Kern Espinoza
CS260
11 October, 2021

Assignment 1 Write-Up

2)

a) Time taken to find “Mertz” in unsorted list:

Elements	Duration
100	0.0008 ms
1,000	0.006 ms
10,000	0.08 ms
100,000	0.87 ms

b) The countLastName function simply iterates through the entire array one time, comparing each element with the given lastName value. The time complexity would be **O(*n*)**, because the number of iterations is always equal to the size of the array.

c) In my estimate, the amount of time to process 1,000,000 names (on this machine) should be roughly **8 ms**.

4)

a) Time taken to find “Mertz” with modified binary sort:

Elements	Duration
100	0.0003 ms
1,000	0.001 ms
10,000	0.002 ms
100,000	0.01 ms

b) The countLastNameInSorted function has two main parts to it. First, a binary search is used to locate a record with the matching last name, to be used as a starting point.

Secondly, a *while* loop iterates to the left until it finds the first record with that name, and then another *while* loop iterates to the right until it finds the last record with the matching last name. The time complexity of the first part should approach O(*log n*), because of how the array decreases in size with each iteration. I think the second component of this counting function has a time complexity of O(*1*), because each *while* loop runs an unknown number of times which does not necessarily scale with the number of elements (it depends on how many different people have the same last name). Therefore, the time complexity of this algorithm should be **O(*log n*)**.

c) As a result, I'd think that a search of 1,000,000 elements would take approximately **0.02 ms**, largely because of the "halving" nature of the binary search.

6)

a) Time taken to sort a list using Quicksort:

Elements	Duration
100	1.999 ms
1,000	4 ms
10,000	47 ms
100,000	574 ms

b) The time complexity of Quicksort is much harder to analyze. Like the binary search, Quicksort splits itself every iteration. However, the sizes of each half are not guaranteed to be equal because of how it uses the pivot index approach. In the worst case scenario, each split would result in one partition of size 0 and another partition of size (n-i). In other words, the size of the non-empty partition would decrease linearly (by 1 each iteration). Using the formula for the sum of an arithmetic series, the number of partitions in this worst case scenario should equal *(n^2)/2*. In terms of time complexity, I'd say that the worst case should be represented as O(*n^2*). The best-case would be one where every single partitioning results in two equally-sized halves, because that means that doubling the n value would add only one single extra step. That inverse-square type of scaling can be represented as O($log_2 n$). However, the partitioning function itself also does an O(*n*) amount of work, so increasing the number of elements always linearly increases the time it takes to solve each partition. Performing n amount of work times $\log_2 n$ number of partitions should simplify to *n * log(n) / log(2)*. Disregarding the *1 / log(2)* coefficient gives a time complexity of **O(*n * log n*)**. Extrapolating from observed values and time complexity analysis, I'd estimate that a list of 1,000,000 records would take approximately **7600 ms** to sort using this machine.