complexity theory speedrun

ho boon suan

May 21, 2022 to June 5, 2022, 14:32

Sanjeev Arora and Boaz Barak wrote *Complexity Theory: A Modern Approach* around fifteen years ago. It seems like one of the most modern references; after all, it has the word 'modern' in it. This document serves as my travel log.

Contents

Tiger got to hunt,
bird got to fly;
Man got to sit and wonder 'why, why, why?'
Tiger got to sleep,
bird got to land;
Man got to tell himself he understand.

— Kurt Vonnegut, *Cat's Cradle* (1963)

1 The computational model

Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules called a program.
People create programs to direct processes.
In effect, we conjure the spirits of the computer with our spells.

— Harold Abelson, Gerald Jay Sussman, & Julie Sussman,
Structure and Interpretation of Computer Programs (1985)

Exercise 1.1. We will only describe in detail the TM for addition. We use a TM M with three tapes (input, work, and output) and the alphabet $\{\triangleright, \square, 0, 1, \#\}$. We assume that the input is provided as two natural numbers encoded in binary separated by #; for example, to compute $3 + 5$, the input tape begins with $\triangleright 011\#101\square$. The transition function δ operates as follows (when a tape head does not move or a state is not changed, it is implicitly understood and not stated):

1. On state q_{start}: Move both the input and work heads to the right; change the state to q_{copy}.
2. On state q_{copy}:
   - If the input symbol is not #, then write the input symbol on the work tape, and move all three heads to the right.
   - Otherwise, move the input head to the right; change the state to q_{right}.
3. On state q_{right}:
   - If the input symbol is not $\square$, then move the input head to the right.
   - Otherwise, move both the input and work heads to the left; change the state to q_{add}.
4. On state q_{add}:
   - If the input symbol is #, write 0 to the output and change the state to q_{halt}.
   - If the input and work symbols are both 0, write 0 to the output and move all three heads left.
   - If the input and work symbols are 0 and 1, or 1 and 0, write 1 to the output and move all three heads left.
   - If the input and work symbols are both 1, write 0 to the output and move all three heads left; change the state to q_{carry}.
5. On state q_{carry}:
   - If the input symbol is #, write 1 to the output and change the state to q_{halt}.
   - If the input and work symbols are both 0, write 1 to the output and move all three heads left; change the state to q_{add}.

- If the input and work symbols are 0 and 1, or 1 and 0, write 0 to the output and move all three heads left.
- If the input and work symbols are both 1, write 1 to the output and move all three heads left.

For inputs with numbers of different lengths, we may make a copy of the first number to another tape, then we may compare the lengths of the two numbers by moving both heads together and seeing which number ends first. We may then copy to the approriate tapes with proper padding so that we can apply the standard addition algorithm as before.

To multiply two (nonnegative) numbers, we pass as input a string like $\triangleright$1101#101$\square$ to a 3-tape TM, which encodes 13×5. If we are multiplying an m-digit number X by an n-digit number Y, we copy X to the work tape (so that the work head is at the end of X), move the input head to the end of Y, and move the output head $m + n$ cells to the right. We then multiply X by the ones digit of Y from right to left, and write the result to the output tape, which amounts to either copying X or writing zeroes (since the ones digit of Y is either zero or one). After the work head reads the start symbol $\triangleright$, we shift it back to the end of X and move the input head to the left. We also move the output head $m - 2$ cells to the right (you can do this by following the work head and then moving to the left). Now we multiply X by the tens digit of Y from right to left, but this time we must take care to handle carries by adding a carry state. This process then continues until the input head reads #, in which case the program halts.

Exercise 1.2. Fix a TM M that is described by (Γ, Q, δ). We define an encoding function $\epsilon\colon \Gamma \to \{0,1\}^{\log|\Gamma|}$ to be any injective function such that $\epsilon(\square) = (0,\dots,0) =: 0\dots0$ and that the image $\mathrm{im}(\epsilon)$ is an initial segment of the codomain — that is, with respect to the usual lexicographic order $<$ on binary strings, if $x < \epsilon(b)$, then $x = \epsilon(a)$ for some $a \in \Gamma$. The prototypical example uses the alphabet $\{\mathsf{a},\dots,\mathsf{z},\square,\triangleright\}$, with $\epsilon(\square) = 00000$, $\epsilon(\mathsf{a}) = 00001$, $\epsilon(\mathsf{b}) = 00010$, …, $\epsilon(\mathsf{z}) = 11010$, $\epsilon(\triangleright) = 11011$.

We will read encodings of characters one bit at a time, storing their restrictions in the state register Q. For example, reading the character z in the encoding above, our new machine will read the sequence $\epsilon(\square) = 00000$, $\epsilon(\mathsf{p}) = \underline{1}0000$, $\epsilon(\mathsf{x}) = \underline{11}000$, $\epsilon(\mathsf{x}) = \underline{11}000$, $\epsilon(\mathsf{z}) = \underline{1101}0$, $\epsilon(\mathsf{z}) = \underline{11010}$. We define an activation function α, that takes a binary string s of length n, a bit $b \in \{0,1\}$, and an index $1 \le k \le n$. It then returns the string s with its k-th bit set to the value of b. Thus $\alpha(11000,1,4) = 11010$, and $\epsilon^{-1}(\alpha(\epsilon(\mathsf{x}),1,4)) = \mathsf{z}$; we identify the character with its encoding so that we can write $\alpha(\mathsf{x},1,4) = \mathsf{z}$. The interpretation is that the new machine has read the first three bits of an encoded character, and x is its representation of the bits it has read so far. It then reads the fourth bit and updates the representation to z — when it gets to the $\log|\Gamma|$-th bit, then the original character is found and can then be passed to the original transition function δ.

Our new machine $\widetilde{M}$ will have alphabet $\Gamma' = \{0, 1, \square, \triangleright\}$, and it will have the state set

$$Q' \coloneqq Q \times \Gamma^k \times [\log|\Gamma|] \times \{q_{\mathsf{READ}}, q_{\mathsf{WRITE}}, q_{\mathsf{MOVE}}\};$$

here $[n] \coloneqq \{1, \dots, n\}$. We think of $\widetilde{M}$ as having four state registers.

Interlude. How do we distinguish between $\triangleright \in \Gamma$ and $\triangleright \in \Gamma'$? At first, I thought of just initializing $\widetilde{M}$ with $\triangleright \in \Gamma'$, so that (say) $\triangleright$u$\square$ becomes $\triangleright$10101$\square$. But what if M decides to overwrite $\triangleright$ with a symbol that is only in Γ'? There wouldn't be space for the encoding. So an option is to initialize $\widetilde{M}$ with $\triangleright$ encoded — we have, say, 1101110101$\square$. Similarly, the encoding of $\square$ can have issues. Our solution is to discard $\square$ and $\triangleright$ from Γ', and encode everything in binary, which allows us to handle all the data uniformly. Other possibilities that I thought of include initializing with $\triangleright$000010101$\square$ to align the blocks of character encodings, or restricting to machines M that don't overwrite $\triangleright$.

Starting the machine. The machine $\widetilde{M}$ will cycle through three main modes — READ, WRITE, and MOVE, halting when its Q register reads q_{halt}. The machine begins in the state $(q_{\mathsf{start}}, \square^k, 1, q_{\mathsf{READ}}) \in Q'$.

READ: Read Every Actual Instruction
WRITE: Write Remembered Instructions To Environment
MOVE: Move Over Virtual Environment

READ mode. Now the counter j has value 1, and the state of $\widetilde{M}$ can be written as $(q, x_1 \dots x_k, 1, q_{\mathsf{READ}}) \in Q'$. First we flush the memory by setting

$$\delta'\big((q, x_1 \dots x_k, 1, q_{\mathsf{READ}}), b_1 \dots b_k\big) \coloneqq \big((q, \square^k, 1, q_{\mathsf{READ}}), b_2 \dots b_k, \mathsf{S}^k\big).$$

Now the machine is some state $(q, y_1 \dots y_k, j, q_{\mathsf{READ}}) \in Q'$; we read the encoded letter one bit at a time by setting

$$\begin{aligned}&\delta'\big((q, y_1 \dots y_k, j, q_{\mathsf{READ}}), b_1 \dots b_k\big)\\ &\coloneqq \Big((q, \alpha(y_1, b_1, j) \dots \alpha(y_k, b_k, j), j+1, q_{\mathsf{READ}}),\\ &\qquad b_2 \dots b_k, \mathsf{R}^k\Big) \quad \text{for } 1 \le j < \log|\Gamma|.\end{aligned}$$

When $j = \log|\Gamma|$, we read the final bit, then pass control to WRITE mode:

$$\begin{aligned}&\delta'\big((q, y_1 \dots y_k, \log|\Gamma|, q_{\mathsf{READ}}), b_1 \dots b_k\big)\\ &\coloneqq \Big((q, \alpha(y_1, b_1, \log|\Gamma|) \dots \alpha(y_k, b_k, \log|\Gamma|), \log|\Gamma|, q_{\mathsf{WRITE}}),\\ &\qquad b_2 \dots b_k, \mathsf{S}^k\Big);\end{aligned}$$

WRITE mode. This mode begins by passing the data that was just read to the original transition function δ. If

$$\delta(q, y_1 \dots y_k) = (q', \gamma_2 \dots \gamma_k, D_1 \dots D_k) \in Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k,$$

we can write the new bits one by one from right to left (given $\gamma \in \Gamma$, we write $\gamma[j]$ for the j-th bit of the binary encoding $\epsilon(\gamma)$, indexed from 1):

$$\begin{aligned}&\delta'\big((q, y_1 \dots y_k, j, q_{\mathsf{WRITE}}), b_1 \dots b_k\big)\\ &\coloneqq \big((q, y_1 \dots y_k, j-1, q_{\mathsf{WRITE}}), \gamma_2[j] \dots \gamma_k[j], \mathsf{L}^k\big) \quad \text{for } 1 < j \le \log|\Gamma|;\end{aligned}$$

$$\delta'\big((q, y_1 \dots y_k, 1, q_{\mathsf{WRITE}}), b_1 \dots b_k\big)$$
$$:= \big((q, y_1 \dots y_k, 1, q_{\mathsf{MOVE}}), \gamma_2[1] \dots \gamma_k[1], \mathsf{S}^k\big);$$

MOVE mode. Now we move, then we restart the cycle, passing the new state before we send it off.

$$\delta'\big((q, y_1 \dots y_k, j, q_{\mathsf{MOVE}}), b_1 \dots b_k\big)$$
$$:= \big((q, y_1 \dots y_k, j+1, q_{\mathsf{MOVE}}), b_2 \dots b_k, D_1 \dots D_k\big) \quad \text{for } 1 \le j < \log|\Gamma|;$$

$$\delta'\big((q, y_1 \dots y_k, \log|\Gamma|, q_{\mathsf{MOVE}}), b_1 \dots b_k\big)$$
$$:= \big((q', y_1 \dots y_k, 1, q_{\mathsf{READ}}), b_2 \dots b_k, D_1 \dots D_k\big).$$

We conclude that it takes $\approx 3\log|\Gamma|$ steps of $\widetilde{M}$ to simulate one step of M.

Exercise 1.3. Given a k-tape TM M described by (Γ, Q, δ), we define a new TM $\widetilde{M}$ with a single read-write tape that simulates M. It has alphabet $\Gamma' = \Gamma \times \{\epsilon, \hat{\cdot}\} \times \{\epsilon, \mathsf{R}\}$, where the hat $\hat{\cdot}$ tracks the head location on each tape and where R is the right marker so that the machine knows where to stop the read sweep. The set of states is

$$Q' := Q \times [k] \times \Gamma'^k \times \{q_{\mathsf{init}}, \dots\};$$

the counter is strictly to keep track of which of the k tapes we are currently simulating, and Γ'^k tracks the k symbols read by the k heads of M at any time. We give a high-level overview of the machine's operation, omitting or being sloppy about the details of δ'.

Initialization. Our first task is to encode the input, placing all the input characters on tape 1 (we will say tape j when we really mean the cells $j, k+j, 2k+j, \dots$ that simulate tape j). The initial state is $(q_{\mathsf{start}}, 1, \square^k, q_{\mathsf{init}})$. Here is what we are trying to do:

▷	h	e	y											

▷			h			e			y					

- Copy the first non-▷ character x into the state register; write a blank to the cell.

$$\delta'((q_{\mathsf{start}}, 1, \square^k, q_{\mathsf{init}}), x) = ((q_{\mathsf{start}}, 2, \square^k, q_{\mathsf{init}}), x, \mathsf{R})$$

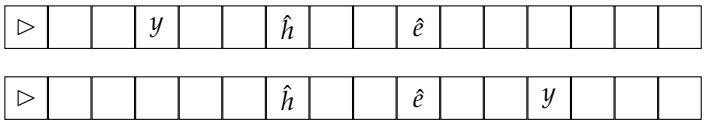
$$\delta'((q_{\mathsf{start}}, 2, \square^k, q_{\mathsf{init}}), x) = ((q_{\mathsf{start}}, 2+1 \bmod k, x\square^{k-1}, q_{\mathsf{init}}), \square, \mathsf{R})$$

- Move the head right until a blank $\square$ is read and we are on tape 1; write $\hat{x}$ to the cell.

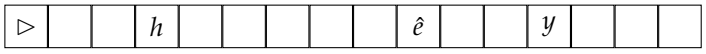
$$\delta'((q_{\mathsf{start}}, 1, x\square^{k-1}, q_{\mathsf{init}}), \square) = ((q_{\mathsf{start}}, k, \square^k, q_{\mathsf{init}}), \hat{x}, \mathsf{L})$$

▷		e	y			$\hat{h}$								

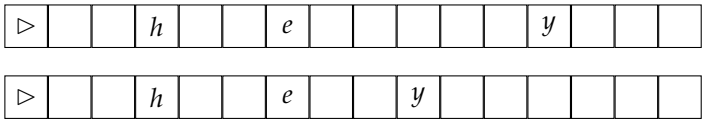
- Move back to $\triangleright$. Then move right until a non-blank symbol is read — copy that symbol x, write a blank to its cell, and then move right until a character with a hat is read. Then move until a blank is read and the counter is 1, then write $\hat{x}$ to the cell.
- Repeat until the first symbol read after moving right from $\triangleright$ has a hat. Then move right to the first blank on tape 1, move k cells to the left, then remove the hat (to mark the end of the input).



- Now we shift the spaced out input back to where it should be. Move to the first hat character. Cut $\hat{x}$ into the state register, move left to the start, then move right and write x to the first blank of tape 1.



- Repeat this, moving right and cutting, then moving left and pasting, the last time being when a non-hat non-blank character is read.



- Finally, move to the start, then write $\widehat{\triangleright}$ to the first k cells, and mark the first empty cell of tape 1.

This concludes the initialization procedure. Notice that every step so far has satisfied the requirements of an oblivious TM.

Read sweep. Sweep from the start to the right marker, storing the hat characters in the registers. When the right marker is read, move it to the next cell of tape 1, then pass the hat characters to the transition function δ. We then store the movements in the Γ^k register — for example, if $\hat{h}$ needs to be moved right, we store $\hat{h}_R$; otherwise we just store $\hat{h}$. Then we sweep back to the start for obliviousness, and pass to the write sweep.

Write sweep. Here we sweep right, updating the heads and symbols to be moved right, then we sweep left, updating the heads and symbols to be moved left. We then pass control back to the read sweep.

In this way, we have simulated a k-tape TM with a single-tape TM that is also *oblivious*. This simulation runs in $O(T(n))$ time.

Remark (Note to self). Admittedly I am uncomfortable with the level of rigor/detail in this chapter, with certain details about Turing machines being glossed over. I think the older books like Papadimitriou or

Hopcroft–Ullman cover such things in more detail, and that Arora–Barak is more focused on getting the reader up to speed with modern stuff. I want to be more comfortable with working on partially specified and ambiguous problems, because that is an important skill at the frontier of research. Then I can dip into the older foundational texts later and retrieve foundational details on a need-to-know basis as and when.

Exercise 1.4.

Exercise 1.5. We designed the machine in exercise 1.3 to be oblivious.

Exercise 1.6. Use the construction of the $T \log T$-time universal TM.

Exercise 1.7. Assuming the tapes of the two-dimensional TM are of the form $\mathbf{N} \times \mathbf{N}$ (here $\mathbf{N}$ includes zero), we may use the Cantor pairing function

$$\begin{aligned} f &: \mathbf{N} \times \mathbf{N} \to \mathbf{N} \\ (i,j) &\mapsto \frac{(i+j)^2 + 3i + j}{2} = \binom{i}{1} + \binom{i+j+1}{2}. \end{aligned}$$

Suppose M has k two-dimensional tapes. We simulate M with a k-tape TM $\widetilde{M}$, encoding each two-dimensional tape with the Cantor pairing function. [This doesn't work! The head doesn't know which cell it is at, only what it is reading.]

Exercise 1.8. We describe a 3-tape TM M that computes LOOKUP. It takes an input like $\triangleright$10110010#101$\square$, which asks for the fifth bit of 10110010, and in this case returns 1. The idea is that we will decrement the index and move the input head until the index reaches zero, and then output the value read by the input head.

We use the alphabet $\{\triangleright, \square, 0, 1, \#\}$.

1. Move the input head and work head right, until the input head reads #.
2. Moving both the input and work heads right, copy the # symbol and the index into the work tape, stopping when the input head reads $\square$.

$\triangleright$	1	0	1	1	0	0	1	0	#	1	0	1				

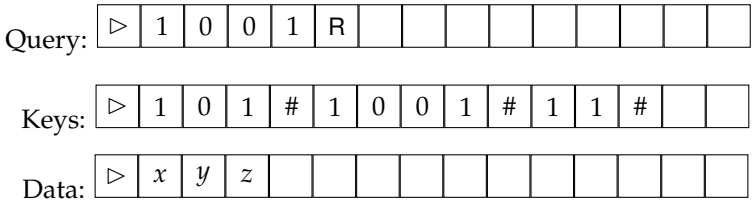
$\triangleright$									#	1	0	1				

3. Move the input head to #, so that it is on the cell before the start of the number to be searched, and also move the work head one cell to the left, so that it is at the start of the index.
4. Now we will repeatedly decrement the index:
   - If the input head reads $\triangleright$, write 0 to the output tape and halt.
   - If the work head reads 0, move it to the left.

- If the work head reads 1, change the 1 to a 0 and move it to the right, writing 1 to each cell until □ is read.
- If the work head reads #, copy the cell currently read by input to the output tape and halt.
- If the work head reads □, move both the input and work heads left.

On an input of length n, the machine halts within $2n^2 + 2n$ steps. Initialization (steps 1 to 3) takes at most $2n$ steps. After that, the input head has to move one cell left at most n times, each time being at most $2n$ steps apart (due to the counter on the work tape decrementing). We conclude that LOOKUP $\in \mathbf{P}$.

Exercise 1.9. The idea is like a hash table. To simulate RAM, we have three tapes — one is the search/query bar to give read or write requests for the RAM, another one stores the memory locations (keys) that have been written to, and the last one stores the bits (data) written to the RAM. For example, suppose someone uses a RAM TM for the following sequence of operations: 101Wx, 1001Wy, 11Wz, 1001R. Then our simulator will end up looking like this:



Each time a write $\llcorner i \lrcorner \mathsf{W} x$ is requested, the keys tape is searched left to right for the index i, with the data head moving right whenever the keys tape reads a #. If the key is found, the data head's current position is overwritten with the symbol x. Otherwise the data head will be on a blank symbol □, and the current key i will be written to the right of the keys tape, with x being written to the right of the data tape. (After a RAM read/write operation is completed, the three heads are moved back to the start.)

I'm not sure about how to do the time analysis here, but I think it's something like this: say the RAM TM takes at most $T(n)$ steps to halt on inputs of length n. Then our simulator will need to simulate at most $T(n)$ many RAM read/write operations. Since each RAM read/write operation sweeps the key tape a fixed number of times, and since the key tape has length $O(T(n)^2)$ (each key has length $O(T(n))$ since the RAM TM can write at most $T(n)$ bits, and there can be at most $T(n)$ keys), we see that our simulation runs in $O(T(n)^3)$ time. [I do not know if it runs in $O(T(n)^2)$ time, and I do not know how to achieve $O(T(n)^2)$ time either.] [*News flash:* From `https://cstheory.stackexchange.com/a/31453/`, I learned that I can instead use two tapes, with one for querying and the other simulating the RAM. The second tape will store both the keys and the data; it can look something like ▷101:x#1001:y#11:z#. Then, since there are

$O(T(n))$ records, and since each record has length $O(T(n))$, the result follows.]

Exercise 1.10.

Exercise 1.11. We describe an encoding system for all Turing machines with some fixed finite alphabet Γ'; we can then encode each symbol of Γ' using $\log|\Gamma'|$ bits, which will give us our desired encoding scheme for Turing machines. We encode a TM M described by (Γ, Q, δ) as a list of ordered pairs — any function $f\colon \{x_1, \dots, x_n\} \to Y$ can be described as a string $(x_1, f(x_1)), \dots, (x_n, f(x_n))$. Since alphabets and state sets can be arbitrarily large for arbitrary TMs, we will encode a symbol in Γ by the symbol A followed by a $\log|\Gamma|$ bit string, and similarly we will encode states by Q followed by a $\log|Q|$ bit string. In this way we can encode every k-tape TM using the alphabet $\{), (, ,, A, Q, 0, 1, \mathsf{L}, \mathsf{S}, \mathsf{R}, \#\}$. Just for concreteness, here is an example of how one tuple in the encoding of a 3-tape TM with 7 states and an alphabet of size 12 might look like:

$$((Q101, A1101, A1011, A0001), (Q011, A1010, A0111, A0101, \mathsf{L}, \mathsf{L}, \mathsf{S}))$$

The entire description of this TM would then consist of such tuples, separated by commas, ordered lexicographically by the inputs. We end the description of the machine with #, and ignore any characters that follow it, so that all TMs admit infinitely many strings — this will be useful later on. Also, malformed strings that don't correspond to any TM will be associated with the trivial TM, that immediately halts on any input.

Notice that we may enumerate Turing machines in this way — every natural number corresponds to some Turing machine, and so the set of all Turing machines is countable.

Exercise 1.12. Given a TM M, we define a function $f_M\colon \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$, where $f_M(x)$ equals the output of M on input x if the machine halts and gives an output, and where we set $f_M(x) = \perp$ if the machine does not halt. This function f_M describes the behavior of M on any input, and we say that M computes f_M. Notice that there can be multiple machines that compute the same f. Also notice that not every function of this form arises from a TM however, as the set of TMs is countable, whereas the set of such functions is uncountable.

In this exercise, we consider the set of *partially computable* functions, defined by

$$\mathbf{PC} \coloneqq \{f\colon \{0,1\}^* \to \{0,1\}^* \cup \{\perp\} \mid f = f_M \text{ for some TM } M\}.$$

Given a non-empty proper subset $\mathcal{S} \subsetneq \mathbf{PC}$, which I think of as a set of Turing machines with some property, we claim that the function $f_{\mathcal{S}}\colon \{0,1\}^* \to \{0,1\}$ defined by

$$\begin{aligned} f_{\mathcal{S}}(\alpha) &\coloneqq [M_\alpha \text{ computes a partially computable function in } \mathcal{S}] \\ &= [f_{M_\alpha} \in \mathcal{S}] \end{aligned}$$

is uncomputable. This is Rice's theorem, which tells us that *all non-trivial semantic properties of programs are undecidable*. Here, we only investigate properties of language that a TM decides, and not the TM itself.

Suppose for contradiction that $M_{\mathcal{S}}$ is a TM that computes $f_{\mathcal{S}}$. We will reduce the halting problem HALT to $M_{\mathcal{S}}$. Define $f_\perp(x) \coloneqq \perp$ for all $x \in \{0,1\}^*$. We consider two cases:

Case 1: $f_\perp \in \mathcal{S}$. Since $\mathcal{S} \subsetneq \mathbf{PC}$, there exists $g \in \mathbf{PC} \setminus \mathcal{S}$. Now, given $\langle \alpha, x \rangle$, we construct a TM M that, on input y,

1. simulates M_α on x and discards the output, then
2. simulates M_g on y and outputs the result.

If M_α halts on x, then M computes g. Otherwise, M computes $f_\perp$. Therefore, $M_{\mathcal{S}}(\llcorner M \lrcorner) = \neg\mathsf{HALT}(\langle \alpha, x \rangle)$, and we are done.

Case 2: $f_\perp \notin \mathcal{S}$. The proof is analogous to case 1, just that we now consider some $g \in \mathcal{S}$, which exists as $\mathcal{S}$ is assumed to be non-empty.

It is also easy to deduce the non-computability of HALT from Rice's theorem — indeed, consider say the set of partial functions that arise from TMs that halt on the input 0. Then this set is a non-empty proper subset of $\mathbf{PC}$, and so membership is undecidable by Rice's theorem. Since $\mathsf{HALT}(\langle x, 0 \rangle)$ would compute membership in this set, we see that HALT must be uncomputable as well.

Exercise 1.13.

Exercise 1.14. (b) While I have nothing to say on these problems at the moment (I will soon remedy this with a proper study of standard algorithms, probably first from Sedgewick–Wayne, and eventually culminating in the grand TAOCP expedition), I just want to note that interestingly you can check triangle-freeness simply by checking if $\operatorname{tr} A^3 = 0$; here A is the adjacency matrix of our graph.

Exercise 1.15.

2 *NP and NP completeness*

In other cases we cannot prove as yet that no decent algorithm for a given problem exists, but we know that such methods are unlikely, because any efficient algorithm would yield a good way to solve thousands of other problems that have stumped the world's greatest experts (see the discussion of ***NP****-completeness in Section 7.9).*

— Donald E. Knuth, *The Art of Computer Programming* 4A (2011)

Remark (On the polytime computability of the Cook–Levin reduction; see also `https://cs.stackexchange.com/q/151929/`). Here I sketch out some details on why the Cook–Levin reduction is polynomial-time computable. If one wants to do this rigorously, one must specify a polynomial-time TM computing $f\colon x \mapsto \varphi_x$. Firstly we can specify the TM with any alphabet Γ we want, and encode each symbol using $\log|\Gamma|$ bits later on. So we will be fine with writing our CNF formula as a string like $(A1 \vee A3 \vee \neg A4) \wedge (\neg A2 \vee \neg A3 \vee A5)$. ...[To do: remarks on modularity; that it suffices to find separate polytime TMs for each of the four clauses in the AND; that we can chain together polytime TMs, etc.]

Exercise 2.1. Let M be a polynomial-time TM and $p\colon \mathbf{N} \to \mathbf{N}$ be a polynomial. We claim that

$$L := \left\{ x \in \{0,1\}^* : \begin{array}{l} \text{there exists } w \in \{0,1\}^* \text{ such that} \\ |w| \leq p(|x|) \text{ and } M(x,w) = 1 \end{array} \right\} \in \mathbf{NP}.$$

Define a TM M' that takes an input of the form (x,u) where $u \in \{0,1\}^{p(|x|)}$ and simulates $M(x,u_1)$, $M(x,u_1u_2)$, ..., $M(x,u_1\dots u_{p(|x|)})$, accepting if any one of these simulations with truncated certificates accepts, and rejecting otherwise. If $x \in L$, then $M'(x,w0\dots0)$ (where $w0\dots0$ is of length $p(|x|)$) accepts, since one of the truncations yields $M(x,w)$. Conversely, if $M'(x,u)$ accepts for some $u \in \{0,1\}^{p(|x|)}$, then some truncation of u is a certificate of length at most $p(|x|)$, and so $x \in L$ by definition.

Exercise 2.2. (i) A certificate would be a two-coloring $c\colon V(G) \to [2]$. A standard greedy algorithm (BFS?) suffices.

(ii) A certificate would be a two-coloring $c\colon V(G) \to [3]$. I can't think of any polynomial-time algorithm for this. [Turns out it is **NP**-complete. Call me if you prove 3COL $\in$ **P**.]

(iii) A certificate would be a permutation of the vertices of the graph, say $v_{\sigma(1)},\dots,v_{\sigma(n)}$, such that for each $1 \leq i \leq n$, the induced subgraph $G[v_{\sigma(1)},\dots,v_{\sigma(i)}]$ is connected. Then the verification procedure would be to, given the i-th connected induced subgraph, check whether there is an edge between $v_{\sigma(i+1)}$ and $v_{\sigma(j)}$ for $1 \leq j \leq i$. A breadth-first search (BFS) would give a polynomial-time algorithm for this problem.

Exercise 2.3. (I think Gaussian elimination gives something like an $O(n^3)$ algorithm for LINEQ?)

Exercise 2.4.

Exercise 2.5.

Exercise 2.6.

Exercise 2.7. (2) Let $L' \in \mathbf{NP}$. Then, since L is **NP**-hard, we have $L' \leq_p L$. Since $L \in \mathbf{P}$, we conclude that $L' \in \mathbf{P}$ as well; thus $\mathbf{P} = \mathbf{NP}$.

(3) Since **NP**-complete languages are **NP**-hard by definition, the forward implication follows from (2). Conversely, suppose $\mathbf{P} = \mathbf{NP}$. Then, since L is **NP**-complete, it is in $\mathbf{NP} = \mathbf{P}$.

Exercise 2.8. Let $L \in \mathbf{NP}$. We prove that HALT is **NP**-hard by giving a Karp reduction $L \leq_p \mathsf{HALT}$. Let M be a verifier for L that runs in polynomial time $p(n)$. Define a TM M' that, on input x, simulates $M(x,u)$ for all $u \in \{0,1\}^{p(|x|)}$, outputting 1 if this simulation accepts at any time, and entering an infinite loop if none of the u are accepted. Given $x \in \{0,1\}^*$, we define $f(x) \in \{0,1\}^*$ by $f(x) := \langle \ulcorner M' \urcorner, x \rangle$. Then $x \in L$ iff $f(x) \in \mathsf{HALT}$, and the result follows as f is computable in polynomial time.

Why is f computable in polynomial time? Let us describe precisely the TM that computes f. [do it!!!]

However, $\mathsf{HALT} \notin \mathbf{NP}$, since $\mathbf{NP} \subset \mathbf{EXP}$, and languages in **EXP** are decidable.

Exercise 2.9. In exercise 2.8, we saw that $\mathsf{SAT} \leq_p \mathsf{HALT}$. If $\leq_p$ were symmetric, then we would have $\mathsf{HALT} \leq_p \mathsf{SAT}$, which implies that SAT is uncomputable, contradicting the fact that SAT admits a brute-force algorithm.

Exercise 2.10. Suppose $L, L' \in \mathbf{NP}$ with polytime verifiers M and M' that run in time $p(n)$ and $p'(n)$ respectively. Then we may concatenate the certificates to form a new certificate. That is, we define a polytime verifier N for $L \cap L'$ that, on input (x,u), simulates M on $(x, u_1 \dots u_{p(|x|)})$, then if an accept state is reached, simulates M' on $(x, u_{p(|x|)+1} \dots u_{p(|x|)+p'(|x|)})$, accepting if M' accepts. Similarly we may define a machine that decides $L \cup L'$ — just accept if at least one of the simulations accepts.

Exercise 2.11. Write L for the language in the question. Then $L \in \mathbf{NP}$ since, given $\langle \phi, 1^n \rangle$, a proof of size at most n of ϕ is a certificate; the idea is that a proof in an axiomatic system can be mechanically verified step-by-step, by checking if each step is consistent with the state obtained from all the previous steps, or something like that. (Also, I would assume that each step of a formal proof has a fixed size.)

For **NP**-completeness, we could reduce $\mathsf{SAT} \leq_p L$ by sending a Boolean formula φ to $(\phi, 1^n)$, where ϕ is the ZF statement that ϕ is satisfiable, and n is the number of variables in ϕ, or some polynomial of that.

Exercise 2.12.

Exercise 2.13. (a)

(b) [I learned this online.] The usual reduction SAT $\leq_p$ 3SAT looks like this:

$$(x_1 \vee x_2 \vee x_3 \vee x_4) = (x_1 \vee x_2 \vee z) \wedge (x_3 \vee x_4 \vee \overline{z}).$$

The problem is that this is not parsimonious — say for $x = 1001$, we see that $z = 0$ and $z = 1$ both give satisfying assignments for the resulting 3CNF formula. The trick is then to force $z = 0$ if x_1 or x_2 is true:

$$(x_1 \vee x_2 \vee x_3 \vee x_4) = (x_1 \vee x_2 \vee z) \wedge (\overline{x_1} \vee \overline{z}) \wedge (\overline{x_2} \vee \overline{z}) \wedge (x_3 \vee x_4 \vee \overline{z}).$$

Repeatedly reducing clauses with more than four literals in this way then gives us our desired parsimonious reduction.

Exercise 2.14. If L is polynomial-time Cook reducible to L', we write $L \leq_T^P L'$, since this notion is sometimes called "polynomial-time Turing reduction." Suppose $L \leq_T^P L'$ and $L' \leq_T^P L''$; thus there exists a polynomial-time TM M that decides L, given an oracle for L', as well as a polynomial-time TM M' that decides L', given an oracle for L''. We wish to prove that $L \leq_T^P L''$, so we must construct a polynomial-time TM M'' that decides L, given an oracle for L''. Giving the oracle for L'' to M', we see that M' is now a polynomial-time oracle for L'. We may then give this oracle to M — it may not be an instant magical oracle, but it gets the job done, since all we need is polynomial-time in the end. And so this lets M decide L in polynomial-time, so we are done.

Let me put it another way. The polynomial-time TM M'' takes an input x, and simulates M, until a point where the oracle for L' is called, in which case M' gets called. Now, we simulate M' until the oracle for L'' is called, which is fine since we do have an oracle for L'' by hypothesis. And so we continue in this way; since polynomials are closed under composition, we find that the TM we construct is indeed polynomial-time.

We prove that 3SAT $\leq_T^P$ TAUTOLOGY. In other words, we construct a polynomial-time TM M that decides 3SAT given an oracle that decides TAUTOLOGY. Given a 3CNF formula ϕ, we negate it and pass $\neg\phi$ to the oracle. If $\neg\phi \in$ TAUTOLOGY, then ϕ is not satisfiable, and so we reject. Otherwise, we accept. (Note that we only reduce it to a DNF; that is, we flip all the literals and swap $\wedge$ and $\vee$. The definition of TAUTOLOGY only requires Boolean formulae; not necessarily in CNF form.)

Exercise 2.15. We have a reduction INDSET $\leq_p$ CLIQUE, which sends a graph to its complement: $\langle G, k\rangle \mapsto \langle \overline{G}, k\rangle$. This is polytime computable; we are just flipping non-diagonal bits in the adjacency matrix.

As for INDSET $\leq_p$ VERTEX COVER, we first establish the following lemma: Let G be an n-vertex graph. Then G has an independent set of size at least k if and only if G has a vertex cover of size at most

$n - k$. *Proof.* Let $S \subset V(G)$ be an independent set of size at least k. Then we claim that its complement $V(G) \setminus S$ is a vertex cover for G. Indeed, given an edge $v - w$ of G, we cannot have both v and w in S; thus one of them belong to $V(G) \setminus S$, which gives the claim. The converse is similar.

As a result, we see that $\langle G, k \rangle \mapsto \langle G, n - k \rangle$ is our desired reduction.

Exercise 2.16.

Exercise 2.17.

Exercise 2.18.

Exercise 2.19.

Exercise 2.20.

Exercise 2.21.

Exercise 2.22.

Exercise 2.23. As we saw in claim 2.4, **P** $\subset$ **NP**. Since **coP** $\subset$ **coNP**, it suffices to prove that **P** $=$ **coP**. This follows, since given a language $L \in$ **P**, we can decide if $x \in \overline{L}$ by taking the polytime TM M that decides L, and negating its output.

Remark. It may seem like the argument in exercise 2.23 can be used to show that **NP** $=$ **coNP**, but this is not the case due to the asymmetry in the definition of a NDTM. In particular, negating the output of an NDTM only gives us an *algorithm* for deciding the complement of a language in **NP**, but not an instantiation of that algorithm as a NDTM.

Exercise 2.24. Suppose $L \subset \{0,1\}^*$ is a language such that $\overline{L} \in$ **NP**. Then there exists a polynomial-time TM M that runs in time $p(n)$ such that $x \in \overline{L}$ iff there exists $u \in \{0,1\}^{p(|x|)}$ such that $M(x,u) = 1$. Thus $x \in L$ iff $M(x,u) \neq 1$ for all $u \in \{0,1\}^{p(|x|)}$; inverting the output appropriately gives the result. The proof of the converse follows the same ideas.

Exercise 2.25. If **P** $=$ **NP**, then **coP** $=$ **coNP**, and so **NP** $=$ **P** $=$ **coP** $=$ **coNP**.

Exercise 2.26. We know that 3SAT is **NP**-complete and TAUTOLOGY is **coNP**-complete. So if **NP** $=$ **coNP**, then both problems are **NP**-complete, and so we have reductions 3SAT $\leq_p$ TAUTOLOGY and TAUTOLOGY $\leq_p$ 3SAT. Conversely, we first note that if $L \in$ **NP** and $L' \leq_p L$, then $L' \in$ **NP** as well, since given a verifier M for L and a reduction f, we may define a verifier M' for L' that, on input (x,u), simulates $M(f(x),u)$. Similarly, if $L \in$ **coNP** and $L' \leq_p L$, then $L' \in$ **coNP**. Therefore, given $L \in$ **NP**, we have $L \leq_p$ 3SAT $\leq_p$ TAUTOLOGY, which implies that $L \in$ **coNP**, and vice versa; thus **NP** $=$ **coNP**.

Exercise 2.27. Recall that $\mathbf{NEXP} = \bigcup_{c \geq 1} \mathrm{NTIME}(2^{n^c})$. Just like how **NP** could be defined in terms of a verifier, we give an alternative definition $\mathbf{NEXP}'$ using a verifier, and prove that $\mathbf{NEXP} = \mathbf{NEXP}'$. We say that a language $L \subset \{0,1\}^*$ is in $\mathbf{NEXP}'$ if there exist polynomials $p, q \colon \mathbf{N} \to \mathbf{N}$ and a TM M, called the verifier, such that

1. *The running time of $M(x,u)$ depends only on the length of x, and this dependence is exponential.* That is, for all $x, u \in \{0,1\}^*$, M runs in time $2^{p(|x|)}$ on input (x,u).
2. *The certificate has length exponential in x.* That is, $x \in L$ if and only if there exists $u \in \{0,1\}^{2^{q(|x|)}}$ such that $M(x,u) = 1$.

Suppose $L \in \mathrm{NTIME}(2^{n^c}) \subset \mathbf{NEXP}$. Then, there exists a 2^{n^c}-time NDTM N deciding L. The sequence of nondeterministic choices leading to an accept state for some $x \in L$ form a certificate u of length $O(2^{n^c})$. We may thus define a deterministic TM M that, on input (x,u), simulates N on the input x, following the sequence of choices encoded by u. Then, for all $x, u \in \{0,1\}^*$, M runs in time $2^{O(|x|^c)}$ on input (x,u), which proves (1). To prove (2), we observe that $x \in L$ iff there exists a sequence of $O(2^{|x|^c})$ nondeterministic choices that lead to an accept state for N; iff there exists a binary string $u \in \{0,1\}^{2^{O(|x|^c)}}$ encoding the nondeterministic choices of N that lead to an accept state, so that $M(x,u) = 1$. Therefore, $L \in \mathbf{NEXP}'$.

Conversely, suppose $L \in \mathbf{NEXP}'$, and let p, q be polynomials and M be a verifier as in the definition of $\mathbf{NEXP}'$. Then we may define an NDTM N that, on input x, guesses a certificate $u \in \{0,1\}^{2^{q(|x|)}}$, and simulates M on the input (x,u), entering an accept state if $M(x,u) = 1$ for any of its guesses, and rejecting otherwise. Since $p(|x|) = O(2^{|x|^c})$ for some $c \geq 1$, we see that N is an NDTM deciding L in $\mathrm{NTIME}(2^{n^c})$, and so $L \in \mathbf{NEXP}$ as needed.

Exercise 2.28. Define a language

$$L = \left\{ \langle \alpha, x, 1^n, 1^t \rangle : \begin{array}{l} \text{there exists } u \in \{0,1\}^{2^n} \text{ such that } M_\alpha \\ \text{outputs 1 on input } \langle x, u \rangle \text{ within } 2^t \text{ steps} \end{array} \right\}.$$

To verify $L \in \mathbf{NEXP}$, we use the definition of **NEXP** given in exercise 2.27. We define a verifier M that, on input $(\langle \alpha, x, 1^n, 1^t \rangle, u)$, computes $M_\alpha(x,u)$; this runs in $O(2^t)$ steps by hypothesis, and thus in $O(2^{|\langle \alpha, x, 1^n, 1^t \rangle|})$ steps. That the certificate has length exponential in the input length follows immediately from how we defined L, and so we conclude that $L \in \mathbf{NEXP}$.

Now we describe a reduction from any language $L' \in \mathbf{NEXP}$ to L. Let p, q be the polynomials associated to L', and M be its verifier. Then the reduction

$$x \mapsto \langle \llcorner M \lrcorner, x, 1^{q(|x|)}, 1^{p(|x|)} \rangle$$

is polynomial-time computable, and so L is **NEXP**-complete.

Finally, if $L \in \mathbf{EXP}$, then for any $L' \in \mathbf{NEXP}$, since $L' \leq_p L$, we may use the reduction f to obtain an exponential-time algorithm

$M \circ f$ for L', where M is an exponential-time TM deciding L. Since **EXP** $\subset$ **NEXP**, it follows that **EXP** $=$ **NEXP**.

Exercise 2.29. Since $L_1 \oplus L_2 = (L_1 \cup L_2) \setminus (L_1 \cap L_2)$, it suffices to prove that **NP** $\cap$ **coNP** is closed under finite unions, finite intersections, and complements. By exercise 2.10 and De Morgan's laws, we see that both **NP** and **coNP** are individually closed under finite unions and finite intersections. Since the complement of an **NP** language is a **coNP** language, and vice versa, the result follows.

Exercise 2.30. [I didn't figure this out myself, but I want to write it down to understand it better.] Let $U \subset 1^*$ be a unary language that is **NP**-hard, and let $f\colon \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time reduction $\mathsf{SAT} \leq_p U$, so that $|f(x)| \leq C|x|^d$ for some constants C and d. We describe a polynomial-time TM M deciding SAT. Given a CNF formula ϕ with variables $x_1, \ldots, x_n$, we describe an n-step algorithm that maintains a set of pairs of the form $(\psi, f(\psi))$, as well as the invariant that after each step, the original formula ϕ is satisfiable iff at least one of the formulas ψ in our set is satisfiable. We start with the set $\{(\phi, f(\phi))\}$. If $f(\phi) \notin 1^*$, then ϕ is not satisfiable, and the algorithm rejects. Otherwise, we add the two pairs $(\phi|_{x_1=\top}, f(\phi|_{x_1=\top}))$ and $(\phi|_{x_1=\bot}, f(\phi|_{x_1=\bot}))$ to our set. We then discard any pair not of the form $(\psi, 1^k)$, as well as repeat pairs, so that we only have one pair of the form $(\psi, 1^k)$ for any k. (We do this because, if $f(\psi) = f(\psi')$, then either both ψ and ψ' are satisfiable, or neither are.) We continue in this way, substituting variable assignments into the formulas, and pruning the search tree at each step. While the original tree has 2^n nodes, the fact that our reduction f is polynomial-time computable means that most of these nodes end up getting pruned. The crucial fact is that there are only Cn^d possible pairs $(\psi, 1^k)$ that can survive — at any step, our set has at most Cn^d pairs.

After n steps, we find that the original formula ϕ is satisfiable if and only if our set contains the pair $(\top, f(\top))$. Thus, we have a polynomial-time algorithm for deciding SAT.

Exercise 2.31.

Exercise 2.32.

Exercise 2.33.

Exercise 2.34.

3 Diagonalization

I dedicate this essay to the two-dozen-odd people whose refutations of Cantor's diagonal argument . . . have come to me either as referee or as editor in the last twenty years or so.
Sadly these submissions were all quite unpublishable; I sent them back with what I hope were helpful comments. A few years ago it occurred to me to wonder why so many people devote so much energy to refuting this harmless little argument — what had it done to make them angry with it?

— Wilfrid Hodges, *An Editor Recalls Some Hopeless Papers* (1998)

Exercise 3.1. [Had to look this up.] Let L denote the language defined in the question. We can reduce the halting problem to L as follows: given an instance $\langle M, x \rangle$ of the halting problem, we define a new machine N_x that, on input y, first simulates M on x for $|y|$ steps. If the simulation does not halt within this time, then N_x halts. Otherwise, it runs an extra $|y|^3 + 100|y|^2 + 200$ arbitrary steps before halting. Then M fails to halt on x if and only if N_x runs in $100n^2 + 200$ time on *all* inputs of size n. (If M halts on x, then it does so in constant time t, and so $N_x(1^t)$ will exceed the time bound.)

[See `https://mathoverflow.net/a/28060/` for interesting ideas on such "Rice-type results."]

Exercise 3.2.

Exercise 3.3.

Exercise 3.4.

Exercise 3.5.

Exercise 3.6.

Exercise 3.7.

Exercise 3.8.

[See also Terry Tao's relativization story: `https://terrytao.wordpress.com/2009/08/01/pnp-relativisation-and-multiple-choice-exams/`]

Exercise 3.9.

Now I am shewn the diagonal procedure and told: "Now here you have the proof that this ordering can't be done here". But I can reply: "I don't know — to repeat — what it is that can't be done *here".*

— Ludwig Wittgenstein, *Remarks on the foundations of mathematics* (1956)