Computational Complexity Theory **Summer HSSP 2018**

Week 2: Defining Computation

Dylan Hendrickson *MIT Educational Studies Program*

2.1 Turing Machines

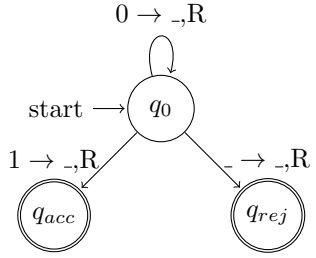
Turing machines provide a simple, clearly defined way of describing algorithms. They turn out to be as powerful as any programming language, but easier to reason about due to their simplicity.

A Turing machine has access to an infinite sequence of cells, called a tape. Each cell has one of a predefined list of symbols; we'll say cells can be blank or contain 0 or 1, though the number of symbols doesn't really matter. The Turing machine also has some number of internal states, including one labelled 'accept' and one labelled 'reject.' At any time, it is in one of its internal states and looking at one cell of the tape. To run the Turing machine for one step, check the state it's in and the symbol in the cell it's looking at. Based on the state and symbol, it can write a new symbol in the cell, move one cell left or right, and switch to a different state. Exactly what it does for each state and symbol is part of the specification of the Turing machine.

To run a Turing machine on some input, we start with the input written on the tape and the Turing machine looking at the first cell of the input, in a specific start state. We then repeatedly run it for one step until it enters either the accept or reject state, at which point we say it has accepted or rejected. Notice that we must represent the input as a string of valid symbols; if the input is a number, we can just write it in binary, but if it's something more complicated, we'll have to find a way to convert it to a string.

(If you don't know binary, it's just a way of writing numbers using only 0 and 1, instead of base ten which uses the ten digits 0 through 9. If a problem in this class involves numbers written in binary, you can usually use numbers in base ten instead if you prefer, by allowing the Turing machine to use all ten digits.)

This will make more sense with an example. We often draw Turing machines, using circles for states and arrows for transitions between states. 'start' indicates that q_0 is the starting state. Arrows between states are transitions, annotated with what they do. If there is an arrow labelled $x \to y$,D from state q_1 to q_2, it means that when the Turing machine is in state q_1 and reads x on the tape, it replaces x with y, move in the direction D (either 'L' or 'R'), and switches to state q_2. For example, the arrow labelled $0 \to \_$,R indicates that when the Turing machine is in state q_0 and reads a 0, it erases the 0 (the blank symbol is represented by $\_$), moves one cell right, and stays in state q_0. The accepting and rejecting states are q_{acc} and q_{rej}; the double circles indicate that the Turing machine halts when it reaches one of those states.



Exercise 1. What decision problem does the above Turing machine answer? Assume that the input is a

string of 0s and 1s, with no blanks (there are still infinitely many blanks past the end of the input). Find a property of the input that determines whether this machine accepts.

To describe a Turing machine, you have to specify the following:

- Input/tape alphabet. This is the set of valid symbols, typically 0, 1, and blank. (Sometimes the input alphabet and tape alphabet are different, meaning you can write symbols to the tape which can't be part of the input.)
- States, including the start state, accept state, and reject states.
- Transitions. For each state (other than accept and reject) and symbol, specify a new symbol, direction to move, and new state.

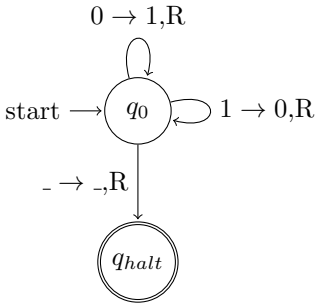
All this can be representing graphically as in the above image.

Exercise 2. Find a Turing machine that, given an input containing only 0s and 1s, says whether it's a palindrome (i.e. is the same written forwards and backwards). Such a machine should accept inputs such as 0110 and 11111, but reject inputs such as 0100 and 0101101.

Exercise 3. Find a Turing machine that checks whether its input is even. Assume the input is a number in binary (or base 10 if you prefer).

Exercise 4. Find a Turing machine that checks whether two strings of 0s and 1s separated by a blank are identical.

Sometimes we want Turing machines that do more than just accept or reject. For example, we might want to be able to add two numbers. In this case, we give the Turing machine a single 'halt' state instead of accept and reject states, and take the output as whatever is written on the tape when it halts. Here's an example:



Exercise 5. What function does the above Turing machine compute? That is, what is its output in relation to its input?

Exercise 6. Find a Turing machine that reverses a string of 0s and 1s.

Exercise 7. Find a Turing machine that adds 1 to its input.

Exercise 8. Find a Turing machine that multiplies its input by 2.

Hopefully you've gotten a sense that Turing machines can be pretty powerful, even if it takes them a while to move back and forth across the tape. For any program you write in your favorite programming language, it's possible to find a Turing machine that does the same thing.

2.2 Complexity Classes

We want to look at Turing machines that run 'quickly' in order to define 'easy' problems. We can't simply put a constant bound on the number of steps the Turing machine takes before halting, since we expect it to take a long time on a very large input. Instead, we'll let the running time grow as a function of the length of the input. Maybe we'll give it 25 steps for inputs of length 5, and 10000 steps for inputs of length 100.

We mostly care about the behavior of the running time for very large inputs. Since there are a finite (but huge) number of inputs of length 100, you could write a Turing machine with the answers to each of those inputs hard-coded; the Turing machine reads across the input and ends up in a different state depending on which length-100 string it is. Such a machine would need about 2^{100} states, but would take only about 100 steps to process inputs of length at most 100. This sort of construction isn't terribly interesting, and doesn't depend on the structure of the problem the Turing machine is solving, so we ignore it by focusing on the asymptotic growth of the running time.

The growth rates we generally think of as 'small' are polynomials: functions like n^2 or n^3, if the input length is n. We say that a Turing machine runs in *polynomial time* if there are constants a and b such that for every input of length n, the Turing machine halts within an^b steps. In other words, there should be a polynomial in the input length that is an upper bound for the running time. There are standard notations for comparing growth rates in general (most commonly $O()$, but also e.g. $o()$, $\Theta()$, and $\Omega()$), but we won't need them in this class. It'll usually be enough to say whether a function grows polynomially or faster than polynomially.

Why do we use polynomial growth instead something more limited, such as linear (an) or quadratic (an^2) growth? Consider the decision problem of determining whether a string is a palindrome. A Turing machine takes quadratic time to solve this problem; it has to check each matching pair of bits of the input (of which there are $\sim n$, and checking each pair requires crossing between them (they're $\sim n$ apart on average). What if we instead defined Turing machines differently, so that they had an extra tape to work with (the extra tape starts blank, the Turing machine has a position on each tape, and the new symbol and movement direction for each tape and new internal state depend on the readings from both tapes)? Then we could make a Turing machine that decides whether a string is a palindrome in only linear time: first, copy the input onto the second tape. Then read one copy of the input forwards and the other copy backwards, comparing them as you go. So a two-tape Turing machine is faster than an ordinary Turing machine. Similarly, real-life computers work in a way that lets them do a lot of things faster than Turing machines.

Even though these different models vary in speed, they're within a polynomial of each other. Being able to do something in linear time on a multi-tape Turing machine doesn't mean you can do it in linear time on a single-tape Turing machine, but if you can do something in *polynomial* time on a multi-tape Turing machine, then a single-tape Turing machine can also do it in polynomial time. To see this, you can design a single-tape Turing machine that simulates a two-tape Turing machine by marking off two sections of its tape to serve as two separate tapes. It then takes some time to go back and forth between the two sections, but the overhead is only polynomial, so if the two-tape machine runs in polynomial time, so does the single-tape machine simulating it. Polynomial time is resilient to changes in the model of computation, which makes it a natural class to consider.

Definition 2.1.

- **P** is the class of decision problems L such that there is a polynomial-time Turing machine that decides L.
- **NP** is the class of decision problems that can be verified in polynomial time. That is, a decision problem L is in **NP** iff there is a polynomial-time Turing machine M such that for each $x \in L$, there is a certificate c for x where $M(x, c)$ accepts, and whenever $M(x, c)$ accepts, x is in L.
- **coNP** is $\overline{\textbf{NP}}$, the class of decision problems that can be verified false in polynomial time.

The definition of **NP** may be confusing; some examples of **NP** problems and certificates will help. It's very similar to the alternate definition of **RE** we saw last week. You can also think of **NP** as being the class of decision problems that can be solved by a polynomial-time Turing machine which is allowed to make lucky guesses; you guess the value of a certificate and then see if it worked. Such a machine is called a *nondeterministic* Turing machine, which is where the **N** comes from.

It should be clear that $\mathbf{P} \subset \mathbf{NP}$ and $\mathbf{P} \subset \mathbf{coNP}$; if you can solve something in polynomial time, then you can verify it just as quickly. Can you prove or disprove that $\mathbf{P} = \overline{\mathbf{P}}$?

Here are some decision problems in **P**:

- Given a completed Sudoku grid, is it solved correctly?
- Given a number, is it prime? (this is hard to show)
- Given some numbers and the location of mines in Minesweeper, is the arrangement of mines valid?
- Given a (directed) graph and two vertices s and t, is there a path from s to t?
- Given a boolean formula and assignments to variables, is the formula true?

Here are some decision problems in **NP** but not known to be in **P**:

- Given a Sudoku puzzle, does it have a solution?
- Given numbers n, a, and b, does n have a factor between a and b?
- Given some numbers in Minesweeper, is there an arrangement of mines satisfying them?
- Given a graph, does it have a Hamiltonian path?
- Given a boolean formula, does it have a satisfying assignment?

For each of these **NP** problems, the complement problem is in **coNP**. The factoring problem is also in **coNP**; the prime factorization of n serves as a certificate to prove it doesn't have a factor between a and b. So this problem is in $\mathbf{NP} \cap \mathbf{coNP}$; the other problems are not known to be in **coNP**.

Sometimes we don't care how long a program takes to run, and instead care about how much memory it uses. A Turing machine runs in *polynomial space* if there are constants a and b such that for every input of length n, the Turing machine visits at most an^b cells of the tape.

Definition 2.2. **PSPACE** is the class of decision problems L such that there is a polynomial-space Turing machine that decides L.

What about verifying in polynomial space? We could define **NPSPACE** similarly, but it turns (not obviously) out that $\mathbf{PSPACE} = \mathbf{NPSACE}$; if you can verify something in polynomial space, you can solve it in polynomial space too.

A Turing machine uses only one cell per step, so a polynomial-time machine runs in polynomial space. Thus $\mathbf{P} \subset \mathbf{PSPACE}$. Suppose you have an **NP** problem. The certificates must be polynomially long, since otherwise the polynomial-time verifier wouldn't be able to read the whole certificate. We can make a polynomial-space Turing machine that writes each possible certificate to its tape and runs the verifier on it, then increments the certificate. This machine solves the **NP** problem, so $\mathbf{NP} \subset \mathbf{PSPACE}$; similarly $\mathbf{coNP} \subset \mathbf{PSPACE}$.

Here are some decision problems in **PSPACE** that aren't known to be in **NP** or **coNP**:

- Given a Rush Hour puzzle, is it solveable?
- Given a quantified boolean formula, is it true?
- Given a level in Super Mario Bros., is it solveable?
- Given a position in Othello/Reversi, can white force a win?

If the tape has k cells, there are roughly 2^k configurations of the tape, so a polynomial-space Turing machine can take an exponential amount of time. Nobody knows for sure whether **P** = **PSPACE**, but it seems extremely likely that there are problems in **PSPACE** but not in **P**; you can do strictly more with polynomial space than you can with polynomial time.

2.3 Challenge Problems

Here are some harder problems in case you want more practice designing Turing machines. I also highly recommend playing Manufactoria (`http://pleasingfungus.com/Manufactoria/`), a Flash game about building devices a lot like Turing machines.

Exercise 9. Find a Turing machine that checks whether the first of two numbers is greater than the second. Assume the numbers are written in binary (or base 10) and separated by a blank.

Exercise 10. Find a Turing machine that checks whether a number (in binary/base 10) is a multiple of 3.

Exercise 11. Find a Turing machine that adds two numbers separated by a blank.

Exercise 12. Find a Turing machine that checks whether a string contains the same number of 0s and 1s.

Exercise 13. Find a Turing machine that checks whether its input is of the form 0^n1^n, i.e. some number of 0s followed by the same number of 1s.

Exercise 14. Find a Turing machine that multiplies its input by 3.

Exercise 15. Find a Turing machine that multiplies two numbers separated by a blank.

Exercise 16. Find a Turing machine that checks whether a number is prime. Does it run in polynomial time?

Exercise 17. Find a Turing machine that checkes whether a string of open and close paretheses is valid (e.g. '()()' and '((()(()))' are valid but '(' and '())(()' aren't). You can either let the Turing machine use '(' and ')' as symbols, or interpret 0 as '(' and 1 as ').'