CS 6743 **Lecture 1** [1] **Fall 2007**

Introduction to Complexity Theory

The following three 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 prime factor?*

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 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}.

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*.

Very recently (4 years ago), there was a major breakthrough: an efficient algorithm for primality testing 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 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 bases is that factoring is not efficient, so factoring large numbers is hard.

Informally, Complexity Theory = Computability Theory with Limited Resources (e.g., time, space, randomness, ...), and it aims to understand the notion of *efficiency*.

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

[1] These course notes are mostly a modification of notes by Valentine Kabanets (`http://www.cs.sfu.ca/~kabanets/710_2007/`). Some parts are based on course notes for CS 364 as taught at the U. of Toronto by Stephen Cook and modified by subsequent instructors. Current version done by Antonina Kolokolova.

Functions and decision problems

First, we introduce some notation.
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. 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* problem is a function from strings to strings $f : \Sigma^* \to \Sigma^*$.

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 $f : \Sigma^* \to \{yes, no\}$.

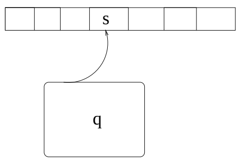
For example, given integer, is it prime?

A *language* associated with a given decision problem f is the set $\{x \in \Sigma^* \mid f(x) = yes\}$. For example, SAT $=\{$ satisfiable Boolean formulas$\}$.

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 "polynomial-time" algorithm.

Turing machines



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; we will use the tape infinite to the right.

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.

A Turing machine M is specified by giving the following:

- Σ (a finite input alphabet).
- Γ (a finite tape alphabet). $\Sigma \subseteq \Gamma$. $\not b \in \Gamma - \Sigma$.
- Q (a finite set of states). There are 3 special states:
  - q_0 (the initial state)
  - q_{accept} (the state in which M halts and accepts)
  - q_{reject} (the state in which M halts and rejects)
- $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (the transition function)
  (If $\delta(q, s) = (q', s', h)$, this means that if q is the current state and s is the symbol being scanned, then q' is the new state, s' is the symbol printed, and h is either L or R, corresponding to a left or right move by one square.

The Turing machine M works as follows on input string $x \in \Sigma^*$.
Initially x appears on the tape starting from its left end, with infinitely many blanks to the right, and with the head pointing to the leftmost symbol of x. (If x is the empty string, then the tape is completely blank and the head is pointing to the leftmost square.) The control is initially in state q_0.
M moves according to the transition function δ.
M may run forever, but if it halts, then it halts either in state q_{accept} or q_{reject}. (We will mainly be interested in whether M accepts or rejects; there are definitions of what it means for a Turing machine to output a string: e.g., the output is what is left on the tape when the machine halts)

Definition 1 M accepts *a string* $x \in \Sigma^*$ *if* M *with input* x *eventually halts in state* q_{accept}. *We write* $\mathcal{L}(M) = \{x \in \Sigma^* | M \text{ accepts } x\}$, *and we refer to* $\mathcal{L}(M)$ *as the language accepted by* M.

(Notice that we are abusing notation, since we refer both to a Turing machine accepting a string, and to a Turing machine accepting a language, namely the set of strings it accepts.)

Example 1 PARITY= the set of all binary strings that have an even number of 1s = $\{x | x \in \{0, 1\}^*, x$ has even number of occurrences of symbol $1\}$.

A very simple Turing machine recognizes the set of such strings in time linear in the number of symbols in the strings. For that, it moves once through the string from left to right, at each point remembering whether the number of 1s it has seen so far is even or odd. If by the time it reaches a blank symbol it has seen an even number of 1s, the Turing machine accepts, and if it has seen an odd number of 1s, it rejects.

State q	*Symbol* s	*Action* $\delta(q, s)$
q_0	0	$(q_0, \not b, R)$
q_0	1	$(q_1, \not b, R)$
q_0	$\not b$	$(q_{accept}, \not b, L)$
q_1	0	$(q_1, \not b, R)$
q_1	1	$(q_0, \not b, R)$
q_1	$\not b$	$(q_{reject}, \not b, L)$

A more complicated example is a Turing machine recognizing the set of palindromes.

Example 2 PAL = the set of even length palindromes = $\{yy^r | y \in \{0, 1\}^*$, where y^r means y spelled backwards.

We will design a Turing machine M that accepts the language PAL$\subseteq \{0, 1\}^*$. M will have input alphabet $\Sigma = \{0, 1\}$, and tape alphabet $\Gamma = \{0, 1, \not b\}$. (Usually it is convenient to let Γ have a number of extra symbols in it, but we don't need to for this simple example.) We will have state set $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$.

If, in state q_0, M reads 0, then that symbol is replaced by a blank, and the machine enters state q_1; the role of q_1 is to go to the right until the first blank, remembering that the symbol 0 has most recently been erased; upon finding that blank, M enters state q_2 and goes left one square looking for symbol 0; if 0 is not there then we *reject*, but if 0 is there, then we erase it and enter state q_3 and go left until the first blank; we then goes right one square, enter state q_0, and continue as before.

If we read 1 in state q_0, then we operate in a manner similar to that above, using states q_4 and q_5 instead of q_1 and q_2.

If we read $\not b$ in state q_0, this means that all the characters of the input have been checked, and so we accept.

Multi-tape Turing Machines

Definition 2 *A k-tape Turing machine has k tapes and k heads (although still only one control unit). Thus a transition function is* $\delta : (Q - \{q_{accept}, a_{reject})x\Gamma^k \rightarrow \{Q, \Gamma^k, \{L, R\}^k\}$.

Palindrome can be solved faster on a two-tape Turing machine by the following algorithm. Here, a Turing machine has two tapes (and two heads), and it starts with its input written on the first tape, and both heads at the leftmost position. When we write the transition function, list the state, then the symbols being read at the first and second tape, respectively, then the direction of movement for the first and second tape.

1) Copy the input to the second tape: $\{q_0, 0, \not b\} \rightarrow \{q_0, 0, 0, R, R\}$ and the same for 1s.
2) Move the first head to the beginning of the tape; leave the second head at the end of the input (to imitate a head staying at one place, use two movements, one left and one right).
3) Now start moving first head right and the second left, comparing the symbols they are looking at. If at any moment they look at different symbols, reject, otherwise accept when the first head sees the blank.

We now formally define the notion of worst case time complexity of Turing machines.

Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to *halt* (i.e., terminate in one of the two final states) on input x. (Each step is an execution of one instruction of the machine, and we define $t_M(x) = \infty$ if M never halts on input x.)

Definition 3 (Worst case time complexity of M) *The worst case time complexity of M is the function $T_M : \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$ defined by*

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^*, |x| = n\}.$$

If T_M is polynomial in $|x|$, we say that the Turing machine runs in polynomial time; if it is linear in $|x|$, then in linear time.

Church-Turing thesis

Church-Turing thesis: A Turing machine captures exactly the intuitive notion of an algorithm. That is, anything that can be algorithmically solved (in the intuitive sense of "algorithmically") can also be solved by a Turing machine.

An extension of the Church-Turing thesis is as follows:

Extended Church–Turing Thesis *Everything* efficiently *computable on a physical computer (in the intuitive sense) is* efficiently *computable by a Turing machine (in the formal sense).*

Quantum computers pose a potential challenge to this thesis, but they haven't been built yet.