Problem 8: Who Doesn't Want to be a Millionaire

By Nalin Ranjan

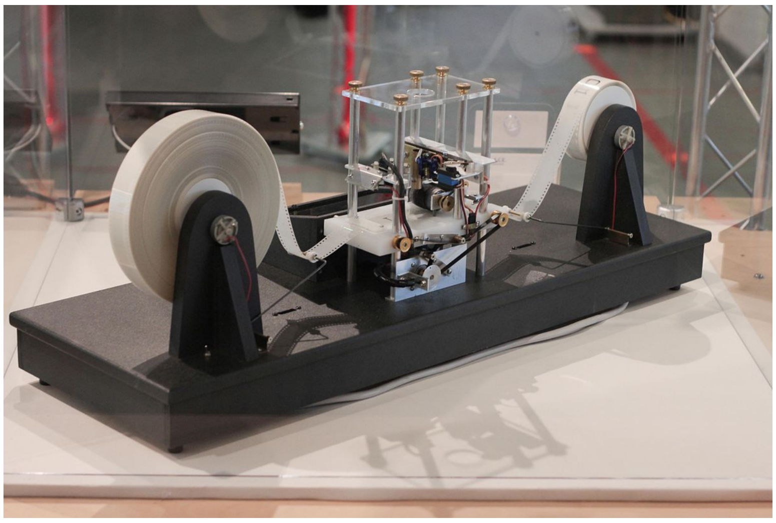
1 Background: Turing Machines

How many computers do you interact with on a daily basis? Just your laptop and phone, you might say. The reality, however, is that you probably interact with ten, maybe even more, computers on an average day. Digital accessories like clocks, watches, or cameras, household items like smoke detectors or washers, routers that your browsing requests go through, the checkout registers at stores, and the motion-activated lights in your dorm hallways, are all computers that you deal with, whether you know it or not!

Remarkably, all the computers we know of — from your digital clock to the most powerful IBM super-computer — are no more than a glorified version of what's called a *Turing Machine*. Think of a Turing machine as a little box with a finite set of instructions, indexed by a *state*. At least one of these states will be a *halting state*. At any moment in time, the machine will be in one particular state, which in turn tells it what instructions to follow. It can then traverse a tape with *symbols* and, upon reading a symbol, will follow the current instruction it has to determine what it does next. The content of the instructions that the Turing machine follows are very simple. Specifically, an instruction must be of the following format:

- Read the tape symbol at the machine's current location.
- (Optional) Replace the tape symbol at the machine's current location with another valid symbol. What it replaces the tape symbol with can depend on both the current state and the symbol it just read.
- (Optional) Change the state of the machine to another valid state. Like the previous step, the new state can depend on both the current state and the symbol it just read.
- Move either one step to the left or one step to the right on the tape.

Once the Turing machine reaches a halting state (if it does), the computation is over. The content of the tape will have likely changed, and depending on how the instructions are written, the final content of the tape might represent something useful! Computerphile has a wonderful explanation of how Turing machines work, complete with cool animations here.



A real-life Turing machine. The instruction logic is stored in the middle box, and it traverses the tape according to its instructions to perform a task. [Image Credit: Wikipedia]

2 First Problem (7 points)

A function $f : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if there exists a deterministic Turing machine that halts in exactly $f(n)$ steps on an input of n ones (for any n). That is, the Turing machine executes exactly $f(n)$ instructions before reaching a halting state. Prove that for all natural numbers k, there exists a function $f(n)$ that is $\Theta(n^k)$ that is time-constructible by explicitly demonstrating a Turing machine that halts in $f(n)$ steps.

Hint: To properly describe a Turing machine, you will need to describe five of its properties: 1) the set of possible states Q, 2) the set of possible tape symbols Γ, 3) the initial state $q_0 \in Q$, 4) The set of halting states $F \subseteq Q$, and 5) a transition function δ, which takes in a tape symbol and the machine's state and returns three things: whether the machine should move left or right, the new symbol to write on the tape, and the new state that the machine should be in. (You can think of the transition function as full set of instructions that the Turing machine has.)

Hint 2: If you have time, you can try implementing and simulating your Turing machine http://morphett.info/turing/. *You don't have to do this.*

3 Background: Languages, P vs. NP, and the Time-Hierarchy Theorem

Basics: At its simplest, complexity theory is concerned with classifying computational tasks based on the number of steps and amount of memory needed to complete it. For the rest of this problem, let's forget about memory demands of a task. The more steps required to complete a task, the "harder" it is.

Computational problems are formalized as *languages*, which are just sets of finite-length bitstrings. A *decision problem* is a question of the following form: "Given a language L, decide if a given bitstring is in L." Since a decision problem only depends on the given language, we will use the two terms interchangeably. A (deterministic) Turing machine solves a decision problem L in $f(n)$ steps if it always returns the correct answer for every possible finite input and performs no more than $f(n)$ computations, where n is the length of the inputted bitstring. We refer to the set of decision problems solvable in $f(n)$ steps by a (deterministic) Turing machine as DTIME($f(n)$).

The Time-Hierarchy Theorem: One of the seminal questions of complexity theory was "Are some computational problems non-trivially harder than others?" (At this point, you're probably rolling your eyes. Isn't the answer *obviously* yes?) While we do know the answer to this question, similar questions that are less general remain unsolved, the most famous of which is the Millennium Prize Problem P vs. NP (which if you solve, you will win a million dollars). The Time-Hierarchy Theorem, proved in 1965 by Juris Hartmanis and Richard Stearns, was a fundamental result that answered this question partially, stating that for any time-constructible function $f(n)$, there exist decision problems in DTIME($f(n)$) but not in DTIME$\left(o\left(\frac{f(n)}{\log f(n)}\right)\right)$. One important consequence, given what you proved in part (i), is that there are decision problems that can be solved in $f(n) = \Theta(n^k)$ operations that cannot be solved in $O(n^{k-1})$ operations!

P vs. NP: For some decision problems L, if some bitstring b is in L, there may be a quick way to verify that it is if we are given a "certificate." More formally, a *verifier* for a problem L is a program that takes in a bitstring b of length n and another bitstring $C(b)$, called a *certificate*, as input and always correctly decides whether b is in L. An *efficient verifier* is a verifier that always runs in a number of steps that is at most some polynomial in n.

If a problem L has an efficient verifier, then we say that $L \in$ NP. As an example, consider the SUBSET-SUM problem, which asks whether there exists a subset of a given set of integers that sums to zero. If the answer is yes, then we can easily be convinced if we are given a subset that sums to zero (the *certificate*). All we need to check is that the certificate presented actually exhibits a valid subset of the original set, and that its elements sum to zero. (This procedure is the *verifier*.) Since this procedure only requires a number of operations linear in the size of the original input, we know that our verifier is *efficient*, and therefore the SUBSET-SUM problem is in NP.

The set of decision problems that can be solved in polynomially many steps by some deterministic Turing machine is called P. While it's clear that every problem in P is in NP, we don't know if there are problems that are in NP but not in P. The P vs. NP question asks if P = NP. Colloquially, the question asks whether there exist problems where it is "easy" to verify a solution, but "hard" to come to one in the general case.

Remarkably, Stephen Cook proved that the decision problem 3-SAT is solvable in polynomial time if an only if P = NP, since being able to solve 3-SAT in polynomial time allows us to solve any problem in NP in polynomial time. He showed this by demonstrating a polynomial-time reduction of any arbitrary problem in NP to 3-SAT, which basically shows that having a "blackbox" that can solve 3-SAT efficiently would make it possible to solve the original problem efficiently. Why is this remarkable? It tells us that 3-SAT is more or less the "hardest" problem in NP!

4 Second Problem (8 points)

What's so tantalizing about the P vs. NP problem is that so many statements which we have every reason to believe are true would suffice to prove that P $\neq$ NP. For example, the existence of one-way functions – functions that are efficiently computable, but hard to invert – would suffice. (Much of cryptography and your information's security relies on their existence!) In this problem, you'll be exploring a premise that, if true, would imply P $\neq$ NP. (Though you won't win a million dollars because the premise is false.)

We might notice that a lot of the problems in NP admit a very efficient verifier. To check if a list is sorted, we just need to run through the list and see if each element is greater than the element before, which at most linearly many steps in the size of the input. The same goes for verifying a solution to the SUBSET-SUM problem. We might hypothesize, then, that

> **Premise**. There exists a verifier for every problem in NP that takes at most some $f(n) = \mathcal{O}(n^a)$ steps, where a is some absolute constant.

(As usual, n is the length of the input to the original decision problem.) For example, if $f(n) = n^2$, then we are hypothesizing that there exists a verifier that runs in no more than quadratically many steps (in the size of the original input) *for every problem in NP*. Can you show why this statement would imply that P $\neq$ NP?

Hint: You may want to do some additional reading on 3-SAT, Cook's theorem, and reductions. (Also, I didn't tell you about the Time-Hierarchy Theorem for nothing!) And as always, come and ask us if you need clarification or if you just want to learn more!