1. INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

October 2023

The Analysis of Algorithms course aims to introduce formal frameworks in which we can explore various issues related to computer science, in an unambiguous and rigorous manner. Having had experience programming, you should posses some intuitive understanding of what computers are, what they can do and how well they can do it. But, while intuition does go a long way, we assert that a formal approach is needed to complement it and provide us with certainty about what we're doing and how we're doing it.

The course is structured into three distinct chapters:

1. **Computability Theory** is the branch of Computer Science concerned with addressing questions about what computers are and what they can do. By now, you should be familiar with some things that computers *can* do: multiply two numbers, sort a list, find the shortest path in a graph etc. Perhaps less intuitive is what computers *cannot* do; in a future lecture we will present some problems that are impossible for a computer to solve.

2. **Complexity Theory**. Having established precisely what computers *can* do, we move on to assess *how efficiently* they can do it. In this chapter, we will address questions related to what efficiency is, what computational resources are there, how to evaluate an algorithm without getting bogged down in the details, how to know if a solution is optimal etc.

3. **Correctness**. The first two chapters are somewhat constructive in nature, providing us with techniques for building solutions to given problems. But once we have an algorithm, how can we make sure that it does actually solve the problem that it's supposed to? In this final chapter, we look at some methods that allow us to obtain *formal guarantees* about the behavior of our code.

As one can infer from the course title, *algorithms* will play a central role in our discussion. But what exactly is an algorithm?

1 Algorithms

You are most likely already familiar with some algorithms and how to represent them. For our first example, consider this pseudocode of Euclid's algorithm:

```
inputs: two natural numbers a and b
while (a ≠ b)
    if (a > b)
        a := a − b
    else
        b := b − a

    return a
```

Euclid's repeated-subtraction algorithm

The problem at hand is finding the greatest common divisor between two natural numbers a and b. The algorithm is a "recipe" for doing so, referring to the numbers themselves in a symbolic manner, such that if you were to set the initial value of a to 49 and the initial value of b to 84 and follow all instructions, at the end you'd get the answer: 7.

For now, we shall rely on the following informal definition:

"An algorithm is a finite sequence of precise steps that can be used to solve a problem in a finite amount of time".

There are some key elements of this description that we need to unpack:

- *"finite sequence of steps"* refers to the fact that the description of the algorithm itself has to be finite. In the example above, you can think of each "step" as being one line of the pseudocode.
- *"precise steps"* – while not a very precise phrase in itself, this requirement tries to match the intuition that the following isn't a valid algorithm for finding the GCD of two numbers:

  ```
  inputs: two natural numbers a and b
  return the GCD of a and b
  ```

  This is a finite sequence of steps, but it doesn't tell us anything about *how* exactly we can do that, whereas it's very clear what each line of the previous pseudocode entails.

  Or is it? Looking at the `while` condition, you could ask: "given two numbers a and b, how can I check if $a \neq b$?" Indeed, we could break this operation into a sequence of more basic ones: for example, if a and b are represented as strings of digits, we first compare their lengths and if they match, we go through the pairs of symbols at corresponding positions in the two strings – if at any point we find a mismatch, the numbers are different, else they're the same.

  For now, we won't dwell on the precise definition of "precise", but having such concerns is the right attitude in our quest to obtain a formal description of algorithms.

- *a finite amount of time* – obviously, for an algorithm to be useful it has to give us the answer, without making us wait forever.

Remember that our aim is to establish a rigorous framework for computational matters so we will need to provide a *formal* definition of *"algorithm"*, described in the language of mathematical notation.

But before we get there, we need to address a keyword that was mentioned in our informal definition: *"problem"*.

This is the aim of the first lecture. You will see that our definition is simple and intuitive, requiring only already-familiar mathematical notions.

Having established what a problem is and thus what exactly we can hope to *solve*, in the next lecture we will address algorithms, by introducing a formal *computational model* – the Turing Machine.

2 Problems

For uniformity, we shall phrase all our problems as questions. Some familiar tasks that computers solve are more naturally expressed as imperative statements, e.g. "sort the list [43, 1, 18]!" We will rephrase such a statement as: "what is the sorted form of the list [43, 1, 18]?"

Consider the following questions:

- Is 0 prime?
- Is 1 prime?
- Is 2 prime?
- Is 451 prime?
- etc.

We can see that much of the question structure remains the same; only one part of it – the number – changes. We can generalize the question above as: "Is x prime?"; we call x *"the input"*. Thus problems are "templates" that ask a question about some generic object. If we substitute the generic object with a concrete one, we get one problem instance, with a particular answer.

The generic form of the sorted-list problem from above is: "What is the sorted form of the list l?"

Each problem has one answer, which we call *the output*.

The *meaning* of a problem is such that each possible input corresponds to exactly one output. There's a mathematical notion that fits these ideas very well: *a function*.

Definition 2.1. A *problem* is a function $f : I \to O$ from a set of *inputs* to a set of *outputs*. Although the input and output sets can be infinite (and the input set usually is), *each member of the set is finite*.

We require inputs and outputs to be finite in length to match the final part of our intuitive definition: *"[...] in a finite amount of time"*. If a problem statement contained an object of infinite length[†], then we couldn't ever finish *reading* it, so we could not start solving it. Similarly, if the output is infinite in length, we wouldn't ever finish *writing* it, so we cannot say that we provided the answer in a finite amount of time.

Note 2.1. You might well object at this point that computers can deal with irrational numbers with infinite decimal representations, such as π. Most of the time, though, that is not true; they only work with *finite approximations*.

But computers can indeed handle actual irrational numbers. However they cannot handle *random real numbers*, just a subset of the reals.

π does have finite representations, it's just that they're not strings of decimal digits. For example, consider the Weierstrass definition, which is a way to finitely represent π as a mathematical formula:

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

This point will make more sense later; for now, you don't need to focus on it.

2.1 Decision problems

Definition 2.2. A *decision problem* is a problem with a boolean answer: `TRUE` or `FALSE`.

In other words, a decision problem is a "yes" or "no" question.

Examples:

- Is this number prime?
- Is this graph fully connected?
- Can this program ever trigger a `SEGFAULT`?

During our future discussions on Computability, Complexity and Correctness, we will focus almost exclusively on decision problems. This is because decision problems have the simplest possible answer structure, which will make our analyses succinct.

Obviously, there are many relevant problems in the real world which are not decision problems. We will later see how what we learn about decision problems can be adapted to other types.

2.2 Function problems

Function problems are basically everything else, i.e. problems whose answer is a mathematical object, other than a boolean.

For example:

- What are the prime factors of n?
- What is b to the power of e?
- What is the largest element in list l?

[†]There are areas that explore the notion of infinite-length input, but they are well outside the scope of `AA` . The key term is "type 2 computability theory".

Note 2.2. Our classification of problems is not exhaustive. For example, there are practically useful problems for which multiple correct answers are possible and thus cannot be modelled as functions. Consider the question "What is one prime factor of 12?" Both 2 and 3 are valid answers.

There are other possible models for problems (in particular, the one above is an example of a *search problem*, which can be modeled as a binary relation), but they are outside the scope of `AA` .

For simplicity, we shall limit ourselves to only those kinds of problems that fit Definition 2. Most of the time, we'll discuss decision problems specifically.

Our definitions are too inexact – they don't say much about either the domain or the codomain of functions.

Some problems concern numbers, some concern lists, other concern graphs. These are vastly different objects; we would like a common way of representing all of them, so that we can have a more precise view on how our inputs (and outputs) look like.

In search of a common representation, we will now take a short detour through the field of formal languages. The concepts presented in the following section will be studied in more detail at the `LFA` course in the third year.

3 Alphabets and strings

Definition 3.1. An *alphabet* is a non-empty, finite set whose members we call *"symbols"*.

In Computer Science, the term "alphabet" is used by analogy with natural language alphabets, such as the Latin Alphabet, the Cyrillic Alphabet etc.

However, alphabets are just sets; they're not some fancier mathematical object, but just a convention that makes things easier to interpret in some contexts.

Alphabets are usually denoted by a capital sigma, Σ; here are some examples:

$$\Sigma_1 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

$$\Sigma_2 = \{0, 1\}$$

$$\Sigma_3 = \{\square, \triangle, \circ\}$$

Definition 3.2. A *string* over alphabet Σ is a finite sequence of symbols from Σ.

For brevity, whenever we are talking about *strings*, we will not write them down as $(a_0, a_1, a_2, ..., a_n)$, but we'll just join all the component symbols together: $a_0a_1a_2...a_n$.

Example strings:

$$s_1 = algorithms$$

$$s_2 = 1001110$$

$$s_3 = \square\square\triangle\square \circ \triangle$$

(Usually we will not explicitly state the alphabet from which a string's symbols are drawn, as it will be clear from the context).

The string of length 0, consisting of no symbols is called *"the empty string"* and is denoted by ε[‡].

Definition 3.3. Let Σ be an alphabet. Then Σ^* is *the set of all possible strings over* Σ.

Note that this includes the *empty string*, ε.

For $\Sigma_1 = \{a, b, c\}$:

$$\Sigma_1^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...\}$$

For $\Sigma_2 = \{\square, \triangle\}$:

[‡]We'll avoid confusion between the empty string and a normal symbol by never using Greek Letters in our alphabets.

$$\Sigma_2^* = \{\varepsilon, \square, \triangle, \square\square, \square\triangle, \triangle\square, \triangle\triangle, \square\square\square, \square\square\triangle, ...\}$$

Note that, while Σ^* itself is infinite, each of its members is finite.

4 Representing objects with strings

Any finite object can be represented as a string of symbols. This may be somewhat obscured by how we use pen-and-paper to create complex 2D combinations of symbols, but a structure like:

$$\prod_{i=1}^{n} \sqrt{\frac{a_i^2}{b_i}}$$

can easily be converted to a string like "$\Pi(i = 1; n)(\sqrt{}((a\_i\hat{}2)/(b\_i)))$".

5 Conclusion

From now on, we shall consider only the string representation of inputs; the alphabet will be chosen to suit our needs, on a per-case basis. But we shall see, in a future lecture, that the binary alphabet $\Sigma = \{0, 1\}$ is sufficient for all our purposes.

We can now say that a problem is a function from strings to outputs:

$$f : \Sigma^* \to O$$

For decision problems, because there are only two outputs (`FALSE` and `TRUE`), we shall treat them in a special way.

So we can revisit Definition 2.1 and say that a decision problem is a function from strings to $\{FALSE, TRUE\}$:

$$f : \Sigma^* \to \{FALSE, TRUE\}$$

For functional problems, because outputs are arbitrary objects, we can also talk instead about their *string encoding*. So we can say a function problem is a function from strings to strings.

$$f : \Sigma_1^* \to \Sigma_2^*$$

Note that the input alphabet can differ from the output alphabet; most of the time, we'll consider them the same. Most of the time, we'll consider them to be $\Sigma_b = \{0, 1\}$.