Recursive challenge. Divide and conquer

Recitation 11

In this assignment you will develop your recursion muscle through designing and implementing several recursive algorithms. After all these exercises, you will become a master of recursion and will also excel in Exam 2, which will have several questions of this type.

To save you some typing time, we provide you with the starter code that contains some classes, method headers and tests: starter_code.zip

Part I. Simple recursive exercises [12 points]

For each problem follow the following steps:

1. Think about the simplest possible input for which the solution is computed without recursion. Code this at the start as a ***base case***.
2. Now assume that you know the solution to a subproblem of a smaller size. How would you convert the solution to a subproblem into a solution to a larger problem? Code this as a ***recursive step***.
3. Sometimes you also need to think about how to decrease the problem size in order to make the next recursive call with a smaller input – at each step getting closer to the base case.

These problems might seem too simple. To make your work meaningful and challenging, please follow the following **rules**:

- **You are not allowed to use any loops in your solutions.**
- **You are not allowed to change the signature of a method or pass additional parameters: you should solve the problem as it is given to you.**
- **You cannot use any helper functions: solve the problem with one set of recursive calls.**

Exercise 1. *sumN(int n)*

We start with a simple task similar to factorial. Design a recursive algorithm that takes as an input a positive integer n and returns the sum of int numbers from 1 to n inclusive.

Exercise 2. *sumList* (*Node t*)

Design a recursive algorithm that produces a sum of the data in a linked list of numbers. The linked list is defined by its head node.

Exercise 3. *isSorted* (*Node t*)

This recursive algorithm takes as an input a list of integers (in the form of a head node) and checks if all the elements of the list are sorted. It returns True if all the elements in t are in a non-decreasing order. It returns False otherwise.

Hint: you may need several base cases for this problem: one when the list has only one element remaining, and another when two elements are out of order.

Exercise 4. allStar(String astring)

Design a recursive algorithm which takes as an input String *astring* and returns a new String in which character '*' is inserted between each character of *astring*. Note that the characters are inserted only between the characters of aString. For example if astring="hello", then the output iis "h*e*l*l*o". If astring= "Hi", then the output is "H*i". If astring = "a" the output is "a".

Exercise 5. noX (String astring)

This recursive algorithm returns a new string in which all characters 'x' are removed from *astring*. For example if astring="ex box dox" then the output is "e bo do".

Exercise 6. *mult (int n, int m)*

This recursive algorithm should output the product of the two integers n and m. Since this would be a bit too easy if the multiplication operator * were used, for this problem you are limited to using addition/subtraction/negation operators, along with recursion.

> **Hints**:
> Sometimes it's a good strategy to first solve a simpler problem and then solve the harder part. This is the case with *mult*: first solve the case where both arguments are positive – then deal with the fact that the arguments may be negative. Recall that you cannot add any additional parameters to the function calls.
>
> Check that *mult*() function runs correctly. Try two positive numbers, two negative numbers, and one positive and one negative number as an input.

Part II. Divide and conquer

In this part you are presented with two problems – both are variations of the Binary Search. These problems are challenging in a sense that you need to devise your solutions carefully to preserve the O(log *n*) running time. Please make sure that you discuss your solutions with TAs and have full understanding of the algorithms, before you proceed.

To make sure your algorithms perform at most O(log n) operations, every time you compare a value with the elements of an array – increment the static variable `comparisons`. An example is shown in the solution to Problem 2 provided here: FindDirty.java. Then you should be able not only to test if your algorithms are correct, but also if they have the desired asymptotic running time.

Problem 1. Count in a sorted array [8 points]

You are given a sorted array *A* of *n* integers, and a target element *x*. The goal is to **count** the number of times *x* occurs in *A*. **The goal is to exercise designing a divide-and-conquer algorithm using recursion, not iteration.**

Design an efficient **recursive** algorithm to solve this problem that **runs in time O (log *n*)**. Implement the algorithm according to the method signature in file CountInSorted.java. You can use additional helper methods but the original call should be to the ***`countSorted`*** **`(int [] sortedA, int element)`**.

Before you start coding, consider the following sample inputs:

Example 1:
Input: A = {1, 2, 2, 2, 2, 6, 8, 10}, x = 2
countSorted returns 4.

Example 2:
Input: A = {1, 2, 2, 2, 2, 6, 8, 10}, x =3
countSorted returns 0.

Example 3:
Input: A = {1, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 9}, x =7
countSorted returns 14.

Think: does your solution have the time complexity claimed above for all these inputs?

Problem 2. Search in a 'dirty' array

You are given a long ("infinite') array *A* of *n* characters. The array consists of two parts: the positions from 0 to *m* are filled with normal letters, and starting from *m*+1 and until the end the array is filled with 'junk' characters – let's use '&' to represent 'junk'. The regular letters in the beginning of the array are **sorted**.

You need to devise an efficient algorithm for finding a position of a given (non-junk) letter in *A*. You can use binary search because the letters are sorted.

However to use the binary search you need to know the start and end index of the search interval. We should start with position 0. But the value of *m* is not known, and the size *n* of the array *A* should not be used in your solution. Overall, *m* is several orders of magnitude smaller than *n*.

Your algorithm must run in time O(log ***m***), not O (log ***n***).

Think how you would make it work. Come up with your original ideas before looking at the solution provided here: FindDirty.java.