Algorithms Assignment Report

Ruán Murgatroyd

C22400846

CMPU1001

16/04/2023

Declaration

I hereby certify that all material submitted in this assignment for *Algorithm Design & Problem Solving* in the *School of Computer Science*, Technological University of Dublin is entirely my own work and has not been submitted for any academic assessment other than this submission.

Signed: Ruán Murgatroyd

Introduction

As part of my CMPU1001 Assignment, I was required to create and deliver a series of tasks directly related to the Module Syllabus.

This report documents my assignment, from identification of the requirements to each element of the deliverables. Justifications and further explanations will be provided where necessary.

This report will be laid out in the following manner:

1. Identification of the Project Requirements
    a. Requirements identification
    b. Deliverables
    c. Assignment Creation Process
2. Suitable Data Types
3. List Combination
    a. Sorting Algorithm
    b. Big O
4. Full-Time Student Search
    a. Algorithm
    b. Big O
5. Searching by Surname
    a. Method
    b. Big O
6. C Implementation
    a. Differences from Designed Elements
    b. Justification
    c. Creation Process
7. Conclusion

Identification of Project Requirements

The assignment outlined a specific project which needed to be delivered. It was separated into a requirements specification which must be achieved in five sections.

Requirements Identification

The assignment introduced four new courses in a university, DT265A, DT265B, DT265C, and DT8900. These courses were either Full-Time or Part-Time, and were Higher Diplomas or Master's Qualifiers, including one International Master's Qualifier. The requirement also listed a certain number of students, which I set as the maximum course occupancy of each course.

The brief stated that each course requires its own list of student information.

Deliverables

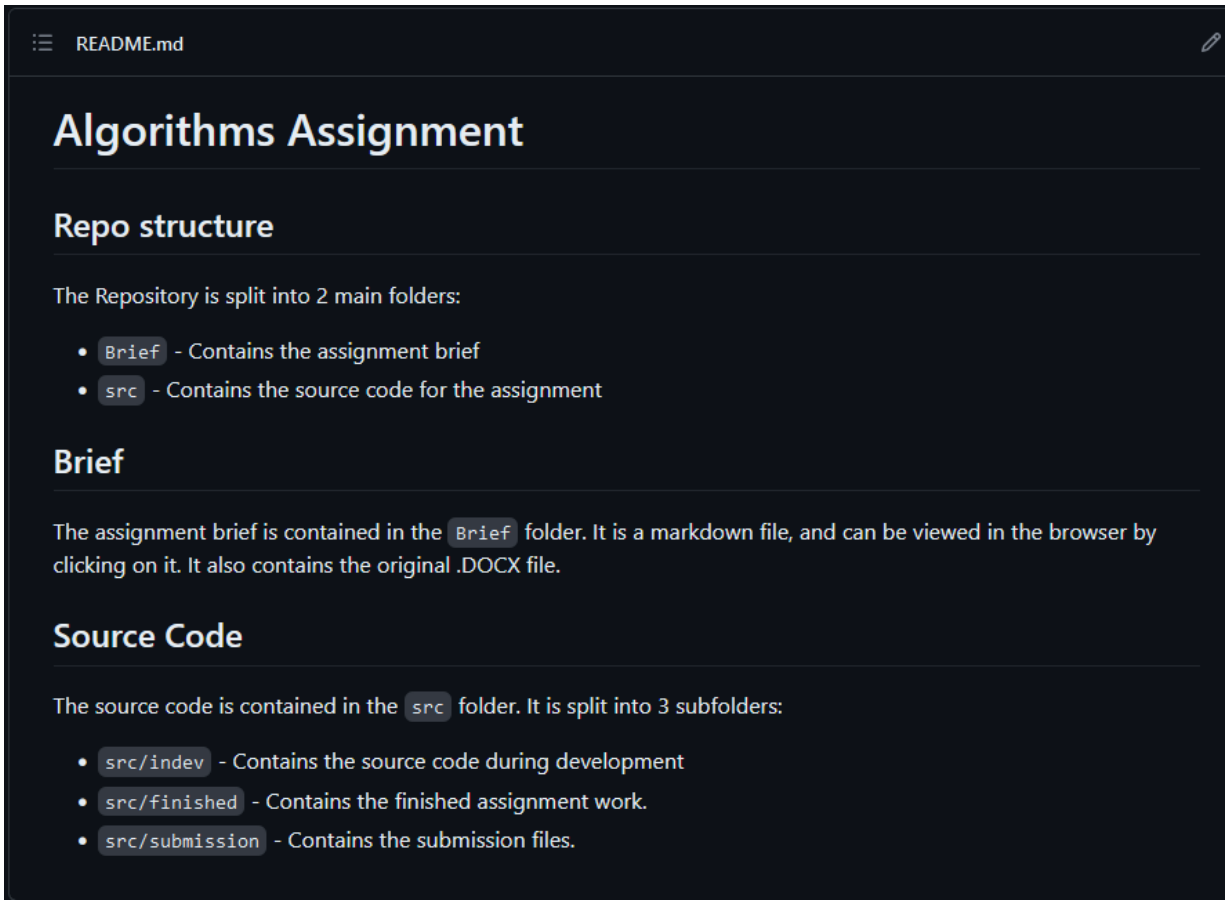
Five deliverables were required in the assignment:

1. A suitable datatype for student information,
2. A flow-chart algorithm for combining the four lists into a single list,
3. A pseudo-code algorithm to search the single list for Full-Time students,
4. A pseudo-code algorithm to search the single list by surname,
5. A working implementation of Parts One – Four in the C Programming Language.

Each of these deliverables will be discussed in-depth in their relevant sections. Each deliverable was also given specific requirements and discussion points which will be discussed in their section.

Assignment Creation Process

The assignment was created using Visual Studio Code, yEd Live. The assignment was stored both on OneDrive and on my personal GitHub account in a private repository (Which will be made public after the results are released). The use of GitHub allowed me to version-control my assignment and have a solid and effective structure.


README.md

Algorithms Assignment

Repo structure

The Repository is split into 2 main folders:

- `Brief` - Contains the assignment brief
- `src` - Contains the source code for the assignment

Brief

The assignment brief is contained in the `Brief` folder. It is a markdown file, and can be viewed in the browser by clicking on it. It also contains the original .DOCX file.

Source Code

The source code is contained in the `src` folder. It is split into 3 subfolders:

- `src/indev` - Contains the source code during development
- `src/finished` - Contains the finished assignment work.
- `src/submission` - Contains the submission files.

The Assignment Repository's README.md

Suitable Data Types

The requirements deliverables stated a ‘suitable data type for manipulating and storing the data having regard to the underlying changeability and the need for instant reporting’ as well as an explanation for my design. To do this, I created two structures, the s*tudentRecord* structure, and the *node* structure.

studentRecord Structure

I identified a few required elements in the studentRecord structure that I included. The structure was as follows:

```
studentRecord:
        FirstName
        Surname
        CourseID
        StudentID
        Year
```

While most of these structure elements are self-documenting, the CourseID and StudentID are elements which I designed to be unique.

CourseID

The CourseID would be a number from 1 to 4, corresponding to the courses DT265A, DT265B, DT265C, and DT8900 respectively. This requires less memory to store, and is easier to parse and manipulate when converted into actual C code.

StudentID

I realised that since I was generating test data for the users, I would need a sufficiently large range of randomly-assignable student IDs to minimise the chance of two students being assigned the same ID. To do this, student ID’s followed the following structure: DT#####.

node Structure

The node structure was a simple two-element structure to make linked lists easy to create, read, and modify.

It was as follows:

```
node:
        studentRecord
        nextNode
```

When recreated in C, the nextNode element would be a pointer to the next node structure in the linked list.

Changeability was another stated requirement. This was achieved by creating two structures for this data type. The separate studentRecord structure allows edits to a single structure for the entire linked list to be updated, and the nextNode pointer creates a simple workflow for adding and removing to the list.

List Combination

The requirement deliverables stated a ‘flow-chart (for) an algorithm that combines the four lists into one main list sorted by surname’. To do this, I had designed a two-step process:

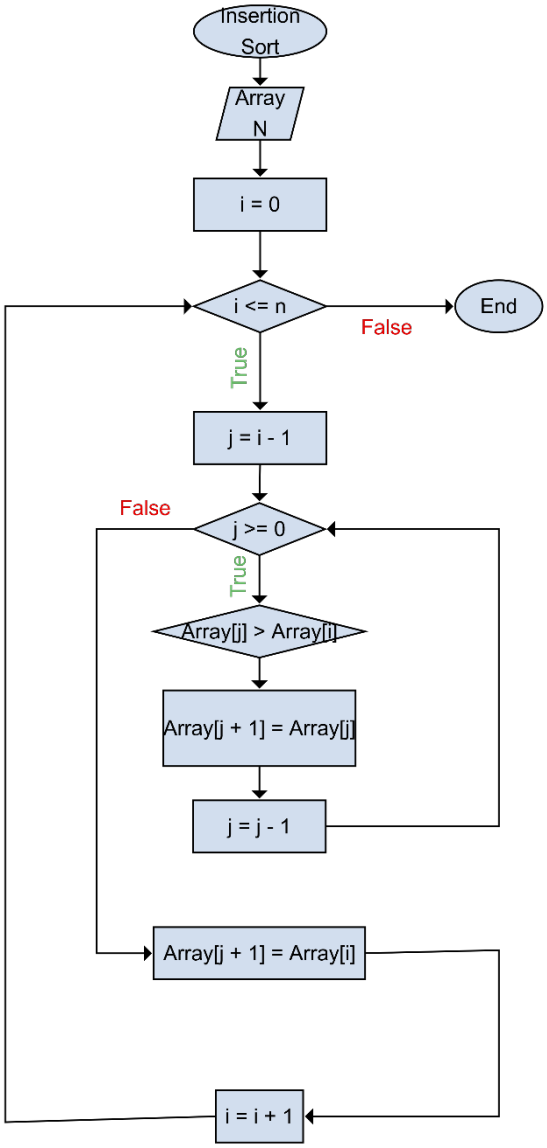
1. Use insertion sort on each of the four lists,
2. Use merge sort to create the main list.

Sorting Algorithms

To do this in Flow-Charts, I had created 4 flow charts, the main code, and three functions.

Full-Size images of each Flow-Chart will be found at the end of the report, after the conclusion.

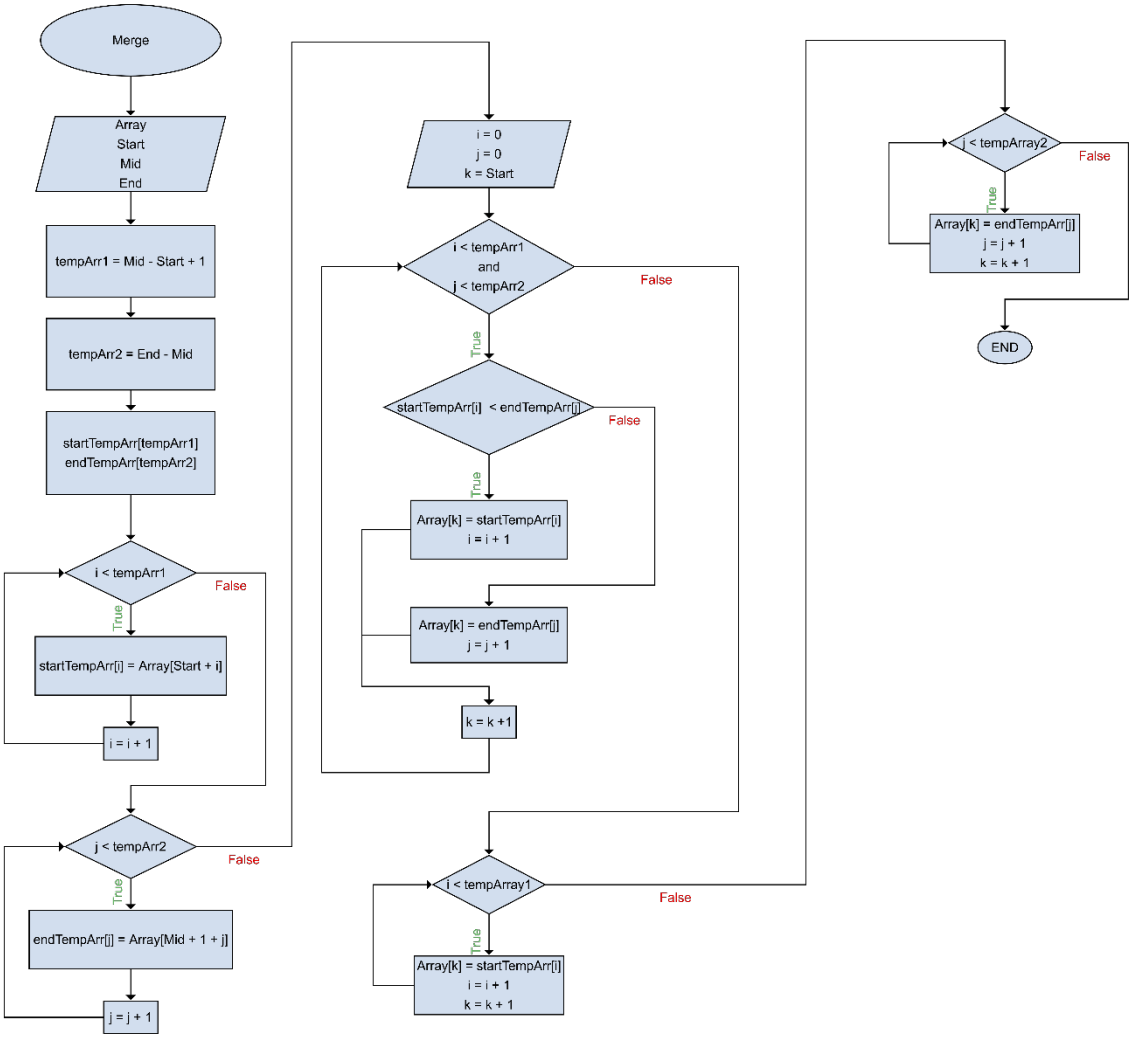
Insertion Sort



The Insertion Sort Function in a Flow-Chart.

This Flow-Chart shows the insertion sort for each of the unique lists. Insertion Sort has a best-case scenario of O(n), and a worst-case scenario of $O(n^2)$.

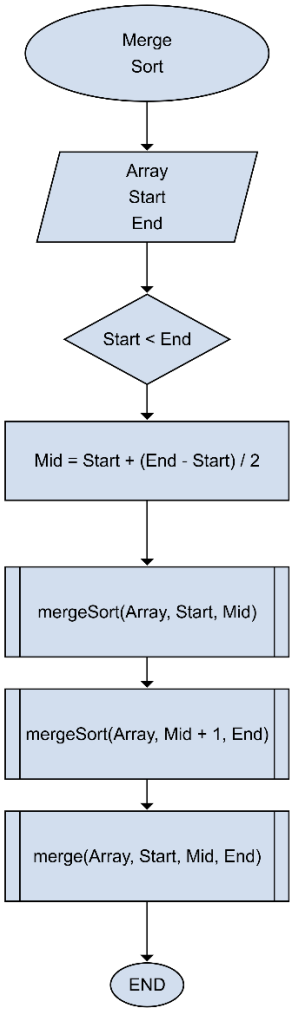
Merge



The Merge Function, a sub-section of Merge Sort

The Merge Function is required for Merge Sort.

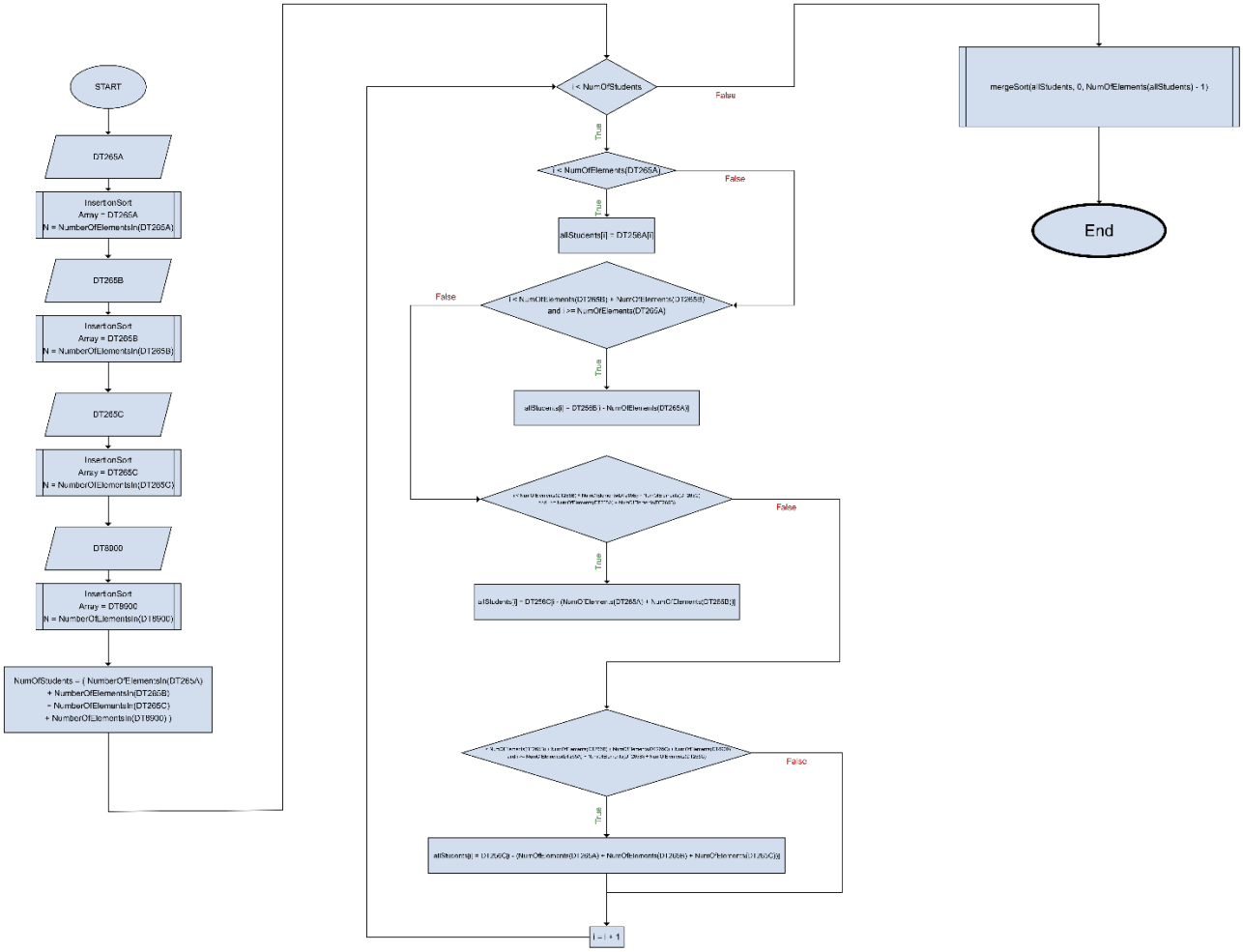
Merge Sort



The Merge Sort function.

This is the Merge Sort implementation in a Flow-Chart. Merge Sort has a worst-case scenario of O(n log(n)).

Main Code



This is the Main block for adding all the lists to a single large list.

Pre-Sorting all of the individual lists, while not necessary, speeds up the merge sorting of the main list.

Full-Time Student Search

```
startStructure studentInfo
      firstName
      surname
      Course // 1 = Part-Time, 2 = Full-Time, 3 = Part-Time, 4 = Full-Time
      StudentID
      Year
endStructure

startProgram findFullTime
      array studentInfo allStudents[42]

      findFullTime(allStudents)
endProgram

startFunction findFullTime(Array)
      While i < LengthOf(Array)
            If Array[i].Course == 2 OR Array[i].Course == 4
                  Print("NAME: " + Array[i].firstName + " " + Array[i].surname)
                  Print("COURSE: " + Array[i].Course)
                  Print("STUDENT ID: " + Array[i].studentID)
                  Print("Year: " + Array[i].Year)
            EndIf
      EndWhile
endFunction
```

The Full-time searcher simply iterates through the list and checks the Course if it is one of the two full-time courses, if it is, it prints all the information of the associated student. It's worst time complexity is O(n).

Searching By Surname

```
startStructure studentInfo
      firstName
      surname
      Course // 1 = Part-Time, 2 = Full-Time, 3 = Part-Time, 4 = Full-Time
      StudentID
      Year
endStructure

startProgram binarySearchList
      array studentInfo allStudents[42]

      Print("Enter Surname to Search: ")
      Input(toSearch)

      Print(binarySearch(allStudents, toSearch, 0, Length(allStudents) - 1))
endProgram

startFunction binarySearch(Array, toFind, Start, End)
      If start <= end
            middle = (start + End) / 2

            If array[midIndex].surname == toFind
                  return array[midIndex]
            Else If toFind < array[midIndex].surname
                  return binarySearch(array, toFind, midIndex + 1, end)
            Else
                  return binarySearch(array, toFind, start, midIndex - 1)
            EndIf
      EndIf

      return -1
endFunction
```

This function uses a Binary Search to find by surname. Binary search allows a best-case O(n log(n)).

In this example, this search is completed on an array instead of a linked list. Using Binary Search on a Linked List has a Big O of O(n)

C Implementation

The implementation of all of these was a large project, but a very fun and enjoyable one.

The final code (Without comments) is nearly 800 lines of code! It has 13 functions and uses five libraries. As such, only sections of the code will be included here, in code blocks. The entire code can be found in 'cMAIN.txt' which is a separately submitted file.

Differences from Designed Elements

To adhere to the rubric as much as possible, the C implementation uses linked lists for its implementation of the 'main list'. As such, while the pseudocode implementation of Question Four uses binary search, the C implementation iterates over each element.

I also implemented a testing data generation function to make the demo and development testing easier.

Creation Process

The creation of the C implementation involved following the programming convention taught in Michael Collins' Programming module. This included function signatures, for loop structure, and more.

```
// Function Signatures

struct node* addStudent(struct studentRecord [], struct
studentRecord [], struct studentRecord [], struct studentRecord [],
struct node *);
struct node* removeStudent(struct studentRecord [], struct
studentRecord [], struct studentRecord [], struct studentRecord [],
struct node *);
void searchStudent(struct node *, char *);
struct node* generateData(struct studentRecord [], struct
studentRecord [], struct studentRecord [], struct studentRecord [],
struct node *, int, int, int, int);
void sortCourseArray(struct studentRecord [], int);
void viewCourseData(struct studentRecord [], int);
struct node* createLinkedList(struct studentRecord [], struct
studentRecord [], struct studentRecord [], struct studentRecord [],
struct node *);
void merge(struct studentRecord [], int, int, int);
void mergeSort(struct studentRecord [], int, int);
void printLinkedList(struct node *);
const char* getCourseName(int);
int modifyCourseOccupancy(int, int);
void findFullTime(struct node *);
```

I decided early on that each of the course lists would be arrays instead of linked lists. This makes the addition, removal, and sorting of each course easier.

I was uncertain if the admission numbers in the assignment brief were the current students, or the maximum students, so I chose for those values to be the maximum students per course.

Once they are all written to, they are all added to a larger array by a merge sort.

One the merge sort is complete, a Linked List is created, and the head is returned.

Looping menus, type-checking, error-handling, and Switch Cases were used where applicable to improve safety and avoid user mistake.

Random Data Generation

The generation of test data was a very enjoyable function to create. Using the rand() function from the Standard Library. It was seeded with the time of the PC.

Using this randomness, test students can have one of twenty first names and twenty surnames. The Student ID is also procedurally generated.

```
printf("\nWriting DT265A's Test Data.\n");
    // Building DT265A's Test Data
    for(int i = 0; i < PTHDLIMIT; i++){

        // Randomly Generating Name
        strcpy(DT265A[i].firstName, firstNames[rand() % 20]);
        strcpy(DT265A[i].surname, surnames[rand() % 20]);

        // Randomly Generating 3-digit Student ID
        char tempStudentID[7] = "DT";
        for(int j = 0; j < 4; j++){
            strcat(tempStudentID, idNums[rand() % 10]);
        }

        strcpy(DT265A[i].studentID, tempStudentID);

        // Adding Course Code
        DT265A[i].courseCode = 1;

        // Adding Random Year
        DT265A[i].year = (rand() % 4) + 1;

        // Modifying the Course Occupancy
        modifyCourseOccupancy(1, 1);
    }

    printf("\nDT265A complete, starting DT265B.\n");
```

`modifyingCourseOccupancy(int, int);`

modifyingCourseOccupancy() is a function which uses global variables to handle the enrolment limits of each course.

This is primarily used to stop users from entering too many students in the addStudent() function.

At runtime, the global variables are set to the #define maxima of each course, modifyingCourseOccupancy() is given the Course number, and the amount to add or remove (Where positive removes that many, and negative adds that many.)

`getCourseName(int);`

In the planning process for this C program, I decided it would be easiest if the courses were stored as integers in the StudentRecord struct. This reduced memory load and the complexity of searching functions, meaning it didn't require a String Comparison.

To turn these Course Integers into human-readable Course Names, getCourseName() was created, which would return the full course code if passed a valid course, and return a NULL otherwise.

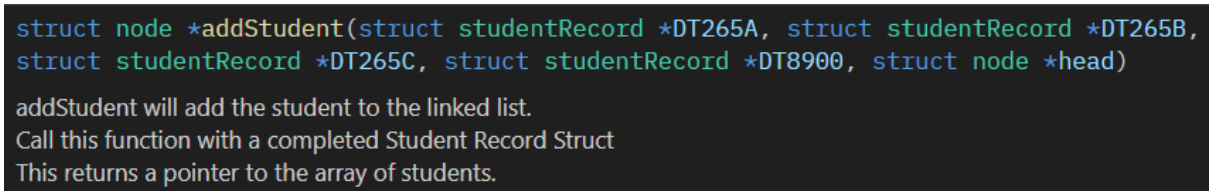
Modularity and Comprehension

The C Program was created with as much modularity as possible in-mind.

The two structs were created to make it easier to add and remove elements, and definitions for commonly used values were made.

The **CLEAR** definition was used to make it much easier to clear the input buffer, and be more human-readable than `while(getchar() != '\n')`.

For such a large project commenting is essential. Each function is given a comment at the top of the function body, to aid use and to work with Visual Studio Code's *IntelliSense.*



An example of IntelliSense adding the comments as documentation.

While I made as much of my code self-documenting, comments were added where complex code would make a first-reading difficult. Neat features of the C language were also commented.

```
// C stores chars as ASCII integers, taking away the ASCII value of '0' from the char number gives an integer value, neat!
// This would cause problems if it was not a number ('A' for example), but the parent if-statement catches that.
```

An example of a comment discussing an interesting quirk of C.

Function signatures were split into three main segments:

CREATING AND MODIFYING DATA:

1. addStudent()
2. removeStudent()
3. generateData()
4. createLinkedList()

SEARCHING AND INTERPRETING DATA

1. searchStudent()
2. findFullTime()
3. getCourseName()
4. modifyCourseOccupancy()
5. printLinkedList()

SORTING

1. sortCourseArray()
2. merge()
3. mergeSort()

Terminal User Interface

The project runs as a Terminal User Interface (TUI). This kind of graphical interface is restrained to ASCII characters, as such, heading were made of -, =, <, and [components, and extra line breaks were used to visually separate information.

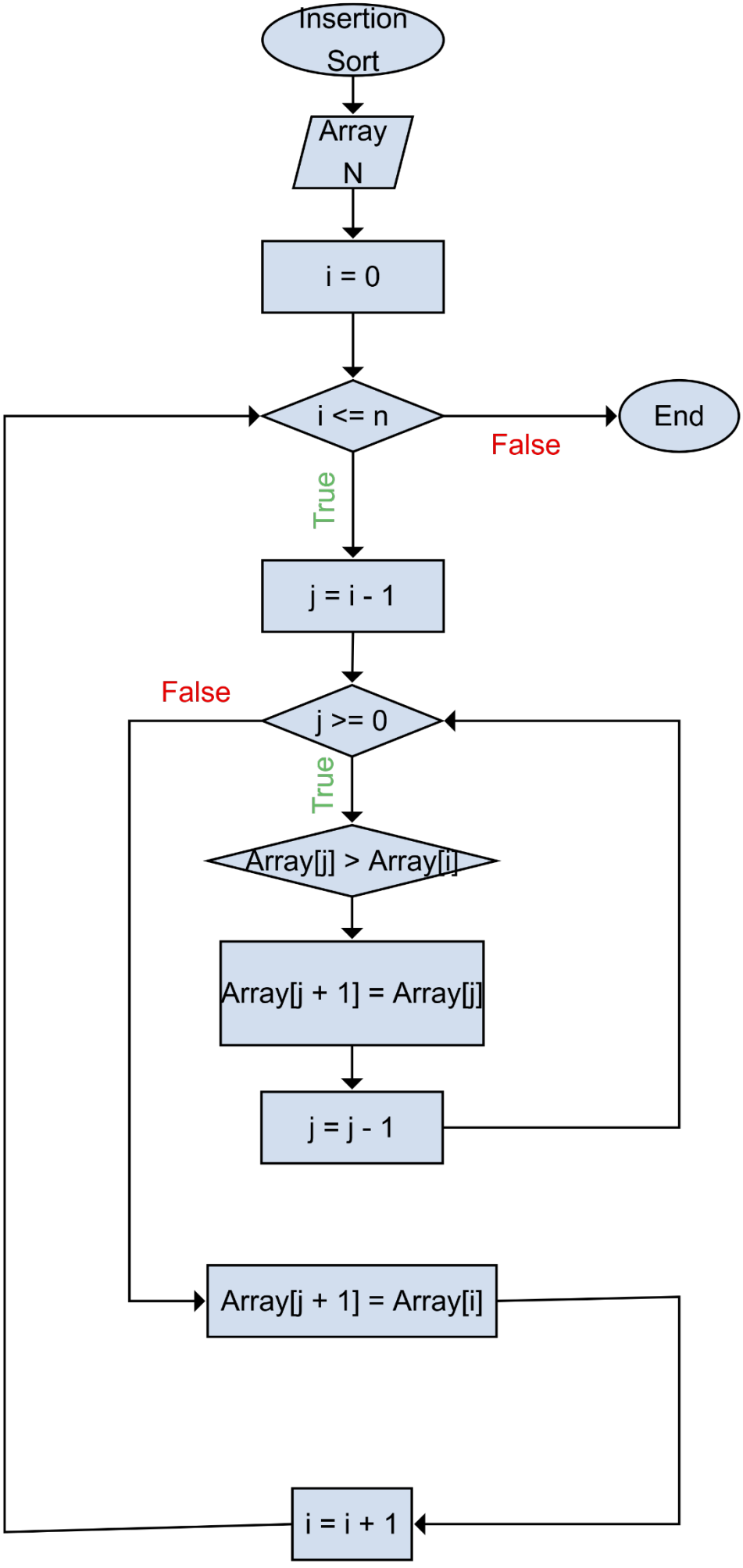
All of this was done to reduce mental load.

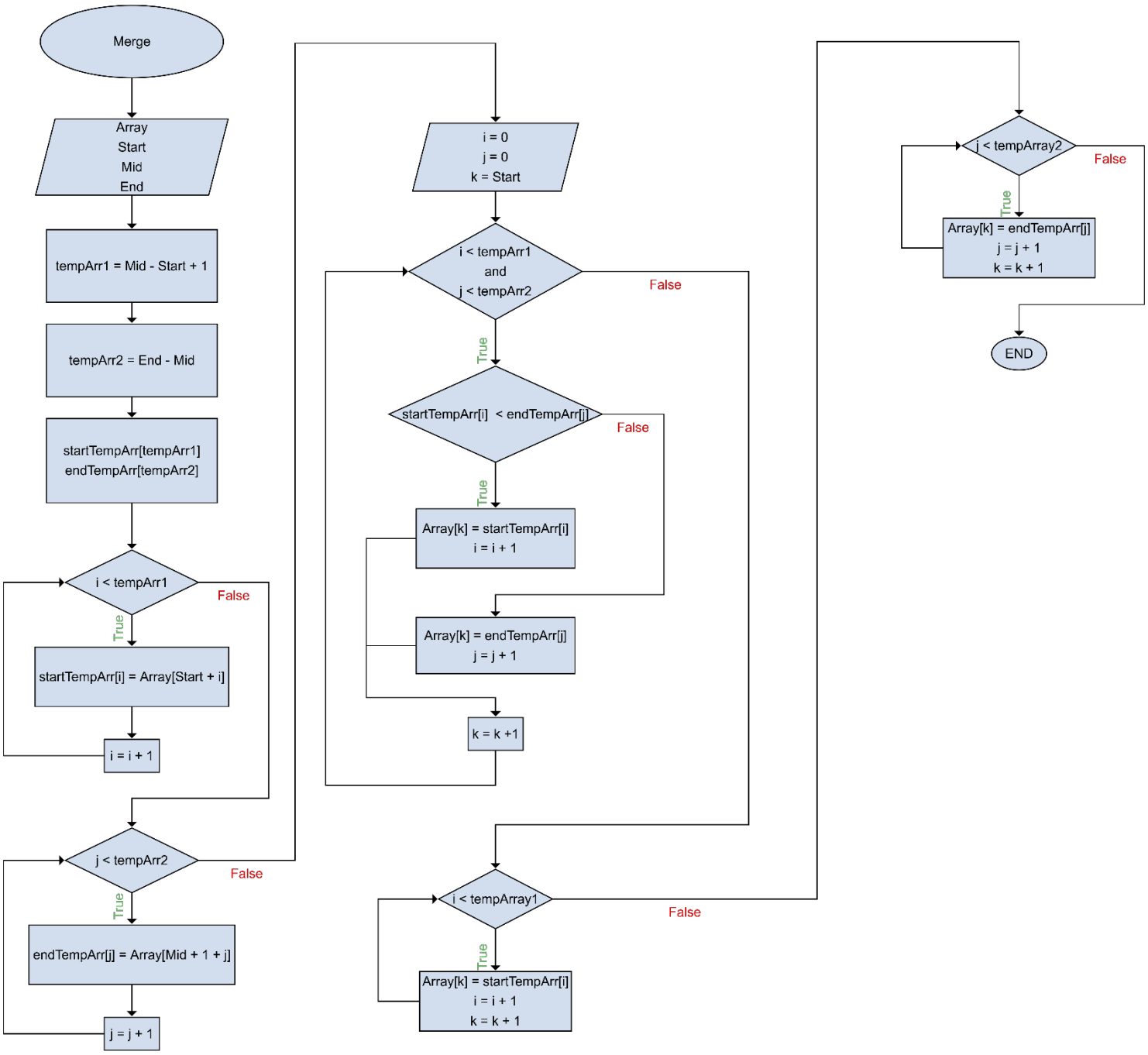
Users would navigate the menus through the entry of numbers, when provided a list, this also reduces mental load and prevents things like buffer overflows if a user entered too much data as a string.

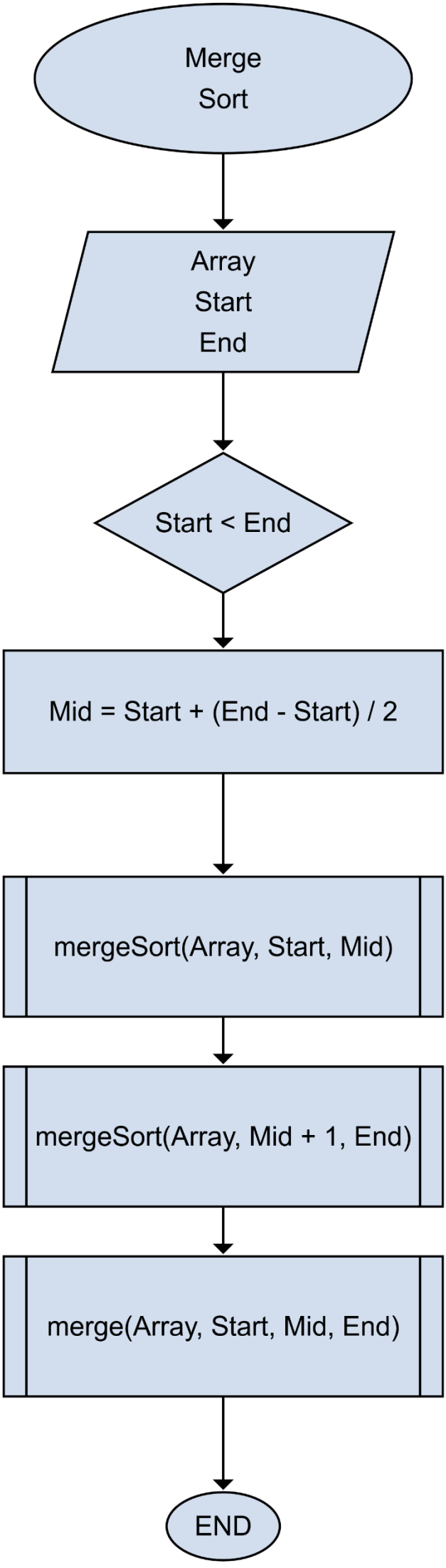
Conclusion

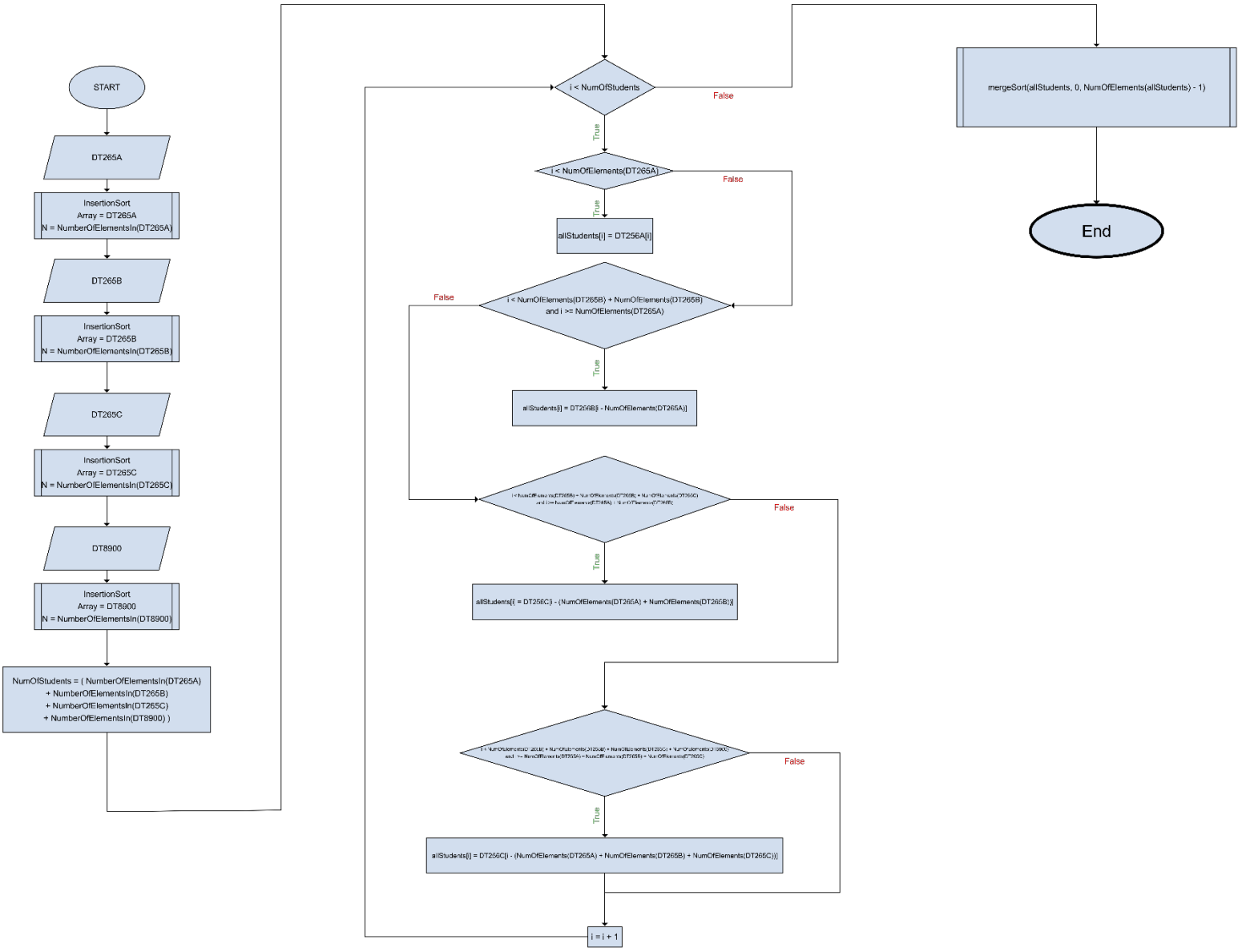
This assignment was a very enjoyable, and complex project which tested my knowledge of multiple sorting and searching algorithms, DMA in C, linked lists and more.

It is also my largest C project to-date, and taught me invaluable skills on the creation, structure, and maintenance of large-scale C projects.


Insertion Sort
Array N
i = 0
i <= n
End
False
True
j = i - 1
False
j >= 0
True
Array[j] > Array[i]
Array[j + 1] = Array[j]
j = j - 1
Array[j + 1] = Array[i]
i = i + 1


Merge
Array
Start
Mid
End
tempArr1 = Mid - Start + 1
tempArr2 = End - Mid
startTempArr[tempArr1]
endTempArr[tempArr2]
i < tempArr1
False
True
startTempArr[i] = Array[Start + i]
i = i + 1
j < tempArr2
False
True
endTempArr[j] = Array[Mid + 1 + j]
j = j + 1
i = 0
j = 0
k = Start
i < tempArr1
and
j < tempArr2
False
True
startTempArr[i] < endTempArr[j]
False
True
Array[k] = startTempArr[i]
i = i + 1
Array[k] = endTempArr[j]
j = j + 1
k = k +1
i < tempArray1
False
True
Array[k] = startTempArr[i]
i = i + 1
k = k + 1
j < tempArray2
False
True
Array[k] = endTempArr[j]
j = j + 1
k = k + 1
END


Merge
Sort
Array
Start
End
Start < End
Mid = Start + (End - Start) / 2
mergeSort(Array, Start, Mid)
mergeSort(Array, Mid + 1, End)
merge(Array, Start, Mid, End)
END


START
DT265A
InsertionSort
Array = DT265A
N = NumberOfElementsIn(DT265A)
DT265B
InsertionSort
Array = DT265B
N = NumberOfElementsIn(DT265B)
DT265C
InsertionSort
Array = DT265C
N = NumberOfElementsIn(DT265C)
DT8900
InsertionSort
Array = DT8900
N = NumberOfElementsIn(DT8900)
NumOfStudents = (NumberOfElementsIn(DT265A)
+ NumberOfElementsIn(DT265B)
+ NumberOfElementsIn(DT265C)
+ NumberOfElementsIn(DT8900))
i < NumOfStudents
False
True
i < NumOfElements(DT265A)
False
True
allStudents[i] = DT256A[i]
i < NumOfElements(DT265B) + NumOfElements(DT265B)
and i >= NumOfElements(DT265A)
False
True
allStudents[i] = DT256B[i - NumOfElements(DT265A)]
False
True
allStudents[i = DT256C[i - (NumOfElements(DT265A) + NumOfElements(DT265B))]
False
True
allStudents[i] = DT256C[i - (NumOfElements(DT265A) + NumOfElements(DT265B) + NumOfElements(DT265C))]
i = i + 1
mergeSort(allStudents, 0, NumOfElements(allStudents) - 1)
End