In this lecture, we go back to the beginning: we will define exactly what we mean by algorithms and how we measure their time complexity. We then introduce the important idea of reductions via the 2Sum and 3Sum problems.

1. Algorithms and 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 2.1** (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 2.2** (Solving a problem). An algorithm solves a problem 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. There are two natural options.

**Line-cost model:** Each execution of a line of an algorithm takes 1 time step.

**Bit-cost model:** Operations on a single bit take 1 time step.
Algorithm 1: Tower(n)

\[
k \leftarrow 1;
\]

\[
\text{for } i = 1, \ldots, n \text{ do}
\]

\[
k \leftarrow 2^k;
\]

\[
\text{return } k;
\]

Neither of these options is ideal. The line-cost model is perfectly reasonable in many cases, but not always. For example, consider the simple algorithm that follows. According to the line-cost model, this algorithm runs in time \(O(n)\). But the value that it returns is

\[
\text{Tower}(n) = 2^{2^{2^{\cdots^n}}},
\]

a number that requires way (way, way) more than \(n\) bits even to write down, so the assumption that the line inside the for loop runs in 1 time step does not correspond in any way to reality.

The bit-cost model suffers from a different problem: it quickly gets too complicated. A line of code might consist of multiplying two positive integers \(a\) and \(b\). What is the time complexity of this operation? For starters, \(\log a\) and \(\log b\) bits are required to encode the two integers in general. And we can as we will see in later lectures, the most efficient algorithm for multiplying integers is far from trivial and has time complexity

\[
O((\log a)(\log b)^{0.59}).
\]

In some settings, it is important to account for this complexity accurately; in many other cases, however, keeping track of this detailed complexity simply makes it harder to understand what is really going on.

The model that we will end up using for this class is in between the two: it is known as the word random-access-memory (RAM) model.

**Definition 2.3 (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.

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 the time complexity of an algorithm according to the word RAM model will correspond exactly to the line-cost model. There will be a few situations, however, where we will refer back to this definition when the notion of an “elementary operation” is not completely obvious.

2. Big-O notation

We are now in a position to measure the time complexity of an algorithm on a specific input. But what we really want to do is define a global measure of the time complexity of the algorithm. We do this using worst-case time complexity.
Definition 2.4 (Worst-case time complexity). The worst-case time complexity of an algorithm $A$ is the function $T_A : \mathbb{N} \rightarrow \mathbb{N}$ obtained by letting $T_A(n)$ be the maximum time complexity of $A$ over any input of size $n$.

Remark 2.5. Worst-case is not the only way we can measure the time complexity of an algorithm as a function of its input size. (One could take the average time complexity over some distribution on inputs of length $n$ instead, for example). But this is a model that is particularly useful and the one we will focus on throughout this course.

We now come to one of the central ideas in the analysis of algorithms: what we care about, when we measure the time complexity of an algorithm, is not the exact expression for this time complexity, but rather its asymptotic rate of growth as the inputs get larger. This is best done using big-$O$ notation.

Definition 2.6 (Big-$O$ notation). Two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ satisfy $f = O(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \leq c g(n)$.

Remark 2.7. Here and throughout,

- $\mathbb{N} = \{1, 2, 3, \ldots\}$ is the set of natural numbers,
- $\mathbb{R}^+$ be the set of positive real numbers, and
- $\mathbb{R}^\geq n_0$ be the set of real numbers that have value at least 1.

Big-$O$ notation is so useful in part because it lets us simplify even very complicated expressions and only worry about the “most significant” aspects of time complexity. For example, we have the following fact.

Proposition 2.8. The function $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ satisfies $f = O(n^7)$.

Proof. For every $n \geq 4$,

$$f(n) = 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi \leq 4n^7 + 100n^3 + n^2 + n \leq 106n^7$$

so $f = O(n^7)$ by the definition with $c = 106$ and $n_0 = 4$. $\square$

One word of warning: since $n^7 \leq n^{100}$ for every $n \geq 1$, it is also correct to say that the function $f$ defined in the proposition satisfies $f = O(n^{100})$. To describe asymptotics of a function more precisely, we need the big-$\Omega$ and the big-$\Theta$ notation. The big-$\Omega$ notation is used to give lower bounds on the asymptotic growth of a function.

Definition 2.9 (Big-$\Omega$ notation). Two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ satisfy $f = \Omega(g)$ if there exist $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) \geq c g(n)$.

The big-$\Theta$ notation is used to show that we have matching upper and lower bounds on the asymptotic growth of a function.

Definition 2.10 (Big-$\Theta$ notation). Two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ satisfy $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

We can use this notation to strengthen our last proposition.

Proposition 2.11. The function $f : n \mapsto 4n^7 + 100n^3 + \frac{1}{3}n^2 + \pi$ satisfies $f = \Theta(n^7)$.

Proof. We have already seen that $f = O(n^7)$. We also have that for every $n \geq 1$, $f(n) \geq 4n^7$ so by definition $f = \Omega(n^7)$ (with $n_0 = 1$ and $c = 4$) and, therefore, $f = \Theta(n^7)$ as well. $\square$

There are also situations where we want to argue that a function grows asymptotically slower than another reference function. We can state this formally with the little-o notation.
Definition 2.12 (little-o notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = o(g)$ if for every $c \in \mathbb{R}^+$, there exists $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $f(n) < c g(n)$.

Remark 2.13. We also have that $f = o(g)$ whenever $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$. This characterization can be easier to work with when the limit exists; feel free to do so.

Similarly, we can say that a function $f$ grows asymptotically faster than another function $g$ using the little-$\omega$ notation.

Definition 2.14 (little-$\omega$ notation). Two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ satisfy $f = \omega(g)$ if and only if $g = o(f)$.

We will be using this notation extensively throughout the course, so it is critical that you are all very comfortable working with it. We will be covering some exercises related to this notation in this week’s tutorial and in the first assignment, but you can also find much more information about it and many other exercises to work on in the textbook.