2018

CMP1124M: Algorithms and Complexity

ASSESSMENT 02 - REPORT

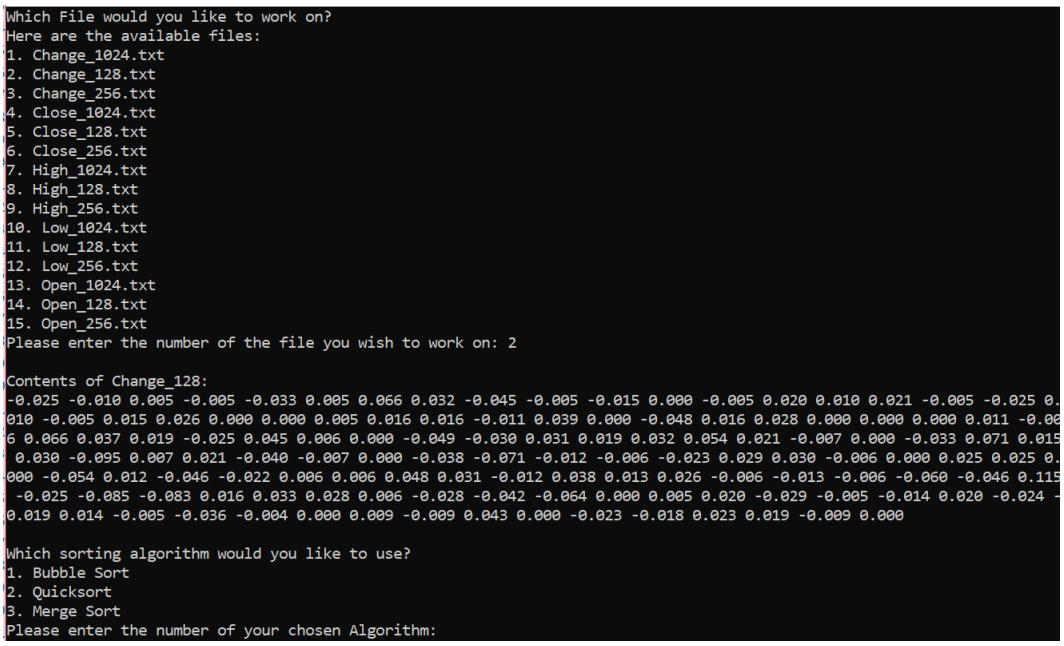
GABRIELLA DI GREGORIO 15624188

UNIVERSITY OF LINCOLN | School of Computer Science

My Application

Demo Video: https://youtu.be/kY8kMTd1Lis

The first feature of my application is the display of all files in the file directory which the user can select from. The program accesses the file directory and finds all the text files inside. Each text file is assigned to an array and printed as a numbered list to the user. Then, the user can simply type the number of the file that they would like to open and work on. Once a file has been selected, the contents of the file is assigned to a string array which is then converted to a float array since the contents of all files is a list of numbers to 3 decimal places, including negative numbers. Because of the negatives and decimal places, using integers would have been an invalid option – I chose to use floats because they allow for an appropriate level of precision, and decimals or doubles are unnecessarily large for this task. So, once this array has been created, the user is then presented with the unordered contents of the file and is then asked which algorithm they would like to use to sort the data.



The user can choose between Bubble Sort, Quicksort and Merge Sort – sorting algorithms will be analysed in detail in the next section. Once an algorithm has been selected, the user is then asked whether they would like their data to be sorted in ascending or descending order. Due to the simple nature of Bubble Sort, the same method runs despite this selection because the difference is dealt with inside the method, but for more complex algorithms such as Quicksort, different methods were needed to deal with the order that is selected. Unfortunately, due to time constraints I was not able to use Merge Sort for descending order however I predict that this would be a similar process as used in the Quicksort method. Once the algorithm has run and the data has been sorted, the user is then presented with their sorted data and they are asked to select which algorithm they would like to use to search for a number.

```
Which sorting algorithm would you like to use?
1. Bubble Sort
2. Quicksort
3. Merge Sort
Please enter the number of your chosen Algorithm: 1

Would you like to sort in ascending or descending order?
Enter 1 for ascending, or 2 for descending
Order: 2

This has taken 8128 steps...

Your file has been sorted:
0.115 0.071 0.066 0.066 0.054 0.048 0.045 0.043 0.039 0.038 0.037 0.033 0.032 0.032 0.031 0.031 0.030 0.030 0.029 0.028
0.028 0.026 0.026 0.025 0.025 0.023 0.021 0.021 0.021 0.020 0.020 0.020 0.019 0.019 0.019 0.016 0.016 0.016 0.016 0.015
0.015 0.014 0.013 0.012 0.011 0.010 0.010 0.009 0.007 0.006 0.006 0.006 0.006 0.005 0.005 0.005 0.005 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 -0.004 -0.005 -0.005 -0.005 -0.005 -0.005
-0.005 -0.005 -0.006 -0.006 -0.006 -0.006 -0.006 -0.007 -0.007 -0.009 -0.009 -0.010 -0.011 -0.012 -0.012 -0.013 -0.014 -
0.015 -0.018 -0.019 -0.022 -0.023 -0.023 -0.024 -0.025 -0.025 -0.025 -0.025 -0.028 -0.029 -0.030 -0.033 -0.033 -0.036 -0
.038 -0.040 -0.042 -0.045 -0.046 -0.046 -0.048 -0.049 -0.054 -0.060 -0.064 -0.071 -0.083 -0.085 -0.095

Which searching algorithm would you like to use?
1. Linear Search
2. Binary Search
Please enter the number of your chosen Algorithm:
```

The user can choose between Linear Search and Binary search to find a specific number of their choice. Again, these searching algorithms will also be looked at in the next section. Once the algorithm has been selected, the user is then asked which number they would like to search for. If the number has been found, both methods return all the positions in which the search item has been found. However, if the search item is not present, using Linear Search will look for the nearest value instead and display this number along with its position will be displayed to the user. Unfortunately, I was not able to use Binary Search to complete this process so if a Binary Search is used, an error message will be displayed to inform the user that their chosen number has not been found. It seemed rather difficult and time consuming to repeat this method with Binary Search without changing the time complexity of the algorithm. The program ends once the search has been completed however with more time, I would have made the program more flexible such as the user being able to search for a different number or restart the program.

```
Your file has been sorted:
0.115 0.071 0.066 0.066 0.054 0.048 0.045 0.043 0.039 0.038 0.037 0.033 0.032 0.032 0.031 0.031 0.030 0.030 0.029 0.028
 0.028 0.026 0.026 0.025 0.025 0.023 0.021 0.021 0.021 0.020 0.020 0.020 0.019 0.019 0.019 0.016 0.016 0.016 0.016 0.01
5 0.015 0.014 0.013 0.012 0.011 0.010 0.010 0.009 0.007 0.006 0.006 0.006 0.006 0.005 0.005 0.005 0.005 0.000 0.000 0.0
00 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 -0.004 -0.005 -0.005 -0.005 -0.005 -0.
005 -0.005 -0.005 -0.006 -0.006 -0.006 -0.006 -0.006 -0.007 -0.007 -0.009 -0.009 -0.010 -0.011 -0.012 -0.012 -0.013 -0.
014 -0.015 -0.018 -0.019 -0.022 -0.023 -0.023 -0.024 -0.025 -0.025 -0.025 -0.025 -0.028 -0.029 -0.030 -0.033 -0.033 -0.
036 -0.038 -0.040 -0.042 -0.045 -0.046 -0.046 -0.048 -0.049 -0.054 -0.060 -0.064 -0.071 -0.083 -0.085 -0.095

Which searching algorithm would you like to use?
1. Linear Search
2. Binary Search
Please enter the number of your chosen Algorithm: 2

Which item would you like to search for?: 0
0 has been found at position(s): 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
```

```
Your file has been sorted:
0.115 0.071 0.066 0.066 0.054 0.048 0.045 0.043 0.039 0.038 0.037 0.033 0.032 0.032 0.031 0.031 0.030 0.030 0.029 0.028
0.028 0.026 0.026 0.025 0.025 0.023 0.021 0.021 0.021 0.020 0.020 0.020 0.019 0.019 0.019 0.016 0.016 0.016 0.016 0.015
0.015 0.014 0.013 0.012 0.011 0.010 0.010 0.009 0.007 0.006 0.006 0.006 0.006 0.005 0.005 0.005 0.005 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 -0.004 -0.005 -0.005 -0.005 -0.005 -0.005
-0.005 -0.005 -0.006 -0.006 -0.006 -0.006 -0.006 -0.007 -0.007 -0.009 -0.009 -0.010 -0.011 -0.012 -0.012 -0.013 -0.014 -
0.015 -0.018 -0.019 -0.022 -0.023 -0.023 -0.024 -0.025 -0.025 -0.025 -0.025 -0.028 -0.029 -0.030 -0.033 -0.033 -0.036 -0
.038 -0.040 -0.042 -0.045 -0.046 -0.046 -0.048 -0.049 -0.054 -0.060 -0.064 -0.071 -0.083 -0.085 -0.095

Which searching algorithm would you like to use?
1. Linear Search
2. Binary Search
Please enter the number of your chosen Algorithm: 1

Which item would you like to search for?: 0.018
The number has not been found. The nearest number is 0.019 at position 34
```

Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

*^ Figure 1**

Before starting my program, I researched to remind myself of the complexities of the different algorithms. I wanted my program to use a variety of sorting algorithms ranging from poor time complexities to efficient time complexities so that I could make comparisons. The table above shows many different options with complexities as good as O(n+k) and as poor as O(n(log(n))^2). My choices were slightly limited since I wanted to mainly use the algorithms that had been covered in the lectures since my previous knowledge would help with my understanding of implementation. However, exploring all of my options was a great way to expand my knowledge of the subject and provided me with some interesting possibilities.

Bubble Sort:

For my 'poor complexity' example, I decided to use Bubble Sort. Although there are several other algorithms with the same worst case time complexity of O(n^2), the implementation of this algorithm was very simple and I had previous knowledge which helped me develop my code. Despite having a poor worst case, Bubble Sort has a good best case of Ω(n) and a good worst case space complexity of O(1) which means it is memory-efficient since it does not require additional space for temporary storage. Furthermore, since the largest data size it needed to sort was 1024, it is unlikely that a worst case scenario would arise. The Bubble Sort is also very appropriate for small data sizes such as the 128 files since it would not take much time to sort such a small number of items. The Bubble Sort works by going through an array and comparing each pair of items that are next to each other, if it is sorting in ascending order, if the item before is greater than the item after, they are swapped. This process repeats until the array is fully sorted which is why the time complexity drastically increases the larger the array. I believe selecting this algorithm gave me an interesting

method to analyse a slow example, however if my main concern was to make the program more efficient, then this would not have been a good choice.

	Size 128	Size 256	Size 1024
Worst Case: O(n^2)	16,384	65,536	1,048,576
Average Case: Θ(n^2)	16,384	65,536	1,048,576
Best Case: Ω(n)	128	256	1024
Space Complexity: O(1)	1	1	1

Quicksort:

Although Quicksort has the same worst case time complexity as Bubble Sort of O(n^2), it sacrifices some memory efficiency (worst case space complexity of O(log(n))) for a much better average case time complexity of Θ(n log(n)). I also had previous knowledge of this algorithm so these two factors led me to choose Quicksort to make an interesting comparison. As the tables show, the average case for Quicksort is drastically more efficient than for Bubble Sort, and the space complexity is only slightly worse so is still rather memory-efficient. However, the best case for quicksort is noticeably worse than Bubble Sort which may mean that this is less appropriate for very small data sizes. Quicksort works by selecting a pivot point within the array then comparing the first numbers in the array (working forwards) with the last numbers in the array (working backwards). For ascending order, if a number near the start of the array is larger than the pivot, it will be swapped with a number near the end of the array which is smaller than the pivot. The side smaller than the pivot and the side larger than the pivot is split and dealt with separately, repeating the process in each half with a new pivot. This continues until no swaps occur and the array is fully sorted. Although this algorithm may be more appropriate for 'average-sized' data, it is inefficient for both small and large data sizes so again, this would not be the best choice for making a program as efficient as possible.

	Size 128	Size 256	Size 1024
Worst Case: O(n^2)	16,384	65,536	1,048,576
Average Case: Θ(n log(n))	269.72	616.51	3082.55
Best Case: Ω(n log(n))	269.72	616.51	3082.55
Space Complexity: O(log(n))	2.11	2.41	3.01

Merge Sort:

Lastly, I chose to use Merge Sort since it has the best worst case time complexity of O(n log(n)) of all the algorithms that I already had an understanding of. Since the worst, average, and best case of the Merge Sort are all the same, the efficiency of this algorithm is reliable and consistent. By using more space and therefore being less memory efficient (space complexity of O(n)), the worst case for this algorithm is much better than both Quicksort and Bubble Sort. For this reason, I decided it would be interesting to compare it. Like Quicksort, this is another 'divide and conquer' method. The array is continuously split until all items are individuals, then each adjacent item is compared and stored in a temporary array in the correct order. This pair is then compared with the next pair and sorted into a

new array of size 4 in order. This process continues until all items have been sorted and the array is merged back together. Although this algorithm has a better time complexity for the worst case of very large or unusual data sets, Quicksort is usually preferred. This is because Merge Sort requires a large amount of secondary storage and Quicksort moves items more quickly into their correct positions.

	Size 128	Size 256	Size 1024
Worst Case: O(n log(n))	269.72	616.51	3082.55
Average Case: Θ(n log(n))	269.72	616.51	3082.55
Best Case: Ω(n log(n))	269.72	616.51	3082.55
Space Complexity: O(n)	128	256	1024

Radix Sort:

If time was not a constraint, I would have used a fourth sorting algorithm and for this I would have considered the Radix Sort because although I am not familiar with this algorithm, it seems to have the one of the best time complexities. Given an ideal situation, the Counting Sort has the best time complexity however it only works with integers so for this task, a Radix Sort would be appropriate. Radix Sort works by doing a digit by digit sort starting from the least significant digit to the most significant and it uses the counting sort as a subroutine.*[1]* I believe this would have been interesting to compare because although in theory it is one of the most efficient algorithms possible, the complexity of O(nk) is dependent on the 'k'. The k is sometimes represented as a constant, which would make radix sort better for a sufficiently large n. However, it often cannot be considered a constant since all n keys are distinct then k would be at least log(n).*[2]* For this reason, I believe it would be interesting to compare whether this sort would have actually been more efficient than comparison-based sorts such as the Quicksort in this situation.

Searching Algorithms

Linear Search:

I decided to use a Linear Search because I already had knowledge of this simple algorithm and I knew it would make an interesting 'poor example' to compare to a more efficient searching algorithm. It works by starting at the beginning of an array and working along one by one to the end of the array, comparing each element to the search item. If a match is found, the index is returned. Although this method is rarely used due to its inefficiency compared to other algorithms it may be well suited to this task since the data sizes are relatively small so should not take long to search. Furthermore, due to its simplicity, I was able to implement a search for the nearest value if the search item is not present. This is a better interaction than simply displaying an error message however if the closest value is far away from the original search then this may take some time to complete. Nevertheless, this provided a useful feature of the application with relatively short development time which would be much harder and more time-consuming to accomplish with more complex algorithms.

	Size 128	Size 256	Size 1024
Worst Case: O(n)	128	256	1024
Average Case: Θ(n)	128	256	1024
Best Case: Ω(1)	1	1	1
Space Complexity: O(1)	1	1	1

Binary Search:

As the tables above and below show, this algorithm is far more efficient than the Linear Search. O(log(n)) results in very short running times and only slightly increases as the data size largely increases. This makes the Binary Search preferred over the Linear Search, however the code is much more complex. It works by repeatedly dividing the array until the item has been found or the entire array has been searched. It begins by assessing the middle number of the sorted array and comparing it to the search item, if it is a match then the process ends, if it is larger than the lower half is split and searched, or the upper half if the middle is smaller. This method required more complex code to search for duplicates and return all the found positions than the Linear Search and was more time-consuming to complete. As well as this, I found it too difficult and time consuming to use the Binary Search method to find the closest value if it had not been found so Linear Search may be more appropriate for that part of the task. Despite this, the Binary Search is an undoubtedly more efficient method and results in a much faster program runtime.

	Size 128	Size 256	Size 1024
Worst Case: O(log(n))	2.11	2.41	3.01
Average Case: Θ(log(n))	2.11	2.41	3.01
Best Case: Ω(1)	1	1	1
Space Complexity: O(1)	1	1	1

Conclusion

In conclusion, I believe I selected a good range of different algorithms to analyse different complexities. From this experience, if I were to choose one sorting and one searching algorithm to make the program as efficient as possible, I would select the Quicksort and Binary Search since they are more efficient and more suited to this task than other options. Moreover, my prior knowledge of these algorithms helped me to understand the implementation rather than struggling to implement an algorithm I was unfamiliar with. Unfortunately, I believe my programming skill was a bottleneck in this task since a large portion of my extensive amount of time invested in this project had to be spent resolving simple programming issues.

References

Figure 1: http://bigocheatsheet.com/

1.Radix Sort, Geeks for Geeks: https://www.geeksforgeeks.org/radix-sort/

2. Radix Sort, Wikipedia: https://en.wikipedia.org/wiki/Radix_sort