Homework 2 writeup

Problem 4

For each of the following recurrences, use the Master theorem to express $T(n)$ as a Theta of a simple function. State what the corresponding values of a, b, and $f(n)$ are and how you determined which case of the theorem applies. Do not worry about the base case or rounding.

1. $T(n) = 9T(n/3) + n^3$
2. $T(n) = \frac{3}{2}T(2n/3) + 3$
3. $T(n) = 2T(n/4) + \sqrt{n}$

Case 1: $T(n) = \Theta(n^{\log_b a})$
Case 2: $T(n) = \Theta(n^{\log_b a} \log n)$
Case 3: $T(n) = \Theta(f(n))$

1. $T(n) = 9T\left(\frac{n}{3}\right) + n^3$

$a = 9$, $f(n) = n^3$
$b = 3$
$\log_b a = \log_3 9$

$= \log_3 3^2$

$= 2 \log_3 3$

$= 2 * 1$
$= 2$

Compare : $n^{\log_b a}$ vs. $f(n)$
n^2 vs. n^3

$\because$ $f(n)$ is dominating it belongs to <u>case 3.</u>
$T(n) \in O(f(n))$

2. $T(n) = \frac{3}{2} T\left(\frac{2n}{3}\right) + 3$

$f(n) = n^0 = 1$

$a = \frac{3}{2}$

$b = \frac{3}{2}$

$\log_b a = \log_{3/2} {}^{3}/_{2}$

$= 1$

Compare : $n^{\log_b a}$ vs. $f(n)$

n^1 vs. n^0

n vs. 1

$n^{\log_b a}$ is dominating and belongs to Case 1.

$\therefore T(n) \in O(n^{\log_b a})$

3. $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$

$f(n) = \sqrt{n} = n^{1/2}$

$a = 2$

$b = 4$

$\log_b a = \log_4 2$

$= \log_{2^2} 2$

$= \frac{1}{2} \log_2 2$

$= \frac{1}{2} * 1$

$= \frac{1}{2}$

Compare : $n^{\log_b a}$ vs. $f(n)$

$n^{1/2}$ vs. $n^{1/2}$

$\sqrt{n}$ vs. $\sqrt{n}$

Here, both are equal therefore it belongs to case 2.

$\therefore T(n) \in O(n^{\log_b a} \log n)$

$\in O(\sqrt{n} \log n)$

Problem 5 (CSCI-665 only)

Recall the Convex Hull problem: Given are n points with coordinates $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. The task is to compute the smallest convex polygon which includes all points. The coordinates are arbitrary real numbers and the polygon should be output as a sequence of its vertices, in clockwise order (starting at any of the vertices). Prove that every Convex Hull algorithm has $\Omega(n \log n)$ running time.

1. GIFT WRAPPING ALGORITHM

GIFT WRAPPING ALGORITHM a.k.a. JARVIS METHOD

We will start by picking the left most bottom point and move in the clock-wise direction to wrap around the convex-hull. This will be done by continuously identifying the maximum polar angle since we are going in the clock-wise direction instead of the anti clock-wise direction. The polar angle will be relative to the previous segment of the point. For n*logn complexity we will consider an example where majority of the points will lie inside the convex hull that we will construct. In that case, all the points will not be the vertices of the convex hull which will give us the worst time complexity of O(n^2). For gift wrapping algorithm the best case and average case complexity will be O(n*logn).

Algorithm:

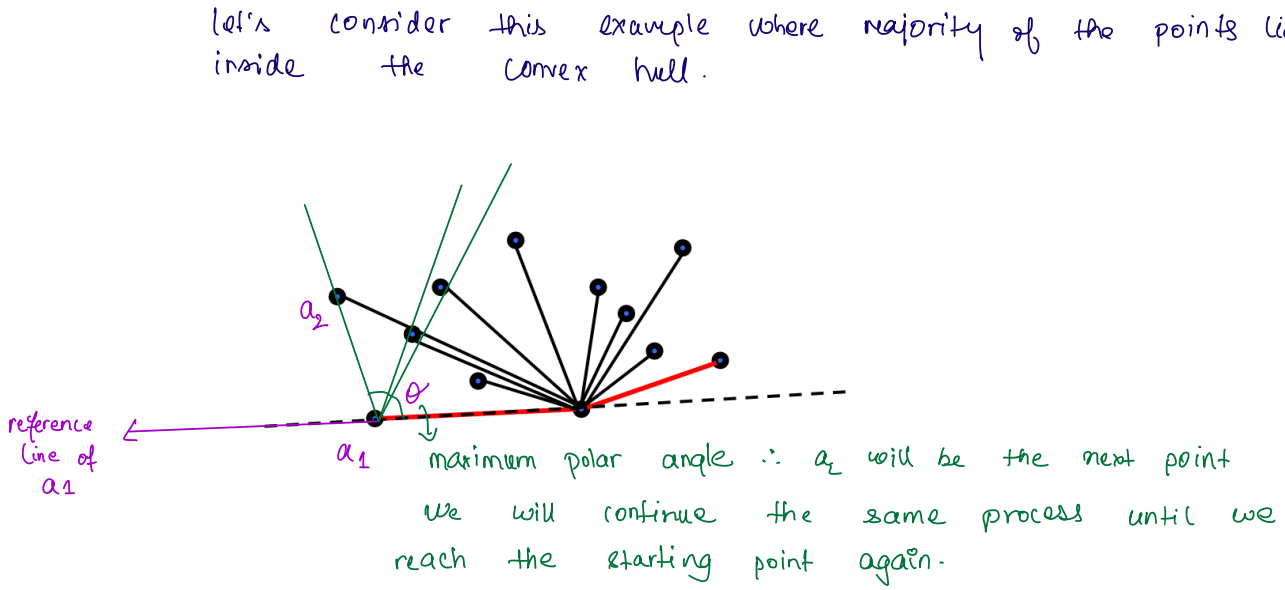
- For finding the left bottom most vertex the complexity will be : O(n), since the number of points as input will be n for the convex hull.

- For computing the polar angle with respect to other points the complexity will be : O(1) to compute any of the angles, so for total n inputs the complexity will be : O(n)

- Repeat the above two steps for the total number of vertices the convex hull will end up with.

- For a convex hull with h vertices where the size of h is asymptotically smaller compared to the total input size of n the complexity will be : O(nh)

And for best case and average case which we are considering here the time complexity will be O(n*logn) for considerably small h.



DIVIDE and CONQUER METHOD

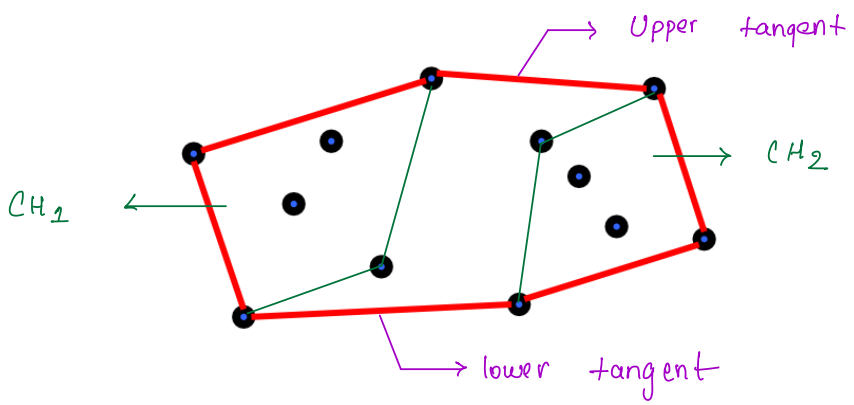
- *Algorithm:*
- For this method, we will start by sorting the points by it's x values(which means we will separate the points into two parts for finding the convex hull).
- After that we will use the recursive algorithm to solve it.
- Incase the input size or the number of points is too small, say it's less than some constant value 5 then we can use brute force to solve it because it will not affect the time complexity.
- Otherwise,
    - Compute the convex hull of the left half (CH1) of the points.
    - Compute the convex hull of the right half (CH2) of the points.
    - Merge the hulls into a single convex hull.
- The cost at each step to merge the convex hull is O(n), then the algorithm will run in O(n*logn) time complexity.
- For merge process we need to find the upper tangent and the lower tangent lines that will connect the two convex hulls and give us the full convex hull.

    - The upper tangent line is the one where all the points lie below it.
        - To get the upper tangent we will select the right most point of CH1 and the left most point of CH2.
        - The upper tangent will be the one when it makes a left turn with the next counter clockwise of CH1 and a right turn with the next counter clockwise vertex of CH2.
        - In case that is not the desired line segment we will check the following conditions:
            - if it does not make a left turn with the next counter clockwise vertex of CH1 then: rotate to the next counter clockwise vertex in CH1

- Else: if the line does not make a right turn with the next clockwise vertex of CH2
  then: rotate to the next clockwise vertex of CH2.
- continue to do so until you find the upper tangent.

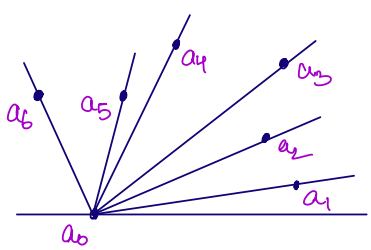
- The lower tangent line is the one where all the points will lie above it.
  - To get the lower tangent we will select the right most point of CH1 and the left most point of CH2.
  - The lower tangent will be the one when it makes a right turn with the next clockwise of CH1 and a left turn with the next counter-clockwise vertex of CH2.
  - In case that is not the desired line segment we will check the following conditions:
    - if it does not make a right turn with the next clockwise vertex of CH1
      then: rotate to the next clockwise vertex in CH1
    - Else: if the line does not make a left turn with the next counter-clockwise vertex of CH2
      then: rotate to the next counter-clockwise vertex of CH2.
    - continue to do so until you find the lower tangent.

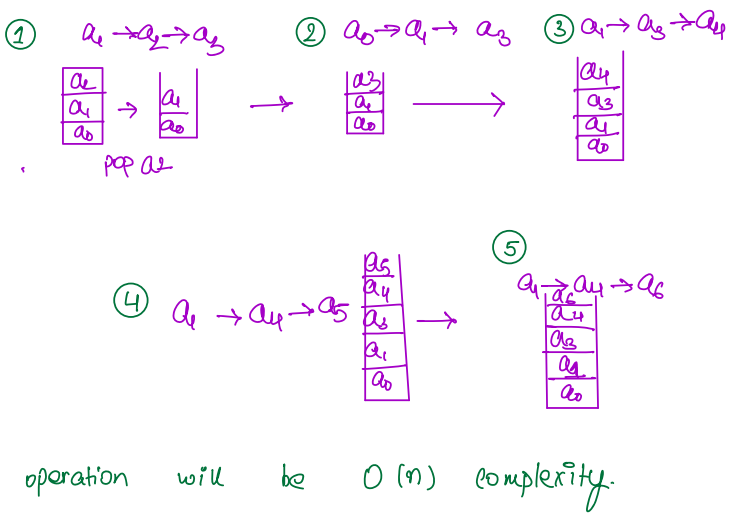
- *Time complexity* for divide and conquer will be : O(n*logn)
  - Sort the values by the x points into two parts : O(n*logn)
  - Recursive algorithm: let that be R(n)
  - If: n (number of input points) <= 5 (for instance), then we can solve it by brute force : O(1)
  - Else:
    - Compute the convex hull of the left half (CH1) of the points : R(n/2)
    - Compute the convex hull of the right half (CH2) of the points : R(n/2)
    - Merge the hulls into a single convex hull : O(n)
- Therefore, R(n) = 2R(n/2) + O(n)
- This will be of complexity O(n*logn) just like Merge Sort.
- Hence, T(n) = O(n*logn) {sorting through all the x's} + O(n*logn) {recursive + rest of the process}
  = O(n*logn)



GRAHAM-SCAN

- First, identify the point with the smallest y-coordinate to start : O(n)
- Next, we will sort all the remaining points by polar angle with respect to the starting point : O(n*logn)
- We will make use of stack here. Push the first three points into the stack including the starting point : O(1)
- For each next point, take into consideration the angle formed by that point plus the two top elements of the stack.
    - For as long as this angle is a non-left turn, pop the element from the top of the stack and consider the next angle.
    - When the angle corresponds to a left turn, add this point to the stack.
- Since points are added to the stack once, and removed at most once, the entire search is O(n) once the points are sorted.
- Therefore, the overall complexity of this algorithm will be : O(n*logn)





This whole operation will be O(n) complexity.

Proof for Divide n Conquer and Graham Scan.

for both these algorithms the complexity will be $(n \log n)$.

$T(n) = 2T\left(\frac{n}{2}\right) + n$ will be same for both of them since, in divide n conquer we are dividing, finding the convex hull and then combine all the points to get the final convex hull. And, in Graham scan, first the sorting is done and then we use stack to find the convex hull.

Proof:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) \geq 2C\left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right) + n$$

$$\geq 2C\left(\frac{n-1}{2}\right)\log_2\left(\frac{n-1}{2}\right) + n$$

$$\geq Cn\log_2(n-1) - C\log_2(n-1) - C(n-1) + n$$

$$\geq Cn\log_2\left[n\left(1-\frac{1}{n}\right)\right] - C\log_2(n-1) - C(n-1) + n$$

$$\geq Cn\log_2 n + Cn\log_2\left(1-\frac{1}{n}\right) - C\log_2(n-1) - C(n-1) + n \quad \text{—eq}^{n}(1)$$

let's assume that $C > 0$, where C will satisfy n; n is a large value.

$\therefore \quad T(n) \geq n\log n$

PROBLEM 1 : *MaxRectangle.java*

Verbal Description

• In this program, we are supposed to find a rectangle with the maximum area from the given histogram which is constructed using the x-coordinates and y-coordinates as input. First, the input from the command line is taken: size of the input, the x,y coordinates.

• The x-values and y-values are stored in separate arrays so that the width and height of the bars can be calculated. Every alternate value in the y-array will give the height of the bar and the width can be calculated by taking the difference of the x-coordinates. Once, all the inputs have been stored in the array we call the *maxRectangleArea()* method and pass the x-array, y-array and the size of the input as parameters.

• To solve this problem a stack has been implemented and the name of the file is Stack.java. It works like a normal, basic stack structure and has the following functionalities: push(), pop(), peek(), isEmpty(), isFull().

• To iterate over all the heights we have created a for-loop, it gives the height of the histogram bars.

• In the first iteration we push the *height_i* onto the stack, since it is empty. Then the i is incremented by 2 since we want alternate values for height. We push the index value of the height in the stack and not the actual value.

• In the next iteration, when we read the second *height_i* we check to see if the stack is empty or, if the top most value is smaller than the value we want to push. If it is true then we push the value, otherwise we jump to the else part of the for-loop since the top most value of the stack is greater than the value. We don't increment the i.

• In the else part, the top most value of the stack is popped and once again we check to see if the stack is empty, if it is empty, then we calculate the width of the height we just popped and compute the area by multiplying the height and width. The computed value of area is stored in the variable *maxArea* by taking the maximum of (maxArea, height*width). But, if the stack is not empty then we calculate the *maxArea* and update the value of *maxArea* if needed.

• Since the value of i wasn't incremented we again read the same height and check the following if-else conditions of the for-loop to check which one holds true. For instance, say, again the top most value of the stack is greater than the *height_i* then, the stack is popped again. The stack is checked using *isEmpty()* method to check if it is empty, if it is then we consider the height of the *height_i (index)* we just popped and take the total width till that height. Here, the left histogram bars are considered and when computing the width we consider all the bar-widths from the bar of *height_i* till the height of the bar just popped from the stack.

• We continue to do so and keep updating the value of *maxArea* until we've iterated over all the heights stored in the array.

• After the for-loop ends we move to the while loop. The purpose of this while-loop is to check the stack and determine if the stack is empty or still full. There could be cases where the for-loop has ended but the stack still has values left in it for which we haven't computed the area yet.

• If the stack has some value still left then we will pop the top most height and again check if the stack is finally empty, if it is, then, we will calculate the *maxArea* by taking the height of the popped index value and getting the width from that bar-height till the left most bar. The computed area is then compared with the *maxArea* and updated if it is greater else it will remain the same.

• If the stack is still full we will calculate the area of the popped height and update the *maxArea* if the value is larger using *Math.max*.

• Once the stack is finally empty we will return the *maxArea* and print the output on the command line.

PseudoCode

```
MaxRectangle{

   Array of x-coordinates;
   Array of y-coordinates;

 main(String[] args){
      Initialize array x with the size of input
      Initialize array y with the size of input

      for  i = 0 till the input_size-1 and i will increment by 1 {
               Store the x-coordinate in x-array
               Store the y-coordinate in y-array
      }

         method to compute the maximum area of a rectangle in a histogram.
      maxRectangleArea(x-array, y-array, size of the input)
      print the maximum area computed

   }


  maxRectangleArea (x-array, y-array, size of input) {

      initialize the stack array with the input size which will store the heights when pushed into it.

      Let the maxArea be 0 which will be updated as we calculate

          iterate through all the heights in the y-array
      for i = 1 till length of y-array-1 {

         if stack is Empty or top-most value of stack <= height_i {
            Push the index i in the stack
            i += 2 (since every alternate value is the height)
```

```
        }

        else{
            get the top most value of the stack

            if stack is empty {
                Calculate width of popped height using x[i-1]-x[1]
                Get the maximum area using Math.max of maxArea and the area we will compute using .
                 the width and the popped height
            }

            else {
                Calculate width of popped height using x[i] -x[stack.peek()+1]  (compute the width of the
                 left bars, from the popped height till the height of the top-most value of the stack)
                Get the maximum area using Math.max of maxArea and the area we will compute using
                 the width and the popped height
            }
        }
    }
```

continue in the while loop until the stack is finally empty.

```
    while stack is not empty {
        get the top most value of the stack

        if stack is empty {
            Get the maximum area using Math.max of maxArea and popped height * width of the
             popped height
        }

        else {
              Calculate width of popped height using (x[top-1] -x[stack.peek()] - 1)
              Get the maximum area using Math.max of maxArea and popped height * width
        }
    }
    return maximum area (maxArea) of the largest rectangle in the histogram
  }
}
```

Proof of Correctness

- This algorithm will always produce the desired output since it handles all the possible cases it can encounter.
- The basic idea behind this algorithm is:
    - When it reads the first height it pushes it onto the stack.
    - In the next iteration when the second height is read it will first check if the stack is empty *or* if the top most value of the stack is less than the value we just read from the array. If these conditions hold true then we will push the value onto the stack.
    - Else, we will pop the value from the stack and check if the stack is empty, if it is then we will compute the area of that bar and read the height again from the array and push

it in, else, we will compute the area of that bar and then check the top most value again
    - In the end, when we have iterated over all the heights we will move on to the while-loop to deal with the remaining heights in the stack and compute the area. In the end, it will return the maxArea.
- This algorithm stores the x-values in one array and all the y-values in a separate array.
- The y-array has all the y-coordinates which is the height of the bars of histogram. In this array, every alternate index will give us the height of the bar for any size of input.
- For width of that specific bar we subtract the (i - (i+2)) to get the width of that bar.
- We push the index values of height in the stack so that it becomes easier to compute the area and the width.

A tight running time estimate for the algorithm

- Tight running time for the aforementioned algorithm will be *O(n)*.
- For best case it can be *O(1)* if the input size is considerably small, say n<=5.

A brief reasoning behind the Running time estimate

- The time complexity of this problem will be *O(n)* where n is the size of the input given. n=x,y coordinates.
- For reading the input from command line and storing it a for-loop is used in the main method which will have a time complexity of the size of the input n : O(n)
- In the maxRectangleArea there is one for-loop and one while-loop (outside the for-loop) which ascertains that the algorithm will always run in O(n) time complexity.
- The for loop will iterate over n inputs therefore the complexity will be O(n). In while-loop for worst case it will have to pop all the n values which are the heights: for that the complexity will be O(n) otherwise it will be less than O(n), for extremely small n lets say n<=5 it will be O(1).
- All the if-else conditions will have a time complexity of O(1).
- Over all complexity will be: O(n)*(for-loop)* + O(n)*(for-loop)* + O(n)*(while-loop)* -> *all the dominating complexities.* *: O(n)*
- This algorithm will have a time complexity of *O(n).*

PROBLEM 2

a) State the recurrence for $T(n)$ that captures the running time of the algorithm as closely as possible.

a) $$T(n) = \begin{cases} O(1) & ; n=1 \\ 2T(n/2) + O(1) & ; n>1 \end{cases}$$

b) Use the "unrolling the recurrence" or the mathematical induction to find a tight bound on $T(n)$.

b) Using unrolling the recurrence,

$$T(n) = 2T(n/2) + 1 \quad \text{—①}$$
$$T(n/2) = 2T(n/4) + 1 \quad \text{—②}$$
$$T(n/4) = 2T(n/8) + 1 \quad \text{—③}$$

② in ① →

$$T(n) = 2(2T(n/4)+1) + 1$$
$$= 4T(n/4) + 2 + 1 \quad \text{—④}$$

using ③,

$$T(n) = 4(2T(n/8)+1) + 2 + 1$$
$$= 8T(n/8) + 8 + 2 + 1$$
$$= 2^3 T(n/2^3) + 2^3 + 2 + 1$$

thus, generalizing →

$$T(n) = 2^k T(1) + 2^{k-1} + 2^{k-2} \cdots 2 + 1$$
$$\therefore \quad T(n) = 1 + 2 + 4 \cdots 2^{k-2} + \cdots + 2^k$$

a GP is formed here.

total terms = $\log_2 n$ //

sum of GP = $a\left[\frac{r^n - 1}{r-1}\right]$

$$= 1\left[\frac{2^{\log_2 n} - 1}{2-1}\right] = n - 1 \Rightarrow n \text{ //}$$

Thus,

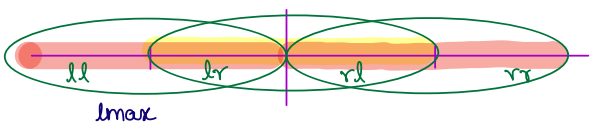
$$T(n) = O(n) \text{ //}$$

c) What does the algorithm do?

- For an input in A and integers `left` and `right`, succinctly describe the meaning of the return variables `maxsum`, `leftalignedmaxsum`, `rightalignedmaxsum`, and `sum`.
- Succinctly decribe the meaning of the value (the first of the four returned values) that we output after running `WHATDOIDO(1,n)`.

c)

- maxsum – holds the maximum value on comparing entire sum of left half with entire sum of right half with right half of left half and left part of right half.



- leftalignedmaxsum – holds the maximum of left part of the maximum left sum and sum of entire left sum with right part of maximum left sum.

- rightalignedmaxsum – holds the maximum of right part of maximum right sum and sum of entire right sum with left part of maximum right sum

- sum – holds the sum of entire left half and entire right half.

- meaning of 1st of the four values that are returned →

The first of the four returned values is maxsum
As explained above, it is the maximum value of the maximum of the left half sum, maximum of the right half sum and sum of left half of the maximum of the right side sum with the right half of the maximum of the left side sum.

PROBLEM 3 : *WeightedInversions.Java*

a) The goal of the problem is to find the sum of the weighted inversions. To do so, a hard comparison i.e., the one where each element is compared with the other element in order to calculate the weight i.e., the difference between the index positions of the arrays is necessary. Without touching each element at least once, it is impossible to calculate the difference between the two elements having inversion. To do so, we have implemented a nested for loop, each of which goes upto the length of the given array. For each iteration of i, all values of j are compared to check for inversion. If there is an inversion, the weight is calculated and summed. At the end of both the nested loops, this calculated sum is returned.

b) For i=0 to length of arr
    For j=0 to length of arr
        Check if element at j < element at i
        Implies a inversion so calculate difference between j and i
        Add the difference to a 'sum'
    Return the sum

c) The goal here to calculate the weight and not just the number of inversions in the array. So, it is absolutely necessary to iterate through each item, compare it with the other element and then actually subtract their indexes in order to get the weight. So, a nested for loop is unavoidable in this case. The first loop compares the first element of the array with the rest, second with the rest in the second iteration and so on. In case there is any inversion, it calculates the difference between the index positions and adds it to the sum. This goes on for the rest of the elements. Thus, the sum now represents the weights of the elements in inversion.

d) Tight running time estimate – O(n^2)

e) The first loop executes for n times because i increments till the length of the array
The next for loop that is nested inside the previous for loop also executes n times because j increments till the length of the array. Hence the time complexity is O(n^2).

Problem 3 (approach 2):

a) The goal of the problem is to find sum of the weighted inversions. In order to do this, we have used the merge sort. The array is initially divided into two halves. Next, the weight of the inversions is recursively counted and added in each of the halves. Then, these two halves are combined using the same logic of merge sort and the weight is computed parallelly with the sorting. In the end, the final weight is the sum of the weight of the left half, weight of the right half, weight computed while combining the halves.

b) mergeSortAndCount(arr, left, right)
    check if left < right
        calculate mid using (l+r)/2
        recursively count and add inversions of left half
        recursively count and add inversions of right half
        recursively count and add inversions while merging the two halves
    return the count

merging the two lists -

```
mergeAndCount(arr, left, mid, right)
    create and initialize the left subarray
    create and initialize the right subarray

    loop till i<length of left and j<length of right
        check if element at i <= element at j
            add this element to sorted array and increment i
        else if element at i > element at j
            this implies a inversion so calculate the weight
            add this weight to the total sum variable
            iterate through rest of the elements of the sorted array as all of them will be in
            inversion with this particular j
            add the weight of these elements to the sum
            add this element (jth) to the sorted array and increment j

    append rest of the elements to the sorted array

logic to find the index for that element from the og array-
    loop till i < length of original array
        check if element at i == target element (i.e ith element)
            if yes, return i
            if no, increment i
```

c) The goal here is to find the sum of the weighted inversions. To do so, and in order to obtain the given complexity, we perform merge sort. Recursively counting the elements using a divide and conquer approach from each half is implemented. The array is first divided into two halves at the mid point. The left half is then divided till single element remains. Now, during the merging process, if an inversion is found between i and j, the algorithm computes its weight. This is done recursively for the two halves. Once this is done, we get two sorted halves that must be merged into one single array. While doing so, the arrays are passed to a function called mergeAndCount where the arrays are inspected element by element and the comparison is done. If an inversion is found, the weight is calculated and added to the final sum. This way, a collective sum of weights is calculated and returned.
The algorithm implements a for loop inside the while loop in order to access the elements after that particular inversion. Meaning-
If the arrays are [2,3,5] and [1,4], the normal algorithm just checks if 2>1, concludes it as an inversion and increments the j. But, here since the arrays are sorted, if 2 is greater than 1, then it means that the rest of the elements of the left array will also be in inversion with that particular j. So, a for loop is necessary here that goes up to the length of the left array and begins from i. Thus, the collective weights of all these inversions are calculated and added to the sum.
Next, while calculating the weight of the elements, we need to ensure that we are considering the index of that particular element from the original array. In order to achieve this, another method called findIndex was created that takes in the target element and finds its index using a simple while loop that iterates through the length of the original array.

d) $O(n^3 \log n)$

e) In the method mergeSortAndCount
Array gets divided recursively, hence complexity is $O(\log n)$
Next, in the mergeAndCount, while loop is implemented to merge arrays and add the weight, so time complexity is $O(n)$
Inside the while loop, a for loop is implemented in order to access the rest of the elements of the left

array as explained above, so time complexity is O(n).
Next, in order to find the index from the original array, another while loop is implemented which goes on till the length of the original array. So, time complexity is O(n)
Hence, total time complexity now becomes O(n3 log n)

Please Note :
We have tried to implement the complexity as told in the problem statement. However, our best attempt gives complexity O(n3 log n). With this algorithm, only 4 test cases pass.
In order to reduce the complexity, we have also written another algorithm with time complexity O(n2). This algorithm passes 6 test cases and fails just 1.
As O(n2) is better than O(n3 log n) we have uploaded the algorithm with complexity O(n2). We have only attached our approach and code for the merge sort approach.

The file with the merge sort code is named WeightedInversions2.java and is stored under the path /home/stu15/s14/ay4034/Courses

The file with the brute force aproach code is named WeightedInversions.java and is stored under the path
/home/stu15/s14/ay4034/Courses/Assignment2/