Overview

1 Overview of Computational Complexity Theory

The overall goal of Computational Complexity Theory is to understand the following: *For each computational task, what is the most "efficient" way to solve it?* There are actually different kinds of resources that you may want to be "efficient" about using; for example:

- Running time.
- Memory usage (or storage space).
- Parallelism (number of processors).
- Communication between different computing components.
- Energy.
- Number of random bits used.
- Number of qubits bits used...

The first two, **time** and **space**, are the basic resources we will focus on the most.

Example (The ideal time complexity for a problem)**.** The computational problem called PATH is: Given an input graph G, and two vertices s and t, decide if there is a path in G from s to

t. (In practice, you might want to find the *shortest* path from s to t, but let's start simple.) The *ideal* outcome for the PATH problem, with respect to the resource of running time, would be:

- You find a fast algorithm for PATH.
- You prove that there is no faster algorithm for PATH.

The first task in this example, finding a fast algorithm, is the domain of Algorithms Theory (as studied in, e.g., Carnegie Mellon's 15-451 course). The second task in the example, proving that a faster algorithm does not exist, is the domain of Complexity Theory (this course, 15-455). As it turns out, proving the *im*possibility of solving a certain task with a certain running time is *extremely difficult*. Computer science researchers have arguably had very little success in proving such impossibilities!

Even though we can only rarely prove "lower bounds" on the complexity of solving various tasks, we still do our best. We set up the mathematical foundations of studying algorithmic complexity, and explore the landscape of problems. One of the main tools we have is *reductions*, which are used to relate the complexity of one problem to that of another. You have probably seen reductions before in the context of NP-completeness, where they are used to show that various problems — e.g., 3-COLORING — are no harder, *and* no easier, than the SAT problem (up to polynomial running time factors). Of course, we famously don't know how hard it is to solve the SAT problem — this is the famous "P vs. NP" problem — but at least we know that however hard it is, that's *also* how hard it is to solve the 3-COLORING problem.

A quotation from a 1988 Complexity Theory paper by Christos Papadimitriou and Mihalis Yannakakis illustrates this point. After showing a reduction from one problem to another, they say that they're

> **"Decreasing the number of questions...**
> **without increasing the number of answers."**

The design of reductions plays a very important role in Complexity Theory. On the other hand, reductions are simply *algorithms*; a reduction from problem A to problem B is like an algorithm R that uses code that solves B as a subroutine within code that solves A. This illustrates another quotation that will inform us throughout the course:

> **"Most of Complexity Theory is... Algorithms."**

2 Complexity versus Computability

You may already know something about *Computability Theory* (also known as *Decidability*), the study of which computational tasks can be solved **at all**. As you probably know, one answer is "not all of them". For example, the HALTING problem is the following: Given as input the source code M of a one-input function, as well as an input x, decide whether $M(x)$ halts (as opposed to going into an infinite loop). Alan Turing proved:

Theorem (Turing's Theorem)**.** *The HALTING problem is undecidable: there is no algorithm that solves it correctly on every input.*

Computational Complexity Theory is a sort of refinement of Computability Theory, where we take the computational problems that *can* be solved, and then ask to what extent they can be solved with *limited resources*. As a point of comparison, in Chapter (**??**) we'll see *Time Hierarchy Theorem*, which is an elaboration of Turing's Theorem that takes into account running-time resources. It shows, for example, that there are computational problems T solvable in $O(n^3)$ time but not in $O(n^2)$ time. The "diagonalization" proof technique used in Turing's Theorem and the Time Hierarchy Theorem is still virtually the only technique we know to prove impossibility results in Complexity Theory.

Unlike in Computability Theory, where most of the obvious big questions were solved long ago, there are still many famous open problems in Complexity Theory. Often they are concerned with whether or not two "complexity classes" are equal or not. Several of these complexity class questions are shown, along with a very high-level "meaning" of the question.

Example (Famous questions about complexity classes)**.**

- $\mathsf{P} = \mathsf{NP}$? *Meaning: Is finding a solution always as fast as recognizing a solution?*

  This is possibly the most famous problem in Computer Science, and is recognized as one of the six most important problems in Mathematics in the Clay Institute's $1 million prize list.

- $\mathsf{P} = \mathsf{NC}$? *Meaning: Is every sequential algorithm efficiently parallelizable?*
- $\mathsf{P} = \mathsf{L}$? *Meaning: Do efficient algorithms ever need to allocate memory? (Or are a constant number of local variables always enough?)*
- $\mathsf{P} = \mathsf{PSPACE}$? *Meaning: If a problem is solvable with a reasonable amount of memory, is it also solvable in a reasonable amount of time?*
- $\mathsf{P} = \mathsf{BPP}$? *Meaning: Can every randomized algorithm be efficiently made deterministic?*
- $\mathsf{P} = \mathsf{QuasiLIN}$? *Meaning: If a problem has an "efficient" (polynomial-time) algorithm, must it have a truly efficient ($O(n \log^c n)$-time) algorithm?*

Exercise. For exactly 1 of the 6 questions above, Complexity Theorists *do* know the answer. The other five are all famous unresolved problems. Guess which one we know the answer to!

Solution. The last one is false. It is known that $\mathsf{P} \neq \mathsf{QuasiLIN}$; this turns out to be a consequence of the Time Hierarchy Theorem mentioned above. For example, there provably exist decision problems (that is, computational tasks where the output is just yes/no) that can be solved in quadratic time (that is, $O(n^2)$ steps) but not in "quasi-linear" time (that is, $O(n \log^c n)$ steps). ■

For the 5 of the 6 questions above, although no one *provably* knows the answer, in each case Complexity Theorists generally have a strong *belief* about what the answer is. When faced with a complexity-theory question like, "Is X always possible?", *almost* always the default belief is "No". So for the open questions above about whether one complexity class equals another, the default guess is "No". But...

Exercise. For exactly 1 of the 6 questions above, Complexity Theorists generally believe the answer is "Yes". Guess which one we think is true!

Solution. $\mathsf{P} = \mathsf{BPP}$ is generally believed to be true. It turns out there are pretty good reasons to believe the following: For every decision problem (yes/no problem) that is solvable in polynomial time using a random number generator, it is also solvable in polynomial time without using a random number generator. The idea is that "good enough pseudorandom number generators" should exist, and in an advanced Complexity Theory class you can see a proof of this, *assuming* that something like $\mathsf{P} \neq \mathsf{NP}$ is true. (More precisely, assuming the SAT problem cannot be solved by circuits unless they have exponentially many gates.) ■

3 Learning objectives

Why might you care to study Computational Complexity Theory?

- Some of you will go on to further deep and rigorous study of computation and Computer Science. Then it goes without saying you will want to learn at least the basics of Complexity Theory.
- For anyone studying Computer Science at the undergraduate level, it's important to work on thinking carefully and rigorously about computation.
- Anyone who does computer programming at all will often face the following situation: You encounter a computational problem. You come up with a reasonably efficient algorithm for solving this problem. But could there be a *much* better one? Studying the basics of Complexity Theory gives you a framework and a set of tools to reason about this.