Rahul Nuggehalli Gopinathan rnn4511@rit.edu

Sharath Navalpakkam Krishnan sxn9447@rit.edu

Homework 2.

Problem 1.

$O(n)$	Step 1	:	Converting n numbers to points(x,y)
$\Omega(n\log n)$	Step 2	:	Convex Hull Algorithm
$O(n)$	Step 3	:	Generate x-coordinate of points to get sorted list of input.
$\Omega(n\log n)$			

To prove Step 2

$$
\begin{aligned}
T(n) &= T_C + O(n) \\
&= O(n\log n) + O(n) \\
&= O(n\log n)
\end{aligned}
$$

Which is also $\Omega(n\log n)$

Thus every algorithm for the convex hull problem runs in $\Omega(n)$. [Since $\Omega(n)$ € $\Omega(n\log n)$]

Hence Proved.

Problem 2. Random pivot quick sort and insertion sort

N	Quick sort -RSA	Quick sort - FSA	Insertion sort - SA	Radix sort - SA	Quick sort -RUA	Quick sort -FUA	Insertion sort - UA	Radix sort - UA
400	0.3055	1.5720	0.0147	0.9804	0.2772	0.1593	0.8915	1.0347
4000	4.4947	9.0025	0.1338	3.1831	4.3677	3.9981	8.3282	3.2752
40000	15.0385	331.2830	1.2699	14.4918	10.8614	9.9908	198.0420	16.2928

*All time in milliseconds. ****RSA**: Random-Pivot Sorted Array; **FSA**: First Element - Pivot Sorted Array; **SA**: Sorted Array; **RUA**: Random-Pivot Unsorted Array; **FUA**: First Element-Pivot Unsorted Array; **UA**: Unsorted Array

I. Quick sort average case time complexity is said to be O(n log n) and worst case is O(n^2).
   a. In our timing experiment, we found that quick sort runs best when provided with an unsorted array and random pivot element. We deduced this to be because of a shallower recursion tree. The tree would have 'log n' height with the n comparisons.

   b. Quick sort performed the worst when provided with a sorted array and pivot being the first element of the array at each iteration. The depth of the recursion tree here, we observed, to be far greater. At each iteration, the algorithm finds each of the elements in the array to be bigger than the pivot, moving them one step to the right and plugging back the pivot in the original place at the end. Therefore, at each step, one element in the array is reduced (sorted) and this gives a complexity of n * (n-1) * (n - 2).. which is O(n^2).

II. The insertion sort, we observed, to be basically the same as quick sort when using the pivot element always being the first element in the array at each iteration. This gives it a complexity of O(n^2).
   a. The insertion sort worked in best case with complexity of O(n). This was when it was given a sorted array. We observed this to be the way the algorithm is built. The algorithm moves the element under investigation at each step from index 'i' to 'i-1' until it finds an element at the index 'i-1' to be lesser. So in a sorted array, this would be O(1) at each step since it's already sorted. The entire complexity is boiled down to O(n).

   b. The insertion sort's worst case and average case then would be O(n^2). Consider an array which is reverse-sorted. Here each element starting from the first is moved 1, 2, 3... n-1 steps back to their natural order index. That gives it a complexity of (n - 1) * (n - 2).. which is again O(n^2).

III. The radix sort is observed to have a complexity of O(n). We have shown this in HW-1 report. Here are a few observations:
   a. The radix sort is sometimes a little slower than the quick sort. This we deduce is due to the fact that there are constant factors in the algorithm affecting its runtime.
   b. We believe that radix sort will remain linear for large N values, when the condition that the range of input will be between 0 .. n^2 - 1. This makes it faster than quicksort which is in best case O(n log n).