2. TURING MACHINES

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

October 2024

In the first part of the AA course we will focus our attention on what it is exactly that computers can or cannot do. We shall study these issues within the framework of "computability theory", a branch of mathematics. This means that our approach will bear all the characteristics of mathematical rigor: we will try to capture intuitive concepts as formal definitions and logical axioms; then we will use inference rules to derive new statements that are *provably true* in our framework.

Our main question can be informally phrased as: *"What problems can algorithms solve?"* In order to approach this question, we shall first need to precisely define what we mean when we use the vague terms: "problems", "algorithms" and "solve".

The concept of "problem" has been addressed in the previous lecture; for our purposes, a problem is a function from a set of strings to another. A key part of this definition is that both the domain and the codomain can be infinite (and the domain usually is), but all their member strings must be *finite* in length.

We now turn our attention towards what an algorithm is and what it means for an algorithm to solve a problem.

1 Informal view

A Turing Machine is a *computational model* proposed by Alan Turing in 1936. It is a rigorous mathematical abstraction, designed to capture the intuitive notion of "computability". However, it can be visualized as a physical device with tangible components and visible *movement*, which makes it very easy to think about without delving into symbolic notation.

- A Turing Machine consists of a **tape**, a reading **head** and an internal **state**.
- The tape is one-dimensional and split into equally-sized **cells**. It extends indefinitely in either direction, so we can never "run out of tape".
- The machine works by going through a series of **steps**. At each step:
  - The machine is in some state and its head is reading some cell.
  - Depending on the combination of internal state and currently read symbol, the machine makes a particular **transition**.
  - When transitioning, the machine does three things:
    1. It goes to a new internal state.[†]
    2. It writes a new symbol on the cell under the head.[‡]
    3. It moves the head to point to the cell immediately on the left or right of the current one; or leaves it where it is.

[†] This "new" state can be the same as the old one; i.e. the machine remains in the same state.
[‡] As with the states, this "new" symbol can be the same as the old one; i.e. the machine leaves the cell unchanged.

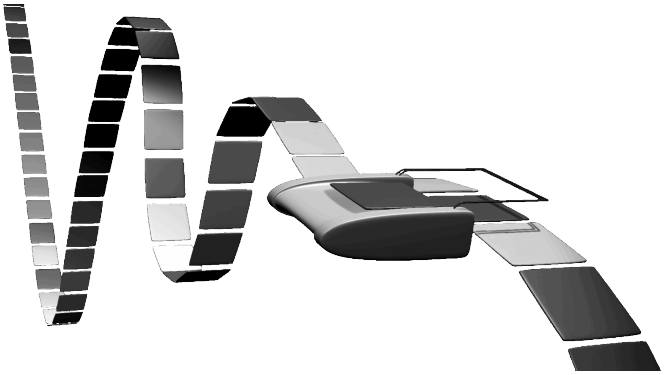

Figure 1: Artistic visualization of a Turing Machine. You can see the tape, split into cells and unbounded in both directions. Here, tape symbols are illustrated as different shades of gray. The printer-like object depicts the "internal control" of the machine – the part that knows how to transition and which memorizes the current internal state. The thin rectangle on its right, hovering above a single cell, is the head.
Image taken from: `https://commons.wikimedia.org/wiki/File:Maquina.png`

2 Formal definition

We shall now summarize the informal presentation from section 1 into a formal definition.

Definition 2.1. A Turing Machine is a 6-tuple: $(Q, \Sigma, \Gamma, B, q_1, \delta)$, where:

- Q is a finite set of **internal states**.
- Σ is a finite set of **symbols** used for the machine input (the word that is initially present on the tape) called **the input alphabet**
- Γ is a finite set of **symbols** that the machine can use during its operation, called **the tape alphabet**; $\Sigma \subsetneq \Gamma$
- B is **the blank symbol**, the default state of a cell that hasn't been yet written, or that has been erased; $B \in \Gamma; B \notin \Sigma$
- q_1 is **the initial state** of the machine before any step is taken; $q_1 \in Q$
- δ is **the transition function** which dictates the behavior of the machine during a single-step (note that the machine can transition into the special states Y, N, H, but cannot "move on" from any of those states):

  $$\delta : Q \times \Gamma \to (Q \cup \{Y, N, H\}) \times \Gamma \times \{\leftarrow, -, \rightarrow\}$$

  The domain captures the idea that a machine's transition is influenced only by the current state and the current symbol read by the head. Based on this pair, the machine makes a transition, changing its state (possibly into one that halts computation), changing the symbol under the head and moving the head: either one cell to the left/right, or keeping it stationary.
  Note that the codomain refers to three special states:

  - Y is **the accepting state**; once the machine reaches this state, its run is over and it signals that the input is **accepted**, i.e. the answer is "yes".
  - N is **the rejecting state**; once the machine reaches this state, its run is over and it signals that the input is **rejected**, i.e. the answer is "no".
  - H is **the halting state**; once the machine reaches this state, its run is over and it signals that the computation is done and the answer to the problem is encoded as the current contents of the tape, starting from the first non-blank character and ending with the last. This is useful for function problems, where the answer is a mathematical object other than a boolean.

In the interest of brevity, we will use Q' to denote the extended set of states $Q \cup \{Y, N, H\}$.

We will also use Γ' to denote $\Gamma \setminus \{B\}$, which we will need to refer to quite often.

The "states" and "symbols" should be thought of as atomic objects. To avoid any confusion, we'll always consider these sets to be disjoint: $\Gamma \cap Q' = \emptyset$.

Similarly, the three head movement directions are not members of any of these sets:

- $\{\leftarrow, -, \rightarrow\} \cap \Gamma = \emptyset$
- $\{\leftarrow, -, \rightarrow\} \cap Q' = \emptyset$

3 Machine description

Having established a formal definition of Turing Machines, we can now start describing individual machines that accomplish various tasks.

Here is our first machine, which we shall name `isEven`. It take its input – a big-endian[§] binary number and checks whether it is even (i.e. the last digit is 0).

$isEven = \{\{q_1, q_2\}, \{0, 1\}, \{\square, 0, 1\}, \square, \delta_{isEven}\}$

$\delta_{isEven}(q_1, 0) = (q_1, 0, \rightarrow)$

$\delta_{isEven}(q_1, 1) = (q_1, 1, \rightarrow)$

$\delta_{isEven}(q_1, \square) = (q_2, \square, \leftarrow)$

$\delta_{isEven}(q_2, 0) = (Y, 0, -)$

$\delta_{isEven}(q_2, 1) = (N, 1, -)$

$\delta_{isEven}(q_2, \square) = (N, \square, -)$

A machine that checks if a number is even.

Before trying to explain how the machine works, we should remark how arcane and intimidating this notation looks like. Fortunately, there are clearer ways of representing a Turing Machine!

3.1 Transition table

The transition function of a Turing Machine is the core of its behavior, so we shall use it as our focal point when designing a clearer representation. The function takes two arguments, which means we can represent it as a 2D table: the rows are states, the columns are tape symbols. A cell contains the triple (next state, written symbol, direction) given by the corresponding combination.

Here is the table representation for δ_{isEven}:

	0	1	$\square$
q_1	$q_1, 0, \rightarrow$	$q_1, 1, \rightarrow$	$q_2, \square, \leftarrow$
q_2	$Y, 0, -$	$N, 1, -$	$N, \square, -$

Looking at the table itself, other information about our machine can be easily extracted: namely the set of tape symbols Γ and the set of states Q.

To be able to figure out the other elements of the 6-tuple, we establish the following conventions:

- the initial state will always be on the *first row* of the table
- the blank symbol will always be denoted by $\square$
- the order of the columns will be such that first come all the input symbols, then the blank symbol $\square$, then all other tape symbols

Now we can represent our entire Turing Machine 6-tuple with a concise transition table; conversely, we can look at such a transition table and identify each element of the 6-tuple.

[§]Most significant digit first.

4 `isEven` explained

The machine starts in the initial state, q_1; this state corresponds to the intuitive concept of "searching the end of the input" – thus, at each step, the head is moved one position to the *right*, leaving each symbol as it is, until a blank symbol is found. This is captured by the first two entries of the table.

Once on the blank symbol, the whole input must have been traversed, so the head needs to be moved one cell *left* onto the last symbol of the input and the machine transitions to state q_2, which corresponds to the intuitive concept of "checking what the last symbol is". This is captured by the last entry in the first row.

If this symbol is a 0, the number is even, so it's accepted by transitioning to state Y, providing a "yes" answer. The other two elements of the transition action (namely the written symbol and the movement direction of the head) are irrelevant because computation halted. But we choose the intuitive option of keeping the symbol as it was (a 0) and holding the head still.

If the symbol is a 1, the number is odd, not even, so it's rejected by transitioning to state N, providing a "no" answer.

Note that in state q_1 the head is only moved right, traversing the whole input and stopping at *the very first* blank symbol. This means that, in q_2, if the blank symbol is encountered, there is only one possibility: the input was empty.

5 The empty input and other arbitrary strings

In the beginning, we said that inputs to our machines are strings of symbols from the input alphabet. When constructing `isEven`, we interpreted the input to be "a number in base 2". However, not every string of 0s and 1s corresponds straightforwardly to a number; for example, what about "00101"?

We can adopt the simple convention of allowing leading 0s, such that "00101", "00000000101" and "101" all represent the same number: five. But there's still the issue of the empty input, consisting of no symbols.

To address this, we'll simply refine the *problem statement*, to ask: *"is the input string a binary representation of a number with optional leadings 0s? If yes, is it even?"*[††]. The empty string is not the binary representation of a number, so the answer is "no".

From now on, we'll keep such subtleties implicit.

6 Using states as memory

We shall soon see that there is a reasonable, coherent definition of *"expressive"* that allows us to say that "a Turing Machine is as expressive as any generic-purpose programming language". At the moment, this hardly seems to be the case and you're excused to suspect that there's not much that a Turing Machine can do.

We take one step towards convincing you of the expressive power of this model, by showing a technique through which we can *"memorize"* information. Examine the following machine, `sameFirstLast`, which checks whether a binary strings begins and ends with the same symbol:

	0	1	□
$start$	$remember0, 0, \rightarrow$	$remember1, 1, \rightarrow$	$Y, \square, -$
$remember0$	$remember0, 0, \rightarrow$	$remember0, 1, \rightarrow$	$expect0, \square, \leftarrow$
$remember1$	$remember1, 0, \rightarrow$	$remember1, 1, \rightarrow$	$expect1, \square, \leftarrow$
$expect0$	$Y, 0, -$	$N, 1, -$	$N, \square, -$
$expect1$	$N, 0, -$	$Y, 1, -$	$N, \square, -$

This time, we have chosen to give each of the five states a suggestive name, instead of generic ones like q_1, q_2 etc. From the machine's point of view, the names are irrelevant; their purpose is simply to make things clearer for us.

We can see here an example of *memorization*: from the **start** state, depending on the contents of the first cell of the input, the machine will transition to one of two states: **remember0** or **remember1**. The states themselves acts as memory, in this case retaining one bit of information.

[††]Note that we could also rephrase this in a way that avoids the concept of "number representation" entirely: *"is the last symbol of the input 0?"*

Some entries in the table might surprise you. As with our first example, we need to think of a way to deal with the empty input; we choose to accept it, by transitioning to the Y state, but this is just a convention, we could just as well transition to N.

But what about the transition on $(expect0, \square)$? Such a situation can never occur:

- if the input was empty, the machine goes from **start** directly to Y and halts, never reaching any other state
- if the input starts with a 1, then the machine will transition to **remember1** and we can see, by looking at the table, that **expect0** can never be reached.
- if the input starts with a 0, then the machine will transition to **remember0**, will remain in this state while the head goes right until the end of the input, then it will transition to **expect0**, as the head moves left onto the last input symbol. Then the machine will transition to either Y or N and halt.

So the machine will never need to "know what to do" if it's in state **expect0** and reads a $\square$ – this situation cannot occur for any input.

We arbitrarily choose to halt computation with a negative answer (i.e. transition to N), leave the symbol as it is and keep the head still. From now on, we shall omit explicitly describing such transitions, by leaving the corresponding entry empty, to keep our tables lighter.

6.1 Multiple bits of memory

In the previous example we showcased how a machine can use states to remember one bit of information. This technique can be extended to an arbitrary number of bits. To illustrate this, consider the following machine, `samePairFirstLast` that checks whether the input starts and ends with the same combination of two bits. For example, 011101011001 is accepted because it starts and ends with "01", while 101111 is rejected, because it starts with "10" and ends with "11".

	0	1	$\square$
$start$	$remember0, 0, \rightarrow$	$remember1, 1, \rightarrow$	$N, \square, -$
$remember0$	$remember00, 0, \rightarrow$	$remember01, 1, \rightarrow$	$N, \square, -$
$remember1$	$remember10, 0, \rightarrow$	$remember11, 1, \rightarrow$	$N, \square, -$
$remember00$	$remember00, 0, \rightarrow$	$remember00, 1, \rightarrow$	$expect00, \square, \leftarrow$
$remember01$	$remember01, 0, \rightarrow$	$remember01, 1, \rightarrow$	$expect10, \square, \leftarrow$
$remember10$	$remember10, 0, \rightarrow$	$remember10, 1, \rightarrow$	$expect01, \square, \leftarrow$
$remember11$	$remember11, 0, \rightarrow$	$remember11, 1, \rightarrow$	$expect11, \square, \leftarrow$
$expect00$	$expect0, 0, \leftarrow$	$N, 1, -$	
$expect01$	$expect1, 0, \leftarrow$	$N, 1, -$	
$expect10$	$N, 0, -$	$expect0, 1, \leftarrow$	
$expect11$	$N, 0, -$	$expect1, 1, \leftarrow$	
$expect0$	$Y, 0, -$	$N, 1, -$	
$expect1$	$N, 0, -$	$Y, 1, -$	

6.2 Checking for palindromes

We move on to a slightly more complicated task: answering the question *"is this binary string a palindrome?"*

When is a string a palindrome? One neat way to think of it is with a recursive definition: a string is a palindrome if:

- it's empty
- its first and last symbols are the same and there's a palindrome between them

Our machine, `isPalindrome` will thus need to check if the word is empty; if not, it will check the first and last symbols for equality, then restart the same process for the string in between:

	0	1	□
$start$	$remember0, □, →$	$remember1, □, →$	$Y, □, –$
$remember0$	$remember0, 0, →$	$remember0, 1, →$	$expect0, □, ←$
$remember1$	$remember1, 0, →$	$remember1, 1, →$	$expect1, □, ←$
$expect0$	$reset, □, ←$	$N, 1, –$	$Y, □, –$
$expect1$	$N, 0, –$	$reset, 1, ←$	$Y, □, –$
$reset$	$reset, 0, ←$	$reset, 1, ←$	$start, □, →$

Notice that state **start** here is almost the same as in the example of section 6, except the initial symbol is *erased* – this is done so that we can later easily find the beginning of "the inner substring".

remember0 and **remember1** are identical to those of `sameFirstLast`; **expect0** and **expect1** are similar, but when a match is found, we can't just accept, we need to start checking the inner substring. So we remove the last symbol (now the tape only contains the inner substring) and move on to a state called **reset**, whose purpose is to bring the head onto the leftmost non-blank symbol, then redo the whole thing by transitioning to **start**.

6.3 Same number of 1s and 0s

We will now showcase a technique for *"marking"* symbols, by making use of Γ, the tape alphabet.

We need to answer the question: *"does this binary string have an equal number of 0s and 1s?"*. How would you solve this, given a very long string written on a piece of paper and a pen?

You would probably look at the first symbol, cross it off, then go symbol by symbol to the right looking for a match[‡‡]. When you find a match, you cross it off as well, then go back to the first unmarked symbol and start over.

If you reach the end of the string when seeking for a match, the answer is "no" – there was some unmatched symbol.

If all symbols are matched, the answer is "yes".

This is exactly how the following machine, `isMatched` works, by using two extra tape symbols: $\emptyset$ and $\not{1}$ to represent crossed out symbols.

	0	1	□	$\emptyset$	$\not{1}$
$start$	$find\_1, \emptyset, →$	$find\_0, \not{1}, →$	$Y, □, –$	$start, \emptyset, →$	$start, \not{1}, →$
$find\_0$	$reset, \emptyset, ←$	$find\_0, 1, →$	$N, □, –$	$find\_0, \emptyset, →$	$find\_0, \not{1}, →$
$find\_1$	$find\_1, 0, →$	$reset, \not{1}, ←$	$N, □, –$	$find\_1, \emptyset, →$	$find\_1, \not{1}, →$
$reset$	$reset, 0, ←$	$reset, 1, ←$	$start, □, →$	$reset, \emptyset, ←$	$reset, \not{1}, ←$

Note that we could also erase the first character of the string, instead of crossing it out.

We also don't actually need to distinguish between crossed out symbols; on paper, that is just an artifact of our pen strokes. But the machine can just as easily overwrite both 0s and 1s with an X, like this:

	0	1	□	X
$start$	$find\_1, X, →$	$find\_0, X, →$	$Y, □, –$	$start, X, →$
$find\_0$	$reset, X, ←$	$find\_0, 1, →$	$N, □, –$	$find\_0, X, →$
$find\_1$	$find\_1, 0, →$	$reset, X, ←$	$N, □, –$	$find\_1, X, →$
$reset$	$reset, 0, ←$	$reset, 1, ←$	$start, □, →$	$reset, X, ←$

7 Representing Turing Machines as prose

So far, we have designed fairly simple machines. But even the 2-bit memorizer in subsection 6.1 had quite a large, unwieldy table. In future lectures, we will need larger machines, for which explicitly presenting the transition table would be a gargantuan, excruciating task, more laborious than illuminating.

To make our lives easier, we shall be using English prose to describe the functionality of larger machines, by combining descriptive phrases to evoke familiar transition tables.

For example, after going through the transition tables presented above, we can now say about a machine:

[‡‡] A 1 if the first symbol was 0, a 0 if the first symbol was 1.

"First, move the head on the last symbol of the input."

and we can easily envision how the explicit transitions would look like.

Now that you have seen the transition table for a machine which checks if an input starts and ends with the same 2-symbol combination, you can easily see that the following machine is possible, and you can imagine how its transition table would look like:

"Check if the input starts and ends with the same 4-symbol combination."

And we can write this simple sentence, instead of presenting a 61×3 table!

Our goal is not to design Turing Machines for the fun of it. The machines are just a useful tool that we employ to get rigorous formal answers to questions related to computability. When designing a machine we want our description to be succinct (so that we don't waste much time with it), but explicit enough to be convinced that such a machine can exist and to have an idea of how its explicit transition table would look like. For example, given the problem: *"does this binary string have more 0s than 1s?"*, it is not very convincing (with what we've seen so far) to describe a machine that solves it like this:

"Check if the input has more 0s than 1s."

But we can employ the techniques already discussed and come up with:

---

1 *If the current symbol is blank, reject.*
2 *Mark the current symbol and move right, searching for a complement symbol.*
2 *If none is found:*
3 *If the marked symbol was a 1, accept.*
4 *If the marked symbol was a 0, reject.*
5 *Else, mark the found symbol, move to the first non-marked symbol and go to 1.*

In the future, if we encounter such a problem, we can rely on this description and confidently describe our machine as:

"Check if the input has more 0s than 1s."

As we continue studying examples of Turing Machines, especially at the labs, we shall be able to abstract more and more functionality into short phrases.

8 References and further reading

Turing Machines were first described by Alan Turing (1912-1954) in his seminal 1936 article: "On computable numbers, with an application to the Entscheidungsproblem" [1]. The paper provides a negative answer to an important mathematical question and introduces the Machines as a way to formalize what a mathematician with a pen and paper can actually do.

While "the Entscheidungsproblem" itself is outside the scope of `AA` , Turing's paper also introduces the concepts of decidability, undecidability, reductions, the Universal Machine etc. which will be the subject of our study during the next four lectures.

The article is freely available online[§§], although we recommend reading Charles Petzold's "The Annotated Turing" [2] – a book built around the article, which includes the necessary historical context and motivation, relevant details about Turing's life and education, as well as later corrections to the mistakes present in the 1936 version.

Bibliography

[1] Alan Mathison Turing et al. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

[2] Charles Petzold. *The annotated Turing: a guided tour through Alan Turing's historic paper on computability and the Turing machine*. Wiley Publishing, 2008.

[§§] `https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-12`