St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 1: Algorithms and Turing Machines
Fall 2023

1 Problems and Solutions

One important aspect of studying the theory of computing is determining precisely what kinds of questions can be answered algorithmically; that is, by a computer. For example, does there exist an algorithm that can tell us whether or not some integer given as input is a prime number?

In order to reason about problems and algorithms for these problems, we must first formalize the notion of a problem itself. Fortunately, this formalization is straightforward: an *algorithmic problem* is a function $A\colon S_{\text{inputs}} \to S_{\text{outputs}}$ mapping some set of inputs to a unique output from some set of outputs. Both the set of inputs and the set of outputs may be either finite or infinite, and we usually encode inputs and outputs as words over some alphabet Σ.

We can further restrict ourselves to considering only those algorithmic problems that map their set of inputs to a two-element set, say $\{0, 1\}$ or $\{\mathrm{T}, \mathrm{F}\}$. In such cases, we end up with something known as a *decision problem*, or a problem for which each input instance corresponds to either a positive/"yes" or a negative/"no" output.

Why should we care about decision problems in particular, though? Well, it makes our job easier if we need only check for one of two possible outputs, instead of checking a possibly arbitrary output from a possibly arbitrary function. In a sense, we can "filter" the outputs of a decision problem by collecting all of the input instances with a positive output and using this to model the corresponding decision problem. Moreover, it is often the case that we can transform an algorithmic problem into a decision problem that matches the spirit of the original; for example, instead of considering the algorithmic problem of finding some satisfying assignment of values to the variables of a Boolean formula, we can consider the decision problem of determining whether a *given* value assignment satisfies a *given* Boolean formula. Effectively, we apply a characteristic function—or a function mapping inputs to the set $\{0, 1\}$—to an algorithmic problem to get a corresponding decision problem.

Now that we're familiar with problems, we turn to the matter of solving problems, and for this we need algorithms. As you might expect, a *solution* to an algorithmic problem is an algorithm that, when given some legal input, produces the correct output. But that leaves us with a question: how do we define an "algorithm"?

1.1 Defining Algorithms Informally

Most introductory courses in computer science define an algorithm to be a set of instructions that we follow in a prescribed way to obtain some solution we desire. This is, naturally, a highly informal and imprecise way of defining algorithms, but it's good enough for most humans to understand at an intuitive level.

To make things just a little more precise—while still keeping ourselves at an overall informal level—we should expect any algorithm we devise to possess certain properties.

Definition 1 (Algorithm—informal def'n)**.** An algorithm for a problem A has the following properties:

1. the algorithm consists of a finite set of instructions I_A;
2. the instructions in I_A can solve any instance of the problem A;

3. each instruction from I_A is applied deterministically; and
4. for each legal input, the output is produced after a finite number of applications of instructions from I_A.

In other terms, an algorithm has a finite set of deterministic instructions that always give us correct outputs and always terminate.

Each of the conditions in Definition 1 intuitively make sense. We need a finite set of instructions because we simply cannot implement an infinite number of instructions in practice. We need our instructions to solve any instance of the algorithmic problem because we want our algorithm to be a solution to that problem. We need each instruction to behave deterministically because we want to know how our overall algorithm behaves, and because we don't want to get two outputs for the same input.

The fourth condition, however, brings about some interesting questions and consequences. If we give a legal input to an algorithm, we know by this condition that we will need to execute a finite number of instructions before we get our output, but *how many* instructions will we need to execute on an input of a particular size or length? We take this to be the *time complexity* of the algorithm. Likewise, how many memory cells must the algorithm use across its execution? We take this to be the *space complexity* of the algorithm.

1.2 Defining Algorithms Formally

Let's now try to introduce a little more rigor into our definition of an algorithm. The informal definition we have is perfectly serviceable if, say, we want to show that some problem *has* a solution, or if we want to establish an *upper bound* on how many instructions need to be executed in order to solve a given algorithmic problem. However, for other equally important tasks, our informal definition simply won't do!

While we can informally show that there exists an upper bound for some algorithmic problem, our definition is insufficient if we want to show that a *lower bound* exists for the same algorithmic problem; that is, if we want to show that no solution exists using time or space less than that given by some bounding function.

Going one step beyond the matter of studying solutions to problems, our informal definition also doesn't allow us to prove that a given problem *has no* solution whatsoever! Indeed, for us even to hope to be able to do this, we need to have some way of enumerating every algorithm. This will be the first step toward formalizing our definition of an algorithm.

Since our algorithms are built up from a finite set of instructions, and since they must terminate, we should be able to come up with a *finite specification* of an algorithm, or a uniform way of representing an algorithm via some encoding. This finite specification can be as simple as writing out a finite-length description of how the algorithm behaves, akin to representing the algorithm in pseudocode or any programming language.

Definition 2 (Algorithm—finite specification)**.** A finitely specified formalization of an algorithm is an effective encoding of an algorithm as a finite word over an alphabet Ω. By "effective encoding", we mean that:

1. for each algorithm, the corresponding set of instructions I_A is encoded as a word $w \in \Omega^*$, and given such a word w the instruction set I_A can be decoded; and
2. there exists an algorithm that takes as input a word $w \in \Omega^*$ and determines whether or not w encodes some algorithm under this formalization.

You might have gleaned from this more formal definition that an effective encoding behaves very similarly to a compiler for a programming language: we can use an algorithm to determine whether some word encodes an instruction (i.e., checking syntax, semantics, etc.) and, if so, we can decode the instruction itself (i.e., compiling).

All the work we've done so far to formalize our notion of an algorithm seems well and good, but unfortunately, our optimism has led us down the garden path. As it turns out, there is *no way* for us to satisfy the formalization we set out in Definition 2!

Why is this? Well, before we prove anything, let's restrict ourselves to considering only decision problems—again, to make our job easier. If we can't formalize the notion of algorithms for decision problems, then naturally we won't be able to formalize the more general case of algorithmic problems.

Our proof that our goal is impossible will use something known as a *diagonalization* argument: we will assume that we can formalize all algorithms using an effective encoding, and show that if we do this, then we will encounter some kind of contradiction.

Theorem 3. *There is no finitely specified formalization of the informal definition of an algorithm given in Definition 1.*

Proof. Suppose by way of contradiction that all informal algorithms can be formalized as words over the alphabet Ω. We can then effectively enumerate all words over Ω^* that encode some algorithm. Suppose this list is denoted by $\{w_1, w_2, w_3, \dots\}$.

Let $f_i \colon \Sigma^* \to \{\mathtt{0}, \mathtt{1}\}$ be the decision problem solved by the algorithm encoded as w_i, where $i \in \mathbb{N}$ and the alphabet Σ contains k symbols. For each $x \in \Sigma^*$, let $\text{num}(x)$ denote the number represented by x in k-ary notation.

Next, define another decision problem $g(x) = f_{\text{num}(x)}(x) + 1 \pmod 2$. (Note that the "mod 2" is necessary because g is a decision problem.) We claim that g has an informal algorithm: we compute $\text{num}(x)$, use this to find $w_{\text{num}(x)}$ in our enumeration of words over Ω^*, retrieve $f_{\text{num}(x)}$, and finally add one to the result. Thus, g must be represented by f_j for some $j \in \mathbb{N}$.

Now, choose x_j such that $\text{num}(x_j) = j$. Then

$$f_j(x_j) = g(x_j) = f_{\text{num}(x_j)}(x_j) + 1 \pmod 2,$$

but since $\text{num}(x_j) = j$, this gives

$$f_j(x_j) = f_j(x_j) + 1 \pmod 2,$$

which is a contradiction. □

Note that this unpleasant outcome didn't rely on any assumptions made on our part regarding specific properties of our formalization; all it required was an effective encoding. This, then, is the sticking point! If we want to formalize our notion of an algorithm, then we sadly cannot assert that all algorithms can be effectively encoded, or else we'll run into the diagonalization problem we saw in the proof. As a consequence, this means we must drop the fourth condition from Definition 1 stating that our algorithms must terminate. If some algorithm A does not terminate, then no algorithm can take the encoding of A as input and determine whether or not that encoding specifies A.

After removing the fourth condition of Definition 1, we must also modify the second condition to read

2. the instructions in I_A can solve any **positive** instance of the problem A **or otherwise run forever**.

Lastly, if we desire, we may remove the third condition of Definition 1 to allow for the consideration of nondeterministic algorithms, but we'll delve more into this matter later.

2 Models of Computation

Recall that one of the major questions of theoretical computer science is whether certain problems can be solved algorithmically, or in other words, whether we can use a computer to solve certain problems. Another major question is closely related: what sort of computer can (or should) we use to answer these problems? Practically speaking, this can be tough—your laptop may be able to solve tougher or larger problems than those your smartphone can solve, by virtue of your laptop having a better processor or more memory. Similarly, your desktop may be able to solve even more problems than your laptop, and a supercomputer would surely be able to solve even more problems than your desktop. What criteria should guide our choice of computer on which to run our algorithms?