Assignment Two – LaTeX Sort, Search, Hash

Rocco Piccirillo
Rocco.Piccirillo1@Marist.edu

March 14, 2019

1 Sort Section

Table 1: Sorts

Sort Name	*Comparisons*	*BigONotation*
Selection	225,569	$O(n^2)$
Insertion	114,309	$O(n^2)$
Merge	18,017	$O(nlog_2n)$
Quick	7,623	$O(nlog_2n)$

1.1 Selection Sort

The amount of comparisons being performed for Selection Search are about 225,569. This is almost equal to the list being squared, and then split in half. Through Big O Notation, we ignore the 1/2. The Big O Notation for Selection Sort is O(n^2). The forumla being followed would be n(n+1)/2 = $n^2 + n/2 = 1/2(n^2) + n$. Each time the sort runs through, it selects the min at each pass, and swaps with the currently selected index (if the found min is lower than it). To get the worst case scenario, you'd need to have the array sorted backwards so going from Z to A. Or for fun, you can potentially break it with an already sorted list.

1.2 Insertion Sort

The amount of comparisons being performed for Linear Search are about 114,309. This is roughly equal to about half of the numbers of strings in the list squared. But, through Big O Notation, we ignore the 1/2. The forumla being followed would be n(n+1)/2 = $n^2 + n/2 = 1/2(n^2) + n$. This is along the same guidelines as selection sort. Though, insertion sort is more optimal because, say you are sorting an already sorted list, it'll get run through once instead of multiple times like the selection sort. Each time the sort checks for the next one in line, it traverses the list, and expands the originally sorted list by one. For this, the array would have to be reversed sorted as well to find the worst possible sort. But, the best, unlike selection sort, would for the array to already be sorted.

1.3 Merge Sort

The amount of comparisons being performed for Merge Sort are about 18,017. This is close to the worst case scenario, which is $O(nlog_2n)$. For Merge Sort we take the array and split it into sub arrays until we eventually get n number arrays of size 1. Through this we merge the arrays back back together. This leads us to getting n comparisons from the start, after splitting the array over and over again, we get log_2 for n. Putting these two together, gets $O(nlog_2n)$. To get this, you'd need to have a merge sort of alternating values.

1.4 Quick Sort

The amount of comparisons being performed for Quck Sort are about 7,623. This is close to the worst case scenario, which is $O(nlog_2n)$. Quick sort is very close to merge sort as in they have the same worst case scenario. For Quick Sort we take the array and find a specific point we base our sort off of. Depending on the point, we either split into an array of bigger numbers or smaller numbers. Both arrays again both pick a specific point to split off of. This occurs until all numbers in the array are in their own arrays. After this return all the arrays back together. To get the worst possible sort, you'd have to made the selected point always the fist or the last point. This doesn't guarantee, but there is always a good chance of happening this way

2 Search Section

Table 2: Sorts

Search Name	$ComparisonsAverage$	$BigONotation$
Linear	333	$O(n)$
Binary	8	$O(log_2n)$

2.1 Linear Search

Linear Search is pretty simplistic. It searches for what it wants, doesn't find it, moves on to the next index. The worst case is O(n). This would be because you can't find your item you're searching for until the last index. For my linear search, depending on the randomly selected words, had an average case ranging from as low as 280 to as high as 380. In class you said how the average case should be about 333. Mine ranged through that average plenty of times just I thought I would showcase my lowest and highest.

2.2 Binary Search

Binary Search is a step in a different direction from linear. This goes to the midpoint and depending on if the search is less than or greater than the midpoint, it then takes the midpoint of the bigger or smaller inputs and repeats till the value is found. The worst case for this is $O(nlog_2n)$. With LogN which is of base 2, we are making the searching space half at each iteration. When searching for a number and it doesn't get selected as the midpoint each time and is chosen last, is when this worst case could occur. The average amount of comparisons I found for binary were about 8.

3 Hash Table

Table 3: Sorts

Hash	$ComparisonsAverage$	$BigONotation$
Linked Hash	3	$O(n)$

3.1 Linked Hash Table

The worst time complexity for hash table would be O(n). This is because, sometimes, too many elements could be chained into the same key, so looking inside the entire key would be take about O(n) time. But, in luckier situations it takes about O(1) for best case which is when we find it in the first index. For my hash I had an average of about 2 compares for each search. But, you wanted us to add a compare for the initial get so it took about 3 compares (technically) to find the selected string.