Algorithm Design

1 MERGE & SORT ALGORITHM

This algorithm is designed to merge and sort by surname, three lists of students, who made up different categories of student entry path. For this algorithm I used three '.txt' files which each contained records in the following format: *ID Type Surname Firstname*

To solve this problem I tried out a number of solutions:

1) The simplest idea I thought about was to sort the 3 files, merge them and sort them again. This seemed like a redundant exercise, as although the files were sorted before merging, I would still have to sort the files after merging, resulting in 4 sorts of the data, and a potential Big O of n^8.

2) Another solution I thought about involved sorting each file individually and then merging the three files together in a new merged file using the Merge algorithm from Merge Sort i.e. comparing the first element in each file and writing the largest/smallest of these to the new file, then the second and third largest/smallest. Then looping back to compare the new top elements in the files. Although this involved sorting the files separately, it seemed very streamlined as the Merge part made the separate sorting no longer redundant.

   This was my favourite solution as I felt it was the simplest and made use of the very powerful idea of divide and conquer. For instance, I am solving smaller problems (sorting the three files separately), then bringing them together as smaller solutions to the bigger problem (merging the three files by comparing the top element in each file).

   The Big O of this solution could potentially have been very large if I had only used a single sorting algorithm like Insertion Sort or Bubble Sort, who have a Big O of n^2. This sort would have occurred 3 times and the merge just once.
   This would have resulted in a Big O of n^7 - $\{n^2 * n^2 * n^2 * n\}$

   If however, I had used the Merge Sort algorithm to sort the three separate files, the Big O would have been much smaller where Merge Sort occurs 3 times in sorting the files separately and is nlog(n) and the Merge algorithm occurs 1 more time in combining the files and is (n).

   However in terms of writing this algorithm in C-code, it became difficult to implement. I could not sort easily on disk, so turned to creating an array in RAM for implementing. At that point it seemed like a lot of waste to be sorting the files 4 times, when I could simply merge them first and then sort the files as one file, requiring only 1 sort. In the end I decided to go with a shorter solution similar to this.

   The final Big O in this scenario is therefore: $4nlog(n^3)$ i.e. { nlog(n) * nlog(n) * nlog(n) * n }

At this point I decided that it would be a good idea if my final implemented algorithm was to focus on merging the files into one file and then sorting them, to keep the Big O reasonably under control.

3) The solution I chose to implement in the end involved reading each of the 3 unsorted files one at a time and writing them, one after the other, into a new file. I would then perform my searching algorithm on this new file.

   Once I had merged the files, I wanted to utilize the power of Merge Sort to divide up my problem, which I was conscious was now a much larger, still unsorted, file than the 3 small files I had started with. So I decided to use Merge Sort to divide up my big file, but not wanting to overload the stack with recursive calls to Merge Sort, I decided that I would amend my base case and add another simple sort to the Merge Sort algorithm.

   So I changed my base case, from an array of 1 size, to an array of 3 elements. At this stage in runtime, the base case reached, the 3-element array would be passed to a Bubble Sort algorithm where it would be sorted quickly and returned to the function call in Merge Sort. Conscious of the fact that this might seem like a wasteful exit from the currently running function of Merge Sort, I implemented a modification to the Bubble Sort, which entailed a sorting flag, to tell the program when Bubble Sort had reached an optimally sorted 3-element array.

   Returning the small sorted sub-array back to Merge Sort, meant that Merge Sort did not have to make as many recursive calls on the stack and the whole sorting is faster and uses less memory.

   Unfortunately these changes did not have a huge effect on the Big O, which always assesses the worst case scenario. To calculate this Big O, we take the Merge Sort which occurs only once and couple it with a Modified Bubble Sort. Bubble Sort itself is usually n^2, and sadly, even using a flag, the worst case scenario is still n^2, but in reality during runtime the flag will more times than not, speed up sorting efficiency.

   The final Big O in this scenario is therefore: $n^3\log(n)$ i.e. $\{n\log(n) * n^2\}$

IMPLEMENTED ALGORITHM

Merge_Sort (A, low, high):

```
If n < 4                          // if the array contains 3 elements
    mod_bubble_sort(A)            // sort the sub-array using mod_bubble_sort
    Return
Else
    mid = (low+high)/2            // split the array into 2 sub-arrays
    Merge_Sort(A, low, mid)       // continue to break down the left side of the array into sub_arrays
    Merge_Sort(A, mid+1, high)    // continue to break down right side of array into sub_arrays
```

```
    Combine(A, low, mid, high)          // bring all the sorted sub-arrays back into the array
END Merge_Sort
```

Modified_Bubble_Sort (A[(records)], N)

```
for i = 0 to i < N and flag do
    flag = 0
    for j = 0 to j < N-1 do
        if A[j].surname > A[j+1].surname                // using strcmp to compare the names
            temp.surname = A[j].surname                 // swap the whole record, not just surname
            A[j].surname = A[j+1].surname
            A[j+1] = temp
            flag = 1
        End if
    End for
End for
END Modified_Bubble_Sort()
```

Combine(A, low, mid, high)

```
leftindex=low                                   // starting head of the left array
rightindex=mid+1                                // starting head of the right array
combinedindex=left                              // starting index of the temp array

// while  not at the end of a sub-array
while leftindex<=mid AND rightindex<=high
// if the head of one array is bigger than the other, place biggest in the temp array
    if A[leftindex]<=A[rightindex]              // using strcmp to compare the names
        tempA[combinedIndex++]=A[leftindex++]   // save left array record into temp
    else
        tempA[combinedIndex++]=A[rightindex++]  // save left array record into temp
```

```
// If end of one subarray reached copy the rest of the other subarray into the temp array
    if leftindex=mid+1                          // if the end of the left array is reached
        while rightindex<=high                  // keep going until the end of the right array
            tempA[combinedIndex++]=A[rightindex++]   // put the right array elements into temp array
    else                                        // otherwise do the right side, similarly
        while leftindex<=high
            tempA[combinedIndex++]=A[leftindex++]

// return the array contents to the original array, copying from the temp array
for i=low to i<=high do
      A[i] = tempA[i]
END Combine()
```

Merge Sort **Big O = log (n)**

This algorithm takes the length of the array to find the mid element in the array. It then recursively splits the array in half, until the number of elements in the array is 3. At this point the smaller array is passed to the Modified Bubble Sort for sorting. Once the array has been split into multiples of 3 and sorted, it is then passed to Combine for merging the small arrays back into one array, now in a sorted order.

Modified Bubble Sort **Big O = n^2**

This algorithm compares two neighbouring elements in a list and swaps them to put them in ascending order. It continues repeating this step until it reaches the end of the list, and then goes back to the start and repeats, looking for any items that are out of order. The algorithm is modified by using a flag to notify the program when no swaps have been made in a single loop of the array. This indicates that all elements are now correctly sorted and the program can stop looking. The array passes back to Merge Sort.

Combine **Big O = n**

This algorithm compares the heads of two arrays that have been split by Merge Sort and sorted by Modified Bubble Sort, and brings them back into a temporary array, in order. It then places the contents of the temporary array, back into the original array. Here the algorithm ends.

The Big O in this scenario is therefore: $n^3 log(n)$ i.e. {log(n) * n^2 * n}

2 CATEGORY SEARCH ALGORITHM

This algorithm is designed to search for all students in the category of 'International'.

For this algorithm, I used a linear search as I was searching not just for one record, but multiple records with the same field. Therefore, I had to treat the data as if it was unsorted, in that the desired records would be scattered around the file.

A linear search works by starting at the beginning, reading one record or object, comparing it to the search parameters and returning if found or not found. In this algorithm, because we are looking for potentially several records of the same group, I have put this linear search inside a loop, reading each record, checking it and then reading the next record until we get to the end of the file. In the case of my algorithm, I have included that the records that are found matching the search parameters, are both displayed on screen in a readable list format, and also written to a file for the user to save if needed for future reference.

An alternative to a linear search would have been to write a sorting part to this algorithm which would have sorted the records by group. However when weighing up the complexity of a sort-search and a plain linear search of 36 records, I decided that a linear search would be sufficient. A larger problem set would probably benefit from a sorting part.

In the absence of adding complexity with a sorting part to this algorithm, I increased the usefulness of this program at implementation by providing functionality to allow the user to input the desired group through the keyboard, thereby increasing the flexibility of a programme that was originally designed to search for a single predetermined group.

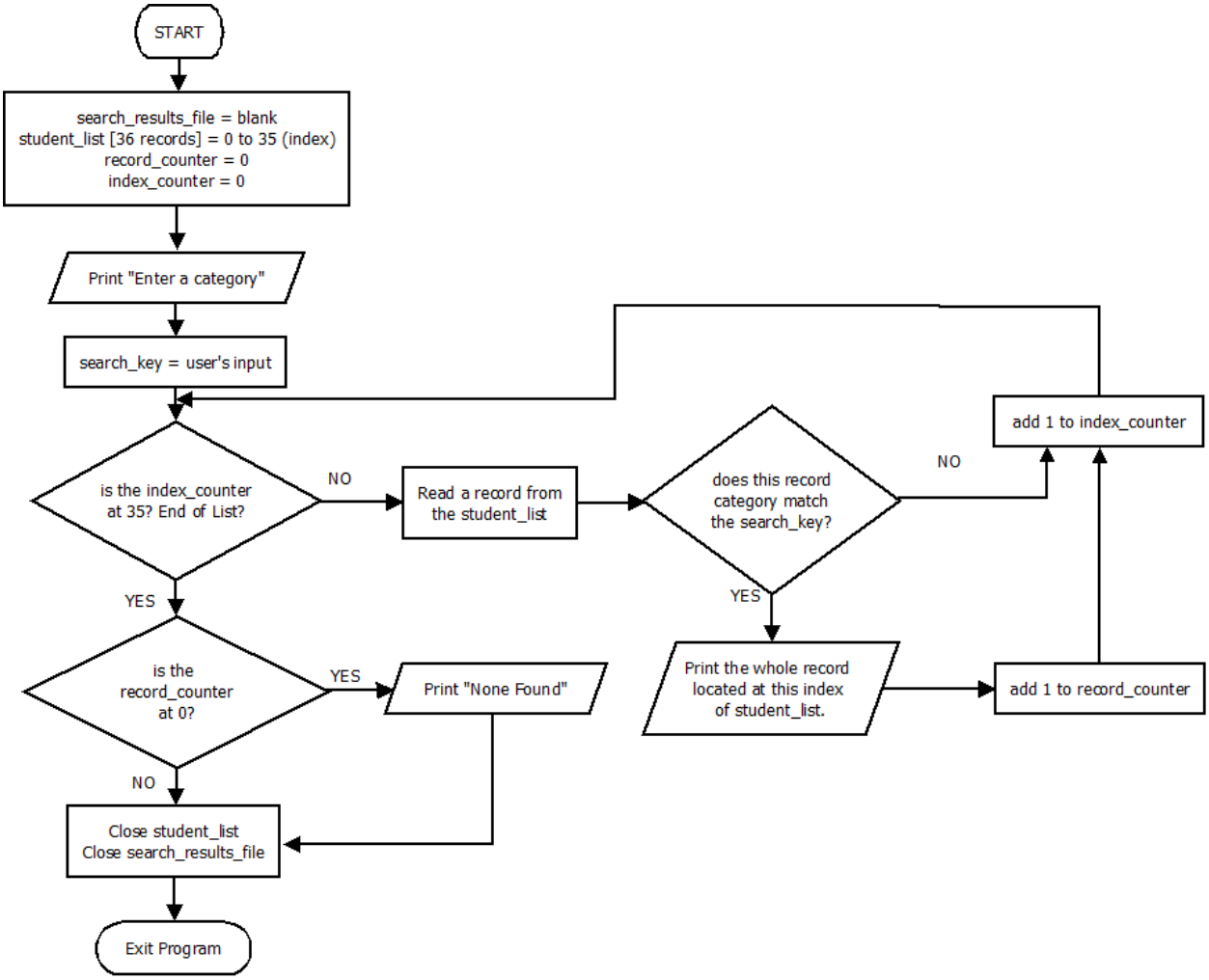
In terms of calculating the Big O, in this algorithm there is only one loop through the file to search for the desired search key. The search is linear and so visits each record only once. When the search key is found it will return that immediately. If the key is not found, the algorithm will have visited every record in the file once (worst case scenario) and return a 'not found' message.

The Big O of this algorithm is therefore: n

FLOWCHART

The following flowchart is written in pseudo-english and is meant as a visual description of the implementation of this algorithm, so that a non-IT client might be able to better understand the process.

PART 2 – SEARCH FOR SPECIFIC CATEGORY OF STUDENTS



3 STUDENT SEARCH ALGORITHM

This algorithm was designed to allow searching for a specific student by surname.

As my list of students existed in a file sorted by surname, I decided that it would be desirable to use a binary search to find the requested student record. Typically a binary search is used with a large, ordered problem set, so these conditions were ideal. I initially began by planning a linear search but as the file was sorted already, it seemed a waste of time and resources to read each object one after another, when my requested record could be the last element. It would be better to use a binary search to skip to the middle of the file and start discarding results that were not required.

In the end I chose the binary search over a simpler linear search, as with a binary search, the problem set is divided in half repeatedly until the search returns found or not found messages.

In terms of the Big O, binary search is infinitely better, reducing a possible n^2 to a very desirable log(n). In terms of implementation in c-code, there was little difference in terms of complexity of the code written or compile time. Rather than read and search on the file, I read the file into an array which will have added to run time slightly, but overall is more efficient.

To allow for flexibility in choosing which student would be searched for, I allowed for the user to input the desired surname into the keyboard. That input is then saved as a search string and while the binary search is reading each record, it check the record's surname and compares it to the search string. If it is the same, it returns the student record as output to the screen.

One thing to note about binary search in this algorithm, is that in implementation, it does not allow for returning results of multiple students with the same surnames. To try account for this, I would suggest adding an additional check for the records on either side of any 'found' records and return those if they also match the search string. During implementation in C-code, I tried this but was unsuccessful in getting the desired results so I have omitted this check from my code for now.

A linear search would get around this, so perhaps a combination of the two searching algorithms would have yielded a complete and faultless solution.

In terms of calculating the Big O, in this algorithm the problem is divided up into smaller problems. The chances of finding the search key are improved by the fact that the algorithm is directing the search left and right down the array's 'branched' sub arrays by comparing the middle element to the search key. Thus the Big O will always be a factor of two, being that the array is split in two each time.

For instance, like with Merge Sort, an array of 8 elements becomes 2 branches of 4, which becomes 4 branches of 2, which finishes with 8 branches of 1 element arrays. That's 3 breaks of the array (worst case scenario) i.e. $\log_2 8 = 3$

The Big O of this algorithm is therefore: log(n)

FLOWCHART

The following flowchart is written in pseudo-english and is meant as a visual description of the implementation of this algorithm, so that a non-IT client might be able to better understand the process.

PART 3 – SEARCH FOR SPECIFIC STUDENT BY SURNAME