ALGORITHMICS - Bachelor of Software Engineering, 2024

LAB GUIDE. SESSION 3

GOALS:

- **Divide and Conquer: recursive models and examples**

1. Introduction

The study of the time complexity and execution time of recursive Divide and Conquer (D&C) algorithms will be addressed, both for the **subtraction** and **division** approaches.

The second part of the lab consists of providing a solution to a specific problem using the D&C technique, measuring the execution times of the algorithm that is implemented and proceeding to the final analysis of the times obtained.

We will continue this lab measuring the times WITHOUT OPTIMIZATION, since in this case of recursive models the JIT seems to produce greater time distortion than in the case of iterative models (seen in the previous sessions). In the tables of times that we will create, if any time exceeds 1 minute, we will indicate Out of Time (OoT).

2. Divide and Conquer by subtraction

You are provided with the following 10 classes that you should try to understand one by one:

`Subtraction1.java` and `Subtraction2.java` classes have an approach by subtraction with `a=1`, which involves a large expenditure of **stack memory**. In fact, it overflows when the problem size grows to several thousands. Fortunately, that type of problems will be better solved with an iterative solution (with loops) than with this solution (subtraction with `a=1`).

`Subtraction3.java` class has an approach by subtraction with `a>1`, which involves a large time of execution (exponential; not polynomial). This means that for a relatively large size problem the algorithm does not end (unfeasible time). The consequence is that we must try not to use "by subtraction" solutions with multiple calls (`a>1`).

YOU ARE REQUESTED TO:

After analyzing the complexity of the three previous classes, you are not asked to make the time tables, but to reason whether the times match the theoretical time complexity of each algorithm.

For what value of **n** do the **Subtraction1** and **Subtraction2** classes stop giving times (aborted)? Why does that happen?

How many years would it take to complete the **Subtraction3** execution for n=80?

Implement a **Subtraction4.java** class with a complexity O(n^3) and then fill in a table showing the time (in milliseconds) for n=100, 200, 400, 800, ... (until OoT).

Implement a **Subtraction5.java** class with a complexity O($3^{n/2}$) and then fill in a table showing the time (in milliseconds) for n=30, 32, 34, 36 , … (until OoT).

How many years would it take to complete the **Subtraction5** execution for n=80?

3. Divide and conquer by division

`Division1.java`, `Division2.java` and `Division3.java` classes have a recursive approach by division, being the first of type $a<b^k$, the second of type $a=b^k$ and the remaining one $a>b^k$.

YOU ARE REQUESTED TO:

After analyzing the complexity of the three previous classes, you are not asked to make the time tables, but to reason whether the times match the theoretical time complexity of each algorithm.

Implement a **Division4.java** class with a complexity O(n^2) (with $a<b^k$) and then fill in a table showing the time (in milliseconds) for n=1000, 2000, 4000, 8000, … (up to OoT).

Implement a **Division5.java** class with a complexity O(n^2) (with $a>b^k$) and then fill in a table showing the time (in milliseconds) for n=1000, 2000, 4000, 8000, … (up to OoT).

4. Two basic examples

`VectorSum1.java` solves the simple problem of adding the elements of a vector in three different ways, and `VectorSum2.java` measures times of these three algorithms for different sizes of the problem.

`Fibonacci1.java` solves the problem of calculating the Fibonacci number of order `n` in four different ways, and `Fibonacci2.java` measures times of these four algorithms for different sizes of the problem.

YOU ARE REQUESTED TO:

After analyzing the complexity of the various algorithms within the two classes, executing them and after putting the times obtained in a table, compare the efficiency of each algorithm.

5. Another task

The problem that is asked to be solved is very well known to us, ordering **n** sortable elements (e.g. of integer type). This problem has three good quasi-linear algorithms (O(n logn)): **Quicksort, Mergesort**, and **Heapsort**. Well, of those three algorithms, two of them are D&C (Quicksort and Mergesort) and will be the objective of the subsequent study that is requested.

YOU ARE REQUESTED TO:

Implement a **Mergesort.java** class that orders the elements of a vector.

Implement a **MergesortTimes.java** class that, after being executed, serves to fill in the following table (analogous to lab 2):

TABLE MERGESORT ALGORITHM

(times in milliseconds and WITHOUT OPTIMIZATION):

We can type "OoT" for times over one minute and "LoR" for times less than 50 milliseconds.

n	***t ordered***	***t reverse***	***t random***
31250			
62500			
125000			
250000			
500000			
1000000			
2000000			
4000000			
8000000			
… (until OoT)			

Fill in the following table, transferring to it the times obtained for the Quicksort in the random case (from lab 2) and those obtained here for the Mergesort, also in the random case. What constant do you get as a comparison of both algorithms?

TABLE MERGESORT vs QUICKSORT

(times in milliseconds and WITHOUT OPTIMIZATION):

We can type "OoT" for times over one minute and "LoR" for times less than 50 milliseconds.

n	***t Mergesort (t1)***	***t Quicksort (t2)***	***t1/t2***
250000			
500000			
1000000			
2000000			
4000000			
1000000			

2000000			
4000000			
8000000			

6. Work to be done

- An `algstudent.s3` **package** in your course project. The content of the package should be the Java files used and created during this session.
- A `session3.pdf` **document** using the course template (the document should be included in the same package as the code files). You should create one activity each time you find a "YOU ARE REQUESTED TO" instruction.

Deadline: The deadline is one day before the next lab session of your group (one week).