Project 3: Sorting Algorithms

CSE3333

The University of Texas Rio Grande Valley

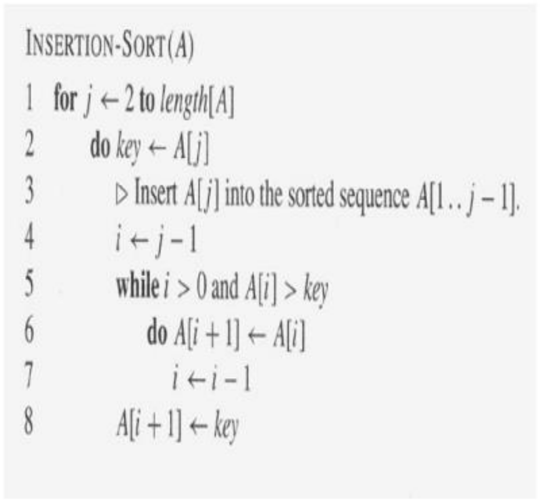
Spring 2018

Dr. Liyu Zhang

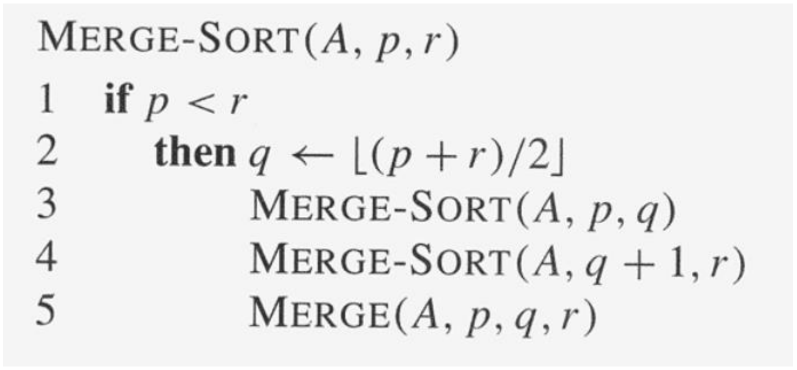
Juan Bermudez

October 18, 2018

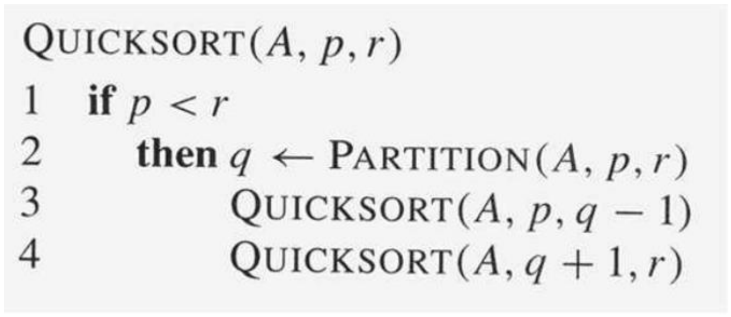
The program's code may be executed on Visual Studio or on an online C++ compiler. All the sorting algorithms work accordingly in sorting integers. The program will sort 3 randomly generated arrays with values from 0 to 1000. The ReverseArray function will sort the arrays in reverse order, from largest to smallest. The comparisons made in each array by an algorithm are also displayed, along with the running times. A persistent issue that kept crashing the program was the size of the arrays. For example, if an array's length was of 10^6, then the program would end. Another issue encountered with extraordinarily large values in an array was the output of negative numbers. For this reason the sizes of the arrays were kept at 10^3, 10^4, and 10^5. The output of the program is given in order for Insertion Sort, Merge Sort, Heap Sort, and Quick Sort. The algorithms were designed by following the pseudocode on the power of chapter 7



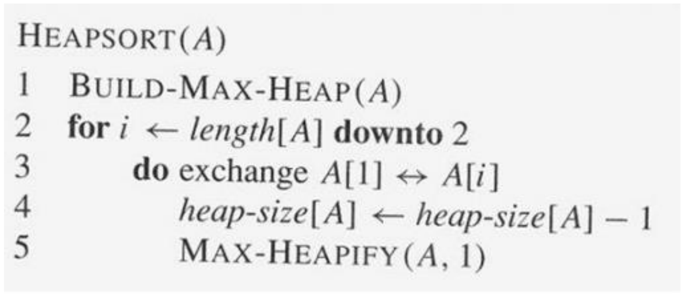
```
INSERTION-SORT(A)
1  for $j \leftarrow 2$ to $length[A]$
2      do $key \leftarrow A[j]$
3         ▷ Insert $A[j]$ into the sorted sequence $A[1 \,.\,.\, j-1]$.
4         $i \leftarrow j-1$
5         while $i > 0$ and $A[i] > key$
6             do $A[i+1] \leftarrow A[i]$
7                $i \leftarrow i-1$
8         $A[i+1] \leftarrow key$
```



```
MERGE-SORT(A, p, r)
1  if $p < r$
2      then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3              MERGE-SORT(A, p, q)
4              MERGE-SORT(A, q + 1, r)
5              MERGE(A, p, q, r)
```

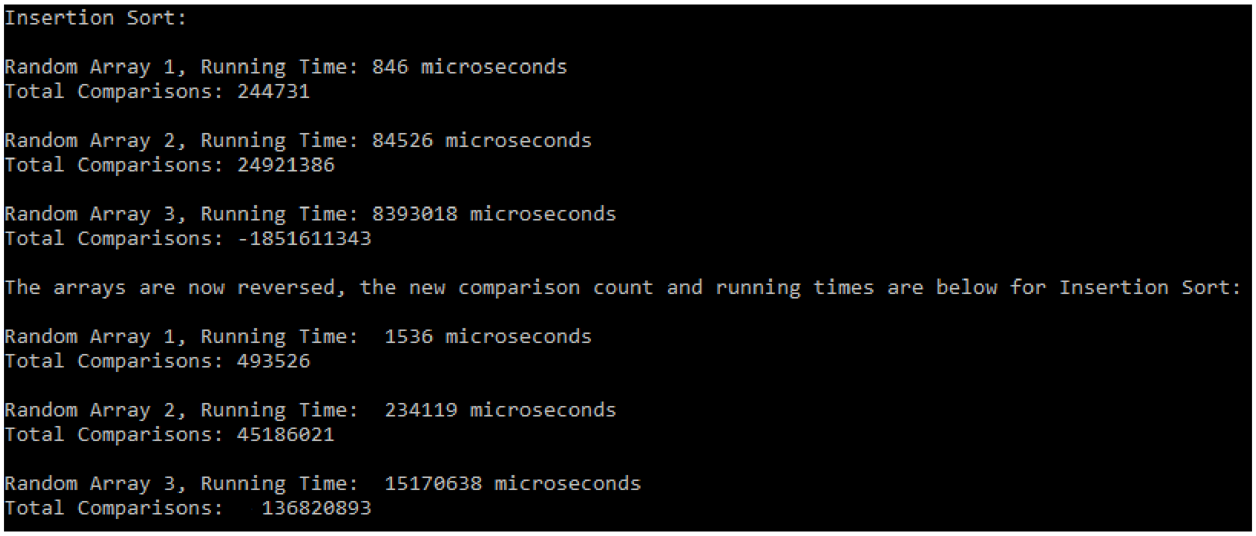


```
QUICKSORT($A$, $p$, $r$)
1  if $p < r$
2     then $q \leftarrow$ PARTITION($A$, $p$, $r$)
3          QUICKSORT($A$, $p$, $q - 1$)
4          QUICKSORT($A$, $q + 1$, $r$)
```

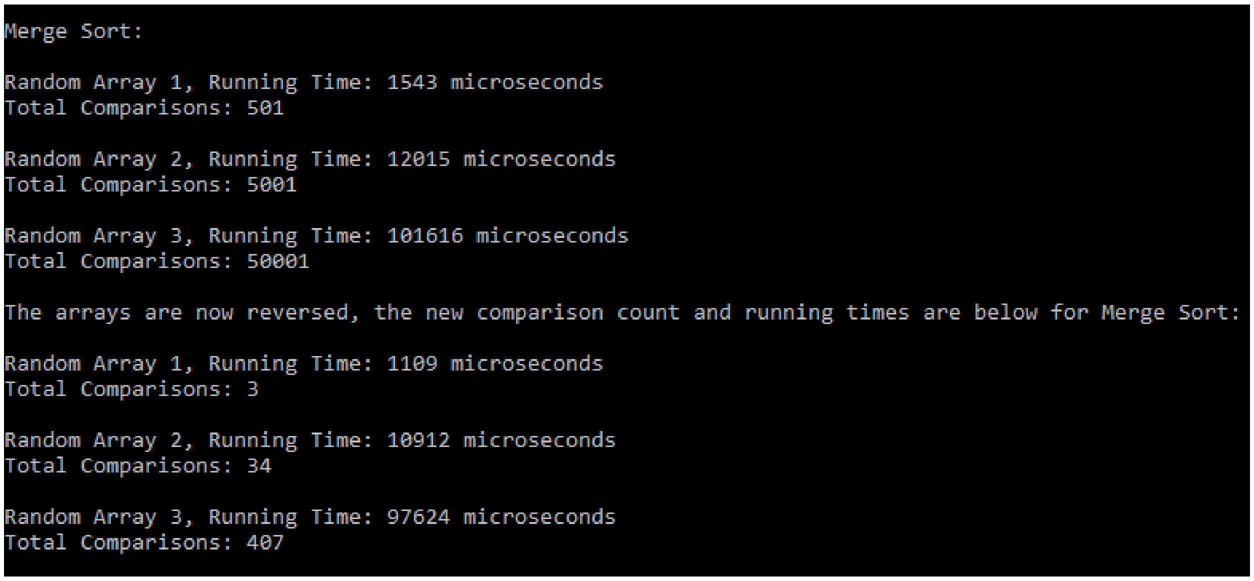


```
HEAPSORT($A$)
1  BUILD-MAX-HEAP($A$)
2  for $i \leftarrow length[A]$ downto 2
3       do exchange $A[1] \leftrightarrow A[i]$
4          $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5          MAX-HEAPIFY($A$, 1)
```

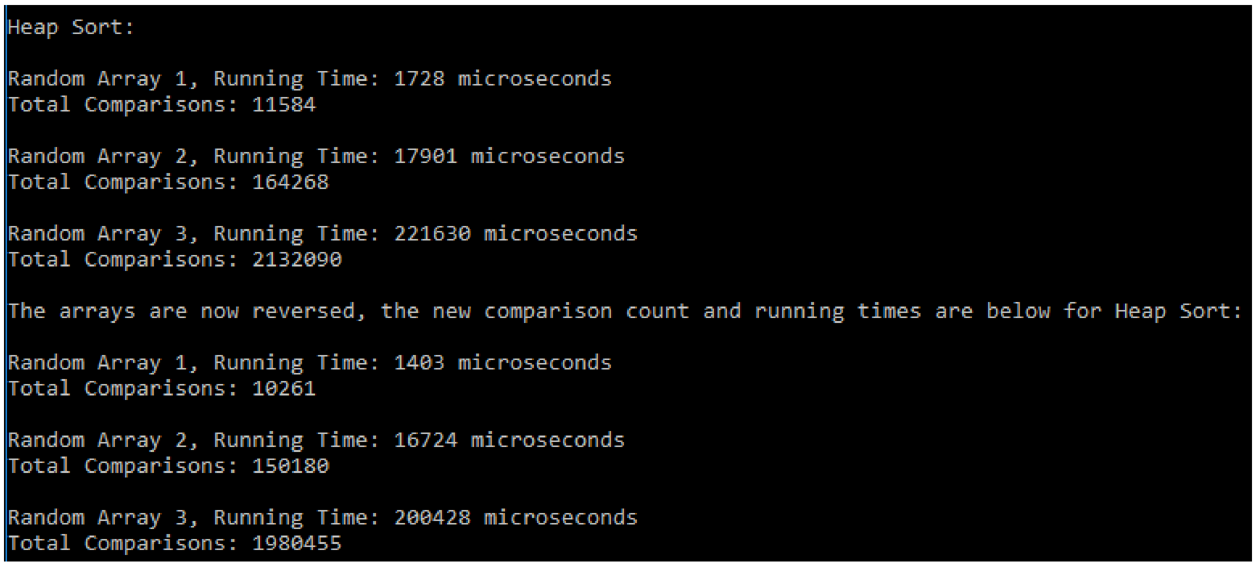
Below are the number of Comparisons for each Algorithm, along with the running times:



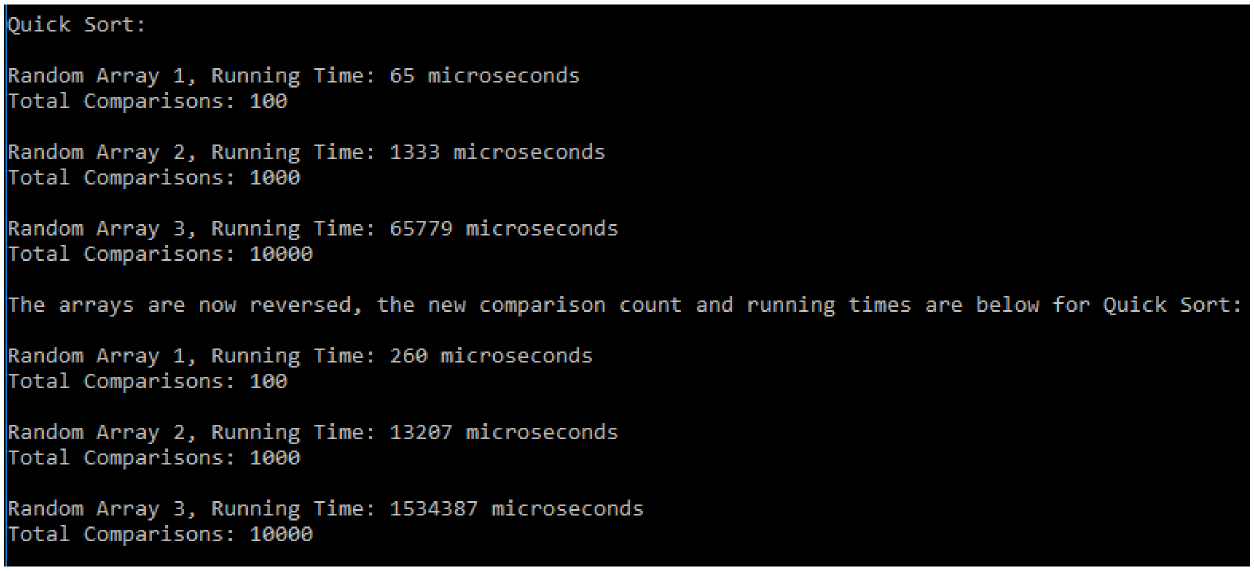
```
Insertion Sort:

Random Array 1, Running Time: 846 microseconds
Total Comparisons: 244731

Random Array 2, Running Time: 84526 microseconds
Total Comparisons: 24921386

Random Array 3, Running Time: 8393018 microseconds
Total Comparisons: -1851611343

The arrays are now reversed, the new comparison count and running times are below for Insertion Sort:

Random Array 1, Running Time:  1536 microseconds
Total Comparisons: 493526

Random Array 2, Running Time:  234119 microseconds
Total Comparisons: 45186021

Random Array 3, Running Time:  15170638 microseconds
Total Comparisons:   136820893
```



```
Merge Sort:

Random Array 1, Running Time: 1543 microseconds
Total Comparisons: 501

Random Array 2, Running Time: 12015 microseconds
Total Comparisons: 5001

Random Array 3, Running Time: 101616 microseconds
Total Comparisons: 50001

The arrays are now reversed, the new comparison count and running times are below for Merge Sort:

Random Array 1, Running Time: 1109 microseconds
Total Comparisons: 3

Random Array 2, Running Time: 10912 microseconds
Total Comparisons: 34

Random Array 3, Running Time: 97624 microseconds
Total Comparisons: 407
```



```
Heap Sort:

Random Array 1, Running Time: 1728 microseconds
Total Comparisons: 11584

Random Array 2, Running Time: 17901 microseconds
Total Comparisons: 164268

Random Array 3, Running Time: 221630 microseconds
Total Comparisons: 2132090

The arrays are now reversed, the new comparison count and running times are below for Heap Sort:

Random Array 1, Running Time: 1403 microseconds
Total Comparisons: 10261

Random Array 2, Running Time: 16724 microseconds
Total Comparisons: 150180

Random Array 3, Running Time: 200428 microseconds
Total Comparisons: 1980455
```



```
Quick Sort:

Random Array 1, Running Time: 65 microseconds
Total Comparisons: 100

Random Array 2, Running Time: 1333 microseconds
Total Comparisons: 1000

Random Array 3, Running Time: 65779 microseconds
Total Comparisons: 10000

The arrays are now reversed, the new comparison count and running times are below for Quick Sort:

Random Array 1, Running Time: 260 microseconds
Total Comparisons: 100

Random Array 2, Running Time: 13207 microseconds
Total Comparisons: 1000

Random Array 3, Running Time: 1534387 microseconds
Total Comparisons: 10000
```

(Reverse Sorted Array means the array was arranged from largest to smallest, and then sorted again)

Insertion Sort:

Insertion Sort				
Array Size	**Regular Sorted Array**		**Reverse Sorted Array**	
	Total Comparisons	**Running Time(micro sec.)**	**Total Comparisons**	**Running Time (micro sec.)**
10^3	**244731**	**846**	**493526**	**1536**
10^4	**24921386**	**84526**	**45186024**	**234119**
10^5	**181611343**	**8393018**	**136820893**	**15170638**

The Insertion Sort algorithm iterates through an entire array, removing an element from the array, and putting it where it belongs from left to right. In it's best running time, insertion sort runs on O(N).
The worst and average case is when an array is in reverse order, and that will be $O(N^2)$. For comparisons, Insertion Sort will have (N-1) on best case, and worst and average case $(1/2(N^2 - N))$.

The results are consistent with the theoretical analysis. The total comparisons and running times for the code represent the average case, showing a linear increase in the output in comparisons and running time. The worst case is represented by the reverse sorted array, which shows the values in comparisons doubling in size.

Merge Sort:

Merge Sort				
Array Size	**Regular Sorted Array**		**Reverse Sorted Array**	
	Total Comparisons	**Running Time(micro sec.)**	**Total Comparisons**	**Running Time (micro sec.)**
10^3	**501**	**1543**	**3**	**1109**
10^4	**5001**	**12015**	**34**	**10912**
10^5	**50001**	**101616**	**407**	**97624**

In Merge Sort, the array is split in half, then each half is recursively sorted, finally the merge algorithm combines the two halves. The Merge Sort algorithm makes most comparisons in the merge procedure where the size of the arrays are N/2. Thus, the equation for comparisons may be linearly (N/2 + N). For both best case and worst case, the Merge Sort algorithm will have a running time of O(N Log N), that is because Merge Sort splits any array in half, so regardless, every element in the array is compared at least once.

Here, the results are consistent with the theoretical part of the Merge Sort algorithm. The comparisons show linear consistency, and running time O(NLOGN)

Heap Sort:

Heap Sort				
Array Size	**Regular Sorted Array**		**Reverse Sorted Array**	
	Total Comparisons	**Running Time(micro sec.)**	**Total Comparisons**	**Running Time (micro sec.)**
10^3	**11587**	**1728**	**10261**	**1403**
10^4	**164268**	**17901**	**150180**	**16724**
10^5	**2132090**	**221630**	**1980455**	**200428**

The Heap Sort algorithm separates the array into a sorted and unsorted region by placing the largest element on the bottom and the smallest to the top. In Heap Sort, the number of comparisons made depends on the way elements are ordered. Regardless, the running time is the same, with O(N LOG N).

Because of this, the results show consistency. The comparisons and the running times in both regular and reversed arrays show similar results, of course the comparisons and times are different, but if seen broadly, then they are consistent. Other trials can be executed to show that the results will remain within range of each other.

Quick Sort:

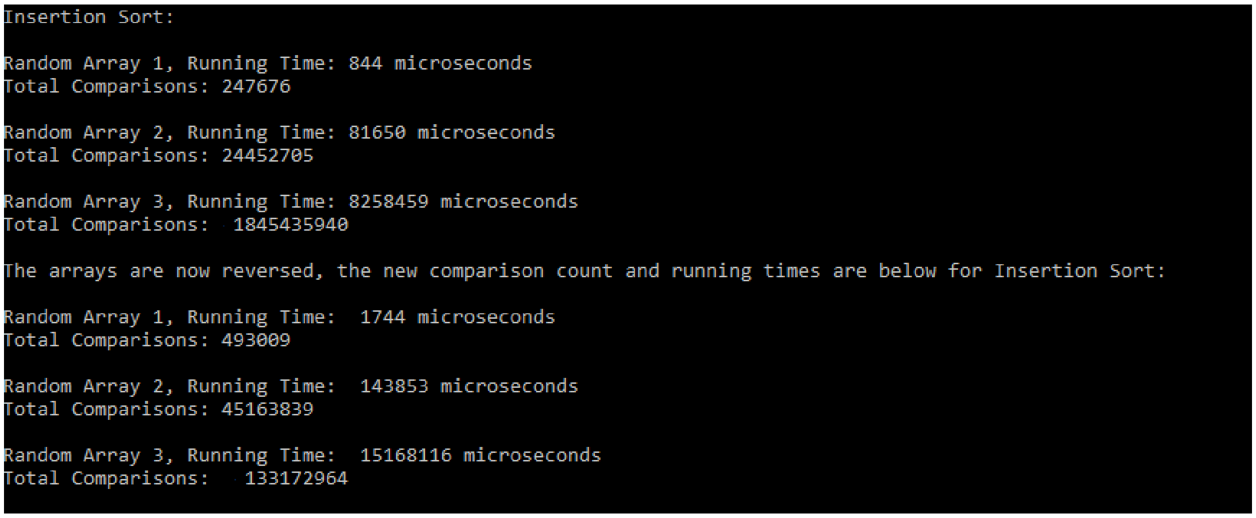
For the quicksort array, the values used are **10^2, 10^3 , and 10^4**. Numbers larger than these would cause the program to exit.

Quick Sort				
Array Size	**Regular Sorted Array**		**Reverse Sorted Array**	
	Total Comparisons	**Running Time(micro sec.)**	**Total Comparisons**	**Running Time (micro sec.)**
10^2	**100**	**65**	**100**	**260**
10^3	**1000**	**1333**	**1000**	**13207**
10^4	**10000**	**65779**	**10000**	**1534387**

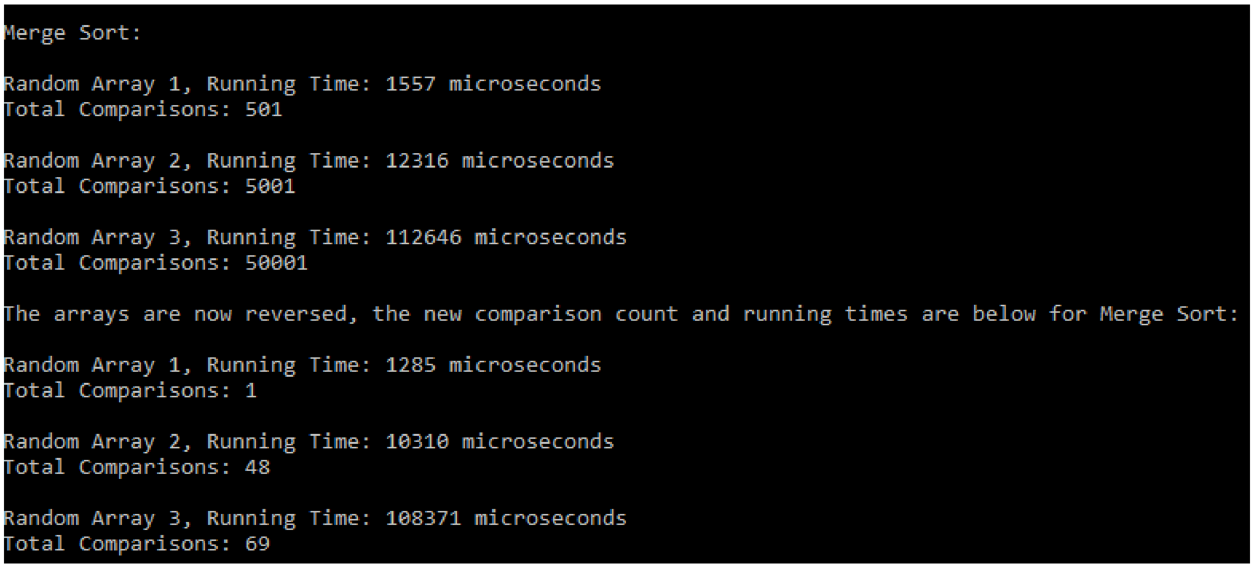
For Quick Sort, the worst case is when the pivot is the largest or smallest element in the array. The worst running time is O(N^2), and the best case is O(N Log N). For this program the pivot was not randomized. So for the Regular Sorted Array, we can assume the best case, and since the reversed array is descending from largest to small in order, then the reverse case must have the worst case.

The comparisons in the quicksort array are mostly done in partition, when they are compared to the pivot. The reason the comparisons are equal to the array is because all elements are compared to the pivot. The results are consistent with the theoretical analysis. The regular sorted array has a faster running time of O(N LOG N) (Best case) and the reversed order array has a running time that doubles when compared to the regular sorted array running time.

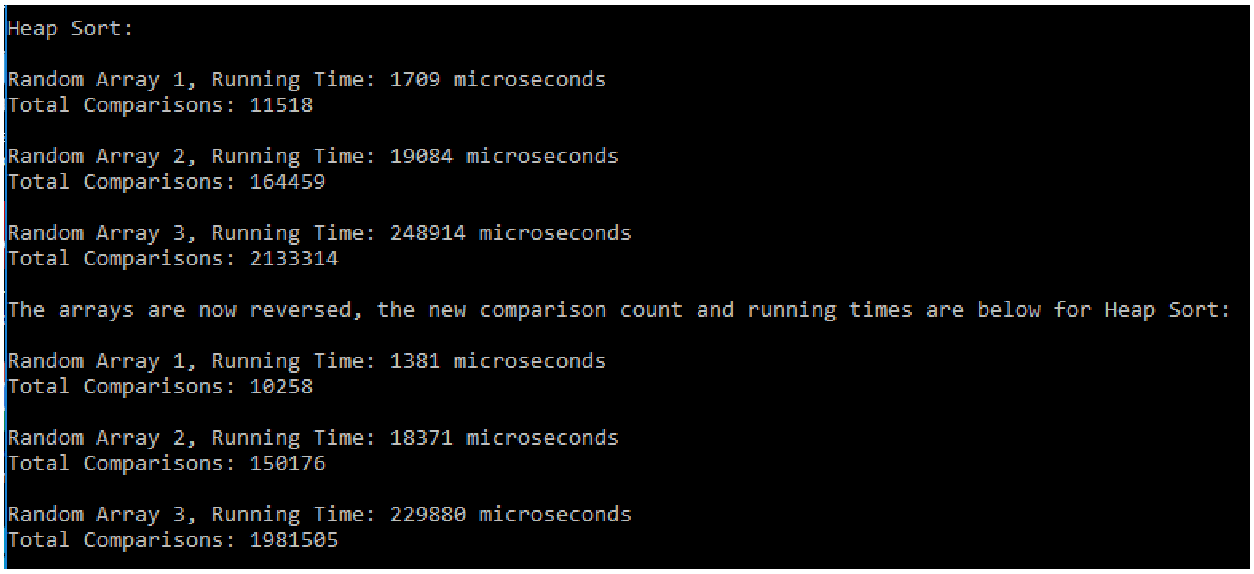
Other Runs:



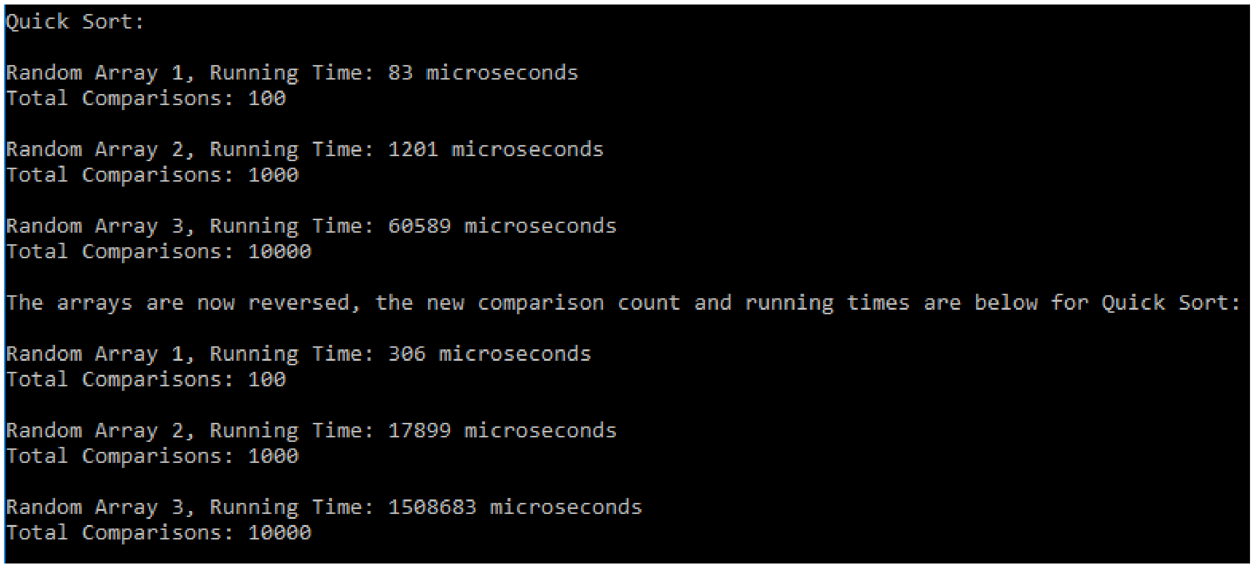
```
Insertion Sort:

Random Array 1, Running Time: 844 microseconds
Total Comparisons: 247676

Random Array 2, Running Time: 81650 microseconds
Total Comparisons: 24452705

Random Array 3, Running Time: 8258459 microseconds
Total Comparisons:  1845435940

The arrays are now reversed, the new comparison count and running times are below for Insertion Sort:

Random Array 1, Running Time:  1744 microseconds
Total Comparisons: 493009

Random Array 2, Running Time:  143853 microseconds
Total Comparisons: 45163839

Random Array 3, Running Time:  15168116 microseconds
Total Comparisons:   133172964
```



```
Merge Sort:

Random Array 1, Running Time: 1557 microseconds
Total Comparisons: 501

Random Array 2, Running Time: 12316 microseconds
Total Comparisons: 5001

Random Array 3, Running Time: 112646 microseconds
Total Comparisons: 50001

The arrays are now reversed, the new comparison count and running times are below for Merge Sort:

Random Array 1, Running Time: 1285 microseconds
Total Comparisons: 1

Random Array 2, Running Time: 10310 microseconds
Total Comparisons: 48

Random Array 3, Running Time: 108371 microseconds
Total Comparisons: 69
```



```
Heap Sort:

Random Array 1, Running Time: 1709 microseconds
Total Comparisons: 11518

Random Array 2, Running Time: 19084 microseconds
Total Comparisons: 164459

Random Array 3, Running Time: 248914 microseconds
Total Comparisons: 2133314

The arrays are now reversed, the new comparison count and running times are below for Heap Sort:

Random Array 1, Running Time: 1381 microseconds
Total Comparisons: 10258

Random Array 2, Running Time: 18371 microseconds
Total Comparisons: 150176

Random Array 3, Running Time: 229880 microseconds
Total Comparisons: 1981505
```



```
Quick Sort:

Random Array 1, Running Time: 83 microseconds
Total Comparisons: 100

Random Array 2, Running Time: 1201 microseconds
Total Comparisons: 1000

Random Array 3, Running Time: 60589 microseconds
Total Comparisons: 10000

The arrays are now reversed, the new comparison count and running times are below for Quick Sort:

Random Array 1, Running Time: 306 microseconds
Total Comparisons: 100

Random Array 2, Running Time: 17899 microseconds
Total Comparisons: 1000

Random Array 3, Running Time: 1508683 microseconds
Total Comparisons: 10000
```