CIS*3490 Assignment 2

1127811

February 14th, 2023

Nathan Starkman

1.1

Pseudocode:

```
ALGORITHM getInvCount
//Finds the amount of inversions for the array of integers [n-1] length
//Input: an array of integers of length n-1,
//Output: The amount of inversions that needed to occur in the array to sort
invCount <- 0
i <- 0
while i <= n-2
    j <- 0
    while j <= n-1
        if arr[i] > arr[j]
            invCount <- invCount + 1
        j<-j+1
    i<-i+1
return invCount
```

Operations:

The number of operations (arr[i]>arr[j]) for the first program is 1249975000, which, is equivalent to saying $\sim\frac{n(1+n)}{2}$

As $\frac{50\,000(50\,001)}{2} = 1250025000 \sim 1249975000$

Therefore, operations is = $\frac{n(1+n)}{2}$

Efficiency class:

The efficiency of the code is O(n^2), as it has two nested loops that iterate over all possible pairs of elements in the array. Specifically, the code has a time complexity of O(n^2), because for an input array of size n, it executes $\frac{n(1+n)}{2}$ comparisons in the inner loop.

1.2

Pseudocode:

```
ALGORITHM Mergesort(A[0..n − 1])
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order and returns the inversion count
    if n > 1
        copy A[0..n/2 − 1] to B[0..n/2 − 1]
        copy A[n/2..n − 1] to C[0..n/2 − 1]
        Mergesort(B[0..n/2 − 1])
        Mergesort(C[0..n/2 − 1])
        Merge(B, C, A)

ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])

//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C and renturns the inversion count

i ← 0;
j ← 0;
k ← 0;
while i<p and j<q do
    if B[i] <= C[j ]
        A[k]← B[i];
        i ← i + 1
        k ← k + 1
    else
        A[k]← C[j ];
        j ← j + 1
        k ← k + 1
if i <= p
    copy C[j..q − 1] to A[k..p + q − 1]
else
    copy B[i..p − 1] to A[k..p + q − 1]
```

Operations:

The operations amount is approximately $3n \log(n)$

Efficiency class:

Based on a program with a $3n \log(n)$ operations amount, the efficiency class of the program is O(n log(n)).

a=2

b=2

f(n) = O(n)

T(n) = aT(n/b) + f(n) = 1T(n/2) + O(n)

SUB PROBLEMS:

$n^{log_b(a)}$ <=> O(n)

n^1 == O(n)

The cost of F(n) and the sub problems are the same, so

T(n) = O(n log n)

1.3

Brute Force Method:

In this method, any time a number is smaller than the ones after, it swaps until it is properly sorted. This is done using i<j. This method takes a significant amount of time, taking 8.2928 seconds. The amount of time to search through a huge array and swap all individual numbers into ascending order does not make this type of code worthwhile

Merge Sort Method:

This program recursively splits the arrays in half until the elements are individually in arrays. Then, it will merge them back in ascending order in one final array. This takes significantly less time then the brute force method, as it only takes 0.0118 seconds according to the program

2.1

Pseudocode:

```
ALGORITHM bruteConvexHull(S[Struct Array of Points from 0...n-1])
   // Goes from leftmost point and checks each candidate edge until convex hull is found
   // INPUT: S is the set of points
   //OUTPUT: Array of the points of the hull
   pointOnHull <- leftmost point in S //which is guaranteed to be part of the Hull(S)
   i <- 0
   repeat
      P[i] <- pointOnHull
      endpoint <- S[0]      // initial endpoint for a candidate edge on the hull
      for j from 1 to n-1
         if (endpoint <- pointOnHull) or (S[j] is on left of line from P[i] to endpoint)
            endpoint <- S[j]   // found greater left turn, update endpoint
      i <- i+1
      pointOnHull <- endpoint
   until endpoint <- P[0]
```

Operations:

The amount of operations is approximately O(nh), where h is the number of hull points found, and n is the input size.

Efficiency Class:

The efficiency class of the program is in the best case, its $O(n^2)$, meanshat it is inefficient. This is because of the nested loop that iterates over all possible points to find the next point in the convex hull.

2.2

Pseudocode:

```
ALGORITHM findConvexHull (pointsSet[0...len], len, leftPoint, rightPoint)
//Creates three arrays to start merge sort for the set
//INPUT: array of all points, the length of the set, and the extreme left and right points
//OUTPUT: Creates a final array of the convex hull

    aboveArr <- array of points of size len
    belowArr <- array of points of size len
    finalArr <- array of points of size len

    belowCount,aboveCount,finalCount <-0

    qsort(pointsSet) //Organize by the x coordinates

    i <- 0
    while i < len
        val <- orientation(leftPoint, rightPoint, pointsSet[i]) //checks where the point is in relation to the extreme points
     if (val is greater than 0)
     aboveArr[aboveCount] <- pointsSet[i]
            aboveCount <- aboveCount + 1
     if (val is less than 0)
     belowArr[belowCount] <- pointsSet[i]
            belowCount <- belowCount + 1




ALGORITHM findHalfHull (pointsSet[0...len], len, leftPoint, rightPoint, finalArr, finalCount)
//divides the arrays into smaller arrays and checks if its along the farthest edge
//INPUT: array of all points, the length of the set, and the extreme left and right points, and the final array to input into
//OUTPUT: populates the final array of the convex hull

    if (the array is empty)
 return

    maxVal <- 0
    maxIndex <- 0
    i <- 0
    while i < len
 val <- oritentation(leftPoint,rightPoint,pointSet[i])
 if pointsSet[i] is greater than 0 and maxVal>val
     maxIndex <- i
     maxVal <- value

    if MaxIndex is not found
 if leftPoint is not in finalArr
     add leftPoint to finalArr
     finalCount <- finalCount +1
 if rightPoint is not in finalArr
     add rightPoint to finalArr
     finalCount <- finalCount +1

ALGORITHM findHalfHull(pointsSet[0...len], len, leftPoint, pointsSet[maxIndex], finalArr, finalCount)
ALGORITHM findHalfHull(pointsSet[0...len], len, pointsSet[maxIndex], rightPoint, finalArr, finalCount)
```

Operations:

I was unable to calculate the exact formula for the operations for each operation, however, the code itself is able to show the amount of operations for each individual run of the program

Efficiency Class:

T(n) = aT(n/b) + f(n)

a and b = 2

T(n) = aT(n/b) + f(n) = 2T(n/2) + O(n)

SUB PROBLEMS:

$n^{log_b(a)}$ <=> O(n)

n^1 == O(n)

The cost of F(n) and the sub problems are the same, so

The time complexity is O(n log n)

<u>2.3</u>

<u>Brute Force Method:</u>

This program takes in the specific left extreme point, and then checks each one by distance to check if it is on the right side of the line. It then checks every point and slowly creates the hull by going around to each point and ensuring that there is no point with a greater x at first and then y to create the convex hull. It takes 0.0167 seconds. Surprisingly, this method is faster then the merge sort method, however both are quite fast. The Jarvis method is an extremely efficient method for finding the convex hull

<u>Merge Sort Method:</u>

This program goes through the array of all points and separates it into two arrays based on its position based on the extreme points asked for. Then, it continues to separate them into smaller arrays until only the ones that are the included in the convex hull. Then, it returns all of the elements that are in the convex hull into a final array. It took 0.0215 seconds to run which is slower then the brute force method, but not by much