Introduction

The following four problems have been known since ancient Greece:

Problem 1 *Given an equation with integer coefficients (e.g., $2x + 5yz = z^2$, or $x + 2 = y$), does it have integer solutions?*

Problem 2 *Is a given number n prime?*

Problem 3 *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 4 *Do given numbers n and m have a common factor (that is, are they relatively prime or not)?*

Problem 2 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 4 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}.

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 proportional to 256, and can be done very fast. 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}.

Very recently (4 years ago), there was a major breakthrough: an efficient algorithm for primality testing (problem 2) was found by Manindra Agrawal and his two undergraduate students, Kayal and 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 3 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 bases is that factoring is not efficient, so factoring large numbers is hard.

Problem 1, 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 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.

Notation: alphabets, languages, encodings and problems

First, we introduce some notation.

Definition 1 *Let Σ (pronounced "sigma", written the same as the summation sign) be a finite alphabet of symbols, By Σ^* (pronounced "sigma star") we mean the set of all finite strings (including the empty string) consisting of elements of Σ. 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 $|x|$.*

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.

For much of the course we will assume that our inputs are binary strings. 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 functionsuch 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.

A *decision* (or *membership*) problem is a function from strings to Boolean values $f : \Sigma^* \to \{yes, no\}$.

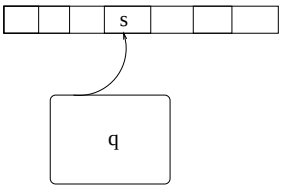
For example, "given integer n, is n prime?"

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

A *function* problem is a function from strings to strings $f : \Sigma^* \to \Sigma^*$.

Turing Machines

In 1936, Alan Turing gave the first definition of an algorithm, in the form of (what we call today) a Turing machine. He wanted a definition that was simple, clear, and sufficiently general. Although Turing was only interested in defining the notion of "algorithm", his model is also good for defining the notion of "efficient" (polynomial-time) algorithm.



A Turing machine consists of an infinite tape and a finite state control. The tape is divided up into squares, each of which holds a symbol from a finite tape alphabet that includes the blank symbol $\not b$. Some definitions give two-way infinite tape; Sipser's book uses tape infinite to the right (so it has a start cell, but no end cell).

The machine has a read/write head that is connected to the control, and that that scans squares on the tape. Depending on the state of the control and symbol scanned, it makes a move, consisting of

- printing a symbol
- moving the head left or right one square
- assuming a new state

The tape will initially be completely blank, except for an input string over the finite input alphabet Σ; the head will initially be pointing to the leftmost symbol of the input.