CS 6902 (Theory of computation) – Lecture 1

Antonina Kolokolova [*]

January 5, 2015

1 Introduction

The following four problems have been known since ancient Greece:

Problem 1 *Is a given number n prime?*

Problem 2 *What are the prime factors of n? If n is known to be a product of two primes p and q, what are p and q?*

Problem 3 *Do given numbers n and m have a common factor (that is, are they relatively prime or not)?*

Problem 4 *Given an equation with integer coefficients (e.g., $2x + 5yz = z^2$, or $x + 2 = y$), does it have integer solutions?*

If you were to rank these problems from hardest to easiest, how would you do it? Your first question here would be what is meant by a problem being computationally easy or hard. Here, we will consider complexity of a problem as the smallest amount of resources, usually time but later also space and other, needed to solve the problem in worst case as a function of the length of the input in binary. That is, we will consider asymptotic upper and lower bounds on the complexity of the problem (although for most problems we will look at the tight bounds are not known).

To illustrate this, consider the well-known sorting problem: given n numbers, rearrange them in order, say from smallest to largest. For example, given $(15, 1, 4, 3)$, output $(1, 3, 4, 15)$.

[*]The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364. Special thank you to Richard Bajona for sharing his scribed lecture notes.

You have seen bubble-sort algorithm of $O(n^2)$ complexity and mergesort of $O(n \log n)$ (where n is taken to be the number of elements, as length of the input is at least as large). Can we pinpoint exactly the complexity of sorting? In this rare case, yes. It is possible to show that no comparison-based sorting algorithm can have worst-case run time better than $O(n \log n)$, and thus mergesort gives a tight bound for asymptotic time complexity of the comparison-based sorting.

The idea behind the lower bound proof is as follows. Represent any run of a possible algorithm as a sequence of comparisons, forming a binary tree where every node compares some a_i and a_j, and depending on the result of the comparison, one of the two possible continuations is chosen. Now, n elements can form $n!$ possible sequences, each sequence corresponding to a distinct path from the root to a leaf of this decision tree. Thus, there are $n!$ leaves of the tree, which means that the height of the tree (and therefore the number of comparisons on the longest path) is at least $O(n \log n)$. Therefore, comparison-based sorting as a problem has lower bound $\Omega(n \log n)$, and, since mergesort gives a matching upper bound, we know the complexity of sorting precisely: it is $\Theta(n \log n)$.

Getting back to the problems above, Problem 1 was solved by Eratosthenes (via his "sieve") around 200s BC. 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 3 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 $r_{i+1} = r_{i-1} \text{rem } r_i$ and $i = i + 1$. Return r_{i-1}. This algorithm uses the fact that if $m = k \cdot n + r$, and d divides both m and n, then d also divides r.

The important difference between the two algorithms is that Euclid's algorithm is very *efficient* (polynomial-time in the number of bits needed to write the inputs), whereas Eratosthenes' algorithm is extremely *inefficient.* The number of operations needed to find out whether two 256-bit numbers are relatively prime is on the order of 256^2, and can be done very fast (note that at every step r_{i-1} has at least one fewer bit than r_{i+1}, since the remainder accounts for less than half of the number, so there are linearly many steps, with one mod taking at most $\log_2(nm)$ time and series quickly decreasing). On the other hand, Eratosthenes's sieve would require the number of operations on the order of 2^{256}; for comparison, the number of atoms in our universe is estimated to be around 2^{200}.

In 2002, just a decade ago, there was a major breakthrough: an efficient algorithm for primality testing (problem 1) was found by Manindra Agrawal (IIT Kanpur) and his two undergraduate students, Neeraj Kayal and Nitin Saxena. So it has taken more than two thousand years to design an efficient algorithm for the problem. Although the notion of efficiency as we know it (time polynomial in the size of the input) appeared only in 1960s, in the work of Allan Cobham.

For problem 2 we do not know an efficient solution. Finding such a solution will have serious consequences for cryptography: one of the assumptions on which security of the RSA protocol is based is that factoring is not efficient, so factoring large numbers is hard. There is a quantum polynomial-time algorithm for this problem due to Shor, however as scalable quantum computers do not exist yet, that algorithm is not practical.

Problem 4, the Diophantine equations problem, turns out to be even more difficult than factoring numbers. In 1900, Hilbert proposed several problems at the congress of mathematicians in Paris that he considered to be the main problems left to do in mathematics; the full list contained 23 problems. You might remember problems number 1 (is there a set of cardinality strictly between countable and reals?) and number 2 (prove that axioms of arithmetic are consistent). His 10th problem was stated as "find a procedure to determine whether a given Diophantine equation has a solution". But in 1970, Yuri Matiyasevich showed, based on previous work by Julia Robinson, Martin Davis and Hilary Putnam, that such a procedure cannot possibly exist. So this problem is significantly harder than figuring out a factorization of a number: no amount of brute force search can tell if some of such equations have an integer solution.

In this course, we will explore the various types of computational resources, and look at a variety of types of computational problems, comparing their complexity.

Functions and decision problems

What precisely do we mean by a problem when we are talking about a computational problem in this way? Let us state some assumptions about the kinds of problems we will be studying in this course, and introduce some notation on the way.

Let Σ be a finite alphabet of symbols, By Σ^* we mean the set of all finite strings (including the empty string) consisting of elements of Σ (note that becuse we only consider finite strings, we are not talking about computing with real numbers). By a *language* over Σ we mean a subset $L \subseteq \Sigma^*$. For a string $x \in \Sigma^*$, we denote the length of x by $|x|$.

A *function* (or *search*) problem is a function from strings to strings $f \colon \Sigma^* \to \Sigma^*$.

For example, given an integer, find its prime factorization; or, given a graph G and two vertices s and t, find a shortest path from s to t in G.

A *decision* problem is a function from strings to Boolean values $f \colon \Sigma^* \to \{yes, no\}$.

For example, given integer, is it prime?

A *language* associated with a given decision problem f is the set $\{x \in \Sigma^* \mid f(x) = yes\}$. For example, PRIME $=\{$ binary strings encoding prime numbers $\}$.