Title:

Algorithms Design & Problem Solving Assignment Report

Student Name:

Robert Vaughan

Student Number:

C15341261

Module Code:

PROG1210

Course Code:

DT228/1

Tables of Contents

- **Introduction……3**
- **Part I……4**

Binary Insertion Sort……4

Big O……7

Bubble Merge Sort Draft……8

Big O……9

Bubble Merge Sort……11

Big O……13

- **Part II……15**

Flowchart……15

Big O……15

- **Part II……16**

Flowchart……16

Big O……16

- **Conclusion……17**

- ## Introduction

The following report analyses what actions were taken to complete all given task within the Algorithm Design and Problem Solving 2015 - 2016 Assignment Brief. Within the assignment, there are three main sections, which are the following:

Part I – Gain a three lists of student names. Each list represents a different type of student which are “Funded”, “Non-Funded” and “International”. One must create an algorithm to merge and sort these lists together to create one class list and to store said result on file. One has the option to implement a search into a pre-existent algorithm and/or merge two algorithms together to create an entirely new algorithm. The algorithm should be presented in Pseudocode.

Part II – One must utilising a searching method to find all international students within the class list. To visually represent this, one is recommended to utilise a Flowchart.

Part III – One must search for a particular student by entering in their surname. Once again, a search method should be utilised to fulfil this aspect of the assignment. A Flowchart should be given to represent said algorithm.

For each Part, the Big O notation for each part of the assignment will be discussed. Finally, each three tasks should be implemented into C Code.

- **Part I**

With the three lists, each representing different types of students, one must write an algorithm “that combines the three lists of students into one list”, according to the assignment brief. Ideally, one must formulate a solution that is highly efficient in for that would of course be the most optimum solution.

Binary Insertion Sort

To begin, an attempt was made at combining some form of search algorithm with a sorting algorithm increase the efficiency of said algorithm. By assessing sorting algorithms, one could deem the best algorithm to merge a search algorithm with is Insertion Sort for it checks every element in an array for instance and the previous sorted elements to find the sorted location for the element being checked. The worst case for Insertion Sort is $O(n^2)$, for it hosts a nest loop within a loop. One could replace the nested loop with some form of a Search to hopefully increase the algorithms performance.

To begin, a recursive Binary Search was written to act as the searching part of the new sorting algorithm. The search should be capable of returning the position to which the value should be placed. The following is that precise algorithm:

```
Search(group[], current[], first, last)

        mid = (first + last) / 2

        if (first >= last)

                if (current < group[mid].surmane)

                        return (first)
                END IF
                else if

                        return (first + 1)
                END ELSE
        END IF

        if (current = group[mid].surmane)

                return (mid +1)
        END IF

        if (current > group[mid].surmane)

                return Search(group, current, mid + 1, last)
        END IF
        else if (current < group[mid].surmane)

                return Search(group[], current, first, mid - 1)
        END ELSE IF

END SEARCH
```

To fully understand the Binary Search, below is a Call Stack that represents the operation of the Search Algorithm with the following sets of numbers:

{2, 3, 4, 5, 6, 7, 8, 1}

The algorithm will check the last element (which hosts the value 1) and will search the previous sorted numbers in the array and discover where it must be placed. Below is the Call Stack for said operation.

16	Search (group[], surname[], first, last)	first = 0, last = 6, mid = 3
16	Search (group[], surname[], first, mid - 1)	first = 0, last = 2, mid = 1
3	Search (group[], surname[], first, mid - 1)	first = 0, last = 0, mid = 0

When then third call is made, the function will meet the first if condition within the algorithm and will return the value 0, which will be the position the value should be placed within the sorted side of the array.

Once this algorithm was completed, merging the Binary Search within Insertion Sort was required. Below is an edited Insertion Sort that will call the Search function:

```
insertionSort(group[], size)

        i = 0
        j = 0
        pivot
        current_first[]
        current_sur[]
        current_type

        for (i = 1 to i < size)

                current_first = people[i].firstname
                current_sur = people[i].surname
                current_type = people[i].type

                j = i – 1

                pivot = Search(group, current_sur, 0, j)

                while (pivot <= j)

                        people[j +1].firstname = people[j].firstname
                        people[j +1].surname = people[j].surname
                        people[j +1].type = people[j].type
                        j = j – 1
                END WHILE

                people[j +1].firstname = current_first
                people[j +1].surname = current_sur
                people[j +1].type = current_ type
        END FOR
END INSERTION
```

```
Search(group[], current[], first, last)

        mid = (first + last) / 2

        if (first >= last)

                if (current < group[mid].surmane)

                        return (first)
                END IF
                else if

                        return (first + 1)
                END ELSE
        END IF

        if (current = group[mid].surmane)

                return (mid +1)
        END IF

        if (current > group[mid].surmane)

                return Search(group, current, mid + 1, last)
        END IF
        else if (current < group[mid].surmane)

                return Search(group[], current, first, mid - 1)
        END ELSE IF

END SEARCH
```

As one may find, the above Insertion Sort is mildly altered to incorporate a call to the Search function. The variable j is assigned (i - 1) so the search only checks the previous elements behind the element being checked. Therefore, the while loop is also edited to accompany these alterations. Below is the operation of the while loop with the previous list of numbers. The Search returned the value 0 to be the position. Therefore pivot is equal to 0 and j is equal to 6.

List before loop commences:

{2, 3, 4, 5, 6, 7, 8, 1}

Iterations

1. {2, 3, 4, 5, 6, 7, 8, 8}
2. {2, 3, 4, 5, 6, 7, 7, 8}
3. {2, 3, 4, 5, 6, 6, 7, 8}
4. {2, 3, 4, 5, 5, 6, 7, 8}
5. {2, 3, 4, 4, 5, 6, 7, 8}
6. {2, 3, 3, 4, 5, 6, 7, 8}
7. {2, 2, 3, 4, 5, 6, 7, 8}

When the loop ends, the value that was being checked is placed into where it should be poisoned to make the list sorted. The value being checked was 1. Therefore, the loop will be placed within element 0, as shown below:

{1, 2, 3, 4, 5, 6, 7, 8}

The Big O - $n^2 + [\overline{\log_2 2} + .. + \overline{\log_2 n}]$

The main question in association with the edit to the above Insertion Sort is if it has become more efficient with the addition of a Binary Search. In a worst case scenario, the original Insertion Sort is unfortunately more efficient in relation to the Big O notation in a worst case scenario. The O notation for Insertion Sort is n^2. However, in a worst case, since the function Search is called that acts as a Binary Search, in theory, it is only adding more iterations to the O notation for it is trying to find the location of where a certain value should be stored if it when sorting the list. Therefore the worst case shall be increased. However, in run time this could be more efficient for it reduces the number of comparisons required within the program, resulting in a faster run time speed. Although, one could argue that utilising the Stack would create too much overhead.

The Big O notation can be calculated by the following algorithm is the following

$$n^2 + [\overline{\log_2 2} + .. + \overline{\log_2 n}]$$

Bubble Merge Sort Draft

Since the original algorithm sadly failed to become more efficient, one may revert to merging two algorithms together. Two algorithms to merge together ideally would be Bubble Sort and Merge Sort. If one could sort the three lists individually with Bubble Sort and then utilise the merge part of Merge Sort to compare the heads of the list and go through said lists by each element until, finally, a sorted list is created. Since Bubble Sort is a commonly utilised sorting algorithm, the only real task is creating a compared heads of two lists algorithm that will formulate a sort list. Below is an attempt at said algorithm in Pseudocode:

```
BubbleSort (origin, pos, people[])

        i = 0
        j = 0
        first_swap[]
        sur_swap[]
        type_swap

        for (i = origin to i < (pos – 1))

                for (j = origin to j < (pos – 1))

                        if (people[j].surname > people[j + 1].surname)

                                first_swap = people[j].firstname
                                sur_swap = people[j].surname
                                type_swap = people[j].type

                                people[j].firstname = people[j + 1].firstname
                                people[j].surname = people[j + 1].surname
                                people[j].type = people[j + 1].type

                                people[j + 1].firstname = first_swap
                                people[j + 1].surname = sur_swap
                                people[j + 1].type = type_swap
                        END IF

                END FOR

        END FOR

END BUBBLE SORT
```

```
Merge(list[], temp_array[], leftindex, left_end, rightindex, right_end, size)

        i = 0

        for (i = 0 to i < size)

                if (list[leftindex].surname < list[rightindex].surname AND leftindex DOES NOT EQUAL
left_end)

                        temp_array[i].firstname = list[leftindex].firstname)
                        temp_array[i].surname = list[leftindex].surname
                        temp_array[i].type = list[leftindex].type

                        leftindex = leftindex + 1
                END IF
                else if (rightindex DOES NOT EQUAL right_end)

                        temp_array[i].firstname = list[rightindex].firstname
                        temp_array[i].surname = list[rightindex].surname
                        temp_array[i].type = list[rightindex].type

                        rightindex = rightindex + 1
                END ELSE IF

        END FOR

        for (i = 0 to i < size)

                list[i].firstname = temp_array[i].firstname
                list[i].surname = temp_array[i].surname
                list[i].type = temp_array[i].type
        END FOR

END MERGE
```

Description of Variables

leftindex	Holds the current element position being checked on the left side
light_end	Hosts a value that prevents the left sides counter incrementing
rightindex	Holds the current element position being checked on the right side
right_end	Hosts a value that prevents the right sides counter incrementing

Each class list will be sorted individually with Bubble Sort. The Merge function will then be called to compare the heads of two of the lists. This means that for the given problem with the Algorithms Design Assignment, the Merge function will be called twice.

The O Notation - $(x^2 + y^2 + z^2) + (4x + 4y) + 2z$

Thankfully, the following algorithm has a better worst case when compared to the Binary Insertion Sort that was created. To calculate the number of notations, one requires three variables. To get a standard worst case, one shall work out the worst case by having 36 elements divided equally into 3 separate lists. The following is an algebraic function to solve this:

n is equivalent to the number of people in each list.

$$\mathbf{3n^2 + 10n}$$

When the algorithm begins, each list will be sorted with Bubble Sort. This means that the algorithm will have (in a worst case) iterated thought each list n^2 times. Therefore, the algorithm would have iterated $3n^2$ thus far and each list is sorted. To continue the algorithm will then begin merging these lists by utilising the comparing of heads method found within a common Merge Sort. A call to said function shall be made twice, the first call compares the heads of the first two lists within the array and store the results in a temporary array and then transfer said array into the original array. Therefore, the total count of numbers being compared and the number of iterations required to transfer the array must be taken into consideration to determine a worst case of iterations.

To break this down the following table highlights these iterations.

Section	Bubble Sort	Bubble Sort	Bubble Sort	Merge Sort	Merge Sort
Iterations	144	144	144	48	72

From the above table, the number of iterations in each section of the designed algorithm is highlighted.

Of course, an algebraic function must be created to highlight the different number of iterations in a worst case scenario.

X – A list with the highest amount of students

Y – A list with the second highest amount of students

Z – A list with the least amount of students

The following algebraic function solves the O notation in the worst case scenario:

$$\mathbf{n = (x^2 + y^2 + z^2) + (4x + 4y) + 2z}$$

As stated previously when discussing an equal number of people in each list, every list must face being sorted by Bubble Sort. Therefore, the number of iterations differs depending on each size of the lists. As an example, we shall use the class list given within the assignment brief as an example. Therefore, the biggest list is 20, the second is 10 and the smallest is 6. Therefore, the number of iterations that occurs after each list is sorted in a worst case is 556. This is due to the fact that Bubble Sorts nest loop checks all elements within an array, creating an n^2 algorithm. The Merge section of the algorithm will be called after all lists are sorted. The first call will compare the heads of the first two lists making it x + y, followed by a loop to store them within an array, making it 2x + 2y. The second merge call will act similarly however it compares all elements of an array making the comparing of heads loop x + y + z and the store loop x + y + z. If one adds these variables together, one gets $(x^2 + y^2 + z^2) + (4x + 4y) + 2z$. Therefore, the total number of iterations is 668, which would be far more efficient than the Binary Insertion Sort discussed previously.

Bubble Merge Sort

Upon reviewing the previous Pseudocode, one may come across a solution that may be more efficient. Instead of utilising a temporary array to hold the merged sorted lists, one could pass an extra array to reduce the usage of an extra loop. Another edit could be the fact that during the merge section of the algorithm, when the when one side of the array is fully checked, the other side is compared is still be compared the side that is complete, which can create an overflow. Finally, the fact that the merge section is not called within the Bubble Sort which technically makes them two separate algorithms. To solve these issues, the following algorithm was designed:

```
BubbleSort (origin, pos, people[], start, pos1, pos2, pos3, merge_array1[], merge_array2[])

        i = 0
        j = 0
        first_swap[]
        sur_swap[]
        type_swap

        for (i = origin to i < (pos - 1))

                for (j = origin to j < (pos - 1))

                        if (people[j].surname > people[j + 1].surname)

                                first_swap = people[j].firstname
                                sur_swap = people[j].surname
                                type_swap = people[j].type

                                people[j].firstname = people[j + 1].firstname
                                people[j].surname = people[j + 1].surname
                                people[j].type = people[j + 1].type

                                people[j + 1].firstname = first_swap
                                people[j + 1].surname = sur_swap
                                people[j + 1].type = type_swap
                        END IF

                END FOR

        END FOR

        if (BUBBLE SORT HAS OCCURRED 3 TIMES)

                Merge(people, merge_array1, start, pos1, pos1, pos2, pos3)
                Merge(merge_array1, merge_array2, start, pos2, pos2, pos3, pos3)
        END IF

END BUBBLE SORT
```

```
Merge(list[], temp_array[], leftindex, left_end, rightindex, right_end, size)

        i = 0

        for (i = 0 to i < size)

                if (rightindex != right_end AND leftindex != left_end)

                        list[leftindex].surname < list[rightindex].surname

                        if (list[leftindex].surname < list[rightindex].surname)

                                temp_array[i].firstname = list[leftindex].firstname
                                temp_array[i].surname = list[leftindex].surname
                                temp_array[i].type = list[leftindex].type;

                                leftindex = leftindex + 1
                        END IF
                        else if (list[leftindex].surname > list[rightindex].surname)

                                temp_array[i].firstname = list[rightindex].firstname
                                temp_array[i].surname = list[rightindex].surname
                                temp_array[i].type = list[rightindex].type

                                rightindex = rightindex + 1
                        END ELSE

                END IF
                else if (leftindex == left_end AND rightindex != right_end)

                        temp_array[i].firstname = list[rightindex].firstname
                        temp_array[i].surname = list[rightindex].surname
                        temp_array[i].type = list[rightindex].type

                        rightindex = rightindex + 1
                END ELSE IF
                else if (leftindex != left_end AND rightindex == right_end)

                        temp_array[i].firstname = list[leftindex].firstname
                        temp_array[i].surname = list[leftindex].surname
                        temp_array[i].type = list[leftindex].type

                        leftindex = leftindex + 1
                END ELSE IF
                else
                        temp_array[i].firstname = list[i].firstname
                        temp_array[i].surname = list[i].surname
                        temp_array[i].type = list[i].type
                END ELSE
        END FOR
END MERGE
```

Description of Variables

start	A variable that is assigned with the value 0
pos1	Contains how many people there on within the first list
pos2	Contains how many people there on within the second list
pos3	Contains how many people there on within the third list
merge_array1	Hosts first two lists sorted and merged
merge_array2	Hosts all three lists sorted and merged
leftindex	Holds the current element position being checked on the left side
light_end	Hosts a value that prevents the left sides counter incrementing
rightindex	Holds the current element position being checked on the right side
right_end	Hosts a value that prevents the right sides counter incrementing

The O Notation - $O((x^2 + y^2 + z^2) + 2(x + y + z))$

With the edit of Bubble Merge Sort, an algorithm is created that is more efficient in sorting the three class lists and merging them together. When all three lists are sorted, Merge will be called with to merge the first two lists in a sorted manner with the comparing of heads tactic. When all heads are compared, the third list is then placed with the first merge array. Therefore, the array is partly sorted. When that Merge call is completed, a second Merge call will compare the heads of the left side of the array that is merged and sort with the right side of the last remaining list of names. Below is an illustration on how this Bubble Merge functions:

{9, 10, 11, 5, 6, 7, 8, 1, 2, 3, 4}

The blue numbers are within the first list, the black numbers are the second list and red numbers are the third list. This is what occurs after the third Bubble Sort Call is made. When the merge begins, the first two lists are compared and placed in a new array.

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
 ^ ^
{5, , , , , , , , , , }

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
 ^ ^
{5, 6, , , , , , , , , }

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
 ^ ^
{5, 6, 7, , , , , , , , }

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
 ^ ^
{5, 6, 7, 8, , , , , , , }

Once one side is completed, the other side is only put through until it reaches its check.

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
 ^
{5, 6, 7, 8, 9, , , , , , }

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
^
{5, 6, 7, 8, 9, 10, , , , , }

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}
^
{5, 6, 7, 8, 9, 10, 11, , , , }

Since the last section of the two merging sections is completed, the rest of the array is placed within it.

{5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4}

When the second Merge Call occurs, the array will be fully sorted. A merge call would have O(n) complexity. Therefore we can edit our previous Big O Notation with this information. To begin we will again look at each list having equal amounts, as seen below:

n is equivalent to the number of people in each list.

$$\mathbf{3n^2 + 2(3n)}$$

The $3n^2$ is the three iterations of the Bubble Sort. The 2(3n) is number of iterations taking place by the Merge function. Each Merge call is O(n) since the loop goes through the array each time called with the use of a loop. Therefore, each call would be 3n, making two calls 2(3n) or 6n.

Now we shall look at the algorithm handling three unique lists.

X – A list with the highest amount of students

Y – A list with the second highest amount of students

Z – A list with the least amount of students

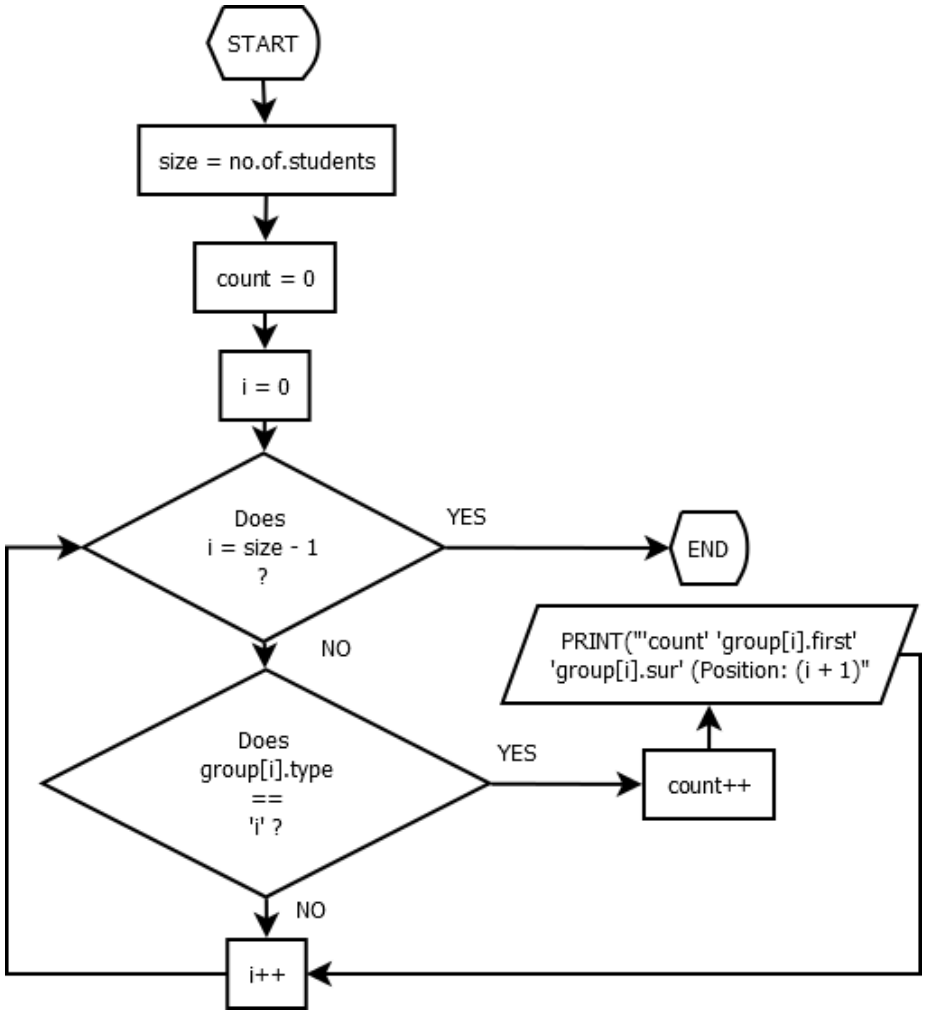
The following algebraic function solves the O notation in the worst case scenario:

$$\mathbf{n = (x^2 + y^2 + z^2) + 2(x + y + z)}$$

Similar to the previous Bubble Merge that was formulated, the Bubble Sort iterations will depends on the size of each list. However, the Merge function iterations now depend on the total amount of all students, making measuring the worst case more simplistic in a sense. The number of iterations with lists of 20, 10 and 6 people is O(608), which is a smaller amount of iterations in comparison to the first draft of the Bubble Merge Sort.

- **Part II**

Part II of the brief is a carry on from Part I, in which one must “write an algorithm to search for all students from International”, according to the assignment brief. From the given brief, I formulated an algorithm that one could claim is highly efficient for it makes use of a Binary Search. Below is the algorithm to which I designed represented with a Flowchart.



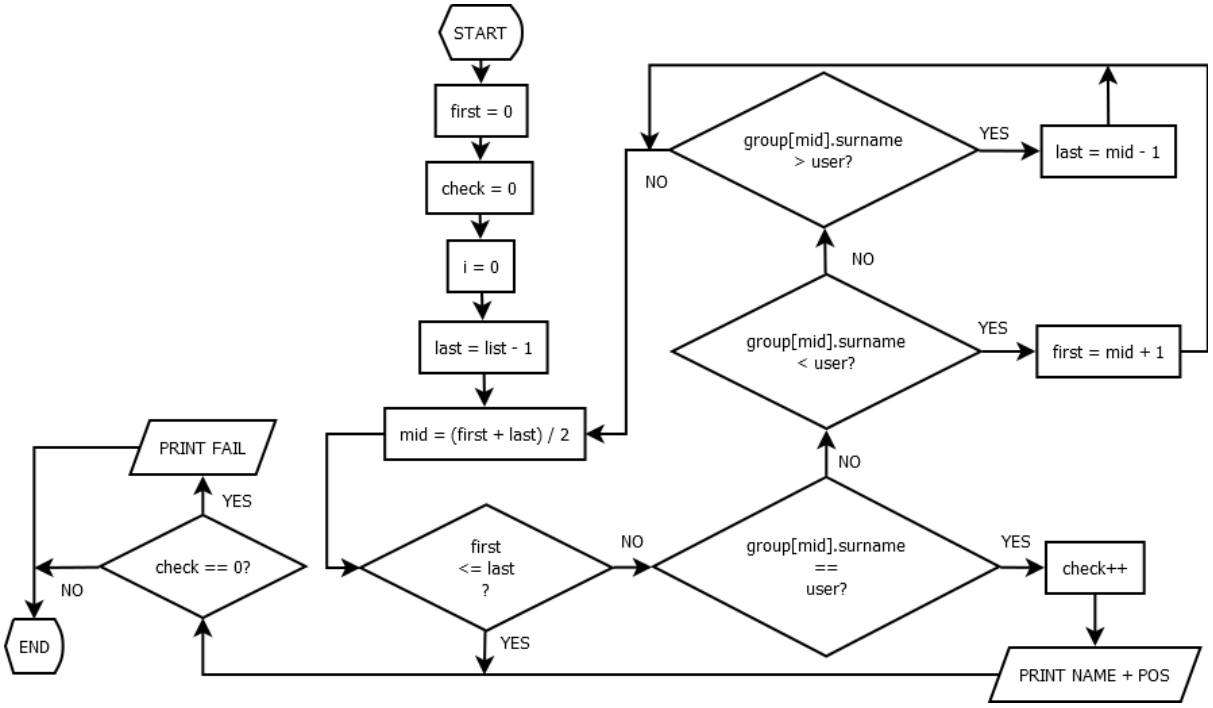
From the above Flowchart, one can find that the purpose of this algorithm is to search for students in that are registered as international and to highlight there position on the list. This would allow administration to take the sorted class list and find in what position of the list is each person on the unsorted international list is on the class list. This could aid one in removing a certain name from the final class list in any point of the year if one student wishes to be removed from the course.

The Big O - O(n)

The Big O notation for the above Binary Search Algorithm is O(n) in a worst case scenario. This is so for the loop condition that occurs within the flowchart is iterative. The algorithm hosts no divide and conquer method. Each name on the class list is checked to see if the each name is marked as an international student or not. Therefore, all names are checked iteratively. For instance, if we have 36 names on the class list, then the O notation is O(36) for the loop will run 36 times since there are 36 names on the list. This is a very simplistic loop and the level of complexity is quite low, as seen in the flowchart.

- **Part III**

In to the third part of the assignment brief, one must design a flowchart that will "search for a specific student by surname". Once again, the best solution for this creating an algorithm that utilises a form of Binary Search. Below is algorithm represented with the aid of a Flowchart.



The above algorithm utilises a simple Binary Search. The algorithm will check the middle element of the of the sorted class list and if the name one is searching for is greater than the middle element, the new first value will be the element after the middle element and division will take place again. If a the name that is being searched is less than the current middle name in the list, the last value will be changed to the middle value minus 1 and division will occur again. If the name is greater than the current name, the first variable will be assigned to the middle value plus 1 and division occurs once again. This process will repeat until a name is found and the loop is broken. If a name is not found, a message will display informing the user that no name exists on the Class List.

The Big O - O($\log_2 n$)

The Binary Search, as discussed in Part I, is a divide and conquer method of searching. Therefore the above algorithm is $\log_2 n$ for it is a divide a conquer approach in finding the correct surname. Therefore the Worst Case for the Algorithm is O(6) for the sum is O($\log_2 36$). Since we are working with a divide and conquer based algorithm for the algorithm is based on Binary Search, then one is working with O($\log_2 n$). n would be the number of elements that are being checked, which is the number of people on the class list. This is a very effect form of search for a surname for the worst case would be a name that is either first or last on the list. Since it is divide and conquer, one also has a better run time for instead of have a notation of O(n), it is reduced to O($\log_2 n$).

- ## Conclusion

As one may note, all aspects of the given assignment brief have been discussed and all tasks with said brief have been completed in a variety of ways. For Part I, a custom Bubble Merge Sort was created after various attempts of making a sorting algorithm from scratch. Part II gave us a conclusion that a useful way to search the list for certain students would be the iterative method. Finally Part III highlighted the use of a divide and conquer algorithm, regarded as the Binary Search to decrease the number of iterations when searching for a specific person. In total, the assignment took a significant amount of hours. This is partly due to the desire to create a custom algorithm from scratch.