Lesson 1: Introduction

CPC UTEC - Lecturers Notes

September 21, 2020

1 Introduction to the Course

1.1 Scope Objectives

This is the *CPC@UTEC* official competitive programming beginners course. In the following 16 weeks we will attempt to introduce you to the world of competitive programming and boost your learning process.

Our main objective is to create a solid foundation for the future members of UTECs CPC team in order to ensure a strong legacy that will be up to the standards of international competitions. Because of this the nature of this lessons will be intensive and will require a lot of your time outside of the classroom.

1.2 Topics of the course

During the following 16 weeks we should cover the following subjects:

1. Algorithmic Analysis
2. Complete Search
3. Divide and Conquer
4. Graphs

1.3 Technical Aspects

In general, when discussing ideas we will work in *pseudocode*, however, coding examples will be in `C++`. Participants can feel free to use whatever programming language they feel more comfortable with.

Most of the learning resources (including those produced by CPC UTEC) will be in English and have code in C++, as this is how information is usually found on the internet.

2 Introduction to Algorithms

2.1 Definition

Algorithms arise as a way to formalize the ways we solve problems. For this reason you will see that the words algorithm and solution will often be used indistinctly. Probably you are familiar with the traditional definition of an algorithm:

> "An finite ordered set of instructions that attempt to solve a problem"

Even thought this definition gives as a general understanding of what an algorithm is, it leaves a very important concept hanging: What is an instruction.

2.2 What it an instruction?

This question is much harder than it seems, as its answer depends on the context it is asked. More specifically on who is executing the instructions. This was one of the main challenges early computer scientists had to face when creating their theoretical models of computers.

Luckily, today we don't have to wonder with the philosophical aspects of this question as we have computers that help us to answer this question in a practical manner.

When we are talking about instructions in an algorithm we are talking about the instructions computers can do natively:

- Arithmetic and Logical operations

- Memory writing and reading
- Conditional branching
- Looping thru instructions

We will also call instruction or *subprocesses* to all the algorithms we know we can build with this set of instructions. When we combine instructions into a subprocess we are abstracting all the primitive instructions a computer does into a single *high level* instruction.

3 Algorithmic Analysis

To talk about algorithms we need to talk about problems, and if there is something almost all problems have in common is that there are a lot of distinct ways of tackling them. This is especially true in the context of CS, where a single problem can be solved by multiple algorithms. However are all of this solutions the same?

If we think about it, there are many ways in which we can measure how good a solution is. Here are just a few indicators we can use to evaluate different solutions to a problem:

- How often it works
- How much it costs
- How long it takes
- How simple it is

In algorithmic analysis we attempt to formalize all of this criteria in order to have a rigorous way of comparing solutions and evaluating their performance.

3.1 Measuring Time

In the context of Computer Science one of the most important measurements in order to determine the performance of an algorithm is how our algorithm takes to solve a problem. We want our solutions to be as fast as possible.

Therefore a very important part of algorithm design is to evaluate the *time complexity* of my algorithm. *Time complexity* is a function that takes as input the input of my algorithm and outputs the time it should take to run. As you can imagine, knowing how long my algorithm will take to run before actually running it is **extremely** helpful.

3.2 Sampling and extrapolation

A first approach for measuring the performance of our solution is to experimentally take time measurements for different input sizes and trying to analyze the trend that the time follows. An easy way of doing this is by plotting the input size against time to get a function we can then *extrapolate*. Using this method we can then experimentally compare the different algorithms we want to test.

For example, let's say I have an array a of number and I want to find the sum of all elements in this array. To solve this problem I make the following algorithm:

```
$s \leftarrow 0$;
**for** $i \leftarrow 1$ **to** $n$ **do**
    $s = s + a_i$;
```

I then implement it in some programming language and run it for arrays of different sizes. I get the following results:

Input Size	Time Taken
1×10^5	0.002
5×10^5	0.008
1×10^6	0.012
5×10^6	0.573
1×10^7	0.122

From this we can estimate that the time our program takes will be directly proportional to the input size. Therefore we can estimate that if we had and array of size 5×10^7, our algorithm would take around 0.5 seconds to run.

As you can imagine this method has some mayor drawbacks:

- We need to implement all the algorithms we want to compare.
- We need to generate valid input for large test cases.
- We are not sure if the extrapolation will be loyal to how our algorithm really scales.

Is there a better way of estimating how long an algorithm will run?