Assignment 3

Name: Hyejin Kim

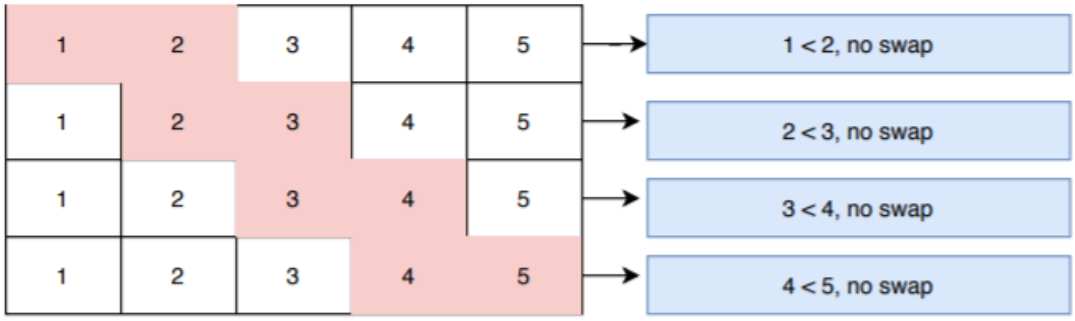
Student Number: 6823116

Date: 6/29/2020

1. Hypothesis

First, for the bubble sort, I assumed that the speed would be improved if an array was already sorted because it is not necessary to swap the values after comparing two elements. In other words, when we sort the array in ascending order, the already sorted array does not have to swap the two elements because the first value is already lower than the second value. Therefore, I thought that the time complexity of sorting the already sorted array is O(n) because there is no need to swap in the inner loop. I expected that this inner loop would not swap the elements in the first iteration, and it breaks the for loop after the iteration. Therefore, since it will take (n-1) comparison (the outer i loop), the time complexity is n-1= O(n) with the already sorted array in the Bubble sort.

<Bubble sort for the already sorted array>



In my code, you could see the inner loop (j for loop, not i for loop) in the method of bubbleSort as follows:

```
for(int i=0;i<s-1;i++){
  for(int j=0,j<s-i-1;j++){
     if(array[j]>array[j+1]{
      int temp = array[j];
      array[j] = array[j+1]'
      array[j+1] = temp;
       }
   }
  }
```

Second, for the Heapsort, I assumed that there is no effect on speed because, no matter if inputs are sorted, the Heapsort should go through the HeapBuild and Heapify functions. It will take O(n) for minHeapBuild() method as well as O(nlog(n)) for minHeap() method whether or not the array was already sorted. When the array is already sorted in the ascending order, the time complexity will take O(n) in order to convert it to a minHeap, and then it removes the last element, which is the largest value in the ascending order, and repeats (n-1) times for minHeapify (minHeap() method in my code). This heapify could be calculated as such:

log1 + log2 + log3 + (...) + log(n-1) = log(1x2x3x(…)x(n-1) = log((n-1)!) =O(nlogn)

Therefore, O(n) (buildHeap) + O(nlogn) (minHeapify) = O(nlogn).

Third, for Merge sort, I assumed that there is no effect on speed even if the array was already sorted. Since the Merge sort also should go through splitting the array until it has only one element no matter what inputs are already sorted. Furthermore, each element should be merged after finishing the split work. Therefore, the time complexity would be the same as the average time complexity with O(nlogn). Specifically, O(logn) for dividing an array into half in every step, and the running time for merge method (in my code) will take O(n). This means O(logn)*O(n)=O(nlogn).

Fourth, for Quicksort, since I chose the median value for partition function, I assumed that it might not affect on the average running time but it would be better than the worst case. In other words, the Quick sort should go through the procedures of partition and dividing the elements no matter if the array was already sorted. However, it is better than the worst case which is O(n^2) because pivot values would be extreme values, largest or smallest, in the worst case. Since in our assignment we chose a median as a pivot value, it is ideal, and the operation will take O(logn) while dividing the array into two equal halves until it is less than the partition size (in my code, I chose 11). And, all of these divided arrays should visit all n inputs, which will take O(n). Therefore, the time complexity is O(n*logn)=O(nlogn). More specifically, n/2 is each divided array into halves, and there are left and right partitions on the basis of pivot, and cn is the partition for all subarrays.

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

T(n)/n = T(n/2)/n/2 + c (divided by n in both sides)

T(n/2)/n/2 = T(n/4)/n/4 + c

T(n/4)/n/4 = T(n/8)/n/8 + c

T(n/8)/n/8 = T(n/16)/n/16 + c

(continued…)

T(2)/2 = T(1)/1 + c

After adding all the above equations, T(n)/n = T(1)/1 + clogn because there are logn of them, and that equals T(n) = cnlogn + n = O(nlogn).

Additionally, I chose the partition selection size as 10 since for very small arrays which has under 20 elements, Quicksort would not be efficient. Rather, for these small arrays, the insertion sort will improve the running time because it does not have to recursively loop through the function, which takes much time. Actually, after few experiments, I was able to observe that it saved the time almost 10 percent when the array was sorted by insertion when the number of elements were under between 10 to 13.

Lastly, for Radix sort, I assumed that there is no effect on speed whether the array is already sorted or not. This is because radix sort is basically a sorting algorithm based on the digits of number. For example, if a number has really long digits or extreme digits, such as 1000000000, it will take much longer time to pass all the way through the digits from the least significant to the most significant digits. In other words, it will take a lot of buckets for the digits. That is, even if the numbers were in the sorted order, the Radix sort should go through all the buckets for the digits to sort the elements. Therefore, the time complexity would be the same as the average running time, which is O(n), because on average time it will take

O(p(n+b)) where b is the number of buckets, like the digits of 0,1,2,3 to 9, and n is the total numbers to sort, and also p is how many times the function has to pass.

2. Experimentation (Results)

< Table of Values - Result>

Algorithm	Running Time (milliseconds)			
	5,000 (Array size)		10,000 (Array size)	
	After sorting	After re-sorting	After sorting	After re-sorting
Bubble sort	33.2468	18.7211	105.0046	64.3915
Heap sort	0.4253	0.1267	0.891	0.2798
Merge sort	1.5392	0.2933	2.729	3.0017
Quick sort	1.6977	0.417	2.6299	0.5624
Radix sort	2.9204	2.4299	3.8757	2.1984

3. Analysis

After testing all the algorithms, even if the Bubble sort took much time to sort compared to other sorts, it showed the most improved running time between after sorting and re-sorting the already sorted array among other sorts, which corresponds to what I expected. Other algorithms were also slightly improved in their speeds, however they didn't show any huge differences between sorting and re-sorting. The results are as follows: In the 5,000-array size, Bubble sort was improved by 14.5257 milliseconds, Heap sort was improved by 0.2986 milliseconds, Merge sort was improved by 1.2459 milliseconds, Quick sort was improved by 1.2807 milliseconds, and Radix sort was improved by 0.4905 milliseconds. Also, in the 10,000-array size, the Bubble sort was improved by 40.6131 milliseconds, Heap sort was improved by 0.6112 milliseconds, Merge sort was "not" improved which means that the running time was increased by 0.2727 milliseconds, the Quick sort was improved by 2.0675 milliseconds, and the Radix sort was improved by 1.6773 milliseconds.

<Table of Values – Time Difference>

Algorithm	Time Difference between after sorting and re-sorting (milliseconds)			
	5,000 (Array size)	rank	10,000 (Array size)	rank
Bubble sort	33.2468-18.7211 =14.5257	1	105.0046-64.3915 =40.6131	1
Heap sort	0.4253-0.1267 = 0.2986	5	0.891-0.2798 = 0.6112	4
Merge sort	1.5392-0.2933 = 1.2459	3	2.729-3.0017 = -0.2727	5
Quick sort	1.6977-0.417 = 1.2807	2	2.6299-0.5624 = 2.0675	2
Radix sort	2.9204-2.4299 = 0.4905	4	3.8757-2.1984 = 1.6773	3

In terms of the different size of the arrays, the Bubble sort in the both size array (5,000 and 10,000 elements) showed a biggest gap compared to other sorts, as mentioned earlier. And, the following ranking was different, depending on the array size as above table. However, Merge sort showed rather the increased time (0.2727 milliseconds) after re-sorting the already sorted array in the 10,000-array size. And, Heap sort, Quick sort, and Radix sort in 10,000 array size showed slightly larger time difference than 5,000 array size. For example, in the Heap sort, the time was 0.2986 milliseconds when the array size was

5,000, but it was decreased by 0.6112 milliseconds after resorting when the array size was 10,000. Also, Quick sort decreased 1.2807 milliseconds after resorting when the array size was 5,000, but in the 10,000-array size, it was decreased by 2.0675 after re-sorting. Radix sort also showed a larger time difference when the array size was 10,000, which was 1.6773 milliseconds, compared to that of 5,000 array size, which was 0.4905.