Dichoso, Aaron Gabrielle
Donato, Adriel Joseph
Natividad, Josh Austin
Razon, Luis Miguel

Documentation Report - MCO1

I. Description of the modified sorting algorithms

To create a sorting algorithm that fits the project's specifications, we first used a standard Quick Sort and Selection Sort algorithm as our foundation. However, the algorithm needed to work with strings instead of integers, so various modifications were made.

First, both functions accept an array of characters as input, referred to as mainString. From the implementation of the algorithm, it is assumed that arrays work similarly to C arrays, wherein the name of the array also denotes the pointer of the first element. By making this assumption, it is possible to derive a suffix given an index by using the formula: mainString + index.

While in the algorithm, it is shown that 2 strings are compared via inequality operators, the algorithm may differ from the actual program. To lexicographically compare strings in the program, the strcmp() function provided by the <string.h> library will be used. A positive integer was given by the function if the value of the first string is greater than the second string, a negative integer is given if the second string is greater than the first string, and a 0 is given if the 2 strings are equal. Using this function, it was possible to sort the suffixes lexicographically.

Additionally, to save space, the suffixes were not stored inside an array that holds strings. Considering that the empirical analysis would consider the testing of n = 65536, it was found to be inefficient to create a two-dimensional array with a max size of 65536 x 65536. Alternatively, we decided to store the corresponding indices of each suffix instead, and derive the corresponding suffixes from a given string using these indices. From this implementation, the Quick Sort and Selection Sort algorithms were modified to take in the given string and the arrays of indices as the input instead. Using the derived suffix, the array of indices would then be sorted appropriately, corresponding to the lexicographical value of the suffix. A supplementary swap function would also be created during programming for both sorting algorithms to aid with the sorting.

All in all, the selection sort algorithm would loop through the indices array, compute for the corresponding suffix string, and obtain the unsorted suffix with the lexicographically lowest value and place its index at the start of the unsorted array until the array is sorted (See Figure 1). The quick sort algorithm, in contrast, assigns the last index of the unsorted array as a pivot value, and places indices which correspond to suffixes with a lexicographically lower value to its left, and those with a higher value to its right. This will be done recursively to the different partitions of the array, splitting it into subarrays, until the entire array is sorted (See Figure 2).

II. Theoretical running time analysis of the algorithms

The theoretical running time of the chosen algorithms can be expressed by determining its respective time complexity. The original Selection Sort algorithm is said to be the most simplistic but tends to be inefficient (Chauhan & Duggal, 2022). The reason being is because its time complexity in its best-case, average-case, and worst-case is all $O(n^2)$. Having an average and worst-case complexity of $O(n^2)$ makes it not suitable for sorting large data sets (Selection Sort, n.d). To be able to find the time complexity of the modified algorithm, we then need to consider its new variables, statements, and count how many times the iterative statements are executed. So, if we compute all these, we then get a time function of $T(n) = n^2 + 3n - 3$ which is equivalent to a time complexity of $O(n^2)$ (See Figure 1).

```
1 SELECTION_SORT (mainString, SuffixIndices, n)
2       for i = 1 to n - 1
3               selected_index = i
4               //copy mainString + SuffixIndices[i] to selected_string

5               for j = i to n
6                       //copy mainString + SuffixIndices[j] to curr_string

7                       if (curr_string < selected_string)
8                               //copy curr_string to selected_string
9                               Selected_index = j
10              //swap SuffixIndices + i with SuffixIndices + selected_index
```

Line 2: $(n-1)-1+1+1 = n$

Line 3: $(n-1)-1+1 = n-1$

Line 4: $n-1$

Line 5: $(n-1)(n-\sum_{i=1}^{n} i + 1 + 1)$

$= (n-1)(n) - \frac{n(n+1)}{2} + 1 = (n-1)(n) - \frac{n^2+n}{2} + 1 = n^2 - n - \frac{1}{2}(n^2+n) + 1$

Line 6: $(n-1)(n-\sum_{i=1}^{n} i + 1) = (n-1)(n) - \frac{n(n+1)}{2} = n^2 - n - \frac{1}{2}(n^2+n)$

Line 7: $(n-1)(n-\sum_{i=1}^{n} i) = n^2 - n - \frac{1}{2}(n^2+n)$

Line 8: n (assuming worst case where current string is always less than selected string, does not matter with growth of function anyways)

Line 9: n (assuming worst case where current string is always less than selected string, does not matter anyways with growth of function anyways)

Line 10: $n-1$

$T(n) = n + (n-1) + (n-1) + (n^2 - n - \frac{1}{2}(n^2+n) + 1)$

$+ (n^2 - n - \frac{1}{2}(n^2+n)) + (n^2 - n - \frac{1}{2}(n^2+n)) + n + n + (n-1)$

$= 6n - 3 + 2(n^2 - n - \frac{1}{2}(n^2+n))$

$= 6n - 3 + 2(\frac{1}{2})(2n^2 - 2n - n^2 - n)$

$= 6n - 3 + (n^2 - 3n)$

$= n^2 + 3n - 3 = O(n^2 + 3n - 3) = O(n^2)$

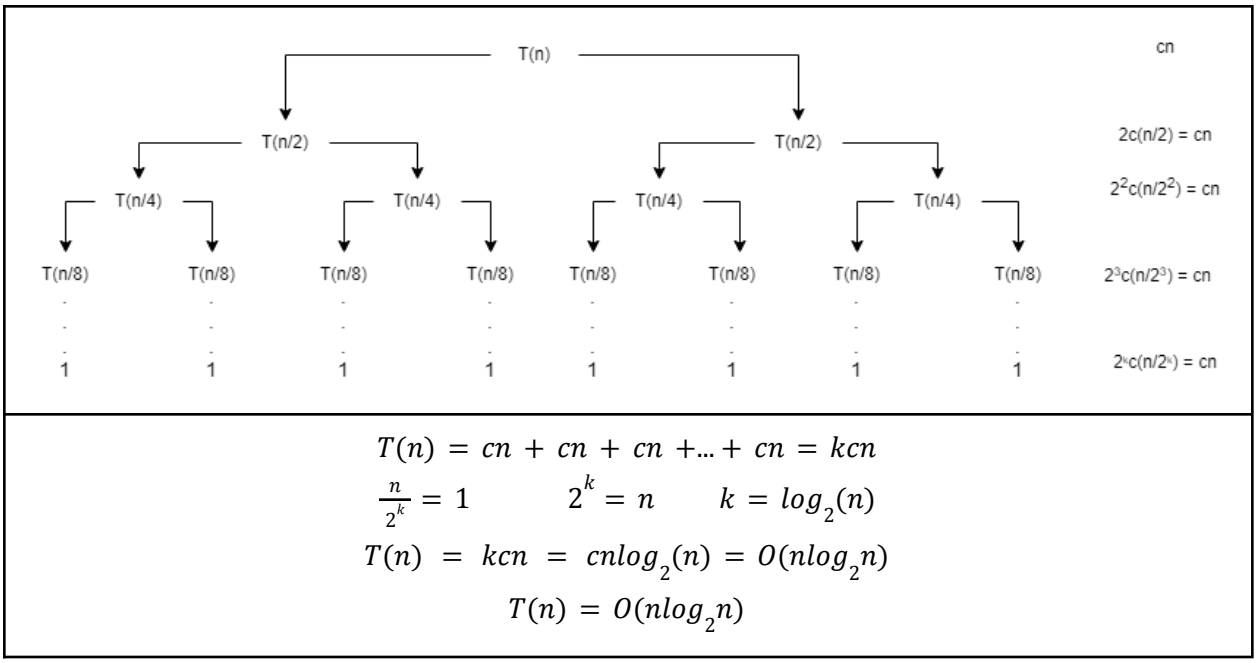
$T(n) = O(n^2)$

Figure 1. Selection Sort Algorithm & Theoretical Running Time Analysis

The Quick Sort Algorithm is considered to be one of the fastest sorting algorithms. It is part of many data processing systems because of its high efficiency and scientific structure (Xiang, 2011). Its time complexity in best-case and average-case is of $O(nlog_2 n)$ and that its worst-case is in $O(n^2)$. To obtain the time complexity of the modified algorithm, we need to analyze it by representing it in a recursion tree to observe how each recursive call behaves. We can then observe that the sum of each level is of cn, and multiplying the height of the recursion tree will give the time function $T(n) = kcn$. With the given time function, we can then conclude that the time complexity is equivalent to $O(nlog_2 n)$ (See Figure 3).

```
QUICKSORT (mainString, SuffixIndices, n, left, right)
1       if left < right
2               q = Partition(mainString, SuffixIndices, n, left, right)
3               QUICKSORT(mainString, SuffixIndices, n, left, q - 1)
4               QUICKSORT(mainString, SuffixIndices, n, q + 1, right)

PARTITION (mainString, SuffixIndices, n, left, right)
1       //copy mainString + SuffixIndices[right] to pivot_string
2       i = left - 1

3       for j = left to right
4               //copy mainString + SuffixIndices[j] to curr_string

5               if curr_string < pivot_string
6                       i = i + 1
7                       //swap SuffixIndices + i with SuffixIndices + j

8       //swap SuffixIndices + (i + 1) with SuffixIndices + right
9       return i + 1
```

Figure 2. Quick Sort Algorithm



$$T(n) = cn + cn + cn + \ldots + cn = kcn$$

$$\frac{n}{2^k} = 1 \qquad 2^k = n \qquad k = log_2(n)$$

$$T(n) = kcn = cnlog_2(n) = O(nlog_2 n)$$

$$T(n) = O(nlog_2 n)$$

Figure 3. Quick Sort Running Time Analysis & Recursion Tree

III. Theoretical analysis of memory usage of the algorithms

The theoretical analysis of memory usage of the chosen algorithms can be expressed by determining their respective space complexity. If we observe the original Selection Sort algorithm, its space complexity is said to be $O(1)$ because it uses two constant spaces for 2 variables to swap elements and 1 to keep pointing to the smallest element in the unsorted array (Trivedi, 2022). Compared to the modified algorithm which takes in strings and sorts them lexicographically, it has various variables and arrays such as mainString, SuffixIndices[], variables i, n, j, etc. So, when considering all the variables in the algorithm, its space function is $S(n) = 4n + 4$ which is equivalent to a space complexity of $O(n)$ (See Figure 4).

Memory Requirement:

$mainString = n$

$SuffixIndices = n$

$n = 1$

$i = 1$

$j = 1$

$selected\_string = n$

$curr\_string = n$

$selected\_index = 1$

$S(n) = n + n + n + n + 1 + 1 + 1 + 1 = 4n + 4 = O(n)$

$S(n) = O(n)$

Figure 4. Selection Sort Theoretical Memory Usage Analysis

Moving on, the space complexity of the original Quick Sort Algorithm is said to be dependent on the size of the recursion call stack, which is equal to the height of the recursion tree (Gautam, 2022). So, when the quick sort is in its worst-case scenario then its space complexity is equivalent to $O(n)$. But, when the algorithm is in its best-case scenario then the space complexity is equivalent to $O(logn)$. Now, if we take a look at the modified algorithm and check its variables which include SuffixIndices[], pivot_string, variables left, right, etc. Then, using all of these variables we can compute the space function of the algorithm which is equivalent to $S(n) = 4n + 5$ and this means it has a space complexity of $O(n)$ (See Figure 5).

Memory Requirement:

$mainString = n$

$SuffixIndices = n$

$n = 1$

$left = 1$

$right = 1$

$i = 1$

$j = 1$

$pivot\_string = n$

$curr\_string = n$

$S(n) = n + n + n + n + 1 + 1 + 1 + 1 + 1 = 4n + 5 = O(n)$

$S(n) = O(n)$

Figure 5. Quick Sort Theoretical Memory Usage Analysis

IV. Empirical running time analysis of the algorithms

To create an empirical running time analysis of the chosen algorithms, the algorithms were implemented using the C programming language. The algorithms were tested in different values of n, ranging from 128 to 65,536. By using the clock() function provided by the time.h library, the actual time taken by the algorithms was calculated by dividing the number of clock ticks that elapsed since the first call of clock() with the defined number of clock ticks per second of the machine CLOCKS_PER_SEC.

As seen in the table below, for the modified Selection Sort Algorithm, the average running time is generally lower when n is around 128 to 4096. However, as n starts to increase at 8192 the average running time also greatly increases as it runs for 2.23 seconds on average. This was also observed for higher values of n whereas the average running time increased significantly at values of n starting from 16,384 to 65,536 (See Table 1).

Selection Sort Execution Time (s)					
# of Elements (n)	RUN #1	RUN #2	RUN #3	RUN #4	AVERAGE
128	0.000171	0.000257	0.000170	0.000166	0.000191
256	0.000451	0.000468	0.000460	0.000510	0.000472
512	0.001855	0.001917	0.002381	0.002933	0.002272
1024	0.009581	0.009370	0.009975	0.009329	0.009564
2048	0.056411	0.059285	0.050971	0.053042	0.054927
4096	0.319065	0.354688	0.321733	0.329811	0.331324
8192	2.161796	2.279300	2.169780	2.324659	2.233884
16384	16.461325	16.172714	15.418023	16.269199	16.080315
32768	187.418082	192.189294	180.160051	182.819803	185.646807
65536	1599.256676	1540.836529	1628.456584	1609.378176	1594.481991

Table 1. Execution Time Results of Selection Sort

For the table of the modified Quick Sort Algorithm, it can be seen that the average running time is steadily increasing as n also increases. It also observed that when n = 128 to 32,768 the run trials are within the range of 0.000091 to 0.232244 or generally in the zeros. However, as n increases to 65,536 then it starts to reach an average time of 1.055137 seconds (See Table 2).

Quick Sort Execution Time (s)					
# of Elements (n)	RUN #1	RUN #2	RUN #3	RUN #4	AVERAGE
128	0.000069	0.000089	0.000145	0.000087	0.000098
256	0.000120	0.000116	0.000113	0.000107	0.000114
512	0.000239	0.000201	0.000190	0.000201	0.000208
1024	0.000452	0.000671	0.000433	0.000455	0.000503
2048	0.001059	0.001112	0.001090	0.001162	0.001106
4096	0.003225	0.003398	0.003510	0.003028	0.003290
8192	0.011129	0.012087	0.011719	0.011111	0.011512
16384	0.039248	0.047538	0.046187	0.046938	0.044978
32768	0.229482	0.256294	0.222826	0.220376	0.232244
65536	0.993032	0.964439	1.128170	1.134905	1.055137

Table 2. Empirical Running Time Analysis of Quick Sort

In the graph below, the two modified algorithms of Selection Sort and Quick Sort were compared based on the running time of the programs. It can be seen that the Selection Sort has a significantly higher running time compared to Quick Sort as n increases. This is seen when n is above 4096 because the Selection Sort Algorithm starts to take 1+ seconds while the Quick Sort Algorithm is still able to run the program below 1 second or in zeros. It also observed that the Quick Sort algorithm only starts to take 1+ seconds when n is around 65,536 whilst compared to the Selection Sort, it has gone up to 1000+ seconds (See Figure 6).

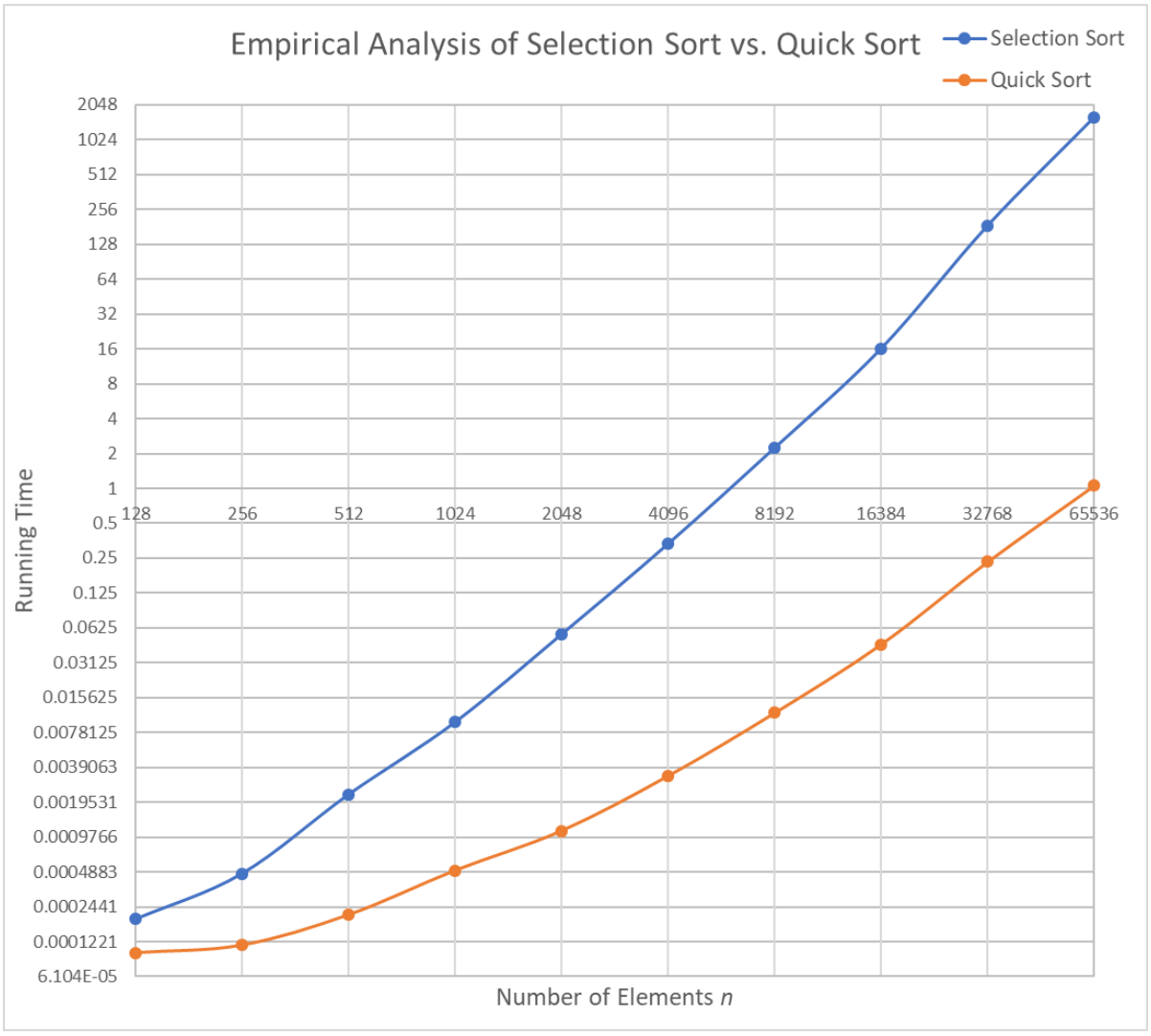


Figure 6. Running Time of Selection Sort & Quick Sort with respect to n in a log-log plot

V. References

Chauhan, Y., & Duggal, A. (2020). Different sorting algorithms comparison based upon the time complexity. Int. J. Res. Anal. Rev., 7(3), 114-121.

Gautam, S. (2022, October 13). *Quick Sort Algorithm*. EnjoyAlgorithms. https://www.enjoyalgorithms.com/blog/quick-sort-algorithm

How to measure time taken by a function in C? (2022, June 21). Geeksforgeeks. https://www.geeksforgeeks.org/how-to-measure-time-taken-by-a-program-in-c

Selection Sort Algorithm. (n.d.). JavaTpoint. https://www.javatpoint.com/selection-sort

Trivedi, A. (2022, May 4). *Space Complexity in Data Structure*. ScalerTopics. https://www.scaler.com/topics/data-structures/space-complexity-in-data-structure/

Xiang, W. (2011). Analysis of the Time Complexity of Quick Sort Algorithm. 2011 International Conference on Information Management, Innovation Management and Industrial Engineering. doi:10.1109/iciii.2011.104

VI. Work Distribution

The work distribution for MCO1 follows as such:

- Aaron Gabrielle C. Dichoso - Algorithm & Program Design
- Josh Austin Natividad - Algorithm Design, Empirical Time Analysis
- Luis Miguel Antonio B. Razon - Algorithm Design, Theoretical Time Analysis
- Adriel Joseph Donato - Theoretical Time, Theoretical Memory Analysis