Programming Assignment 5 (Recursion and Friends)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are going to:

- **do some useless stuff using recursion**,
- **learn recursive binary search, but on a weird array**,
- **find common elements in two sorted arrays, and in the process learn a bit about vector (in C++) or ArrayList (in Java)**
- solve the seemingly useless, but actually very useful problem of **merging k sorted arrays into a single sorted array**, and
- **implement the Mergesort algorithm in a linked list**

To this end, **your task is to implement the following methods**:

- `sumEvenDigits` and `binaryStringsWithMoreOnes` in `Recursion.h`/ `Recursion.java`
- `search` and `maxIndex` in `RotatedBinarySearch.h`/ `RotatedBinarySearch.java`
- `binaryMerge`, `commonElements`, and `kWayMerge` in `MergeSortAndFriends.h`/ `MergeSortAndFriends.java`
- `mergesort` in `LinkedList.h`/`LinkedList.java`

For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not. Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

For each part, you must use recursion (unless stated otherwise). Failing to do so will result in no credit being awarded.

2 Recursion

- `sumEvenDigits` returns the sum of digits in n that are even.

  Hint: This is very similar to the sum of digits example on the notes. You extract the last digit and check whether or not it is even. If it is, then this digit contributes to the overall sum, else it does not.

- `binaryStringsWithMoreOnes` prints all n-length binary strings having more ones than zeroes. If $n = 3$, then all 3-length binary strings are: $000, 001, 010, 011, 100, 101, 110$, and 111. The 3-length binary strings which have more ones than zeroes are: $011, 101, 110$, and 111.

  Hint: Base-case is when length of *str* equals n and number of zeroes is less than the number of ones. In this case, you print the string and return from the function (because you cannot extend it any further). Else if the length of the string is less than n, the recursion rule is to simply extend the string by appending 0 and by 1 separately. Thus, recursively call the function twice – once with arguments: str + "0", $numZeroes + 1$, $numOnes$, and n, and another time with arguments : str + "1", $numZeroes$, $numOnes + 1$, and n.

3 Binary Search a Rotated Array

A rotated array is created as follows. You are given a sorted array, say $[1, 3, 5, 6, 8, 10, 14, 17, 19, 21]$. You splice the array somewhere and attach the second half before the first half. For example, we could splice at the value 8 to create $[8, 10, 14, 17, 19, 21, 1, 3, 5, 6]$; this is as good as rotating the array to the right in a circular way for the desired number of times.

We are given such a rotated array, where all numbers are distinct. Also, assume that the array has actually been rotated (i.e., it is of length at least 2 and is not completely sorted). Our task is to search for a key in a rotated array in $O(\log n)$ time, where n is the array length.

Let's see the main intuition. Focus on the maximum, which is 21 in the above example. If $key \geq$ the first number in the array, then we will binary search the array from the start to the maximum, else we will binary search the array from the maximum to the end. The main objective, therefore, is to find the maximum. Let $lastValue$ be the rightmost value in the array. Let's focus on an index, say x. Observe the following. If $array[x + 1] < array[x]$, then x is the index containing the maximum. If $array[x] < lastValue$, then the maximum lies to the leftside of x, else the maximum lies to the rightside of x. Note that finding the maximum follows a binary-search-like approach.

The pseudo-codes for each function are given next.

Caution: You algorithm must have a complexity of $O(\log n)$. Anything worse, you will be awarded partial credit, even if you get the correct output.

You do not need recursion for the first (i.e., the search) function.

You must use recursion for the second (i.e., finding maximum index) function; you can check the binary search code for ideas.

Algorithm for searching

- Find the index of the maximum value in the array by calling the `maxIndex` function with arguments: the array, its last value, leftmost index, and rightmost index. Call this index $maxInd$.
- If key equals $array[maxInd]$, return $maxIndex$
- If $key \geq array[0]$, then binary search the array from index 0 to $maxIndex - 1$ (both inclusive) and return the index found by binary search
- Else binary search the array from index $maxIndex+1$ to the last index (both inclusive) and return the index found by binary search

Algorithm for Finding maxIndex

- The base-case is when $left$ equals $right$; here, you return $left$.
- Compute $mid = (left + right)/2$
- If value at mid exceeds value at $(mid + 1)$, then maximum is at mid
- Else if value at mid is less than the last value, then the maximum lies to the left side of mid. Hence, recursively call and return the function with arguments: $array, lastValue, left$, and $mid - 1$
- Else the maximum lies to the right side of mid. Hence, recursively call and return the function with arguments: $array, lastValue, mid + 1$, and $right$

4 Friends of MergeSort

Our main goal is to **design an algorithm that merges k sorted arrays of total length N in $O(N \log k)$ time.** This algorithm is especially useful for implementing parallel mergesort by using multiple processors. The algorithm also finds use in sorting numbers on a hard disk directly (known as external sorting).

We will also solve another problem that of finding all common elements in two sorted arrays and returning them. To this end, you are going to use Dynamic Arrays, but not the one that you studied in class, but its C++ and Java implementations.

4.1 Dynamic Arrays and Jagged Arrays

Here, you will use their C++ and Java implementations of dynamic arrays, which are named respectively **vector** and **ArrayList**. Typically, this encompasses use of generics, whereby you can create dynamic arrays of any type (and not just integer). However, you will create integer dynamic arrays here; the syntax is pretty self explanatory on how to extend this to other types (such as char, string, or even objects of a class).

- In C++, to create an integer vector use: **vector⟨int⟩ name;**

  To add a number (say 15) at the end of the vector, the syntax is **name.push_back(15)**.

  Although not needed for this assignment, the following are good to know. To remove the last number, the syntax is **name.pop_back()**. To access the number at a particular index (say 4), the syntax is **name[4]**.

- In Java, to create an integer ArrayList use: **ArrayList⟨Integer⟩ name = new ArrayList⟨Integer⟩();**

  To add a number (say 15) at the end of the array list, the syntax is **name.add(15)**.

  Although not needed for this assignment, the following are good to know. To remove the last number, the syntax is **name.remove(name.size() − 1)**. To access the number at a particular index (say 4), the syntax is **name.get(4)**.

Note that the push_back and add functions are equivalent to *insertAtEnd*, whereas pop_back and remove (with the appropriate index) are equivalent to *deleteLast*.

Jagged Arrays are two-dimensional arrays, where the number of columns vary over each row.

- In Java, a jagged array JA with k rows is created as: **int JA[ ][ ] = new int[k][ ];**
- In C++, a jagged array JA with k rows is created as an array of pointers using the following syntax: **int *JA[k];**

4.2 Task 1: Implement the `binaryMerge` method

This method takes two arrays A and B and their respective lengths $lenA$ and $lenB$ as arguments. It merges A and B into a single sorted array C, and then returns C. This is the first algorithm on the mergesort notes. Recursion is not needed for this part. Once you implement this method, you can run the mergesort function.

Important: In C++, remember to use dynamic allocation when returning an array. So, if you want to return an array X having length 10, it must be declared as **int *X = new int[10];**

4.3 Task 2: Implement the `numCommonElements` method

We want to solve the following problem: given two sorted arrays $A[\,]$ and $B[\,]$ of lengths n and m respectively, we want to find the elements that are common to both the arrays (without repeated counting of the same element) and return the common elements in a dynamic array.

For example, the arrays $A[\,] = \{7, 13, 19, 20, 22, 22, 37, 37\}$ and $B[\,] = \{7, 14, 17, 22, 28, 31, 37, 37, 37, 43\}$ both have 7, 22 and 37. (Note that we count 22 and 37 only once.) On the other hand, $A[\,] = \{7, 13, 19, 20, 22\}$ and $B[\,] = \{14, 17, 21, 28\}$ have nothing in common.

One algorithm for solving this problem is as follows: pick each number in array A and then scan array B to verify if the number exists. It is easy to see that the Big-O complexity of this procedure is $O(mn)$. You can improve this to $O(m \log n)$ or $O(n \log m)$ by applying a binary search approach – pick a number from one array and binary search the other one; typically, you would binary search the larger of the two arrays. Although binary search is a substantial improvement, based on the merge principle, we will devise an $O(m + n)$ time algorithm, which is as good as it can get.

Algorithm for finding the number of common elements

- Create a dynamic array, i.e., an integer vector (in C++) or ArrayList (in Java)
- Initialize $a = 0$ and $b = 0$
- Use a while loop that runs as long as $a < lenA$ and $b < lenB$. Within the loop:
  - If $A[a] < B[b]$ then you increment a
  - Else If $A[a] > B[b]$ then you increment b
  - Else, do the following:
    - Add $A[a]$ to the dynamic array
    - Increment a
    - Skip all repeated occurrences of $A[a]$ in A (using another loop increment a as long as $a < lenA$ and $A[a] = B[b]$).
- Return the dynamic array

Caution: You algorithm must have a complexity of $O(m + n)$. Anything worse, you will be awarded partial credit, even if you get the correct output. Recursion is not needed for this part.

4.3.1 Task 3: Implement the `kWayMerge` method

This method takes the following as arguments: k sorted arrays in a jagged array **lists** (i.e., each row in the jagged array is a sorted array and there are k rows), the length of each row in another array **listLengths**, and the value of k. The task is to return a single merged sorted array.

Although one can merge row 0 with row 1, then merge the merged row with row 2, and so on, the process is slow. The faster way is to carry out a pairwise merge process, i.e., merge rows 0 and 1, then 2 and 3, and so on. This will effectively halve the number of rows in each round, which will allow us to achieve the desired complexity. More specifically,

Base-Case for k-sorted merge

- If k is 1, then there is a single sorted array (i.e., only one row in the jagged array). Hence, simply return the first (and only) row of the jagged array **lists**.
- If k is 2, then there are two sorted arrays. Merge them using the `binaryMerge` method.

  Hence, you call and return the `binaryMerge` method with the following arguments: the first row of **lists**, the second row of **lists**, the length of first row, and the length of second row. The length of rows are in the **listLengths** array.

Recursive Rule for k-sorted merge

- Declare an integer $newK$ initialized to $(k+1)/2$. This is the new number of sorted lists that you are going to end up with after pairwise merging.
- Create a jagged array $mergedLists$ having $newK$ rows. This jagged array is going to store the pairwise merged lists. Create another array $mergedListLengths$ of length $newK$, which is going to store the lengths of the pairwise merged lists.
- Recall that you will merge pairwise, i.e., $lists[0]$ and $lists[1]$ will merge to create $mergedLists[0]$, $lists[2]$ and $lists[3]$ will merge to create $mergedLists[1]$, and so on. Hence, within a for-loop that runs from $i = 0$ till $k/2 - 1$ (inclusive):
  1. set $mergedListLengths[i]$ to the sum of $listLengths[2i]$ and $listLengths[2i+1]$
  2. set $mergedLists[i]$ to the array obtained by merging $lists[2i]$ and $lists[2i+1]$. You have to use `binaryMerge` method to merge $lists[2i]$ and $lists[2i+1]$.
- After the for-loop above, if k is odd, you have to set $mergedLists[newK-1]$ to $lists[k-1]$ and $mergedListLengths[newK-1]$ to $listLengths[k-1]$. This is because if k is odd then the last row in $lists$, i.e., $lists[k-1]$ does not have a pair to be merged with, and so it simply gets copied into $mergedLists[newK-1]$.
- Finally, recursively call and return $kWayMerge$ with the following arguments: $mergedLists$, $mergedListLengths$, and $newK$.

5 Mergesort a Linked List

Merge-sorting a linked list pretty much uses the same ideas that are needed in merge-sorting an array. The key difference is that you cannot access the middle index (or for that matter any index)

directly. Therefore, you will need to traverse the list to get hold of this indexes. Although this may be prohibitive, recall that in merge-sort, you are allowed to $O(right - left + 1)$ complexity when merging the two sorted halves (i.e., you can afford $O(n)$ time over an entire recursion level of merge-sort). We will see how to achieve that by using a global array called $mergedArray[\,]$ whose length is the same as the length of the linked list to be merge-sorted. Following is the pseudo-code.

MergeSort a Linked List

- If $left \geq right$, return, else do the following.
- Compute $mid = (left + right)/2$
- Assign a place-holder $midNode = leftNode$
- Move the place-holder using a loop from $i = left$ to $i < mid$ such that $midNode$ points to the node at index mid
- Recursively MERGESORT(left, mid, leftNode, midNode);
- Recursively MERGESORT(mid + 1, right, midNode's next, rightNode);
- Initialize 3 variable: $k = left$, $tmp1 = leftNode$, and $tmp2 = midNode's\ next$.
- While ($tmp1 \neq midNode$'s next and $tmp2 \neq rightNode$'s next), do the following:
  - If ($tmp1's\ value < tmp2's\ value$), do the following:
    - assign $mergedArray[k{+}{+}] = tmp1's\ value$
    - move $tmp1$ to its next node
  - Else, do the following:
    - assign $mergedArray[k{+}{+}] = tmp2's\ value$
    - move $tmp2$ to its next node
- While ($tmp1 \neq midNode$'s next), do the following:
  - assign $mergedArray[k{+}{+}] = tmp1's\ value$
  - move $tmp1$ to its next node
- While ($tmp2 \neq rightNode$'s next), do the following:
  - assign $mergedArray[k{+}{+}] = tmp2's\ value$
  - move $tmp2$ to its next node
- Reset $k = left$ and assign a placeholder $tmp = leftNode$
- While ($tmp \neq rightNode$'s next), do the following:
  - set $tmp's\ value = mergedArray[k{+}{+}]$
  - move tmp to its next node