Lecture 2: Turing machines, counting arguments, diagonalization, incompleteness, complexity classes

Anup Rao

October 4, 2023

A Turing Machine is essentially a program written in a particular programming language. The program has access to three arrays and three pointers:

- x which is accessed using the pointer i. x is an array that can be read but not written into.
- y which is accessed using the pointer j. y can be read and written into.
- z which is accessed using the pointer k. z can only be written into.

The machine is described by its code. Each line of code reads the bits x_i, y_j, and based on those values, (possibly) writes new bits into y_j, z_k, and then possibly after incrementing or decrementing i, j, k, jumps to a different line of code or stops computing. Initially, the input is written in x and the goal is for the output to be written in z at the end. i, j, k are all set to 1 to begin with. The arrays all have a special symbol to denote the beginning of the tape and a special symbol to denote the blank parts of the tape.

For example here is a program that copies the input to the output using a single line:

1. If x_i is empty, then HALT. Else set $z_k = x_i$ and increment each of i, k. Jump to step 1.

Here is another that outputs the input bits which are in odd locations:

1. If x_i is empty, then HALT. Else set $z_k = x_i$, increment each of i, k and jump to step 2.
2. If x_i is empty, then HALT. Else increment each of i, k and jump to step 1.

The exact details of this model are not important. The main reason we introduce it is to have a fixed model of computation in mind. For example, it is easy to show that adding more tapes or increasing the alphabet size does not change the model significantly, as we shall discuss further next time.

Resources of Turing Machines

Once we have fixed the model, we can start talking about the *complexity* of computing a particular function $f : \{0,1\}^* \to \{0,1\}$. Fix a turing machine M that computes a function f. There are two main things that we can measure:

- Time. We can measure how many steps the turing machine takes in order to halt. Formally, the machine has running time $T(n)$ if on every input of length n, it halts within $T(n)$ steps.
- Space. We can measure the maximum value of j during the run of the turing machine. We say the space is $S(n)$ if on every input of length n, j never exceeds $S(n)$.

The following fact is immediate:

Fact 1. *The space used by a machine is at most the time it takes for the machine to run.*

Robustness of the model: Extended Church-Turing Thesis

THE REASON TURING MACHINES ARE SO IMPORTANT is because of the *Extended Church-Turing Thesis*. The thesis says that *every* efficient computational process can be simulated using an efficient Turing machine as formalized above. Here we say that a Turing machine is efficient if it carries out the computation in polynomial time.

The original (non-extended) thesis made a much tamer claim: that any computation that can be carried out by a human can be carried out by a Turing machine.

The Church-Turing Thesis is not a mathematical claim, but a wishy washy philosophical claim about the nature of the universe. As far as we know so far, it is a sound one. In particular if one changed the above model slightly (say by providing 10 arrays to the machine instead of just 3, or by allowing it to run in parallel), then one can simulate any program in the new model using a program in the model we have chosen.

Claim 2. *A program written using symbols from a larger alphabet Γ that runs in time $T(n)$ can be simulated by a machine using the binary alphabet in time $O(\log|\Gamma| \cdot T(n))$.*

Sketch of Proof We encode every element of the old alphabet in binary. This requires $O(\log|\Gamma|)$ bits to encode each alphabet symbol. Each step of the original machine can then be simulated using $O(\log|\Gamma|)$ steps of the new machine. ■

Claim 3. *A program written for an L-tape machine that runs in time $T(n)$ can be simulated by a program for a 3-tape machine in time $O(L \cdot T(n)^2)$.*

Sketch of Proof The idea is to encode the contents of all the new work arrays into a single work tape. To do this, we can use the first L locations on the work tape to store the first bit from each of the L arrays, then the next L locations to store the second bit from each of the L arrays, and so on. To encode the location of the pointers, we increase the size of the alphabet so that exactly one symbol from each tape is colored red. This encodes the fact that the pointer points to this symbol of the tape. The actual pointer in the new Turing machine will then do a big left to right sweep of the array to simulate a single operation of the old machine. ■

The following theorem should not come as a surprise to most of you. It says that there is a machine that can compile and run the code of any other machine efficiently:

Theorem 4. *There is a turing machine M such that given the code of any Turing machine α and an input x as input to M, if α takes T steps to compute an output for x, then M computes the same output in $O(CT \log T)$ steps, where here C is a number that depends only on α and not on x.*

We shall say that a machine runs in time $t(n)$ if for every input x, the machine halts after $t(|x|)$ steps (here $|x|$ is the length of the string x). Similarly, we can measure the space complexity of the machine. The crucial point is that small changes to the model of Turing machines does not affect the time/space complexity of computing a particular function in a big way. Thus it makes sense to talk about the running time for computing a function f, and this measure is not really model dependent.

Lower bounds—Counting arguments

We have shown that every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size at most $O(2^n/n)$, and on the other hand we show that for n large enough there is a function that cannot be computed by a circuit of size less than $2^n/(3n)$. The lower bound we prove here was first shown by Shanon. He introduced a really simple but powerful technique to prove it, called a *counting argument*.

Theorem 5. *For every large enough n, there is a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size $2^n/3n$.*

Proof We shall count the total number of circuits of size s, where $s > n$. To define a circuit of size s, we need to pick the logical operator for each (non-input) gate, and specify where each of its two inputs come from. There are at most 3 choices for the logical operation, and at most s choices for where each input comes from. So the number of choices for each non-input gate is at most $3s^2$. The number of choices for an input gate is at most $n < 3s^2$. So, the total number of choices for each gate is at most $3s^2 + n$, and the number of possible circuits of size s is at most

$$(3s^2 + n)^s \leq (4s^2)^s = 2^{s \log(4s^2)} < 2^{3s \log s},$$

when $n > 4$.

This means that the total number of circuits of size $2^n/3n$ is less than $2^{3 \cdot \frac{2^n}{3n} \cdot n} = 2^{2^n}$. On the other hand, the number of functions $f : \{0,1\}^n \to \{0,1\}$ is exactly 2^{2^n}. Thus, not all these functions can be computed by a circuit of size $2^n/(3n)$. ■

Indeed, the above argument shows that the fraction of functions $f : \{0,1\}^n \to \{0,1\}$ that can be computed by a circuit of size $2^n/4n$ is at most $\frac{2^{\frac{3}{4} \cdot 2^n}}{2^{2^n}} = \frac{1}{2^{2^{n-2}}}$, which is extremely small.

Similar arguments can be used to show that not every function has an efficient branching program (as you will do on your homework).

Diagonalization

We used counting arguments to show that there are functions that cannot be computed by circuits of size $o(2^n/n)$. If we were to try and use the same approach to show that there are functions $f : \{0,1\}^* \to \{0,1\}$ not computable Turing machines we would first try to show that:

$$\text{\# turing machines } \ll \text{\# functions } f.$$

This approach doesn't seem like it makes any sense at first, because both numbers here are infinite. Luckily, mathematicians have long studied how to compare the sizes of infinite sets.

Recall the definitions of the following sets:

$\mathbb{N} = \{1, 2, 3, \dots\}$	the natural numbers
$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$	the integers
$2^{\mathbb{N}} = \{A \subseteq \mathbb{N}\}$	the set of sets of natural numbers
$\mathcal{Q} = \{i/j : i, j \in \mathbb{Z}, j \neq 0\}$	the rational numbers
$\mathbb{R} = \left\{ \lim_{i \to \infty} x_i : x_1, x_2, \dots \in \mathcal{Q} \text{ is a convergent sequence} \right\}$	the real numbers

To compare the sizes of these sets, we use the concept of countability. A function $\phi : \mathbb{N} \to S$ is said to be surjective if for every $s \in S$, there is an $i \in \mathbb{N}$ such that $\phi(i) = s$.

Definition 6. *A set S is* countable*, if there is a surjective function $\phi : \mathbb{N} \to S$.*

Equivalently, S is countable if there is a list $\phi(1), \phi(2), \ldots$ of elements from S, such that every element of S shows up at least once on the list.

Let us try to understand which of the sets we have discussed are countable.

Fact 7. *$\mathbb{N}$ is countable.*

Proof Consider the list $1, 2, 3, \ldots$. This obviously contains every element of $\mathbb{N}$. ■

Fact 8. *$\mathbb{Z}$ is countable.*

Proof Consider the list $0, 1, -1, 2, -2, 3, -3, \ldots$. This obviously contains every element of $\mathbb{Z}$. ■

Fact 9. *$\mathbb{Z} \times \mathbb{Z} = \{(i, j) : i, j \in \mathbb{Z}\}$ is countable.*

Proof Consider the list

$$(0,0), (1,0), (1,1), (0,1), (-1,1), (-1,0),$$
$$(-1,-1), (0,-1), (1,-1), (2,-1), \ldots,$$

shown in Figure 1. This list contains every element of $\mathbb{Z} \times \mathbb{Z}$. Indeed, we are enumerating all pairs (i, j) where the $\max\{|i|, |j|\}$ is 0, then all pairs where $\max\{|i|, |j|\}$ is 1 and so on. Clearly, every pair occurs somewhere in the list. ■

Fact 10. *$\mathcal{Q}$ is countable.*

Proof Since $\mathbb{Z} \times \mathbb{Z}$ is countable, just take the list of all pairs from $\mathbb{Z} \times \mathbb{Z}$, and discard an entry if $j = 0$ and replace it with i/j if $j \neq 0$. This gives an enumeration of $\mathcal{Q}$. ■

The interesting thing is that some sets can be shown to be uncountable, using the technique of *diagonalization*.

Fact 11. *$2^{\mathbb{N}}$ is not countable.*

Proof Suppose there was some list of sets $A_1, A_2, \ldots$. Then consider the set

$$T = \{i : i \in \mathbb{N}, i \notin A_i\}.$$

We claim that T is not in the list. Indeed, suppose $T = A_j$ for some j. Then if $j \in A_j$, $j \notin T$ by our construction, and if $j \notin A_j$, then $j \in T$. In either case, $T \neq A_j$. ■

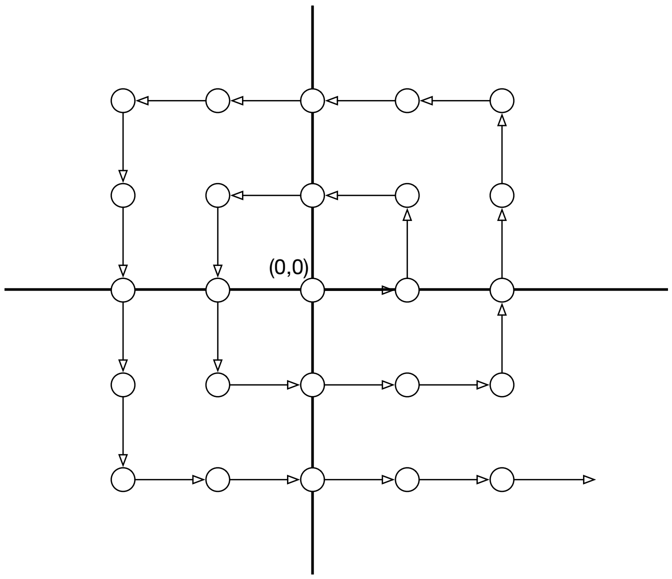

Figure 1: Enumeration of $\mathbb{Z} \times \mathbb{Z}$.

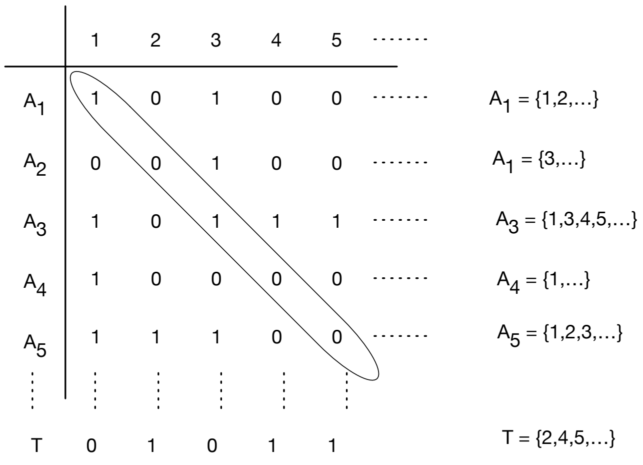
1 2 3 4 5 ....
A1 1 0 1 0 0 .... A1 = {1,2,...}
A2 0 0 1 0 0 .... A1 = {3,...}
A3 1 0 1 1 1 .... A3 = {1,3,4,5,...}
A4 1 0 0 0 0 .... A4 = {1,...}
A5 1 1 1 0 0 .... A5 = {1,2,3,...}
T 0 1 0 1 1 T = {2,4,5,...}

Figure 2: Diagonalization of a list of sets.

The proof we just used is called a proof by diagonalization, because we can think of doing it using the picture described in Figure 2. We encode each set in our list using a binary string. The set T we picked is obtained by taking the set that is obtained by choosing something that disagrees with the diagonal in the picture.

It was discovered by Cantor

A very similar idea can be used to show that the real numbers are not countable:

Fact 12. $\mathbb{R}$ *is not countable.*

Proof Every real number can be thought of as a number with a potentially infinite decimal expansion.

Suppose $r_1, r_2, \ldots$ is an enumeration of the real numbers. Consider the real number $t = 0.d_1 d_2 \ldots$, where the i'th digit d_i is chosen so that

d_i is not the same as the i'th digit of r_i. Then t is a real number that does not occur anywhere in the list of r_i's, since it disagrees with the i'th number in the i'th digit after 0. ■

A very similar idea gives an impossibility result for Turing Machines.

Theorem 13. *There is a function that is not computed by any Turing Machine.*

Before we see the the simple proof, let us point out that this is philosophically a very powerful fact. A consequence of it is that assuming the Church-Turing Thesis is true, there are some ways to manipulate information that can never occur in the universe. It seems hard to imagine a physical process that violates the Church-Turing thesis, and it also seems hard to stomach the fact that the universe cannot manipulate information in a particular way, yet one of those two (admittedly wishy washy) strange things must happen.

We shall need some notation before discussing the proof. Given a string α, we write M_α to denote the Turing Machine whose code is α.

Proof Consider the function $f : \{0,1\}^* \to \{0,1\}$ defined as follows:

$$f(\alpha) = \begin{cases} 1 & \text{if } M_\alpha(\alpha) = 0 \\ 0 & \text{else.} \end{cases}$$

No Turing Machine can compute this function, for if there was some machine that could, then let γ denote the binary encoding of its code. Then we have that $M_\gamma(\gamma) = f(\gamma)$, but this contradicts the definition of f, since if $f(\gamma) = 0$, then $M_\gamma(\gamma)$ cannot be 0, and if $f(\gamma) = 1$, $M_\gamma(\gamma)$ cannot be 1. ■

You may object that the uncomputable f that we found above is very unnatural, but actually it is not hard to come up with natural examples that are also impossible to compute using Turing Machines.

For example, we can define the function $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ that takes as input two strings α, x, and then decides whether $M_\alpha(x)$ halts or runs forever. This seems like a very useful function to compute, but it is also uncomputable.

Theorem 14. HALT *is not computable by a Turing Machine.*

Proof Suppose it was. Then consider the machine M that on input α first simulates $\mathsf{HALT}(\alpha, \alpha)$. If the answer is that $M_\alpha(\alpha)$ halts, then M simulates $M_\alpha(\alpha)$ and outputs the opposite of its output. If $M_\alpha(\alpha)$ does not halt, then M outputs 0. Then M computes the uncomputable function f above. ■

Gödel's Incompleteness Theorem

Diagonalization was also used to prove Gödel's famous incompleteness theorem. The theorem is a statement about proof systems. We sketch a simple proof using Turing machines here.

A proof system is given by a collection of axioms. For example, here are two axioms about the integers:

1. For any integers a, b, c, $a > b$ and $b > c$ implies that $a > c$.
2. For any integer a, $a + 1 > a$.

Given a list of such axioms, a proof is a sequence of statements that uses the axioms to prove that a statement is true. For example, to prove that $a > b$ implies that $a + 1 > b$, we can combine the assumption $a > b$ with the axiom $a + 1 > a$ and the first axiom, to prove $a + 1 > b$.

Prior to Gödel's work, mathematicians were trying to axiomatize all of mathematics. They were looking for a set of finite axioms that could be combined to prove any proof statement. Godel proved that this a doomed project.

A set of axioms is *consistent* if the axioms don't contradict each other. The set of axioms is complete if every true statement can be derived from the set of axioms. Godel proved:

Theorem 15. *Every consistent finite set of axioms is incomplete.*

We give an alternate proof due to Chaitin. Given $x \in \{0,1\}^*$, its Kolmogorov complexity $K(x)$ is the length of the shortest program α such that $M_\alpha(.) = x$. Namely it is the length of the shortest program that outputs x. For each $x \in \{0,1\}^*, N \in \mathbb{N}$, let $S_{x,N}$ be the statement

$$K(x) > N.$$

Fact 16. *For every N, there is an x for which $S_{x,N}$ is true.*

Proof There are only a finite number of programs of length N, so for each N, there are only a finite number of x's such that $K(x) \leq N$. This means that almost all statements $S_{x,N}$ are true. ■

To prove Godel's theorem, suppose there is some finite set of axioms A. Consider the following program M_N:

- Enumerate over all pairs (x, α), where $x \in \{0,1\}^*$, $\alpha \in \{0,1\}^*$. If α describes a proof of $S_{x,N}$ using the axioms A, output x.

If the finite set of axioms were complete, M_N would always halt, since it would find some string x and a proof α proving $S_{x,N}$. But the program M_N can be described using just $O(\log N)$ bits, and it outputs a string x for which $K(x) > N$. For N large enough, this is a contradiction, and so A must be incomplete.

Complexity Classes

let us talk *complexity classes*. We are interested in classifying functions according to their complexity, so it makes sense to lump functions into sets of similar complexity:

Definition 17. *Define* $\mathsf{DTIME}(t(n))$ *to be the set of functions*

$$\mathsf{DTIME}(t(n)) = \{f : \{0,1\}^* \to \{0,1\} | f \textit{ is computable in time } O(t(n))\}.$$

Similarly,

Definition 18. *Define* $\mathsf{DSPACE}(s(n))$ *to be the set*

$$\mathsf{DSPACE}(s(n)) = \{f : \{0,1\}^* \to \{0,1\} | f \textit{ is computable in space } O(s(n))\}.$$

Once we have these definitions, we can try to define what it means for a function $f : \{0,1\}^* \to \{0,1\}$ to be *efficiently computable*. A reasonable definition of efficient computation should allow enough time to read all of the input, which takes $\Omega(n)$ time. So we should definitely include $\mathsf{DTIME}(n)$ in our set of efficiently computable functions. Further, if one algorithm calls another as a subroutine, and both are efficient, we would like to say that the combined algorithm is also efficient. The minimal class satisfying these assumptions is the class

Definition 19. $\mathbf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c)$.

Of course there is a whole spectrum of classes above P. For example:

Definition 20. $\mathbf{EXP} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{n^c})$.

And,

Definition 21. $\mathbf{E} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{cn})$.

For space bounded computation, we need to have enough space to manipulate pointers into the inputs, which takes $\log n$ bits, before we get interesting classes. The first such class is:

Definition 22. $\mathbf{L} = \mathsf{DSPACE}(\log n)$.

Definition 23. $\mathbf{PSPACE} = \bigcup_{c \geq 1} \mathsf{DSPACE}(n^c)$.

Obviously if $t(n) = O(t'(n))$, then $\mathsf{DTIME}(t(n)) \subseteq \mathsf{DTIME}(t'(n))$. But is the containment strict? Does giving a Turing Machine more time actually allow it to compute things that it cannot compute without the extra time?