DIT181 Data Structures and Algorithms: Assignment 2

Maximum points you can get from Assignment 2 is **100 points.** Distribution of points within questions has been given in parentheses below.

- **Criteria to pass "Assignments" sub-course (3.5 HEC)**: From each assignment (i.e. Assignment #1, Assignment #2 and Assignment#3) you have to get minimum 50 points (out of 100 points for Assignments #1and #2, and out of 130 points for Assignment #3).

- **Information about Deliverables:** You will submit a pdf file for the answers to questions that do not require programming. For the answers consisting of implementation in Java, you will submit the java skeleton file that you worked on and completed.

- **Submission Deadline:** 25.02.2018 at 08:00 am in the morning SHARP!!

Recursion

Question 1 During Hands-on Programming Session, you calculated the time complexity of the following code for Fibonacci numbers assuming that each addition is $O(1)$?

```
import java.math.BigInteger;
public static BigInteger fib(int i) {
          if(n <= 1)
                    return BigInteger.ONE;
          return fib(n-1).add(fib(n-2));
}
```

You are now asked to implement a faster version of this method **(10 points)** and give its complexity **(2 points)**.

Question 2 Solve the complexity of the following recurrences using the "Master Method":

- $T(n) = 4T(n/2) + \Theta(n^2)$ **(5 points)**
- $T(n) = 5T(n/2) + \Theta(n^2)$ **(5 points)**

Question 3 During Lecture #7 and Hands-on Programming Session #3, you learned about the pseudocode and implementation of the recursive version of the maximum subarray algorithm. Implement the method `maxInterval()`that finds the sub-array within a given array where elements are in ascending order and difference between the last element and the first element of this sub-array is maximum. Please note that you will need to define an auxiliary method. **(13 points)**

Question 4 Implement the method `median()`, which should return the median element of the array. Use the divide and conquer technique, and the method `partition()`. The method should have $O(n)$ time complexity in the average case, assuming that the order of the elements in the array is random. (Note: $median = a[\lfloor(low + high)/2\rfloor]$) **(15 points)**

Sorting Algorithms

Question 5 In this question, you are asked to implement merge sort **(12 points)** and complete quicksort algorithm (whose implementation you started during hands-on programming session #4.) **(8 points).** Solving this question will require you to work with the `LabSorting.java` file, which you have been working on during Hands-on Programming Sessions #2 and #4. During Hands-on Programming Session #2, you have been shown how to implement bubble sort and you also worked on insertion sort. During Hands-on Programming Session #4, you were asked to start implementation of quicksort algorithm.

Question 6 (30 points) You are asked to do a benchmarking for bubble sort, insertion sort, merge sort and quicksort algorithms, and fill in Table 1, below for the execution time (CPU seconds) it takes for each of those algorithms for each case. Each case is a combination of the array size (e.g., 100 elements or 100000 elements) and how that array is sorted (e.g., sorted in ascending order, sorted in descending order).

BENCHMARK SUMMARY		**Execution Time (CPU seconds)**	
		sorted (ascending order)	**sorted (descending order)**
bubble sort	**array size:** 100		X
	array size: 100000		
merge sort	**array size:** 100		
	array size: 100000		
insertion sort	**array size:** 100		
	array size: 100000		
quicksort (pivot: median)	**array size:** 100		
	array size: 100000		
quicksort (pivot: A[0])	**array size:** 100		
	array size: 100000		

Table 1: Summary of benchmarking study

- Arrays to be sorted will consist of integers x in the range $1 \leq x \leq N$. For instance, if array size N = 100 that is sorted in descending order, then array A = [100, 99, 98, 97, …, 3, 2, 1]. If array size N = 100000 that is sorted in ascending order, then array A = [1, 2, 3, 4, …, 98, 99, 100, …, 900, …, 99999, 100000].
- Each sorting algorithm is supposed to sort the input array in ascending order.
- You will implement two different cases of quicksort algorithm in your benchmarking code.
  - For the case "quicksort (pivot: median)", pivot is the median of all elements in the array.

- For the case “quicksort (pivot: A[0])”, pivot is first element of the array.

- As you can see in Table 1 above, there are 20 different cases and each empty cell in the Table corresponds to a case.
  - For instance, the cell in Table 1 that is marked with “X” corresponds to the case where *bubble sort algorithm is used to sort an array of size 100, that is already sorted in descending order*.
  - For implementation and execution of each case, you will get **1 point**, which sums up to **20 points**.

Further Questions:

In order to get the remaining **10 points**, you will need to answer the following questions, after having completed the benchmarking and filling in Table 1.

- Is there an algorithm, which performed worst in all of the cases? If so, which algorithm? **(2 points)**
- Compare the performance of quicksort algorithm for the following two cases: (1) pivot is the median of all array elements; (2) pivot is the first element of the array, for the case when the input array is sorted in ascending order. **(2 points)**
- Compare the performance of quicksort algorithm for the following two cases: (1) pivot is the median of all array elements; (2) pivot is the first element of the array, for the case when the input array is sorted in descending order. **(2 points)**
- Compare the performance of insertion sort to the rest of the remaining algorithms, for sorting short arrays of size 100? How about when array size is large (i.e., with 100000 elements)? **(2 points)**
- How does performance of merge sort change for different cases? **(2 points)**