**Q.1**

 def divide\_and\_conquer(self, points):

        if not points:

            return []  # time complexity = O(1)  #space complexity = O(1)

        if len(points) <= 2:

            if len(points) == 3 and not self.is\_counter\_clockwise(points[0], points[1], points[2]):

                points[1], points[2] = points[2], points[1] # time complexity = O(1) #space complexity = O(1)

            return points # time complexity = O(1) #space complexity = O(1)

        mid = len(points) // 2 # time complexity = O(1) #space complexity = O(1)

        left = self.divide\_and\_conquer(points[:mid]) # time complexity = O(n/2) #space complexity = O(n/2)

        right = self.divide\_and\_conquer(points[mid:]) # time complexity = O(n/2) #space complexity = O(n/2)

        return self.merge\_hulls(left, right) # time complexity = O(n) #space complexity = O(n)

total **time complexity** of the divide and conquer = O(nlogn). This is because we are dividing the points into two halves until the base case and then recursively performing the convex hull for each half and merging the two hulls together.

**The recurrence relation and master theorem**

T(n) = aT (n/b) + O(nd)

a = the number of pieces the points were divided into or the breaking factor = 2

b = denominator of fraction = 2

d = the exponent = 1

Rule = O(nd) if 1 > a/bd

O(ndlog n) if 1 = a/bd

O(nlogba) if 1 < a/bd

T(n) = 2T(n/2) + O(n)

= 2/2

= 1

1 = 1

Therefore, Time complexity T(n) = O(nlogn)

**Space Complexity** : total space complexity = O(n) because we are storing the points in a list and recursively computing the convex hull for each half and merging the two hulls.

The space used in splitting the points into halves is O(n). also, despite the function calls itself recursively creating new subpoints, the number of level of recursion is (n). Even though there are lots of subpoints at each level of recursion, the space used at any given time is equal to the number of points which is (n).

Therefore, space complexity = O(n)

def is\_counter\_clockwise(self, point1, point2, point3):

        return (point2[1] - point1[1]) \* (point3[0] - point2[0]) > (point2[0] - point1[0]) \* (point3[1] - point2[1]) # time complexity = O(1) #space complexity = O(1)

This is\_counter\_clockwise function performs arithmetic operation. Therefore, time and space complexity is constant O(1), O(1), respectively. The function does not use any space.

def is\_clockwise(self, point1, point2, point3):

        return not self.is\_counter\_clockwise(point1, point2, point3)

This is\_clockwise function performs arithmetic operation. Therefore, time and space complexity is constant O(1), O(1), respectively. The function does not use any space.

def find\_upper\_tangent(self, Left, Right):

        p = max(Left, key=lambda x: x[0]) #Time Complexity O(n) on the left, space complexity is O(1)

        q = min(Right, key=lambda x: x[0]) #Time Complexity O(n) on the right, space complexity is O(1)

        temp = (p, q) #Time Complexity = O(1), space complexity = O(1)

        done = False #Time Complexity = O(1), space complexity = O(1)

        while not done:

            done = True #Time Complexity = O(1), space complexity = O(1)

            # Find the clockwise neighbor of p in L

            point\_indices = {point: i for i, point in enumerate(Left)} #Time Complexity O(n), space complexity is O(n)

            r = Left[(point\_indices[p] + 1) % len(Left)] #Time Complexity = O(1), space complexity = O(1)

            while not self.is\_lower\_tangent(temp, Left):

                temp = (r, q)  #Time Complexity = O(1), space complexity = O(1)

                p = r #Time Complexity = O(1), space complexity = O(1)

                r = Left[(point\_indices[p] + 1) % len(Left)] #Time Complexity = O(1), space complexity = O(1)

                done = False #Time Complexity = O(1), space complexity = O(1)

            # Find the counterclockwise neighbor of q in R

            point\_indices = {point: i for i, point in enumerate(Right)} #Time Complexity = O(n), space complexity = O(n)

            r = Right[(point\_indices[q] - 1) % len(Right)] #Time Complexity = O(1), space complexity = O(1)

            while not self.is\_lower\_tangent(temp, Right):

                temp = (p, r)  #Time Complexity = O(1), space complexity = O(1)

                q = r #Time Complexity = O(1), space complexity = O(1)

                r = Right[(point\_indices[q] - 1) % len(Right)] #Time Complexity = O(1), space complexity = O(1)

                done = False #Time Complexity = O(1), space complexity = O(1)

        return temp

**Time complexity**

finding the initial point

p = max(Left, key=lambda x: x[0]) scans the points on the left to find the max x-coordinates and it take (n) amount of time

q = min(Right, key=lambda x: x[0]) scans the points on the right to find the max x-coordinates and it takes (n) amount of time

The outer loop (while not done) can run up to (O(n+ n)) (from the left and right sides) times in the worst case

(on the left side)

The point\_indecies takes (n) amount of time

the inner loop in the point\_indecies can run up to (n) in the worse case

(on the right side)

The point\_indecies takes (n) amount of time

the inner loop in the point\_indecies can run up to (n) in the worse case

the total time complexity is O(n+n)2 = O(2n)2 = O(n)2: the left and right sides

**Space complexity:**

Storing p and q (initial points) require constant space O(1)

Creating point\_indecies on the left require O(n) space

Creating point\_indecies on the right requires O(n) space

The temporary variables (temp, r, and done) require O(1)

Therefore space complexity is O(2n) = O(n)

This function does not have a recurrence relation. The time complexity is quadratic and the space complexity is linear

def find\_lower\_tangent(self, Left, Right):

        intial\_point\_1 = max(Left, key=lambda x: x[0]) # time complexity = O(n) #space complexity = O(1)

        intial\_point\_2 = min(Right, key=lambda x: x[0]) # time complexity = O(n) #space complexity = O(1)

        temp = (intial\_point\_1, intial\_point\_2) # time complexity = O(1) #space complexity = O(1)

        done = False # time complexity = O(1) #space complexity = O(1)

        while not done:

            done = True # time complexity = O(1) #space complexity = O(1)

            # Find the counterclockwise neighbor of intial\_point\_1 in L

            point\_indices = {point: i for i, point in enumerate(Left)} # time complexity = O(n) #space complexity = O(n)

            r = Left[(point\_indices[intial\_point\_1] - 1) % len(Left)] # time complexity = O(1) #space complexity = O(1)

            while not self.is\_upper\_tangent(temp, Left): # time complexity = O(n) #space complexity = O(1)

                temp = (r, intial\_point\_2)  # time complexity = O(1) #space complexity = O(1)

                intial\_point\_1 = r # time complexity = O(1) #space complexity = O(1)

                r = Left[(point\_indices[intial\_point\_1] - 1) % len(Left)] # time complexity = O(1) #space complexity = O(1)

                done = False    # time complexity = O(1) #space complexity = O(1)

            # Find the clockwise neighbor of intial\_point\_2 in R

            point\_indices = {point: i for i, point in enumerate(Right)} # time complexity = O(n) #space complexity = O(n)

            r = Right[(point\_indices[intial\_point\_2] + 1) % len(Right)] # time complexity = O(1) #space complexity = O(1)

            while not self.is\_upper\_tangent(temp, Right):

                temp = (intial\_point\_1, r)   # time complexity = O(1) #space complexity = O(1)

                intial\_point\_2 = r # time complexity = O(1) #space complexity = O(1)

                r = Right[(point\_indices[intial\_point\_2] + 1) % len(Right)] # time complexity = O(1) #space complexity = O(1)

                done = False # time complexity = O(1) #space complexity = O(1)

        return temp

**Time complexity**

finding the initial point

p = max(Left, key=lambda x: x[0]) scans the points on the left to find the max x-coordinates and it take (n) amount of time

q = min(Right, key=lambda x: x[0]) scans the points on the right to find the max x-coordinates and it takes (n) amount of time

The outer loop (while not done) can run up to (O(n+ n)) (from the left and right sides) times in the worst case

(on the left side)

The point\_indecies takes (n) amount of time

the inner loop in the point\_indecies can run up to (n) in the worse case

(on the right side)

The point\_indecies takes (n) amount of time

the inner loop in the point\_indecies can run up to (n) in the worse case

the total time complexity is O(n+n)2 = O(2n)2 = O(n)2: the left and right sides

**Space complexity:**

Storing p and q (initial points) require constant space O(1)

Creating point\_indecies on the left require O(n) space

Creating point\_indecies on the right requires O(n) space

The temporary variables (temp, r, and done) require O(1)

Therefore space complexity is O(2n) = O(n)

This function does not have a recurrence relation. The time complexity is quadratic and the space complexity is linear

def is\_upper\_tangent(self, line, points):

        point1, point2 = line # time complexity = O(1) #space complexity = O(1)

        for p in points:

            if p != point1 and p != point2 and self.is\_counter\_clockwise(point1, point2, p): # time complexity = O(n) #space complexity = O(1)

                return False # time complexity = O(1) #space complexity = O(1)

        return True

        #total time complexity = O(n) #total space complexity = O(1)

for p in points loop iterates over all points in the points list. The loop runs (n) times making the time complexity O(n) amount on time.

The function takes a constant amount of space = O(1)

def is\_lower\_tangent(self, line, points):

        point1, point2 = line # time complexity = O(1) #space complexity = O(1)

        for p in points: # time complexity = O(n) #space complexity = O(1)

            if p != point1 and p != point2 and self.is\_clockwise(point1, point2, p): # time complexity = O(1) #space complexity = O(1)

                return False

        return True

for p in points loop iterates over all points in the points list. The loop runs (n) times making the time complexity O(n) amount on time.

The function takes a constant amount of space = O(1)

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

        upper\_tangent = self.find\_upper\_tangent(left, right) #time complexity = O(n^2) #space complexity = O(1)

        lower\_tangent = self.find\_lower\_tangent(left, right) #time complexity = O(n^2) #space complexity = O(1)

        merged\_hull = [] #time complexity = O(1) #space complexity = O(1)

        # Add points from left starting from upper\_tangent[0] to lower\_tangent[0]

        point\_indices = {point: i for i, point in enumerate(left)} #time complexity = O(n) #space complexity = O(n)

        slope = point\_indices[upper\_tangent[0]] #time complexity = O(1) #space complexity = O(1)

        while True:

            merged\_hull.append(left[slope]) #time complexity = O(1) #space complexity = O(1)

            if left[slope] == lower\_tangent[0]:

                break

            slope = (slope + 1) % len(left) #time complexity = O(1) #space complexity = O(1)

        # Add points from right starting from lower\_tangent[1] to upper\_tangent[1]

        point\_indices = {point: i for i, point in enumerate(right)} #time complexity = O(n) #space complexity = O(n)

        slope = point\_indices[lower\_tangent[1]] #time complexity = O(1) #space complexity = O(1)

        while True:

            merged\_hull.append(right[slope]) #time complexity = O(1) #space complexity = O(1)

            if right[slope] == upper\_tangent[1]:

                break

            slope = (slope + 1) % len(right) #time complexity = O(1) #space complexity = O(1)

2

        return merged\_hull #time complexity = O(1) #space complexity = O(1)

**Time complexity**:  
finding upper and lower tangents:

upper\_tangent = self.find\_upper\_tangent(left, right), lower\_tangent = self.find\_lower\_tangent(left, right) both have a time complexity of O(n2) respectively

the point\_indicies of the left is O(n)

the point\_indices of the right is O(n)

the points to the merged\_hulls: the while loop iterates from the upper to the lower tangent in the left and takes O(n) times

the second while loop iterates from the lower to the upper tangent in the right and takes O(n) times

the time complexity is: O((n2) + (n2) + (n) + (n)) = O(n2)

**Space complexity:**

The upper and lower tangent uses O(1) amount of space

The point\_indicies take (n) amount of space on both left and right of the hull

The merged\_hull also takes (n) amount of space on both left and right of the hull

Therefore the space complexity = O(n + n + n + n + 1 + 1) = O(n)

**Recurence relation and the master theorem**

T(n) = aT (n/b) + O(nd)

a == 2

b = = 2

d = O(n2)

Rule = O(nd) if 1 > a/bd

O(ndlog n) if 1 = a/bd

O(nlogba) if 1 < a/bd

T(n) = 2T(n/2) + O(n2)

= 2/22

= 2/4

= 1/2

1 > ½

Therefore time complexity T(n) = O(n2)

**(2)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X | Y1 | Y2 | Y3 | Y4 | Y5 |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0014 | 0.0015 | 0.001 | 0.0011 | 0.004 |
| 1,000 | 0.0108 | 0.0104 | 0.0103 | 0.0095 | 0.009 |
| 10,000 | 0.1308 | 0.1147 | 0.1151 | 0.1162 | 0.123 |
| 100,000 | 1.2963 | 1.2976 | 1.3373 | 1.2553 | 1.2672 |
| 500,000 | 6.4397 | 6.3927 | 6.8345 | 6.3837 | 6.3577 |
| 1,000,000 | 13.6628 | 13.4054 | 12.9558 | 13.5377 | 13.0063 |

|  |  |
| --- | --- |
| X | y |
| 10 | 0.0 |
| 100 | 0.0018 |
| 1,000 | 0.010 |
| 10,000 | 0.12016 |
| 100,000 | 1.29074 |
| 500,000 | 6.48166 |
| 1,000,000 | 13.3136 |

A screenshot of a calculator

Description automatically generated

A graph on a graph paper

Description automatically generated

**Discussion of pattern**

Base on the mean value, the growth pattern appears as O(nlogn) complexity. This is due to the fact that it uses merge sort to divide the points into halves and merge them back. As the mean (y) is ploted against the x(data), there is an increase in the mean(y) which is neither linear nor quadratic but rather a logarithmic. This means that as x increases the value of y grows faster.

Assumption

As it is seen on the graph the data points are assumed to follow an (O(n \log n)) growth pattern.

**(3)**

Empirical

A screenshot of a calculator

Description automatically generated

In plotting the plotting the mean (y) against x(data) value, the growth pattern proves that the increasing rate of change in the mean as the data increases align with logarithmic.

**(4):**100 points

A graph of a uniform point

Description automatically generated with medium confidence

1,000 points

A black and white diagram with numbers and dots

Description automatically generated with medium confidence