Algorithm Design Assignment – Brian Willis

Part 1 – Sorting and Merging

a. A strong contender for the sorting component is Merge Sort. This is because it has an efficiency of n(log n), which is good for large volumes of data and also because it already incorporates the merging of lists, meaning it could potentially be easily adapted to facilitate the merging of the multiple lists of students required in this assignment.

Another possibility is Quicksort, since it often proves to be a fast solution. However, this is not guaranteed and the worst-case scenario actually places this method’s efficiency at n2. In addition, this sorting algorithm requires more moving around of elements for each step compared to Merge Sort, meaning it could require more processing, especially for the large lists which would have to be handled for this assignment.

For merging the lists, one solution would be to use the same method of merging as is used in Merge Sort, which involves adding the lowest out of the current values (of either the student numbers or names) in the source lists to the current position in the new list and repeating until the source lists are exhausted. This will create a fully merged list ordered in a somewhat logical fashion. This is a relatively simple algorithm, making use of one loop, giving it a complexity of n.

Another option for merging is to simply place the sorted lists one after another in the new list. This very simple solution would be barely affected by the volume of data it would handle, performing essentially the exact same amount of work each time. This means the Big O would be 1. If the start of each section were noted and accessible to a searching program, it could also potentially allow increased speed when searching. However, it would not create a completely sorted list, meaning that if it were required to find an element without knowing which section it would be in, the search could take up a large amount of processing and time.

I decided to use Merge Sort for the sorting due to its efficiency and suitability for large volumes of data. For merging, I chose the solution which adds the lowest out of the current values in the source lists because even though it is not particularly fast, it should allow for a consistent level of performance when searching the resulting list.

b. There would be some advantages to using a linked list in this situation. It would minimise the need to move elements around in the lists while sorting when compared to using an array. This should help to optimise the speed of the sorting. In a linked list, the data is not necessarily stored in a logical order. Instead, each element contains the address of the next element, meaning they are linked sequentially. Unfortunately, this could complicate the searching component of the assignment if it were used.

An array could also be considered. The main advantage of using an array would be easier access to any given element without having to follow a sequence of addresses. An array holds the data in the correct sequence, with an element’s position in memory determining its place in the array. This sometimes leads to a disadvantage when sorting large lists, since to insert an element into the middle, all elements after it must be shifted forward. However, this would be less of a problem when using merge sort, since this algorithm essentially builds the array from scratch anyway. In addition, searching with an array is quite straightforward compared to linked lists

I decided to use arrays to store the data chiefly because of their ease of access, since searching will make up an important part of the assignment.

c. For this solution, it is assumed that for student records across all colleges, the first and second names, student number and college are available. The records are expected to be supplied in one list (in array form) for each different college. It is also assumed that there will not be duplicate student numbers due to them coming from different colleges.

The solution will use merge sort to sort the three lists separately so that they can be merged into one array easily, using the student’s surname to order them, since student number conventions will be inconsistent. When there are multiple students with the same surname, they will be sorted using their first name and any students which also have matching first names will be sorted using their student number. This comparison will be handled by a function. For each array, the function will recursively split them into smaller portions until there is just one element, at which point they will be reinserted into the previous level in a sorted manner using the merging algorithm. Once the three lists are individually sorted, the merging algorithm will be applied to them again to sort them all into one list.

The methods for merging the lists during sorting and when merging the different lists into one differ in how they will be implemented and how many lists of data they are given, but merge on the same principle. They work by receiving the sorted data to be merged and starting at the beginning of each list. The lowest out of the current values is added to the new array and the algorithm moves on to the next element in the array which the chosen value came from. This is repeated until the source lists have been exhausted. The version for use during sorting accepts one array as input, with values showing it the locations of the areas to be merged, while the version for merging the complete lists accepts three arrays, along with their respective sizes and merges them all from start to finish.

The merge sort algorithm, when coupled with this merging algorithm is considered to have an complexity of n(log n) [1]. The additional merge will be outside the running of the sorting algorithm and, as such, should have no effect on the program’s complexity.

d. Psuedocode Algorithms:

Mergesort(A, low, high)

IF low = high

Return

ELSE

Mid = (low + high) / 2

Mergesort(A, low, mid)

Mergesort(A, mid + 1, high)

Merge2(A, low, mid, high)

ENDIF

END

Merge2( A, low, mid, high)

P1=low

P2=mid+1

Count=0

WHILE P1<mid+1 OR P2<high+1

IF Compare(A[P1], A[P2])=0 AND P1<mid+1 AND P2<high+1

New[Count]=A[P1]

Count = Count+1

P1 = P1+1

ELSE IF P1<mid+1 AND P2=high+1

New[Count]=A[P1]

Count = Count+1

P1 = P1+1

ELSE IF Compare(A[P2], A[P1])=0 AND P2<high+1

New[Count]=A[P2]

Count = Count+1

P2 = P2+1

ELSE IF P1=mid+1 AND P2<high+1

New[Count]=A[P2]

Count = Count+1

P2 = P2+1

ENDIF

ENDWHILE

Count=0

WHILE Count<high+1

A[count+low]=New[count]

Count=count+1

ENDWHILE

END

Merge3(A, sizea, B, sizeb, C, sizec)

ia=0

ib=0

ic=0

Count=0

WHILE ia<sizea OR ib<sizeb OR ic<sizec

IF Compare(A[ia], B[ib])=0 AND ia<sizea AND ib<sizeb

IF Compare(A[ia], C[ic])=0 AND ic<sizec

New[count]=A[ia]

Count=count+1

Ia=ia+1

ELSE IF IF ia<sizea AND ic=sizec

New[count]=A[ia]

Count=count+1

Ia=ia+1

ENDIF

ELSE IF ia<sizea AND ib=sizeb

IF Compare(A[ia], C[ic])=0 AND ic<sizec

New[count]=A[ia]

Count=count+1

Ia=ia+1

ELSE IF IF ia<sizea AND ic=sizec

New[count]=A[ia]

Count=count+1

Ia=ia+1

ENDIF

ELSE IF Compare(B[ib], C[ic])=0 AND AND ib<sizeb AND ic<sizec

New[count]=B[ib]

Count=count+1

Ib=ib+1

ELSE IF ib<sizeb AND ic=sizec

New[count]=B[ib]

Count=count+1

Ib=ib+1

ELSE

New[count]=C[ic]

Count=count+1

Ic=ic+1

ENDIF

ENDWHILE

Write new array to file

END

Compare(recA, recB)

Count=0

WHILE end of recA.surname is not reached AND end of recB.surname is not reached

IF recA.surname[count]>recB.surname[count]

Return(1)

ELSEIF recA.surname[count]< recB.surname[count]

Return(0)

ENDIF

Count=count+1

ENDWHILE

WHILE end of recA.firstname is not reached AND end of recB.firstname is not reached

IF recA. firstname[count]> recB. firstname[count]

Return(1)

ELSEIF recA. firstname[count]< recB. firstname[count]

Return(0)

ENDIF

ENDWHILE

WHILE end of recA.studentnumber is not reached AND end of recB. studentnumber is not reached

IF recA. studentnumber [count]> recB. studentnumber [count]

Return(1)

ELSEIF recA. studentnumber [count]< recB. studentnumber [count]

Return(0)

ENDIF

Count=count+1

ENDWHILE

END

Part 2 – Searching

a. A straightforward solution for handling students who have attended more than one of the colleges is to notify the user about potential duplicate entries. Because the records were sorted so alphabetically according to student names, all records of students with the same name will be adjacent in the array to the record which has been found, meaning it would be easy to inspect these records. When the searching program finds a student record, any records with the same name could be searched and if there were one from a different college, a notification could be presented to the user, informing them of the possible duplicate.

If duplicate records had to be merged into one, the administrator could be given the ability to specify the two student numbers of the student in question. The program would search for these in the file (an error message would display in the event of both being found, but with non-matching student names) and upon finding them, modify the college record in one of the records to represent both of the colleges the student has attended. The other could have the values modified so that future searches would not ever match with it and also so that a deletion program could easily find it.

To perform the searching itself, the binary search would be a suitable option. This works by comparing the middle element in the list to the search term. If it does not match, it calls itself, passing the middle as either the new minimum or maximum value, depending on whether it was before or after the search term. This recursive algorithm has a complexity of n(log n). It would be well suited because it works roughly the same regardless of where in the list the search term is.

An alternative to this could be linear search. This is a much simpler method than binary search. It goes through each element in the list one by one iteratively until it either finds the search term or reaches the end. This means it has a complexity of n, suggesting it would be less suited for the large size of this problem compared to binary search. However, there are some merits to using linear search. Its simplicity means that it is easy to implement and while it may consume time while operating, it will not require as much memory or processing power as the recursive binary search.

I selected to use the binary search algorithm, taking a student’s full name as the search term in conjunction with displaying a notification of possible duplicate students. Providing such a warning is simpler and makes the algorithm more self-contained than relying on manually merging the records, which would require prior knowledge of their presence. Using binary search will ensure consistent speed wherever a record may be located in the file. It will need to differ slightly from normal implementations, since often there will be more than one student surname, the primary search term. When this occurs, the partially matching elements will instead be searched with linear search, since a more straightforward algorithm is more suited for a small number of elements.

b.

binarysearch(A, searchsurname, searchfirstname, min, max)

mid=(low+high)/2

IF A[mid].surname=searchsurname

i=mid

IF A[mid].firstname>searchfirstname

WHILE A[i-1]>=searchfirstname

i=i-1

ENDWHILE

ELSE IF A[mid].firstname<searchfirstname

WHILE A[i]<searchfirstname

i=i+1

ENDWHILE

ELSE IF A[mid].firstname=searchfirstname

WHILE A[i-1]=searchfirstname

i=i-1

ENDWHILE

ENDIF

IF A[i].firstname<>searchfirstname

PRINT Student not found

ELSE

WHILE A[i].firstname=searchfirstname

PRINT A[i].studentnumber

PRINT A[i].firstname

PRINT A[i].surname

PRINT A[i].college

IF A[i+1].firstname=searchfirstname AND A[i].college<>A[i+1].college

PRINT Warning: Potential duplicate student records

ENDIF

i=i+1

ENDWHILE

ENDIF

ELSE IF compare (A[mid].surname, searchsurname)=1

binarysearch(A, searchsurname, searchfirstname, min, mid)

ELSE

binarysearch(A, searchsurname, searchfirstname, mid, max)

ENDIF

END

References:

1. Melhorn K, Sanders P, Algorithms and Data Structures: The Basic Toolbox, 1st ed, Berlin, Springer-Verlag; 2008