CS 3719 (Theory of computation) – Lecture 1

Antonina Kolokolova *

January 8, 2019

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. Still, this is one of the very few examples where we have a significantly faster quantum algorithm than any non-quantum one.

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.

2 Notation and definition of a problem

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.

- Here, define an *alphabet* to be any finite non-empty set. Usually, we denote an alphabet by the greek letter capital sigma Σ. Note that this is the same symbol as is often used for summation, but it should be understandable from the context which meaning of Σ to use.
  - Unary alphabet: an alphabet which contains exactly one letter.
  - Binary alphabet: alphabet with two symbols, usually called $\{0, 1\}$
  - English alphabet: for example, $\Sigma = \{a, b, c, d, \ldots, z\}$.
- A *string* over an alphabet is a finite sequence of letters from that alphabet. E.g., 0110 is a string over binary alphabet, and *aaabaa* can be a string over $\{a, b\}$ or, say,

$\{a, b, c, d\}$. The special *empty string* denoted ϵ can be a string over any alphabet. The length of a string is a number of symbols in it; the length of ϵ is 0.

- A set of all finite strings over a given alphabet Σ is denoted Σ^* ("sigma star"). For example, $00110 \in \{0,1\}^*$. Sometimes we are only interested in strings of length at most n; then we write a set of them as Σ^n (the length of a string x is the number of symbols in x, denoted $|x|$).

- A *language* is a set of strings over some alphabet. For example, English language (ignoring capitalization) is a subset of strings over English alphabet (together with symbol $-$). Prime numbers $\{2, 3, 5, 7, 11, 13, 17...\} \subseteq \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$. Two special languages are the empty language $\emptyset$ and the language Σ^* of all finite strings over an alphabet Σ. Since our alphabets are finite sets, our languages will be countable sets (possibly finite, although interesting ones are infinitely countable).

For much of the course we will assume that our inputs are binary strings. That is, our default alphabet would be $\Sigma = \{0, 1\}$; the set of all binary strings is $\{0, 1\}^*$. But in most problems we consider the inputs are more complicated – graphs, pairs of numbers and so on. In this case, we can still encode these objects as binary strings without losing much space. For example, a pair of numbers (x, y) could be encoded very easily by making a binary string in which odd digits are the digits of the numbers, and even digits are, say, 1 except the two digits 00 used as a delimiter between the numbers (so $101, 011$ can be encoded as 11011100011111). As another example, graphs can be encoded as binary strings representing the adjacency matrix of a graph; usually for convenience when encoding graphs the matrix is preceded by the number of vertices in unary, then 0, then the matrix. So an undirected graph on 4 vertices with edges $(1, 3), (2, 4), (3, 4)$ becomes 111100010000110010110. One more note about the encoding: the same problem can be encoded in multiple different ways (for example, we could have used a different order of vertices in the graph, or a more compact pairing function such as Cantor's pairing function for numbers). We usually imply that our algorithm knows precisely the kind of encoding of a problem it is getting as an input. One more note: we could have encoded the input in unary (that is, a number, say, 5 would be represented as 11111, rather than 101); but it would make our encoding exponentially longer to write down. You can check for yourself that going from base 2 to any other constant basis does not change the lenght of the encoding that much.

Now we are ready to define a *problem*. We will have two main types of problems, depending on whether the answer can be arbitrary long, or is just a "yes/no".

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 $\}$.

In this course, we will mainly study models of computation that compute decision problems (in most cases the complexity of solving decision and function problems is very similar).

3 Computability and complexity

When we talk about complexity of solving a problem (or whether it can be solved at all), it is always with respect to a given model of computation. The two main models we will consider, which many of you have seen before, are the Finite State Machines (or Finite Automata) which are described in terms of states, and state changes as a reaction to the environment (i.e., input) and a "finite automaton with a memory", a Turing machine. Intuitively, an example of a finite automaton is something like a simple coke machine, or a sensor-operated bus door; a Turing machine, which can revisit its input multiple times and write things down is a full-blown computer. There is an intermediate model of a Pushdown Automaton (where the memory access is limited to be a first-in-last-out pipe) with an intermediate complexity.

Our first question would be computability, that is, whether there is a way to solve a problem in a given model at all. In particular, we will show that all models we consider have limitations: there are problems that they just cannot solve. You might have seen the famous Halting problem and the argument that there is no Turing machine that solves it; we will revisit this in a few weeks.

Once we establish that a specific problem can be solved in a model, the next question would be the amount of resources it takes to do so. We will mainly talk about time complexity for Turing machines (that is, the amount of time it takes to solve a problem on a Turing machine as a function of the length of the input, worst-case for any given input length). Most often we will describe upper bounds on the complexity of solving problems using O-notation, or look at relative complexity of problems using reductions.