Turing Machines: Lecture 1

1 We Need to Go Beyond Pushdown Automata

Why: reminder.

- We started this class with finite automata.
- It turned out that there are some languages – e.g., the language

$$\{a^n b^n, n = 0, 1, 2, \ldots\} = \{\Lambda, ab, aabb, \ldots\}$$

  that cannot be recognized by finite automata.
- To recognize such languages, we added a stack to a finite automaton. This led to pushdown automaton.
- It turns out that there are some languages – e.g., the language

$$\{a^n b^n c^n, n = 0, 1, 2, \ldots\} = \{\Lambda, abc, aabbcc, \ldots\}$$

  that cannot be recognized by pushdown automata either.
- Since adding *one* stack is not enough, a natural idea is to add *two* stacks. Such devices are known as *Turing machines*, after Alan Turing, one of the founders of computer science.

Comment.

- While Turing machines are, in effect, finite automata with two stacks, this is *not* how they are usually described.
- Their usual description comes from history – in the 1930s where they were invented, the most sophisticated device was a tape recorder, so the Turing machine simulates a tape recorder.
- Later on, we will learn how to describe them as finite automata with two stacks.

Who was Turing. Alan Turing was a professor interested in a topic that at that time, before actual computers were invented, was rather philosophical: what can be computed in principle? Before computers were invented, this topic was as important as now the topic of how to communicate with aliens from other planets.

In 1936, he published his fundamental paper on this topic. For two years, no one read this paper. How do we know that?

- His proof contained an error, which is easy to see if you read the paper.
- However, the only person who noticed this error was Turing himself, he published a correction in 1938.

Not only the topic was too science fiction-y at this time. It did not help readability that half of the title was in German. This is easy to explain: until Hitler came to power in 1933 and destroyed a lot of German science, Germany was the place where most science took place – it is sufficient to recall Einstein, Schroedinger, etc., and German was an international language of science. If a Russian or an American scientist wanted to publish a paper so that people would read it, he had to publish it in German – and they did! Only after 1933, when Hitler forced many German scientists out of Germany and many of them moved to the US, English became an international language of science.

So for many years no one paid attention to Turing's work – until the Second World War started in 1939. One of the problems that the allies faced was trying to decode German codes. This required a lot of tries, so there was a need to automate the corresponding computations. The British government started looking for specialists in computation – and found Turing. He helped a lot, German codes were broken, Turing became a hero.

After the war, the situation was not so good for him. The problem was that Turing was gay, and in England at that time, this was punishable by jail.

- During the war, the authorities looked the other way, since Turing was too important.
- However, after the fact, they started pushing him.

They did not want to put the war hero in jail, so they offered him an alternative – hormonal treatment. This "treatment" made him feel very bad, both physically and mentally, and he killed himself – by poisoning an apple with cyanide and biting into it. (This is, by the way, where the Apple's logo may have come from.)

2 What Is a Turing Machine

General idea. A Turing machine consists of:

- a *head* – a finite automaton that can be in different states, and
- a *tape* on which symbols are written;
  - in the beginning, the first cell is empty, and the rest of the tape should contain the input;
  - at the end, the first cell should again be empty, and the rest of the tape should contain the result of the computation.

In the beginning, the head is in the special start state, and it is located at the first cell of the tape.

Depending on the state and on the symbol, the machine can do one or more of the following three things:

- it can change the symbol in the cell,
- it can go to another state, and
- it can move one step to the left (we will denote it L) or one step to the right (we will denote it R).

The Turing machine stops when it gets to a special halt state. At this moment, the head should again point to the very first cell, and this cell should be empty.

Comment. The word "halt" comes from Turing: halt is the British word for stop. British stop signs do not say Stop, they say Halt. Similarly, queue is a British word for what we call line.

3 Unary Numbers

Let us present a simple example of a Turing machine. This machine will deal with objects which are even simpler than the usual binary numbers – unary numbers. What are unary numbers?

- In a usual decimal system, we use 10 digits: 0, 1, 2, ..., 9.
- In a binary system, we use 2 digits: 0 and 1.
- In a unary system, we use only one digit: 1.

How can we describe different numbers by using only one digit? This is how this is done for the first Roman numbers (that you can see, e.g., on the old clocks):

- 1 is represented as 1;
- 2 is represented as 11;
- 3 is represented as 111; etc.

4 Turing Machine for Adding 1 to a Unary Number

Adding 1. Let us start with the simplest possible operation on numbers: adding 1. We will illustrate it on the example of adding 1 to number 2, so that the result will be $2 + 1 = 3$.

Let us describe the corresponding states in pictures. To make the pictures clearer, we will:

- describe the empty cell by a dash –,
- indicate where the head is by making the corresponding symbol underlined, and
- the state of the head will be written after the state of the tape.

In these notations, the original configuration of the Turing machine – corresponding to the unary number 2 – has the following form:

$\underline{-}$	1	1	–	–	...

start

The desired final configuration should have the following form:

$\underline{-}$	1	1	1	–	...

halt

How can we do it: idea. To make it easier to understand the problem, let us assume that you are hired to paint one more section of the bridge. You are hanging on a suspended platform. You can signal to move you to the left or to the right.

Also, your boss is really into micro-managing, he wants to know every minute what exactly you are doing (we all had bosses like that).

An additional problem is that at each moment of time, we only see one section of the bridge.

So, how can we do it? Here is a natural idea:

- first, we are at the beginning section of the bridge, where there is no color needed; we motion to move us one step to the right; we are reporting to the boss that we are moving;
- if the section of the bridge that we see is already painted, we again ask to move us to the right; we report to the boss that we are still moving;
- eventually, when we see an un-painted section of the bridge, we paint it, and ask to be moved to the left; we report to the boss that we are going back;

- if we see a painted section of the bridge, we again ask to move us to the left, and report to the boss that we are still going back;
- finally, when we see an unpainted starting section of the bridge, we report, to the boss, that our job is done.

How can we do it: rules. Let us describe this in terms of rules:

- if we are in the start state, and we see blank, then we move to the right and get into moving state; this can be described as follows:

$$\text{start}, - \to \text{R}, \text{moving}$$

- if we are in the moving state and we see 1, we go right; the state does not change – it remains the same moving state, so we do not need to explicitly describe the new state:

$$\text{moving}, 1 \to \text{R}$$

- if we are in the moving state and we see blank, then we replace this blank with 1, move one step to the left, and change the state to back:

$$\text{moving}, - \to 1, \text{L}, \text{back}$$

- if we are in the back state and we see 1, we continue going left:

$$\text{back}, 1 \to \text{L}$$

- finally, if we are in the back state and we see blank, we stop:

$$\text{back}, - \to \text{halt}$$

How these rules will work on our example. Let us trace, step-by-step, how these rules will work on our example of adding 1 to $n = 2$. In this example, we are already described the initial state:

$\underline{-}$	1	1	–	–	...	start

Here, the head is in the state start, and the symbol that we see is blank, so we use the rule "start, $- \to$ R, moving": we move the head to the right, and change the state to moving. As a result, the state of the Turing machine will take the following form:

–	$\underline{1}$	1	–	–	...	moving

Now, we are in the state moving and we see 1. So, according to the corresponding rule, we move one step to the right:

–	1	$\underline{1}$	–	–	...	moving

Again, we are in the state moving and we see 1, so we again move to the right:

–	1	1	$\underline{-}$	–	...	moving

Now, we are in the state moving, and we see blank. So, according to the corresponding rule, we replace blank with 1, move one step to the left, and change the state to back:

–	1	$\underline{1}$	1	–	...	back

Now, we are in the state back and we see 1, so we move to the left:

–	$\underline{1}$	1	1	–	...	back

Again, we are in the state back and we see 1, so we move to the left:

$\underline{-}$	1	1	1	–	...	back

Finally, we are in the state back and we see blank, so we change to the final state halt:

$\underline{-}$	1	1	1	–	...	halt

This is exactly what we wanted: number 3.

Try it yourself on other examples: $n = 0$ and $n = 1$.

Comment. In this class, we only study *deterministic* Turing machines. Just like for the deterministic finite automaton, once you know the state and the symbol, there should be only one possible transition – similarly for Turing machines that we study, once we know the state of the head and the symbol, our actions should be uniquely determined.

If you have two rules like

$$\text{moving}, - \to 1, \text{R}$$

and

$$\text{moving}, - \to 1, \text{L}, \text{back}$$

what is the Turing machine supposed to do if the head is the state moving, and it sees a blank symbol? Should it follow the first rule or the second rule?

For those who are curious: there *are* also non-deterministic Turing machines, the textbook mentions them – but in this class, we do not study them.

5 Subtracting 1 from a Unary Number

Idea. This is similar to adding 1 from a unary number. The only difference is that when we reach the first blank space – meaning that we have over all the 1s, we need to get back and delete the last 1. Then, we go back.

Rules. Here are the rules for the Turing machine:
start, – → R, moving
moving, 1 → R
moving, – → deleting, L
deleting, 1 → –, back, L
back, 1 → L
back, – → halt

Example. Let us show how it will work for $n = 2$.

$\underline{-}$	1	1	–	–	–	...	start
–	$\underline{1}$	1	–	–	–	...	moving
–	1	1	$\underline{-}$	–	–	...	moving
–	1	$\underline{1}$	–	–	–	...	deleting
–	$\underline{1}$	–	–	–	–	...	back
$\underline{-}$	1	–	–	–	–	...	back
$\underline{-}$	1	–	–	–	–	...	halt

6 Adding 1 to a Binary Number

We need to consider binary numbers. We dealt with unary numbers only to have a very simple example. Real computers do not use unary numbers. they use binary numbers. So how do we deal with binary numbers? Let us also start with the simplest possible operation: adding 1.

How are binary numbers represented in a computer? When write down a number, we start with the most significant digit. For example, the number

$$13 = 1 \cdot 10^1 + 3$$

starts with 1.

Similarly, when we write down a binary number, we start with the most significant digit, i.e., with the highest bit. For example, the binary representation of the decimal number 13, i.e.:

$$13_{10} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 1101_2$$

start with the first 1.

Some computers store binary numbers the same way: highest bits first. However, most computers store binary numbers in the opposite direction: lowest bits first. For example, $13_{10} = 1101_2$ is stored as 1011.

Why? Because computers want to process numbers fast. When we add two numbers, e.g., when we add 1998 and 13, we start with the lowest bits:

```
   .
 1998
+  13
-----
    1
```

then we move to the next lowest, etc., until we get the answer

```
  .     .
 1998   1998   1998
+  13  +  13  +  13
-----  -----  -----
   11    011   2011
```

Similarly with binary numbers, and similarly with subtraction or multiplication: we always start with the lower bits.

If we store a number our usual way, highest bit first, then the computer will waste some time getting to the lowest bit before it can start performing the arithmetic operation. So, to speed up computations, numbers are usually stored lowest bit first.

This is how we will store binary numbers. To make our models as close to real computers as possible, in our Turing machines, we will also store binary numbers lowest-bit first.

How the ++ operation is implemented in a computer. To understand how adding 1 can be implemented on a Turing machine, let us recall how adding 1 – i.e., the ++ operation – is implemented in the actual computers.

One may think that they are just a particular case of the usual addition, but this is not how they are implemented. Adding 1 is usually a separate hardware supported operation. Indeed, let us recall how, e.g., we add 1, step-by-step to the binary number 1011.

```
   .
1101  1101 1101 1101
+  1  +  1 +  1 +  1
----  ---- ---- ----
   0    10  110 1110
```

In general:

- we start with the lower possible digit;
- if we see 1, we replace it with 0 and go to the next digit;
- finally, if we see 0 or blank, we replace it with 1.

Comment. Seeing blank is possible, e.g., when we add 1 to $7_{10} = 111_2$:

```
  .    .      .
 111  111    111    111
 + 1  + 1   +  1    + 1
 ---  ---   ----   ----
   0   00    000   1000
```

Another comment. At the end of this lecture, we explain why ++ is a separate operation: it is very efficient. This additional information *will not* be on the test, but it can be for extra credit.

Let us describe this as rules for a Turing machine:

- we start with the start state and see blank; in this case, we go right and change to state moving;
- if we are in the state moving and we see 1, we replace it with 0 and continue moving right;
- if we are in the state moving and we see 0 or blank, we replace this symbol with 1, go left, and change to state back;
- when we are in the state back and we see 0, we continue going left (and remain in the state back);
- finally, when we are in the state back and we see blank, we halt.

Here are the corresponding rules:

start, – → R, moving
moving, 1 → 0, R
moving, 0 → 1, L, back
moving, – → 1, L, back
back, 0 → L
back, – → halt

Tracing this Turing machine. Let us trace these rules on the example of the number $13_{10} = 1101_2$ (which is stored as 1011):

$\underline{-}$	1	0	1	1	–	...	start

At first, we move to the right and change to state moving:

–	1	0	1	1	–	...

moving

Then, we replace 1 with 0 and continue going to the right:

–	0	0	1	1	–	...

moving

Since we see 0, we replace it with 1, go left, and go into the state back:

–	0	1	1	1	–	...

back

In the state back, we see 0, so we continue going left:

–	0	1	1	1	–	...

back

We are in the state back, and we see blank, so we halt:

–	0	1	1	1	–	...

halt

Try it yourself on other examples of binary numbers.

7 Adding 2 to a Binary Number

Main idea. When we add $2_{10} = 10_2$ to a binary number, the last bit of 10_2 is 0, so the last bit of the sum does not change, but for other bits, we have the same algorithm as before. Let us first give an example:

```
        .     .
  111    111    111    111
+ 10   + 10   + 10   + 10
----   ----   ----   ----
   1     01    001   1001
```

Here is the resulting algorithm:

- we skip the last bit,
- after that, if we see 1, we replace 1 with 0;
- if we see 0 or blank, we replace them with 1 and start going back.

Here are the corresponding Turing machine rules:

start, – → R, skip
skip, 1 → R, moving
skip, 0 → R, moving
moving, 1 → 0, R
moving, 0 → 1, L, back
moving, – → 1, L, back
back, 0 → L
back, 1 → L
back, – → halt

Here is a tracing on the example of 111 + 10:

$\underline{-}$	1	1	1	–	–	...	start
–	$\underline{1}$	1	1	–	–	...	skip
–	1	$\underline{1}$	1	–	–	...	moving
–	1	0	$\underline{1}$	–	–	...	moving
–	1	0	0	$\underline{-}$	–	...	moving
–	1	0	$\underline{0}$	1	–	...	back
–	1	$\underline{0}$	0	1	–	...	back
–	$\underline{1}$	0	0	1	–	...	back
$\underline{-}$	1	0	0	1	–	...	back
$\underline{-}$	1	0	0	1	–	...	halt

8 Subtracting 1 from a Binary Number

The operation −− (subtracting 1) is similar to ++. the only difference is that:

- in ++, first, 1s are replace with 0s, and then 0 is replaced with 1;
- here, first, 0s are replaced with 1s, and then 1 with 0.

Here is an example:

```
   .     .     .
11000 11000 11000 11000 11000
-   1 -   1 -   1 -   1 -   1
----- ----- ----- ----- -----
    1    11   111  0111 10111
```

Here are the corresponding rules for a Turing machine:

start, – → R, moving
moving, 0 → 1, R
moving, 1 → 0, L, back
back, 1 → L
back, – → halt

Here is tracing:

$\underline{-}$	0	0	0	1	1	–	...	start
–	$\underline{0}$	0	0	1	1	–	...	moving
–	1	$\underline{0}$	0	1	1	–	...	moving
–	1	1	$\underline{0}$	1	1	–	...	moving
–	1	1	1	$\underline{1}$	1	–	...	moving
–	1	1	$\underline{1}$	0	1	–	...	back
–	1	$\underline{1}$	1	0	1	–	...	back
–	$\underline{1}$	1	1	0	1	–	...	back
$\underline{-}$	1	1	1	0	1	–	...	back
$\underline{-}$	1	1	1	0	1	–	...	halt

Try it yourself on some other input.

9 From a Finite Automaton to a Turing Machine

Turing machines for computing and Turing machines for checking. So far, we had Turing machines for *computing*. Turing machines can also be used to *checking* whether a certain property is true. In this case, there is no state halt, instead, there are two final states: accept and reject:

- If a Turing machine ends up in a state accept, the word is accepted.
- If a Turing machine ends up in a state reject, the word is rejected.

How to go from a finite automaton to a Turing machine: general algorithm.

- for each state of the finite automaton, we add a similar state of the Turing machine;

- at first, when are in the start state and see blank, we move right and go into the starting state of the finite automaton;
- then, we simply follow the rules of the finite automaton, with the only change that now, every time we move R;
- if at the end – i.e., after we have moved to a blank cell following the checked word – we are in the final state of the finite automaton, we get into the state accept;
- if at the end – i.e., after we have moved to a blank cell following the checked word – we are in the state which is not final for the original finite automaton, we get into the state reject.

Example. Let us consider the following finite automaton with two states: the starting state s, a final state f, and the following transitions:

$$\text{s, } 0 \to \text{s; s, } 1 \to \text{f; f, } 0 \to \text{f; f, } 1 \to \text{s.}$$

This automaton accepts all binary words with odd number of 1s, e.g., the word 010. Indeed, for the word 010, we have the following transtions:

$$\text{s, } 0 \to \text{s; s, } 1 \to \text{f; f, } 0 \to \text{f.}$$

We end up in a final state, so the word is indeed accepted.

Based on the above algorithm, we get the following Turing machine rules:

$$\text{start, } - \to \text{R, s;}$$
$$\text{s, } 0 \to \text{R, s; s, } 1 \to \text{R, f; f, } 0 \to \text{R, f; f, } 1 \to \text{R, s;}$$
$$\text{f, } - \to \text{accept; s, } - \to \text{reject.}$$

Let us trace this Turing machine on the example of the word 010:

$\underline{-}$	0	1	0	–	...	start
–	$\underline{0}$	1	0	–	...	s
–	0	$\underline{1}$	0	–	...	s
–	0	1	$\underline{0}$	–	...	f
–	0	1	0	$\underline{-}$	...	f
–	0	1	0	$\underline{-}$	...	accept

Try it yourself on some other example.

10 Why Adding 1 Is More Efficient Than General Addition

When we add two N-digit numbers, we need, in general, at least N bit operations. How many operations do we need, on average, to add 1?

- If the number ends with 0, we just replace it with 1, so we only need 1 bit operation. How many are such cases? We have two possible bits: 0 and 1, so 0 happens in 1/2 of the cases.
- If the number ends with 01, we replace it with 10, i.e., we need 2 bit operations. This happens in one of $2^2 = 4$ possible endings, so it happens in $1/2^2$ of the cases.
- If the number ends with 011, we replace it with 100, i.e., we need 3 bit operations. This happens in one of $2^3 = 8$ possible endings, so it happens in $1/2^3$ of the cases, etc.

So, the average number a of bit operations is equal to:

$$a = \frac{1}{2^1} \cdot 1 + \frac{1}{2^2} \cdot 2 + \frac{1}{2^3} \cdot 3 + \ldots$$

Let us divide each term by 2. Then:

- $\frac{1}{2^1} \cdot 1$ becomes $\frac{1}{2^2} \cdot 1$,
- $\frac{1}{2^2} \cdot 2$ becomes $\frac{1}{2^3} \cdot 2$, etc.

So

$$\frac{a}{2} = \frac{1}{2^2} \cdot 1 + \frac{1}{2^3} \cdot 2 + \ldots$$

Subtracting this expression for $a/2$ from the above expression for a, we get

$$a - \frac{a}{2} = \frac{a}{2} = \frac{1}{2^1} \cdot 1 + \frac{1}{2^2} \cdot (2-1) + \frac{1}{2^3} \cdot (3-2) + \ldots,$$

i.e.,

$$\frac{a}{2} = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots$$

Now, we can again divide this formula by 2 and get

$$\frac{a}{4} = \frac{1}{2^2} + \frac{1}{2^3} + \ldots$$

Subtracting this expression from the above expression for $a/2$, we get $a/4 = 1/2$, so $a = 2$. Thus:

- adding 1 requires, on average, 2 bit operations,
- while the general addition requires N bit operations – e.g., 64 for 64-bit numbers.

So, of course, adding 1 is much much faster than the general addition!