break

# Loop to find lower points

        while(slope\_change): **#O(n) = O(n) + O(n)**

          # Loop for RIGHT POLYGON

            while(slope decreases): **# O(n) worse case we visit the small set of n**

# calculating the slope is a simple division thus O(1)

                slope = find\_slope(l\_low\_anchor\_point, right\_pol[i])

                if(slope < l\_tan\_slope):

                    r\_low\_anchor\_point = right\_pol[i]

l\_tan\_slop = slope

                    i -= 1

                else:

                    break

# Loop for LEFT POLYGON

            while(slope increases): **# O(n) worse case we visit the small set of n**

# calculating the slope is a simple division thus O(1)

                slope = find\_slope(left\_pol[i], r\_low\_anchor\_point)

                if(slope > l\_tan\_slope):

                    l\_low\_anchor\_point = left\_pol[i]

l\_tan\_slop = slope

                    i += 1

                else:

                    break

# List in clockwise order

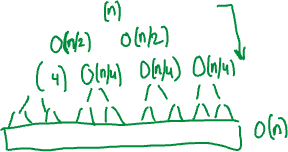
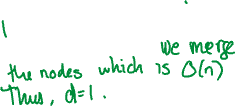
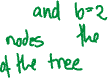
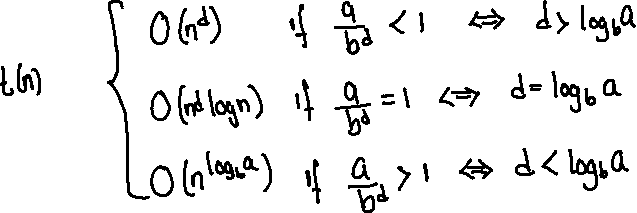
**# adding the different elements to a list that will be returned would require in worst case O(n) if we need to go through all points in the list**

poly = left\_pol[:l\_low\_anchor\_point] +

right\_pol[r\_anchor\_point:r\_low\_anchor\_point] +

left\_pol[l\_low\_anchor\_point:]

        return poly



**3 – 4 Theoretical vs Empirical analyses**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **10** | **100** | **1000** | **10,000** | **100,000** | **500,000** | **1,000,000** |
| **t** | 0.000 | 0.001 | 0.008 | 0.068 | 0.621 | 3.720 | 7.699 |
| **t** | 0.000 | 0.001 | 0.007 | 0.062 | 0.615 | 3.601 | 7.691 |
| **t** | 0.000 | 0.001 | 0.009 | 0.061 | 0.617 | 3.501 | 7.721 |
| **t** | 0.000 | 0.001 | 0.005 | 0.060 | 0.613 | 3.602 | 7.704 |
| **t** | 0.000 | 0.001 | 0.006 | 0.060 | 0.621 | 3.701 | 7.604 |
| **mean** | **0.000** | **0.001** | **0.008** | **0.064** | **0.619** | **3.702** | **7.697** |

**Chart, line chart

Description automatically generated**

**Schematic

Description automatically generated with low confidence**

The graph from the empirical analysis showed, at first, a more linear behavior as the number of points increased. However, we can see that after 100,000 points the graph starts to look more linear. This is expected since the theoretical complexity analysis and the master theorem demonstrates that the time complexity for this algorithm should be O (n \* logn). The second graph shows what a function 0.1\*x\*logn looks like. Which is pretty similar to what we obtain from the empirical analysis.

If it takes 0.008 (y) to find the convex hull of 1000(x) we can calculate a constant of proportionality as CH(Q) ≈ k·g(n) where CH(Q) is the time taken by the algorithm an g(n) is the O(n\*log(n)) and k is the constant of proportionality. Thus, for n = 1000 we obtain that CH(Q) = 0.008s and g(n) = g(1000) = n\*logn = 1000\*log(1000) = 3000 and k is:

k = CH(Q) / g(n) = 0.008 / 3000 = 0.00000266666

**5 Example of 100 and 1000 points**

Example with 100 points:

Graphical user interface

Description automatically generated

Example with 1000 points:

Chart

Description automatically generated