(S1-21\_DSECCZG519)

(Data Structures and Algorithms Design)

Academic Year 2021-2022

**Assignment 2 – PS05 - [ Confectionery]**

**Design Document**

**Contribution Table:**

| **Sl. No.** | **Name** | **ID NO** | **Contribution %** |
| --- | --- | --- | --- |
| 1 | Shehza Fathima | 2021C104174 | 100% |
| 2 | Nithin Krishnan | 2021C104176 | 100% |
| 3 | Jinju Nankandiyil | 2021C104118 | 100% |

**Developed Functionalities:**

| **Functionalities Expected** | **Implemented using Functions** |
| --- | --- |
| Function that inturn calls functions that prepare confectioneries list and sorts them | prepareConfectioneries(inputfile) |
| Creates confectioneries list of the given size and initializes with entries like: [ [‘’, 0], [‘’, 0] ] | initConfectioneries(size) |
| Adds the chocolate names / counts into the confectioneries list based on type (name / count) | addToConfectioneries(itemArray, itemType) |
| Sorts the confectioneries (based on incr param in ascending or descending order) using divide and conquer recursive methodology (Merge Sort) internally it calls the merge() function once the array has been split into atomic elements | orderConfectioneryDnQ(confectioneries,incr) |
| Sorts the confectioneries using the iterative approach of merge sort. It internally calls the merge srt function. Inr: True - sorts in ascending order, else sorts in descending order | orderConfectioneryIncr(incr) |
| Writes the sorted output of both the recursive and iterative merge sort approaches to the “outputPS05” file. | writeConfectionery |

**Data Structure Used:**

The given problem can be solved using a two dimensional list as an optimum data structure. For handling of confectioneries we used a **two dimensional Array**, implemented using **2-D List** in python, the representative code structure is as follows.

[

['Campco', 278],

['Amul', 231],

['M&M', 156],

['Nestle', 123],

['Mars', 99],

['Parle', 89],

['Cadbury', 60]

]

Where,

* 0th index in the array Item holds the name of the Chocolate brand
* 1st index in the array Item holds the count of Chocolates

**Assumptions:**  
We assume the input format is given in the complete form. That is, the implementation assumes that the data is not blank and has counts corresponding to the given number of chocolate brands. Basic validation of trimming has been implemented.

**Runtime Analysis:**

The following section gives the algorithm for each of the four main functions used in the assignment along with their runtime analysis:

1. **initConfectioneries(size):** This function creates the confectioneries list and initializes it with empty values [‘’, 0].

**Algorithm:**

* 1. Initial count = 0
  2. For item count till size
     1. Initialize the confectioneries list by [‘’, 0]
     2. count = count + 1

**Complexity:**

The function executes the for loop till the length of the confectioneries list. Hence if n is the number of candidates, the worst case complexity of the function is **O(n)**

1. **addToConfectioneries(itemArray, itemType):** This function adds the confectioneries brands into the confectioneries list and the count based on the itemType, if ‘CHOC’ it adds the brand name else adds the count to the confectioneries list.

**Algorithm:**

* 1. Initialize
     1. i = 0
     2. Split the items in items array using the ‘/’ delimiter
  2. For each split item
     1. Remove preceding and trailing space and extra characters
     2. Depending on if itemtype is chocolate name or count
        1. Add confectionery name in confectioneries[i][0] index for itemtype == ‘CHOC’ (chocolate name)
        2. Add confectionery count in confectioneries[i][1] index for itemtype != ‘CHOC’ (chocolate count)
     3. Increment i by 1

**Complexity:**The function uses a single iteration. Hence for worst case ‘n’ is the number of confectioneries **O(n)** is the complexity.

1. **orderConfectioneryIncr():** The function calls the ‘iterMergeSort()’ function underneath and passes the confectioneries list and the order in which the confectioneries need to be sorted.  
   1. **iterMergeSort(cnfList, order):** Applies merge sort to the confectioneries list based on the order passed as parameter to the function

**Algorithm:**

* 1. Initialize
     1. Width = 1
     2. n = size of confectioneries list
  2. while (width < n)
     1. Initialize left = 0
     2. while (left < n)
        1. Calculate right index as
           1. right = minimum( left + ((width \* 2) - 1), n - 1)
        2. Calculate middle index as
           1. middle = floorValue( (left + right) / 2)
        3. if (width > floor(n / 2) ):
           1. Update middle = right - (n % width)
        4. Call iterMerge(cnfList, left, middle, right, isIncrOrder)
        5. Update left index:
           1. left = left + width \* 2
     3. Update width
        1. width = width \* 2
  3. Finally return the confectioneryList

**Complexity:**

The function uses nested iterations using 2 while loops. The outer loop executes for log n times as “width = width \* 2” the iterative parameter is being multiplied by 2. And the inner loop executes for n times including the function call for merging the inputs. Hence for worst case ‘n’ is the number of confectioneries, the worst case complexity of the function is **O(n x log n)**

* 1. **iterMerge(self, confList, leftIndex, middleIndex, rightIndex, isIncrOrder):** The merge function iteratively merges the left and right subtree after it has done sorting of the respective arrays.

**Algorithm:**

* 1. Initialize
     1. n1 = middleIndex - leftIndex + 1
     2. n2 = rightIndex - middleIndex
     3. Create the left and right sub arrays
        1. LeftSubAry[i] ← confList[0] to confList[leftIndex + i]
        2. RightSubAry[i] ← confList[0] to confList[middleIndex + i + 1]
     4. I = 0, j = 0, k = 0
  2. Sort and Merge the arrays
     1. while i < n1 and j < n2 do:
     2. Sort the left, right sub arrays and merge
        1. if ( ( isIncrOrder && ( LeftSubAry[i][1] > RightSubAry[j][1]) )
        2. || ( not isIncrOrder && (LeftSubAry[i][1] < RightSubAry[j][1] ) ) ):
           1. Assign confList[k] = RightSubAry[j]
           2. Increment j, j = j + 1
        3. Else:
           1. Assign confList[k] = LeftSubAry[i]
           2. Increment i, i = i + 1
        4. Increment k, k = k + 1
  3. If (i < n1):
     1. Copy Left sub array elements to confectioneries list
     2. confList[k] ← LeftSubArray[i]
  4. If (j < n2):
     1. Copy Right sub array elements to confectioneries list
     2. confList[k] ← RightSubAry[j]

**Complexity:**

The function uses a single while loop for the merge functionality. The worst case ‘n’ is the number of elements in the left subtree and right subtree, the worst case complexity of the function is **O(n)**

**Overall Complexity of Merge Sort:** Hence the overall complexity for the merge sort using iterative approach is **O(n)** executed **O(log n)** times. That is **O(n log n)**

1. **orderConfectioneryDnQ(conf, isIncrOrder):** This function has two parts: the splitting part and then call to the sort and merge function.  
   1. **orderConfectioneryDnQ(conf, isIncrOrder):** The function uses a **recursive** approach for the **merge sort** algorithm. Uses divide and conquer mechanism. The order in which the sort needs to be done is passed as isIncrOrder, a boolean value, True means sort in ascending order and false cause the merge sort in descending order.

**Algorithm:**

1. **Basic condition**
   1. if length of confectioneries list is <= 1, then
      1. return confectioneriesList
2. Calculate the middle Index = floorValue(length of (confectioneriesList) / 2)
3. Recursively call to break down the input to atomic list items
   1. Left = orderConfectioneryDnQ(conf[0 → mid], isIncrOrder)
   2. right = self.orderConfectioneryDnQ(conf[mid → length],isIncrOrder)
4. Sort and Merge the broken atomic list by calling the merge function
5. Finally return output of merge ( left, right, isIncrOrder)

**Complexity:**

The function executes recursively each time dividing the input array by 2 hence T(n/2) + T (n/2) + time for merging the inputs. Then finally calling the merge algorithm which does n comparisons in the worst case. Hence it executes in **O(n log n)** times if n is the number of elements in the confectioneries list.

* 1. **merge(self, left, right, isIncrOrder):** The function sorts the input left subtree and right subtree and merges them into one list and finally returns the sorted list.

**Algorithm:**

1. Initialization
   1. i = 0; j = 0;
   2. Initialize an empty array Temp = []
2. Repeat while (i < length of (left) ) && (j < length of (right) ):
   * 1. If (isIncrOrder && (left[i][1] < right[j][1] ) )   
        || ( (! isIncrOrder) && ( left[i][1] > right[j][1] ) ):
        1. Add to temp, contents of left[i]
        2. Increment i
           1. i = i + 1
     2. Else:
        1. Add to temp, contents of right[j]
        2. Increment j
           1. j = j + 1
3. Copy remaining elements of Left Sub Array to main array
   * 1. temp ← left[i to length]
4. Copy remaining elements of Right Sub Array to main array
   * 1. temp ← right[j to length]
5. Finally return output temp

**Complexity:**

The function does n comparisons in the worst case. Hence if n/2 is the number of elements in the left subtree and right subtree, the algorithm performs n comparisons in total. Hence it executes in **O(n)** times for sorting and merging.

**Overall Complexity of Merge Sort:** Hence the overall complexity for the merge sort using recursive approach is **O(n)** sorts and merges executed **O(log n)** times. Hence total complexity is **O(n log n)**

1. **writeConfectionery(confectionery, type):** The function based on the type - if type is true means iterative approach, writes the line ‘Iterative Solution’ else ‘Divide & Conquer’ to the file. It extracts the confectioneries name and count separately and writes them to the file against the respective approach.

**Algorithm:**

1. Open the file in append mode
   1. Based on sorttype if true:
      1. Add line “Iterative Solution:”
      2. Else Add line “Divide & Conquer:”
   2. Extract the chocolate brand name from confectioneries write to file against ‘Chocolate:’
   3. Extract the confectioneries count against each brand write to file against ‘Number:’

**Complexity:**

The function iterates over the confectioneries list for extracting the list of chocolate brands and chocolate count. Hence if n is the number of chocolate brands and count, the worst case complexity of the function is **2xO(n)** or **O(n).**

Overall the time complexity to sort the confectionaries in worst case will be **O(n log n)**