1 Why Complexity Theory?

Ancient Greeks considered the following problems.

Problem 1: Is a given number $N$ prime?

Problem 2: Do given numbers $N$ and $M$ have a common factor?

Problem 3: Given a number $N$, find its factorization.

Problem 4: Given a polynomial equation (e.g., $x^2 + y^2 = z^2$), does it have an integral solution?

Problem 1 was solved by Eratosthenes (via his “Sieve”): Write down all integers less than or equal to $N$. Then cross out all even numbers, all multiples of 3, of 5, and so on, for each prime less than $\sqrt{N}$. If $N$ remains on the list, then $N$ is prime.

Problem 2 was solved by Euclid via what is now known as Euclid’s Algorithm for finding GCD of $M$ and $N$: Initially set $r_0 = M$, $r_1 = N$, and $i = 1$. While $r_i \neq 0$, assign $r_{i+1} = r_i - r_{i-1}$, rem $r_i$ and $i = i + 1$. Return $r_{i-1}$.

The important difference between the two algorithms is that Euclid’s algorithm is very efficient (polynomial-time in the sizes of its inputs), whereas Eratosthenes’ algorithm is extremely inefficient.\(^1\)

Many other problems in mathematics still have inefficient algorithmic solutions. For example, no efficient algorithm is known for Problem 3 (integer factorization); though, there is a quantum polytime algorithm due to Peter Shor.

For some problems, no algorithms could be found for a long time. Turing (1930’s) formalized the notion of an algorithm (via Turing machines), and showed that certain problems have no algorithmic solution! This is truly one of the deepest results in mathematics.

For example, Problem 4 (solving diophantine equations) was shown to be undecidable (i.e., impossible to solve by a computer algorithm) in the 1970s by Yuri Matiyasevich (resolving Hilbert’s famous 10th problem from 1900).

\(^1\)In 2003, a polytime algorithm for Primality Testing was discovered by Agrawal, Kayal, and Saxena!
2 Efficient computation

After first computers were built, people realized that, in practice, efficient algorithms are what one needs. The notion of an efficient algorithm was formalized in the 1960’s: the class \( P \) of problems solvable in polynomial time was defined, and \( P \) was equated with the class of efficiently solvable problems. Thus, complexity theory was born! (There are good reasons why \( P \) is used to capture the notion of “efficiently computable”. For one thing, the class \( P \) turns out to be independent of the details of any particular computer model (the computer architecture), and also can be naturally defined in the “machine-independent” way without using the notion of algorithms (say using logic).)

Informally,

\[
\text{Complexity Theory} = \text{Computability Theory with Limited Resources (e.g., time, space, randomness, ...)}
\]

3 Goals and applications of complexity theory

Main goal of Complexity Theory: obtain a complete taxonomy of interesting problems according to the amount of computational resources needed to solve them.

It is still a very distant goal. The field is still young, with many exciting open problems. It addresses some of the most fundamental questions in Computer Science and all of Mathematics.

Some recent successes. While many important problems are still unresolved, complexity theory has seen a number of breakthrough developments. Some recent ones include:

1. Deterministic polytime algorithm for primality testing [AKS04].

2. Deterministic logspace algorithm for deciding the \( st \)-connectivity in undirected graphs [Rei05].

3. PCP Theorem: every mathematical proof can be efficiently “encoded” so that its validity can be efficiently (randomly) verified with only 3 queries into the encoded proof (literally 3 bits of information) [AS98, ALM+98].

4. Hardness implies Pseudorandomness (hard functions are used to build cryptographically secure pseudorandom generators). ([BM84, Yao82], plus many recent developments in cryptography [HILL99], and in derandomization of randomized algorithms [NW94, IW97, Uma03]).

The last item above suggests at least one application for computational hardness: cryptography bases its security on the computationally hard problems (e.g., RSA assumes that
Factoring integer numbers is difficult). Also, computationally hard problems can be used to build pseudorandom generators which in turn can be used to turn randomized algorithms into deterministic ones (the process called derandomization), without much loss in efficiency.

4 This course

In this course we’ll see what (little) is known, what is conjectured, and why it’s so hard to answer the many open questions in complexity theory. Our focus will be on the beautiful ideas and algorithmic techniques behind complexity results. It will be fun!

References


