CS 341: ALGORITHMS (S18) — LECTURE 1
INTRODUCTION

ERIC BLAIS

The purpose of CS 341 is to learn how to *design* and *analyze* efficient algorithms. This is something that can be done simply with pen and paper, and does not require any advanced knowledge of any programming language. And yet, this is also one of the most important skill to master if you want to be a great computer scientist. To see why this is the case, it's helpful to look at a simple example.

1. A first example: Fibonacci numbers

Definition 1.1. The *Fibonacci sequence* of numbers is the famous sequence of integers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ defined by the rule

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

It is very easy to turn this rule into an algorithm for generating any Fibonacci number.

Algorithm 1: Fib1 (n)

```
if $n = 0$, return 0;
if $n = 1$, return 1;
return Fib1 $(n-1)$ + Fib1 $(n-2)$;
```

Is this a good algorithm for computing Fibonacci numbers? It is certainly a *correct* algorithm, as it follows the definition exactly, but is it fast enough to be practical? We answer this question by completing *(time complexity) analysis* of the algorithms.

Theorem 1.2. *The* Fib1 *algorithm has time complexity that is* exponential *in its input n.*

Proof sketch. Let $T(n)$ denote the number of basic operations performed by the Fib1 algorithm on input n. Since Fib1 calls itself recursively, its running time satisfies the inequality

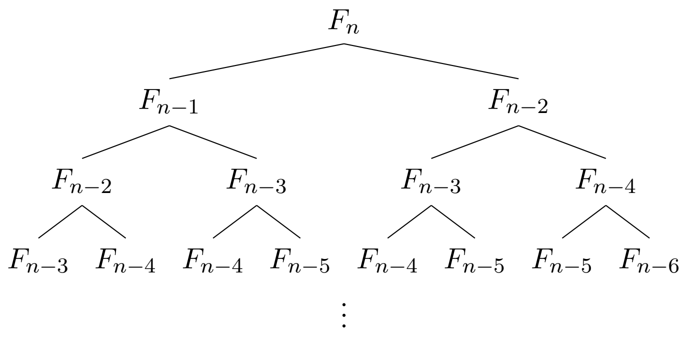
$$T(n) \geq T(n-1) + T(n-2).$$

This is an expression that should seem familiar from our definition of Fibonacci numbers! Indeed, this relation confirms that $T(n) \geq F_n$, so the time complexity of Fib1 grows at least as fast as the Fibonacci numbers. The fact that those numbers grow exponentially fast in terms of n is not trivial but is also not important for this course.[1] □

[1] If you're very curious, you can show taht $F_n \geq 2^{n/2}$ for every $n > 6$ using a proof by induction, or you can read up on Binet's formula to obtain a closed-form expression for F_n.

This is bad news! In practice, this means that we have no hope of computing F_n for even reasonably small values of n. For example, even with all of the world's computing power devoted to compute F_{50} with this algorithm, we still wouldn't get the answer for millions of years.

Can we do better than FIB1? This is the *algorithm design* component of this course. It's always useful to try to understand why an algorithm might be slow when we're trying to build a more efficient one. To understand FIB1, we can draw the tree of recursion calls made by the algorithm.



By inspecting the tree, we see a clear inefficiency: the intermediate values $F_{n-1}, F_{n-2}, F_{n-3}, \ldots$ are each recomputed multiple times within this tree. Eliminating this inefficiency can be done easily with a simple application of the *dynamic programming* technique, a powerful tool that we will explore in more detail in this class.

```
**Algorithm 2:** FIB2 ($n$)
if $n = 0$ return 0;
Create an array $A$ of size $n + 1$;
$A[0] \leftarrow 0$;
$A[1] \leftarrow 1$;
for $i = 2, \ldots, n$ do
    $A[i] = A[i-1] + A[i-2]$;
return $A[n]$;
```

The correctness of FIB2 is again easy to establish, and when we examine its time complexity, we see that it is much better than that of FIB1: the total number of basic computational steps it performs is now *linear* in n. This means we can now easily compute F_{50} on any computer. (Or $F_{5000000}$, if we want.)

2. Overview of the class

Goal. By the end of this class, you should be able to tackle a wide variety of computational problems just like we did for the Fibonacci sequence above so that you can *analyze* the time complexity of algorithms, *design* better algorithms in many cases, and *recognize* when some problems are intractable.

Topics. The topics we will cover to help you achieve this goal are grouped as follows:

(1) Analysis of algorithms
(2) Greedy algorithms
(3) Dynamic programming
(4) Graph search algorithms
(5) NP-completeness

Resources. All the course information and relevant links are found on the website

`www.student.cs.uwaterloo.ca/~cs341`

The lecture notes and tutorial notes will be posted throughout the term on the Learn page for CS 341. We will also use Piazza for discussions, as well as Crowdmark and Marmoset for the assignments.

The main textbook for the class is Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms* (3rd edition). It is not required but is highly recommended. Another great alternative textbook that is also highly recommended is *Algorithms* by Dasgupta, Papadimitriou, and Vazirani. There are also other alternatives that are on reserve at the library; see the website for details.

Marking scheme. The marks breakdown for the class is as follows:

- 30%: 5 assignments throughout the term
- 25%: Midterm
- 45%: Final exam

All sections of the class have the same assignments, midterm, and final exam. Each assignment will include multiple problems that require written solutions and a programming question.

How to succeed in CS 341. There are a few tricks that will help you succeed in CS 341 and master all the tools and techniques we introduce in the class.

- Bring pen and paper to class, and nothing else. Lecture notes will be provided so you should not worry about taking complete notes in class. But there will be many pauses in the lectures where you will be asked to think about a problem before the answer is revealed. The best way to do this will be in small groups, with pen and paper. Doing so actively (whether you do find the answer or not) will help you learn the material much more effectively than if you're just here to listen.
- Complete the homeworks *by yourself.* This will be the single most important factor determining your success in the course.
- Attend tutorial sessions. This is a new component for CS 341. The idea is to have extra practice at solving problems, on top of the assignments, in a setting where you can also get guidance and support from IAs. Take advantage of them!
- Ask questions! We are here to help. You are encouraged to discuss the material with other students, to ask questions on Piazza, to visit instructors or TAs during office hours, etc.

3. First definitions: the computational model

To make our exploration of the efficiency of algorithms more precise, we need to first establish some fundamental definitions. Let's start at the very beginning: what is an algorithm?

Definition 1.3 (Algorithm)**.** An *algorithm* is the description of a process that is:

- effective, (we can carry out each step, or basic operation, of the process)
- unambiguous, (there is no room for interpretation of the steps)
- and finite (we can write it down with a finite number of lines)

which takes some *input* and halts and generates some *output* after a finite number of steps.

A formal description of an algorithm starts with a list of the elementary operations (or steps) that the algorithm can carry out. We will not do so explicitly very often in this course: almost everywhere, the set of elementary operations will be the same as the ones you have seen in CS 240 (reading or writing to a specific index in an array, adding/subtracting/multiplying numbers, comparing two numbers, applying logical operators, etc.). And we describe the algorithms informally using pseudocode. This will allow us to specify the algorithm precisely enough that we can analyze it, but without so much implementation detail that it obscures the main ideas of the algorithm.

Before examining the time complexity of algorithms, we will want to make sure that the algorithm actually does what we want it to. Formally, this means that we first define a *problem* by specifying the instances of the problem (which correspond to the inputs of an algorithm) and the valid solutions for each instance. Then we can formally define what we mean when we say that an algorithm "solves a problem".

Definition 1.4 (Solving a problem)**.** An algorithm *solves* a problem P if for every instance I of the problem P, when the algorithm receives I as input and is run, it outputs a valid solution to P.

In this class, it's not enough to identify algorithms that solve a given problem; we also want to measure their *time complexity.* To do so, we must first define the model of computation that we consider.

Definition 1.5 (Word RAM model)**.** The *Word RAM* model is the computational model in which for an algorithm run on an input of size n,

- the memory of the algorithm is broken up into *words* of length w (typically, $w = \lceil \log n \rceil$), and
- any elementary operation (read, write, add, multiply, AND, etc.) on any single word in memory takes 1 time step.

Remark 1.6. In the definition above we have ignored a fundamental question: how do we measure the *size* of the input? The correct answer to this question is to count the number of bits required to encode the input. But, to make our lives easier, we will also have cases where the size n measures some other quantity, such as the number of entries when the input is an array, or the number of vertices when the input is a graph.

The Word RAM model balances the needs of our model to be realistic (so that algorithms that we show to be faster than alternatives really are in practice!) while still being simple (so that we can actually measure the time complexity of algorithms within the model!).

As a general rule of thumb, for most of this class you will *not* need to worry about the specifics of the model of computation; our time complexity will correspond to the number of "basic operations" performed by the algorithm. But in the few situations where it is not completely obvious what exactly constitutes a "basic operation", this definition will be useful.

4. Notes

This lecture is closely modelled on Chapter 0 of *Algorithms* by Dasgupta, Papadimitriou, and Vazirani.

More information on the Fibonacci sequence can be found on the *Online Encyclopdia of Integer Sequences (OEIS)* at `https://oeis.org/A000045`. The OEIS is a vast repository of insight on a multitude of different sequences.

And if you can't get enough of the Fibonacci sequence itself, the *Fibonacci Quaterly* (`https://www.fq.math.ca/`) is an entire journal devoted exclusively to current research on this sequence.