Algorithms and Complexity Assessment 2 Report

My system is designed with the purpose of being able to sort and search through data provided by the Met Office. I have achieved this with an object orientated approach, which requests multiple inputs from the user based on what they may or may not want to do (e.g. do a search, sort in ascending or descending order, etc.). Functions are then called based on that choice, in order to sort through data and allow it to be searched, or to do anything else that may be required. Data is outputted to the console, in a clear, formatted way, and remains legible and consistent, even when there are hundreds or thousands of lines being outputted. Each task starts with the user being asked what to do, and selecting one of multiple given options (by typing 1, 2, 3 etc.). I used lists for the data instead of arrays, as I feel that it makes the program more effective in the future, as the text files could be added to, or have values taken away, without any change being needed to the code, whereas since using arrays requires a length to be declared with the array, this would be much more complicated to implement. The text files that are required are loaded into the lists from the start, instead of loading them when needed, as this allows it to be done while the system is waiting for a user input in the menu.

Tasks 3 and 4 were then implemented by sorting the chosen data source (months or years) using a quick sort, and then running a Binary Search on the sorted data (it has to be sorted for a Binary Search to work) in order to find the value that was entered by the user (e.g. 1940 for years, or 2 for February). If the value doesn't exist, an error is given that returns the user back to the main menu. However, if it does exist, the system will then return every occurrence of that value, and display all of its associated values (e.g. the max temp). This is done differently with months and years, as with months, I used the knowledge that once I have found one instance of it, I can then add on 12 to get every instance of that month. With years, the data was already in the correct order compared to the other data sources, so I just used the index I found for the instances of the years as the index of each other corresponding value. The quick sort was implemented using a recursive function, which checks values against a 'pivot' (the center value), in order to determine whether the values before it need moving to the other side of the pivot, or vice versa. This is repeated until every value has been checked, and made sure they are in the right order. The Binary Search is also recursive, and this works by finding the middle value, and if it is too small/big compared to the value being searched for, it will remove the first/last half of the data set, and runs again until it either finds the value, or decides that it doesn't exist. The added complication for my implementation is that there could be multiple instances of the value being search for, so I have a small algorithm inside my Binary Search that cycles forwards and backwards once it finds what it is looking for, and adds every other instance as well.

Task 5 was achieved by using an array with each of the months in their chronological order, and referring to them using this instead of listing them alphabetically.

Task 6 and 7 were implemented using a function called Linear Search, as my implementation of the program forced me to use a less than optimal method of finding the values. For example, to find the maximum values, I was using a quick sort to organize my data into order, then finding the last value (and moving backwards to find all other instances of the max value), and using these as my values. However, to find the corresponding values of the maximum instances, I couldn't do a Binary Search on the unsorted list, as that would require it to be ordered. So, I had to use a searching algorithm that instead cycled through every value in the list and retrieved all of the indexes, in their unsorted form, and then used those to display all of the corresponding values to each instance of the max value. For the minimum, I only changed this up by using the first values in the sorted data instead of the last. And for the median, I decided to use the middle value, and then display every instance of that appearing (to use the true middle value wasn't possible as the data set had an even amount of numbers).

Task 8 was implemented throughout, by simply ensuring every bit of data is outputted alongside whatever was being sorted, in the correct order depending on its actual meaning (the right values match up).

Task 9 was implemented by creating more lists and ensuring they were used throughout the entire program.

In my application, I used a Quick Sort to do most of my sorting. I used this algorithm, as it has an expected time efficiency of O(nlogn), and memory efficiency of O(1). This means that with my relatively large amount of data, this is arguably the most effective algorithm I could have used, since O(nlogn) tells me that this algorithm is far more efficient as a data size grows than the other sorting algorithms, which tend to have an expected time of $O(n^2)$ (other than the Merge Sort). This is also more efficient than the Merge Sort, as that has a memory efficiency of O(n), since more data needs to be stored at once (each branch of the tree will need returning to). However, the Quick Sort does have a worst case time of $O(n^2)$, meaning there is the chance that it performs as badly as sorts like the Bubble Sort and the Insertion Sort, which would make the Merge Sort the better alternative, seeing as that has a worst time of O(nlogn).

I also gave the user a choice of whether to use a Quick Sort or Bubble Sort when displaying data in ascending order, to show the implementation of both sorting algorithms. Implementing both shows how much simpler the bubble sort is, which explains why it has such a poor time efficiency. In a usual full release program, the sorting algorithm used would be hidden from the user; however I gave the option of the two in this to show a larger range of potential algorithms.

Another point to consider is that, although there is a fairly large amount of values being worked with in this project, there aren't enough to see any difference in the speed of the operation. If I had used a Bubble Sort instead of a Quick Sort, there would be no visible difference in the time taken to run the program. The choice in sorting algorithm would be more impactful if the data set were in the millions of values, as then, the Bubble Sort would take a much longer amount of time to run than the Quick Sort (unless it was the worst case for the Quick Sort).

For searching in my program, I had to use a combination of a Linear Search and a Binary Search, as they both served different purposes; the Linear Search was used when I couldn't guarantee that the data would be ordered, and the Binary Search would be used when I could. The reason that I used a Binary Search as much as possible was because the Linear Search has a time efficiency of O(n) operations, whereas the Binary Search is O(log2n), which is significantly faster. I could have used an Interpolation Search, which is technically the most efficient searching algorithm at a time complexity of O(log(log(N)), but this is more suited to much larger data sets, due to its increased complexity. My Binary Search is a slightly less efficient algorithm than it should be, due to the fact that it has to return a set of values, instead of a single one. Although this wouldn't change the time complexity, it does slightly decrease the performance, however since this is a requirement of the program, it is a necessary decrease.

To show how fast the Binary Search is, Skiena (1998) stated that using a printed dictionary, using the Binary Search algorithm could find any word in a printed dictionary within a maximum of 20 runs, highlighting how efficient this algorithm is. More importantly, when a user runs my program, the time taken for the sort and search to be processed isn't even noticeable, which is impressive to say there could be 1000 values being sorted then searched at a time.

Both my Quick Sort and my Binary Search are recursive algorithms, which doesn't necessarily affect the performance, however this has made the algorithms far simpler to understand at first glance, and made it easier to go back to my algorithms and make any changes where necessary, since a recursion algorithms should look much cleaner and well laid out than an iterative algorithm, as it is just more intuitive.

My overall system is effective at meeting the listed requirements, however I feel like there others options I could have taken in building the system, that may have been more optimal than my current implementation. For example, I haven't grouped my data together in any way, and have instead opted to use different methods to use the one value I found to find everything corresponding to it. One way I could have done this is to use a class to keep all the data together, then refer to the class as an object for each instance of the data, which would make it easier to refer to relevant data. This method wouldn't necessarily improve the performance of my system, but would have made it more well laid out, which helps for future reference if I go back to the system, maintain it, or if someone else were to work on it.

The least efficient part of my system is the use of the Linear Search, which could slow part of my system down if larger amounts of data were used, whereas the Binary Search and Quick Sort could hold up to much larger data sources, as they have more efficient time complexities.

The link for my video: https://www.youtube.com/watch?v=7hkkZPC45VY&feature=youtu.be

References

Skiena, Steven S. *The Algorithm Design Manual*. Santa Clara, CA: TELOS--the Electronic Library of Science, 1998.