Algorithms Assignment

DT228-1/ PROG1210: Algorithms Design and Problem Solving

Stephen Pearson

Contents

Part 1 Pseudocode: Merging and Sorting Student Lists

```
//Created  Four .CSV files with the following data columns:
char  firstName [20]
char surname    [20]
char dateOfBirth [12]
char studentNum [12]
char gender [12]

//#Define used to replace the number of students in each course
DT265A [12]      - Part Time
DT265C [10]      - Part Time
DT265B [14]      - Full Time
DT8900 [06]      - Full Time
ALL_STUDENTS [DT265A+ DT265C+ DT265B+ DT8900] // Total of students =
42

//Create a structure defining each student record
char  firstName [20]
char surname    [20]
char dateOfBirth [12]
char studentNum [12]
char gender [12]
char course [12] // to be appended during the read process

//Declare an array of structures to store our records
A[ALL_STUDENTS]

Read in file “DT265A.csv” to A[0-11]
Append course, “DT265A” to A[0-11]

Read in file “DT265C.csv” to A[12-21]
Append course, “DT265C” to A[12-21]

Read in file “DT265B.csv” to A[22-35]
Append course, “DT265B” to A[22-35]

Read in file “DT8900.csv” to A[36-41]
Append course, “DT8900” to A[36-41]
```

```
// Sorting the combined list of student records by surname
If ALL_STUDENTS < 40
      InsertionSort(A)
else  MergeSort(A,  0, ALL_STUDENTS -1)

//See MergeSort function below for Merge sorting algorithm

//See InsertionSort function below for Insertion sorting algorithm

//Output the sorted array
for i=0; i<ALL_STUDENTS; i++
      Print A[i]
```

//MergeSort Function

```
Merge_Sort(A, int low, int high)

//Declare temporary array used for sorting
TempA[ALL_STUDENTS]
//Declare counter variables
int leftIndex, rightIndex, combinedIndex, i

//Divide the array recursively until n is a single element
If n < 2

      Return
      Else
      mid = (low + high ) / 2
      Merge_Sort (A , low, mid)
      Merge_Sort (A, mid+1, high)
      Merge(A, low, mid, high)

// Function to merge the divided lists
 Merge(A, low, mid, high)

leftindex=low
rightindex=mid+1
combinedindex = low

while leftindex <= mid AND rightindex <= high
      if A.Surname[leftindex] <= A.Surname[rightindex]

            tempA[combinedIndex] = A[leftindex]
            combinedIndex++
            leftIndex++
      else
            tempA[combinedIndex] = A[rightindex]
            combinedIndex++
            rightIndex++


If leftindex = mid + 1
      While rightindex <= high
            tempA[combinedIndex] = A[rightindex]
            combinedIndex++
            rightIndex++
```

```
else
      while leftindex <=mid
            tempA[combinedIndex] = A[leftindex]
            combinedIndex++
            leftIndex++

For i=low to i<=high do
      A[i] = tempA[i]
```

```
//InsertionSort Function
InsertionSort(A)
int i, j

tempA;

for(i=1; i < ALL_STUDENTS-1; i++)


      j = i;

      while( j > 0 && A[j-1].Surname > A[j].Surname )

            tempA = A[j]

            A[j] = A[j-1]

            A[j-1] = tempA

            j--

return 0
```

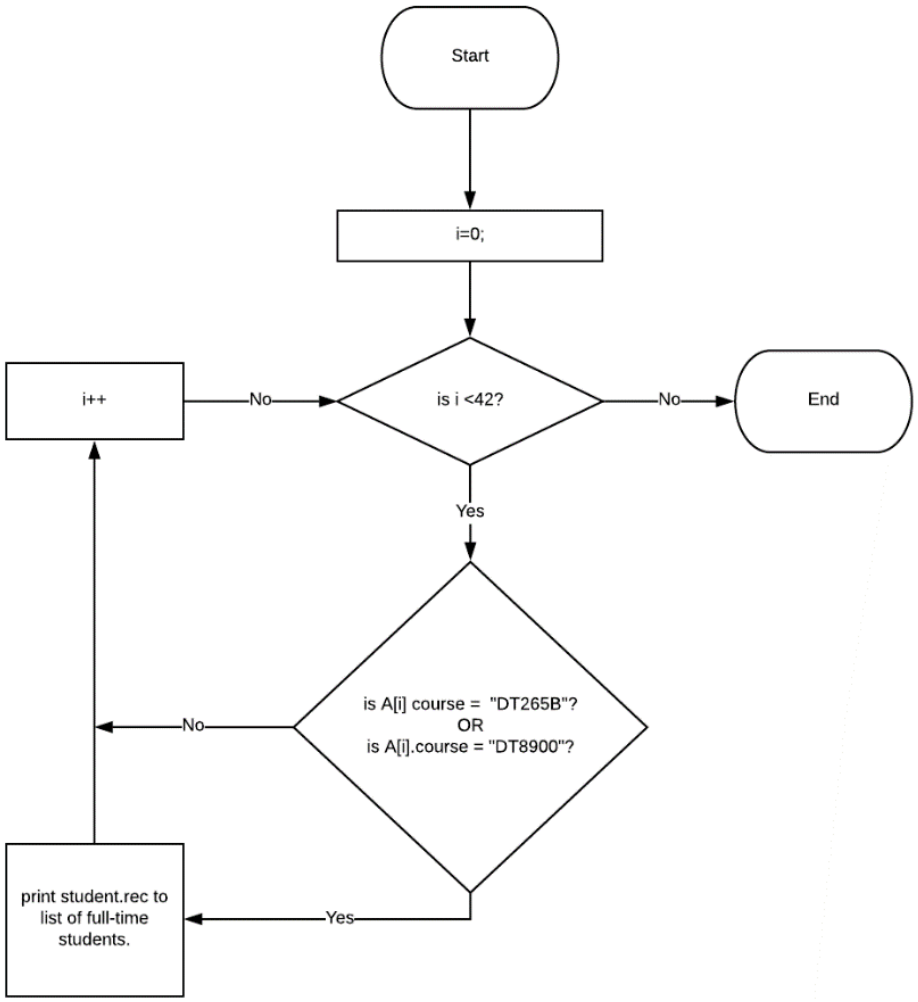
//Big O

The performance of this algorithm will change once the number of elements in the array exceeds 40 and the MergeSort function is used. This ensures that for large data sets, the complexity of the algorithm will not exceed a worst case performance of O(n log n). This is due to the divide and conquer nature of the algorithm, this allows the algorithm to scale up well. The division of the array into smaller lists requires a complexity of O(log n) while the merging process will occur within O(n). It is a stable sort which will not vary depending on whether the given array is nearly sorted. For smaller data sets (in the case of this algorithm the cutoff point is 40), the Insertion sort algorithm will be used. While the difference in performance between Merge and Insertion sort is smaller at this level, there is a potential gain for the insertion sort if the dataset is nearly sorted.

The big O of this combined sorting algorithm in the worst case in practice would be O(N^2). This would be the performance of the algorithm in the case if the array of elements was below the size of 40 and reversed in order. In the case of nearly sorted set of data, the number of swaps required would increase the performance of the algorithm up to a best case of O(n) comparisons where only a single swap might be required.

Part 2 Flowchart: Linear search for finding all Full-time students

Assuming an array of structures sorted by Surname.



Big O:

This linear search algorithm has a complexity of N as it simply passes through the entire data set with a single loop. The complexity of the algorithm will scale in line with the size of the element set it is processing. This makes it suitable for smaller arrays such as our problem but would cause issues with large set of data.

Part 3 Pseudocode: Searching For A Specific Student by Surname

```
//Declare searchKey string to store the surname of the student the user
is searching for
char searchkey[20]
//Declare counters for the linear duplicate search
int resultIndex , i, j

while searchKey != "exit"
      Prompt user ("Enter student surname or 'exit" to quit)
      Read in searchKey
      //call binarySearch function (see page below)
      resultIndex =  binarySearch (A , 0 , ALL_STUDENTS-1 , searchKey)

      if resultIndex == -1
            Print "Student not found"
            Return resultIndex
      else
            //checking for all matching surnames
            i = resultIndex
            j = resultIndex
            //loop through elements until they no longer match searchKey
            while A[i] == searchKey
                  i--

            //once loop has ended increment i by 1 to the position of
            the first match
            i = i + 1

            while A[j] == searchKey
                  j++

            //once loop has ended decrement j by 1 to the position of
            the final match
            j = j - 1

            //Output results of the search
            for resultIndex = i , resultIndex < j + 1; resultIndex ++
                  Print A[resultIndex]
```

```
//Recursive binarySearch function

binarySearch( A, low, high, searchKey)
int middle = (low + high) / 2

while low <= high
      //base case if match is found
      if A[middle].Surname == searchKey
            return middle

      // if searchKey later in the alphabet, call binarySearch from
      middle+1 up to high
      else if A[middle].Surname <= searchKey
            return binarySearch ( A, middle+1, high, searchKey)
      else
      // if previous in the alphabet, call binarySearch again from low
      to middle-1
            return binarySearch ( A, low, middle -1 , searchKey)
//failure condition in case element is not found
return -1
```

Big O

The performance of the binary search itself in the worst case will be a complexity of O(log n). As with the merge sort algorithm, this is due to the divide and conquer nature of the algorithm, with the array discounting half of the array with each pass. In the case where the student being searched for lies directly in middle of the array, the best case performance will be O(1). It is important to note that the binary search can only be used in this case because the data in the array has already been sorted by surname.

Once a match has been found a linear search is used to locate any values which may match student's surname we are searching for. In this case this adds another layer of complexity to the searching algorithm, with the number of existing duplicates plus two (the loop in each direction will exceed the bounds of the duplicate set in each direction by one). Thus the final worst case performance of the algorithm will be O(log N) + (D + 2) where D is the set of surnames matching our search.