Lecture 1: Introduction

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

January 6, 2014

1 What is an algorithm?

First of all, welcome to this course! Our goal will be to introduce you to some of the basic concepts in computer science.

The most important such concept is that of an **algorithm**. An algorithm gives **precise** instructions to carry out a task based on well-understood primitive steps. Additionally, we want to guarantee that if we try to carry this out, the result will be produced in a **finite number of steps** (i.e., the task will end). Algorithms have been around for much longer than computers. Actually, until 1946, "computer" just meant a person that can compute.

To understand what an algorithm is, think of the recipe for cooking pancakes. You will have a list of ingredients, or **inputs**: sugar, flour, milk, oil, baking soda, chocolate chips (each in some quantity). There is a desired **output**: yummy pancakes! And there is a precise sequence of steps that should be followed to achieve it:

1. Pour flour, sugar, chocolate chips and milk in a bowl
2. Mix **until** the batter is smooth
3. Heat up the grill
4. Put baking soda in the bowl
5. Mix **again** the batter is smooth
6. Pour oil on the grill and let it heat for 10 sec
7. **If** the grill is hot enough, pour the batter on the grill
8. Cook until brown, then flip and cook until brown

In general, a cooking recipe can be thought of as an algorithm, which has **inputs**, **outputs** and a **sequence of instructions** describing how the output can be obtained from the input. Folding origami is a similar example, where the input is a sheet of paper and the output is a swan, or some other desired shape. Can you think of some other examples?

Mathematical algorithms have been around since the antiquity. Mayans had algorithms for predicting solar eclipses centuries in advance. Egyptians used algorithms to build pyramids. Indians had algorithms for factorizing polynomials. Greeks had algorithms for all kinds of geometric constructions, like bisecting angles.

Of course, in our daily lives there are also examples for which there is no well-defined "algorithm". E.g., coaching someone in sports, learning how to behave in order to make friends, or making certain scientific discoveries are not describable by precise sequences of steps. For some tasks, there are algorithms but they do not always work very well. Predicting the weather or the stock market are such examples; people strive to improve the existing algorithms for these problems.

Let us consider now the problem of adding up two fractions:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

(For simplicity, assume that we are not interested at the moment in simplifying the resulting fraction; we will explore simplification later). The inputs are the two initial fractions, the output is also a fraction. The procedure described by the equation above is **general-purpose**, which means that it will work for any two well-defined fractions. In general, we will be interested in general-purpose algorithms, i.e. algorithms that produce the **correct** answer for any inputs that are of a valid type. During the course, we will see examples of how we can **prove** that an algorithm is correct.

Aside from correctness, another aspect that we will study in detail is the **efficiency** of algorithms. For instance, adding two fractions requires 3 multiplications and 1 addition, to produce the numerator and denominator of the resulting fraction. We will talk about ways of "counting" these operations and expressing the efficiency/complexity of algorithms during the course.

A large part of computer science is also devoted to coming up with good languages and representations for designing and expressing algorithms. The languages that are understandable by a computer are called **programming languages**. At the intersection of computer science and computer engineering, people study the structures or machines for executing algorithms. We will briefly discuss this aspect as well.

Now let's come back to the problem of adding two fractions, and try to express it in a way that is understandable by a computer. For this purpose, a fraction is a pair of integers (a, b). We usually write it $\frac{a}{b}$ but the visual appearance does not matter. Integers are a **primitive data type**, i.e. they appear in all programming languages, and arithmetic with integers is available as well. Fractions will be a data type that is built on top of integers. In order to make an algorithm for adding fractions **generic** (i.e. able to work for any two fractions), we will use **variables** to hold the numerators and denominators of the two inputs, as well as those of the output. Our little algorithm looks as follows:

Input: (a, b), (c, d) where a, b, c, d are integers
Output: (n, m), a pair of integers holding the result of the addition.
Algorithm:

```
1. $n = a * d + b * c$

2. $m = b * d$
```

Each multiplication and addition is a **primitive operation**. Each row above represents a **step** in the algorithm. In this case, our algorithm uses 4 primitive operations to compute the answer. This number is **independent of the input of the algorithm** (it is the same regardless which fractions we have to add). So, we will say that the **time complexity of the algorithm is constant** (or “big-oh 1”). We will define this more precisely later. In general, the time complexity may vary depending on the data we receive.

In general, in most programming languages there are primitive (basic) data types, typically integers, booleans (can be true or false), floats (used to represent real numbers in a computer - obviously up to a finite precision!) and characters. These come with primitive operations that are defined on them (e.g. addition and multiplications for integers, logical “and”, “or” and “not” for booleans). The first thing you learn in a programming language is the basic data types and operations. We will talk about these in Java (the programming language we use in this course) in Lecture 3. Complex data types, such as strings, records, lists etc, can be built from these. Any operations on the complex data types will have to be described. More than half of the course is devoted to complex data types.

2 Example: Finding the maximum of an array

Consider the problem of finding the largest element in an array of numbers. The English prose description looks as follows:

> To find the maximum element of an array, initialize m to the value of the first element. Then, for each subsequent element, if that element is larger than m, replace m with the value of that element. Return the value of m.

This is human readable, but too vague and too verbose to be useful for a computer.

The binary language specification might look like:

```
0101010110110011010101001101010010101010100101010110110010
```

This is very precise but not readable.

The specification in Java is as follows:

```
int findMax(int a[]) { //a is the array of integers
int m=a[0]; //m will hold the maximum
for (int i=1; i<a.length; i++)
   if (m<a[i]) m=a[i];
return m;
}
```

This is precise and human readable, **especially if you use comments**, but you need to know Java to write it. Some of the things I wrote are very Java-specific (like a.length, which is the number of elements in the array). If you wrote it in Lisp, the specification would look very different, even though it would solve the same problem. (By the way, if this looks completely foreign to you, you should be taking COMP-202 instead of COMP-250!)

The pseudocode looks as follows:

```
**Algorithm:** findMax($a, n$)
**Input:** an array $a$ of $n$ numbers
**Output:** The largest element in the array
$m \leftarrow a[0]$
**for** $i \leftarrow 1$ **to** $n - 1$ **do** {
  **if** $m < a[i]$ **then** $m \leftarrow a[i]$
}
**return** $m$
```

Usually, in pseudocode we will use constructs similar to programming languages:

- Assignments: $m \leftarrow a[0]$
- Conditionals: **if** $m < a[i]$ **then** ...
- Loops: **for** $i \leftarrow 1$ **to** $n - 1$ **do** ...
- Calls to other subroutines

But we will also use freely mathematical notation, which you cannot do usually in a programming language. We will sometimes assume that someone gives us a black box to solve a particular problem (so we can call it as a subroutine). And sometimes we will specify steps less precisely, if it is clear what they need to accomplish.

How much time does this algorithm take to execute? As discussed last time, we will measure number of primitive operations instead of time. In this case, the number of primitive operations is proportional to the number of elements in the array n: no matter what array we get, we still have to look at every element in order to correctly determine the maximum. So we will say that the complexity is "on the order of n"; we will denote this in a couple of lectures by $O(n)$, read "big-oh of n.

3 How do we express algorithms?

There are different ways in which we could express an algorithm. One is to use a human-level language, like English. This is easily understandable by humans, but is often ambiguous and hard to understand by a machine. For instance, think of the ways in which you could specify a recipe for a professional cook, or for an 10-year old (like my daughter) or for a robot. The specification would have to be a lot more precise in the last two cases. If we are going to specify an algorithm for a computer, then it has to be specified in a language that the computer understands. Unfortunately, a computer only understands one language: strings of bits (0s and 1s). This is called binary language, and is the lowest level in writing programs. But it is really difficult for humans to write and read binary programs, which means that they will make lots of mistakes. So instead we write programs in some kind of "high-level" **programming language** that is then translated by another program, called the **compiler or interpreter**, into **binary language**. Over the years, people have come up with many high-level programming languages, e.g. Java, C, C++, Lisp, Fortran, Perl, Python, ML etc. In this course we will focus mainly on Java. But this requires knowledge of the programming language. Annoyingly enough, these change all the time! Also, writing the program involves specifying a lot of details, e.g. about how the data is kept. In a first pass at thinking about a problem, this may be cumbersome.

Somewhere in between, we want a way to describe algorithms that does not depend on a particular programming language, but that is unambiguous and easily implementable, and easy to read for people. This is called **pseudocode**. We'll have variables, assignments, conditionals, loops, etc. Sometimes, we will allow ourselves a little more flexibility when it is clear how a part of an algorithm should be implemented. In this course, we will often write our algorithms in pseudocode. Pseudocode is often the first pass at a solution for a problem. But it has to be followed by an implementation in an actual programming language; however, this step of translating pseudocode into a programming language is often easy. The little algorithm we discussed above is an example of pseudocode, and we will discuss more examples next lecture.

4 What makes a good algorithm?

There are several important features that we will look for in a good algorithm:

1. **Correctness**. Of course, we would like the algorithm to always return the right solution. This will be the case for most algorithms that we talk about in this class. However, sometimes it is very hard to find exactly the right solution (for example, if you are trying to find the maximum of a continuous function specified somehow, in a closed interval). In such cases, we might settle for an algorithm that just gives the right solution most of the time (i.e., with high probability), or one which gives "approximately" the right solution. In many algorithms we will try to **prove** that they are actually correct.

2. **Efficiency**. We would like the algorithm both to be **fast** and to require a small amount of **memory** for extra variables. We will discuss in detail ways of measuring the speed of an algorithm

3. **Simplicity**. This is usually a "softer" requirement than the other two. But basically we want our algorithms and programs to be easy to understand and analyze, easy to implement, easy to debug and easy to maintain (modify, add functionality etc). This is especially important in the software industry, where you work in teams to build large pieces of software, which are used over multiple decades. Unfortunately, it is hard to measure how complicated software is (do you count lines of code? classes? variables?). There is a whole field of research in software engineering related to this issue. We will be less concerned with analyzing this in the course, though you are always encouraged to write your code as "cleanly" as possible.