Turning Points in the Information Sciences
Tutorial on Turing Machines and the Halting Problem

In the most recent lecture, you learned about Turing Machines and the Halting Problem. These are difficult concepts to understand. Turing Machines are very abstract. It seems hard to believe that they really are as powerful as the fanciest of modern computers. And the proof that the Halting Problem is undecidable at first looks more like a dirty trick than a mathematical proof. This tutorial is about getting a better intuitive understanding of these concepts. This will also help you get a grip on the corresponding homework assignment.

Form a group of two to five students (three probably would be ideal). Then work through the exercises below, step by step. They mostly are about explaining things to one another. Each exercise is annotated with a suggested duration in minutes. If you need significantly more time, stop and move on to the next exercise. After the tutorial, spend some time on your own to go over everything once more and to tie up any remaining loose ends.

Exercise 1: Halting Problem (30m)

Task: *Work through the proof of the undecidability of the Halting Problem presented in class!*

(a) **Structure.** First, try to understand the high-level structure of the argument. It's what is known as a *proof by contradiction*. To show that there exists no program that would solve the Halting Problem (HP), we start by assuming the opposite, namely that there in fact *does* exist such a program. We can then reason about this hypothetical program. If we can show that our assumption ultimately leads to a contradiction, then we know that that assumption must have been wrong. So then we know that there really exists no program that would solve the HP. In other words, our proof has this structure:

> **Claim:** No program can solve the HP. (This is what we want to show.)
> **Assumption:** The claim is false. Such a program *does* exist. Let's call it H.
> ⟨*some clever stuff happening here*⟩
> **Contradiction!** We arrived at statement that is obviously false.
> **Conclusion:** So our assumption must be false. Thus, the claim is true. QED

(b) **Black box.** It's important to understand that assuming that H *exists* does not require us to have even the foggiest idea of how H might work. We simply assume *that* it works. Indeed, our ultimate goal is to show that this existence assumption is wrong, so trying to figure out how H might work not only would be unnecessary but also hopeless.

Now draw a "black box" diagram that represents H, just as done on the slides:

- H has two incoming arrows, representing the two inputs to H: one arrow for the program P to be tested, and one for the data x for which P is to be tested.
- H has two outgoing arrows, representing its two possible outputs: *yes* or *no*, indicating the (correct!) answer to the question "does P halt on x?".

We're still missing the "clever stuff" in the middle of the proof. So let's do that part.

(c) **New program.** Given that we assumed the existence of H, we can construct a new program $H^\star$ that uses H as a subroutine. Now draw the diagram for $H^\star$:

- $H^\star$ takes a single input, namely (the source code of) a program P of our choice.

- This is what $H^\star$ does. First, $H^\star$ calls H, using P *both* as the "program input" and the "data input". (You can do that: at the end of the day, every program is also a piece of data; it's just a string of symbols after all.) Let's write $H(P, P)$ for the result of this computation, which—by definition—will be either *yes* (in case P halts on P) or *no* (in case it doesn't). Then $H^\star$ uses this result as follows:
  - If $H(P, P)$ equals *yes*, continue by entering an infinite loop.
  - Otherwise, stop.

  So in case P halts on P, our program $H^\star$ will keep running forever; and in case P does not halt on P, our program $H^\star$ will halt as soon as it confirmed this fact.

(d) **Contradiction.** Now let's think a bit about this new program $H^\star$. How does it behave? Specifically, how does it behave if you use (the source code of) $H^\star$ *itself* as the input? Will it halt eventually or will it keep going forever? Let's explore both possibilities:

- Suppose $H^\star$ eventually halts on $H^\star$. For this to happen, $H(H^\star, H^\star)$ must equal *no*. But that means that $H^\star$ does *not* halt on $H^\star$. That's a contradiction.
- Suppose $H^\star$ never halts on $H^\star$. For this to happen, $H(H^\star, H^\star)$ must equal *yes*. But that means that $H^\star$ actually *does* halt on $H^\star$. That's also a contradiction.

There are no other cases. So we found a contradiction for every possible scenario. In other words, we managed to derive a contradiction from our original assumption that some program H that can solve the HP supposedly exists.

This concludes the proof (read again the first part above to make sure you really understand it does): no program out there can possibly solve the HP.

Exercise 2: Program Correctness (15m)

Task: *Show that the problem of checking program correctness is another undecidable problem!*

During the lecture, we saw the claim that, not only, humanity had not yet figured out how to check whether a given program might generate an error when run with a given piece of data as input, but also that humanity will never manage to build a machine that would be able to perform this task. But details were a bit vague and no reason was given for this extraordinary claim. This exercise is about understanding this claim and the fact that it really is true.

(a) **Problem definition.** We start by giving a clear definition of the problem we want to investigate. Consider the following question regarding the behaviour of programs:

"Given a program P and some data x, will P generate an error when applied to x?"

Let's call this the *Correctness Problem* (CP). Solving it means writing a program that will answer our question for any P and any x. The claim is that doing so is impossible.

(b) **Understanding the problem.** We will try to prove this claim by relating it to what we know about the HP. But let's first eliminate some possible misunderstandings.

First, what's an error? Probably the best-known example is *division by zero*. Recall that you cannot divide a number (say, 42) by 0. The fraction $42/0$ is not a well-defined number (in particular, ∞ is not a number!). If you write a program that asks the computer to perform such a division operation, it will generate an error at runtime.

It's important to realise that a program running forever is not an error in the technical sense in which we use this term here. On the contrary! Some kinds of programs we

actually *want* to keep running forever. Think of the software monitoring operations in a nuclear reactor. Or think of the operating system running on your laptop.

We will think of a program generating an error as that program *crashing*, and thus also halting. In practice, not every error will lead to a crash. So, to be precise, our analysis here only pertains to these most serious of errors, the ones causing a total crash.

(c) **Structure.** To show that we cannot solve the CP we are going to *reduce* the HP to it. What this means is that we are going to show that any method that solves the CP can be turned into a method that solves the HP. As we already know that the HP cannot be solved, this means that the CP cannot be solved either.

Write down the high-level structure of this argument in a similar fashion as we did above for the first exercise (*Claim*, *Assumption*, ..., *Conclusion*). Before continuing, convince yourself that, once we have filled in the "clever stuff" in the middle (i.e., the reduction itself), we really have a valid argument for the undecidability of the CP.

(d) **Reduction.** So how can you reduce the HP to the CP? Suppose, for the sake of contradiction, we have a magic program C that can, for any program P and any data x, tells us (correctly!) whether P will generate an error when applied to x.

For any given program P (that might or might not halt) we can construct a variant P' of that program (that might or might not be errorneous) as follows. For any data x:

- First, execute P with input x and catch any errors.
- Then, divide 42 by 0.

Think about how this altered program P' behaves. If P does not halt (and thus also never generates an error), P' will be busy with the first line forever and never reach the part where it would have to divide 42 by 0. If P does halt (with or without an error), then P' will reach the second line and immediately generate an error. So we know:

$$P \text{ halts on } x \text{ if and only if } P' \text{ generates an error when applied to } x.$$

But this means that, to find out whether P halts on x, we can first construct the corresponding P' and then ask our magic program C to check whether P' generates an error when applied to x. So we have successfully reduced the HP to the CP.

Now convince yourself that this really concludes the proof.

Exercise 3: Turing Machine Simulator (15m)

Task: *Play around with one of the many Turing Machine simulators you can find online!*

A nice one is available at `turingmachinesimulator.com`. Don't worry too much about the details. People might use slightly different definitions of what a Turing Machine is than we have done, and that's ok. All of these definitions are ultimately equivalent. Run through a couple of the examples available online to get a feel for how the machine executes a program.

Exercise 4: Turing Machines and Palindromes (30m)

Task: *Program a Turing Machine to check whether a given string of letters is a palindrome!*

(a) **Palindromes.** A palindrome is a word that reads the same backwards and forwards. Examples include ABBA and MADAM. For the sake of simplicity, we will restrict attention to words that only use the letters A and B. They need not be words in English, but any string of letters is fine. So ABBA and BBB are palindromes, but BABA is not.

(b) **Problem specification.** What do we want to achieve? In the initial state, there is a finite string of A's and B's on the tape, with the head of the Turing Machine reading the first letter. We are looking for a program that will delete the string in case it is a palindrome, and that otherwise will stop without deleting the entire string.

The objective is to arrive at a general understanding how one would go about writing such a program. You can also work out all the details and test your program on `turingmachinesimulator.com`, but that definitely will take longer than 30 minutes.

(c) **Idea.** To get an idea for how to approach this problem, put yourself in the shoes of the Turing Machine. You can only look at a single cell of the tape at a time. You can memorise as much information as you like, but you must do so by giving every state of your mind you might possibly experience during the computation a unique name (such as "state 42") and you must restrict yourself to using the information encoded in this manner. So think carefully about what information you really need to memorise, and what information you can happily forget. Then, based on your current state of mind and the symbol you see on the tape, you must decide on three things: which state to change to, which symbol to write, and whether to move left or right.

The most important trick you need to know about when writing programs for a Turing Machine (and, frankly, and other kind of program as well) is this: try to *decompose* the complex task you are facing (here: of checking for palindromehood) into simpler subproblems. Before reading on, discuss amongst yourself how to get started.

So here's how you can check whether a given word is a palindrome:

- Read the first letter of the word. Remember whether it's an A or a B. Delete it.
- Move to the right, until you reach the end of the word. (*How do you do that?*)
- Check whether the last letter is the same as the one you remembered.
  - If *no*, stop (as then it's definitely not a palindrome).
  - If *yes*, delete the last letter and move back to the front of the word.
- Repeat these three steps until there's nothing left to do.

A crucial observation here is that the problem you are left with after you have deleted the first and the last letter of the word (and returned to the front) has *the same structure* as the original problem. It's just a slightly smaller version of that original problem. This is known as *recursion*. It is a central concept in Computer Science and you'll encounter it again and again in the years to come. For example, if you start out with AABBAA, then after the first round you are left with ABBA. Whether ABBA is your original word or whether it is a subproblem you encounter while you are dealing with AABBAA does not matter. Your approach is the same in both cases. Sounds kind of obvious? But when fully understood, it is an extremely powerful programming technique.

(d) **Implementation.** Now write your program. What states do you need? For each state, the head might read a *blank* (␣), an A, or a B (you won't need any additional symbols for this particular program). So for each state and each of these three symbols, write down a rule that determines which state to change to, which symbol to write, and which

direction to move in. If you omit the rule for a certain state/symbol-combination, this means that the program will stop when the machine reaches that configuration.

Remember the thing about decomposition. Start by writing small programs for specific subtasks, and then think about how to put everything together.

By convention, the starting state is usually called q_0. You could do that and you could call the other states q_1, q_2, and so forth, but it might be more helpful if you pick descriptive names, such as `q_AT_END_A`, to refer to the state where you read A as the first letter of the word and you are now at the end of the word.

Here is a small part of the program you need to write:

> In state `q_START` upon reading A, switch to `q_READ_A`, write ␣, and move right
>
> In state `q_READ_A` upon reading A, switch to `q_READ_A`, write A, and move right
>
> In state `q_READ_A` upon reading B, switch to `q_READ_A`, write B, and move right
>
> In state `q_READ_A` upon reading ␣, switch to `q_AT_END_A`, write ␣, and move left

In case the first letter of the word is an A, this program fragment will have the Turing Machine move across the entire string and then stop above the final letter.

Now complete this program to obtain a full program for recognising palindromes. It will be roughly three times as long as the program fragment above.