Turing Machines

1 Quick Review – How many steps?

We have seen how to analyze the running time of a program by counting the number of steps.
For example:

```
void g6 (char[] p){
  elapse (8 steps);
  for (nat i=0; i<p.length(); i++){
    elapse (5 steps);
    for (nat j=i; j<p.length(); j++){
    }
      elapse (2 steps);
  }
}
```

Remember that a "step" is supposed to be a fixed amount of time.

This kind of analysis is widely used and convenient. But how do we get the basic step counts in each part of the code? They are just assumptions (or guesses). For example, many people analyze sorting algorithms by assuming that each comparison of two values is "one step". That's certainly a helpful assumption but it ignores the fact that comparing two big numbers takes longer than comparing two small numbers.

To reason about running time in a rigorous way, and avoid the risk of sweeping any time costs under the carpet, we need a precise *Model of Computation* that fully specifies the steps. The model we'll be looking at is the *Turing Machine (TM)*, invented by Alan Turing in 1936. A TM takes a very conservative view of what constitutes a step, so it serves as a gold standard. If your algorithm is fast on a Turing Machine, it's indisputably fast!

2 Complexity Notations

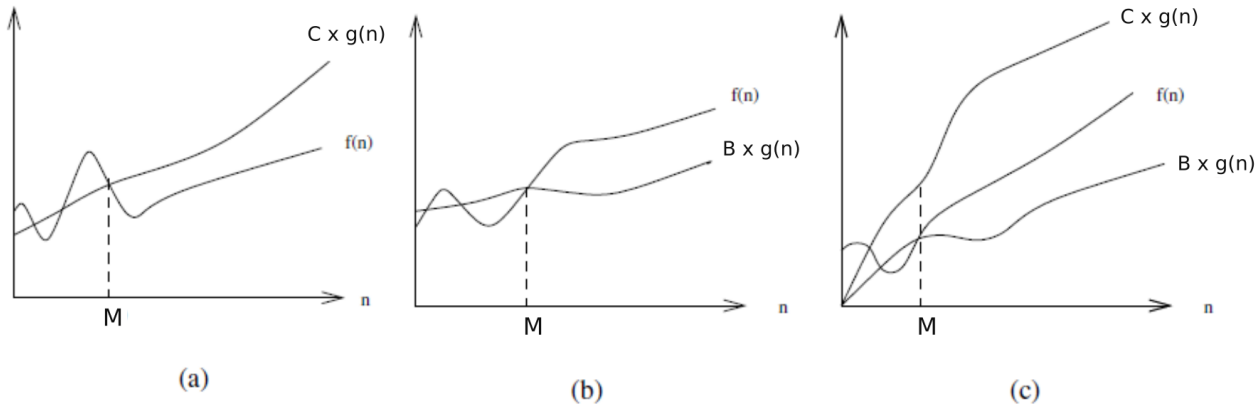
Let f and g be functions from $\mathbb{N}$ to the nonnegative reals.

(a) We say that $f \in O(g)$, or informally "$f(n)$ is $O(g(n))$", when g is an upper bound for f up to a constant factor. That is: there are numbers M and C such that for $n \geqslant M$ we have $f(n) \leqslant C \times g(n)$.

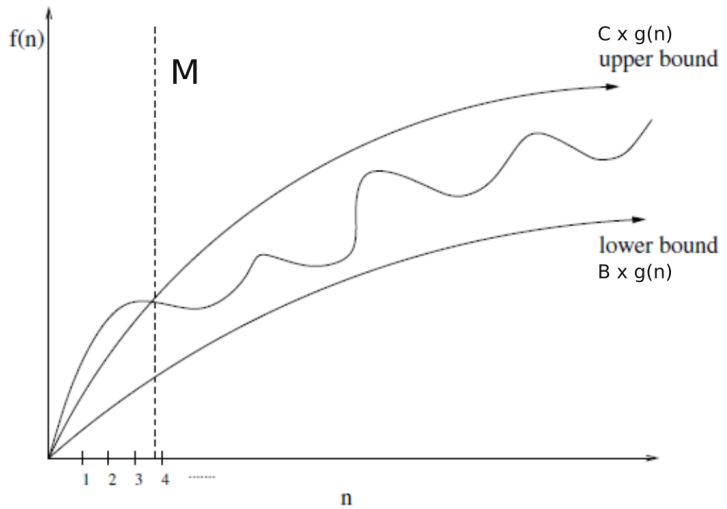
(b) We say that $f \in \Omega(g)$, or informally "$f(n)$ is $\Omega(g(n))$", when g is a lower bound for f up to a constant factor. That is: there are numbers M and $B > 0$ such that for $n \geqslant M$ we have $B \times g(n) \leqslant f(n)$.

(c) We say that $f \in \theta(g)$, or informally "$f(n)$ is $\theta(g(n))$", when both of the above conditions hold. That is: there are numbers M and C and $B > 0$ such that for $n \geqslant M$ we have $B \times g(n) \leqslant f(n) \leqslant C \times g(n)$.

The following figure illustrates the above complexity notations.



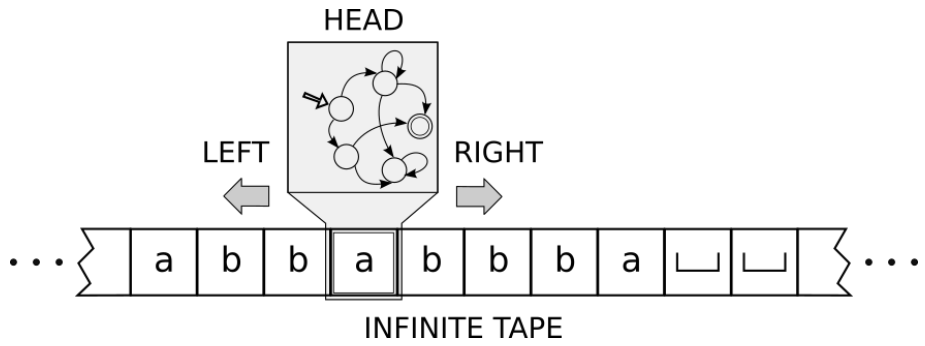
With these notations, we can be more precise about complexity. For example, if we say that the worst case running time is $O(n^2)$, it might in fact be linear, but if we know that it is $\theta(n^2)$ then it really is no better than quadratic, because it will be within the lower and upper bounds of complexity. The upper and lower bounds that are valid for $n > M$ smooth-out the behavior of complex functions, we would like to have a tight-bound to ensure our estimations are precise, as shown below:



3 What is a Turing Machine?

"A Turing machine can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer. In a very real sense, these problems are beyond the theoretical limits of computation.
The Turing machine model uses an ***infinite tape*** *as its unlimited memory. It has a tape* ***head*** *that can read and write symbols and move around on the tape. Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs accept and reject are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting. "* (Sipser, 2013).

In simple words, a Turing machine is a simple formal model of mechanical computation, and a universal Turing machine can be used to compute any function, which is computable by any other Turing machine. A Turing machine has finitely many states (like a DFA) but it also has an external memory: an infinite tape, divided into cells. The machine has a *head* that sits over one cell of the tape. The Turing machine can read and write symbols on the tape and move left or right (or stay put) after each step. Unlike a DFA, once a Turing machine enters an accept or reject state, it stops computing and halts. The following diagram shows a general representation of a Turing machine:



Before giving a Turing machine, we first specify two finite sets:

- The *Tape Alphabet* T, which includes a "blank" character ␣ (also shown above). At any time, each cell contains a character in T, and there are only finitely many cells with a non-blank character. We will usually take $T = \{\mathtt{a}, \mathtt{b}, ␣\}$. The set of non-blank characters $\{\mathtt{a}, \mathtt{b}\}$ is called the *input alphabet* (Σ).
- The set V of *Return Values*.

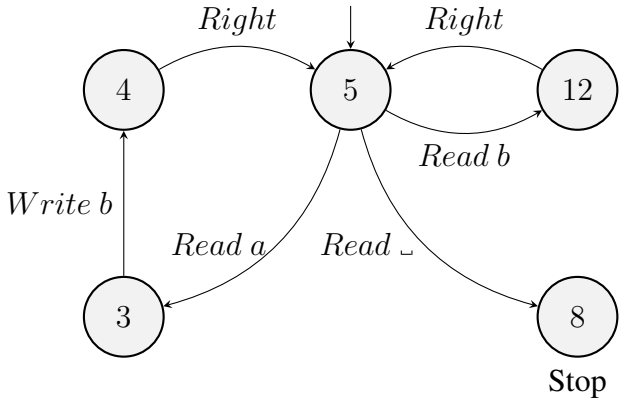
For $V = \{\mathsf{true}, \mathsf{false}\}$, the instructions available are the following:

- Read, which may result in `a` or `b` or ␣
- Write `a`
- Write `b`
- Write ␣
- Move Left
- Move Right
- No-op, which does nothing
- Return True (accept)
- Return False (reject)

If V is singleton then the Return instruction is usually just written Stop. It is important to keep note of the following points:

- Where is the head at the start?
- Where should the head be at the end?

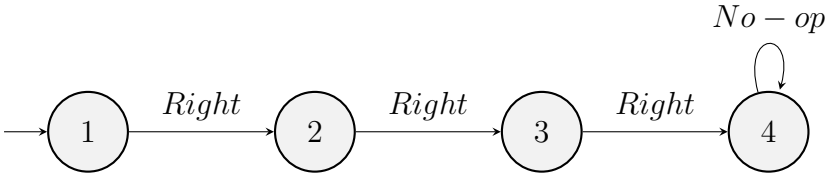
The following machine starts on the leftmost cell of an a, b-block on an otherwise blank tape. It moves to the right, converting every a to b, and halts on the cell to the right of the block.



For example:

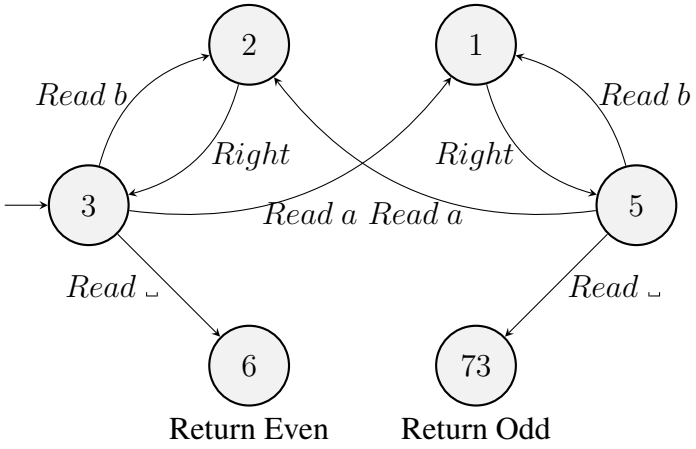
$\dot{\texttt{b}}$aba	5	Read b
$\dot{\texttt{b}}$aba	12	Right
b$\dot{\texttt{a}}$ba	5	Read a
b$\dot{\texttt{a}}$ba	3	Write b
bb$\dot{\texttt{b}}$a	4	Right
bbb$\dot{\texttt{a}}$	5	Read b
...		

The following machine moves three steps to the right and waits forever.



3.1 Parity Checking Example

The following "parity checking" machine has:
Tape Alphabet: $T = \{␣, a, b\}$, Return Set: $V = \{\text{Even}, \text{Odd}\}$
It starts on the leftmost cell of an a, b-block on an otherwise blank tape, and it ends on the cell to the right of the block, saying whether the number of a's is even or odd.



More formally, a Turing machine consists of the following, over T and V:

- A finite set of X states
- An initial state $p \in X$.
- A transition function δ from X to

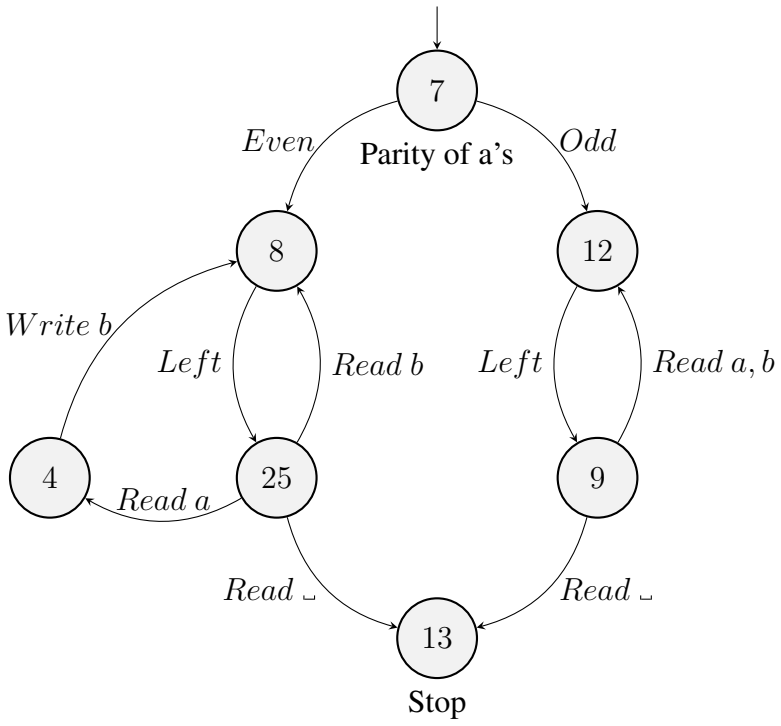
$$\begin{array}{rl} X^T & \text{(Read instructions and change state)} \\ +T \times X & \text{(Write instructions and change state)} \\ +X & \text{(Move head left and change state)} \\ +X & \text{(Move head right and change state)} \\ +X & \text{(No-op and change state)} \\ +V & \text{(Return a value from set } V\text{)} \end{array}$$

The above parity-checking TM can be formally described as:

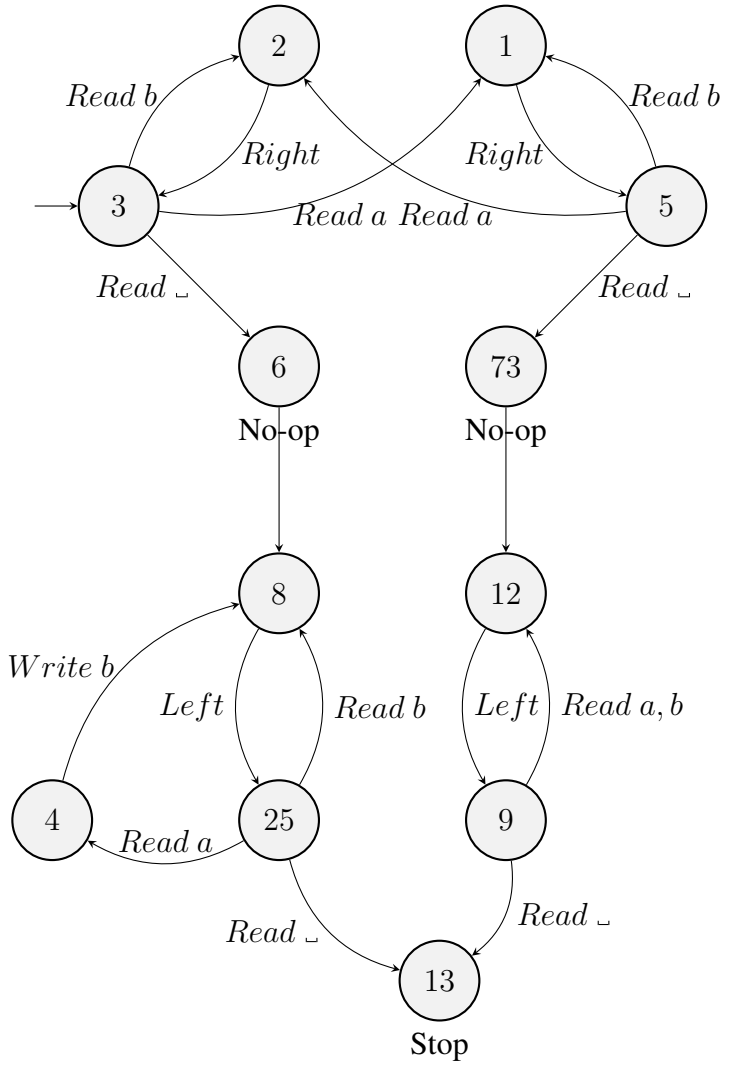
$$\begin{array}{ll} X \triangleright & (\{3, 2, 6, 73, 5, 1\}, \\ p \triangleright & \{3\}, \\ \delta \triangleright & \{3 \longmapsto \mathit{Read}(a \mapsto 1, b \mapsto 2, \sqcup \mapsto 6) \\ & \ 2 \longmapsto \mathit{Right}\ 3 \\ & \ 6 \longmapsto \mathit{Return}\ \mathit{Even} \\ & \ 5 \longmapsto \mathit{Read}(a \mapsto 2, b \mapsto 1, \sqcup \mapsto 73) \\ & \ 73 \longmapsto \mathit{Return}\ \mathit{Odd} \\ & \ 1 \longmapsto \mathit{Right}\ 5 \\ & \}) \end{array}$$

3.2 Macros

A convenient way of writing a program is using *macros*, which is a single instruction that abbreviates a whole program. To get the full program out of a *program with macros*, we need to *expand* all of them. Here is an example, using the parity checker that we saw above:

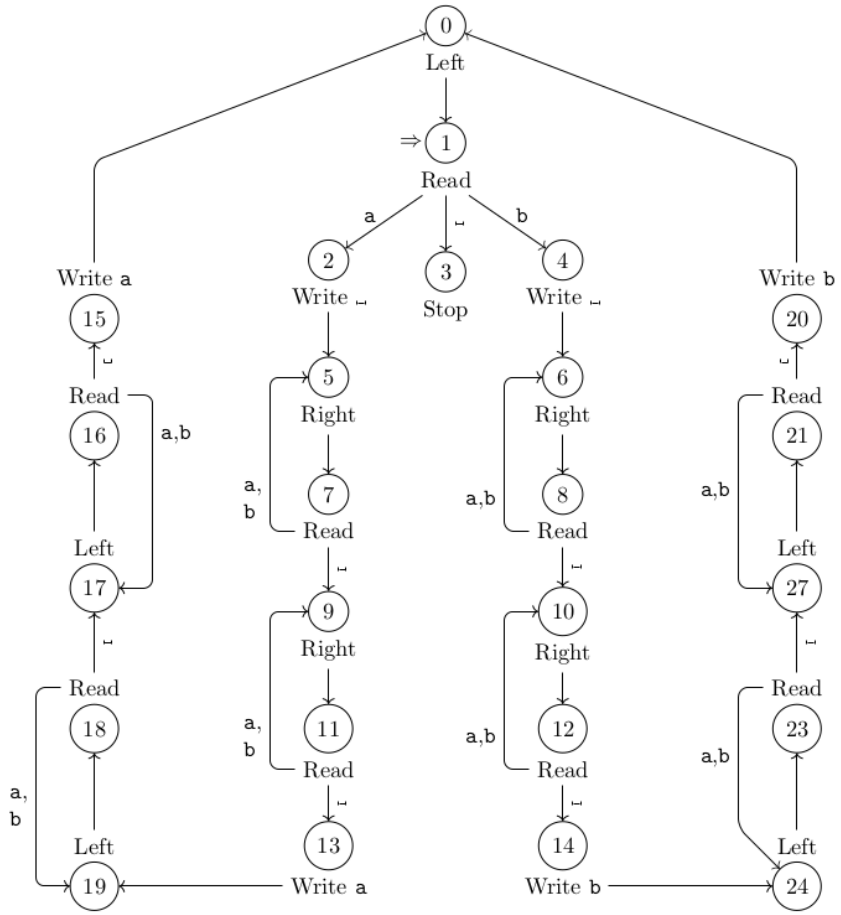


We can see that the state 7 is a macro, which abbreviates the parity checking of a i.e. whether the number of a's is even or odd. To obtain the full program, we expand this macro, which means that we replace "Parity of a's" with the parity checker's definition. Anything that points to the macro, will now point to the initial state of the definition. Likewise, if the state with the macro is initial, the initial state of the definition is initial state of the expanded program. For example, the arrow pointing to the starting state 7 will now point to the starting state 3 of the macro definition. Each Return instruction of the parity checker is replaced by a No-op, leading to the appropriate next state of the main program i.e. the next state will be the one that results from V after the macro.



3.3 Hacking

When implementing some programs, we may need to resort to a hacking approach. For example, we would like to build a Turing machine that starts at the rightmost character of an a, b-block on an otherwise blank tape and places a reversed copy of the input string to the right, with a *blank* character in between the original and reversed strings. The machine should halt with the head on the blank cell to the left of the original block. For example, at the start: abbaḃ
We expect to get the following output : ␣abbab␣babba
The following Turing machine implements the desired program:



Note how this machine is forced to use a $blank$ to record the position currently being copied. Whether the copied character is a or b is included in the state. We can work out the precise number of steps of this program, in terms of the length of the initial block. The general outline of the steps undertaken by this TM are given as under:

- Record the current character, whether its a or b
- Replace it with a $blank$ (hacking!)
- Move two steps to the right and write this character
- Then go back and replace the $blank$ with a or b (whatever was there before)
- Move one step to the left
  - Record the current character, whether its a or b
  - Replace it with a $blank$ (hacking!)
  - Move right to the central $blank$ character
  - Move right to the next $blank$ and write this character
  - Move left to the central $blank$ character
  - Move left to the character we just *blanked-out*
  - Replace the $blank$ with a or b (whatever was there before)
  - Move one step to the left, if its $blank$ then stop
  - Otherwise, begin the cycle again.

The running time is evidently *quadratic*: $1 + 2 + 3 + \cdots + n$ times a constant! It can be shown that the copy-reverse task cannot be solved any faster than this.

4 Turing Machine Variants

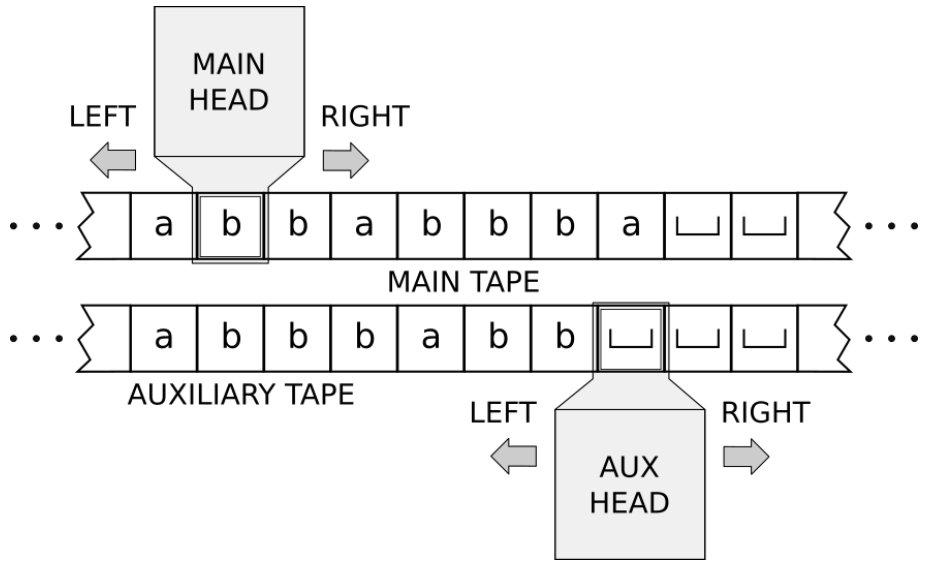
As we have seen, because the notion of Turing machine is so conservative, programs can be intricate and run slowly. Let us consider some more liberal variants.

4.1 Auxiliary Characters

Suppose that, in addition to the input alphabet and the blank, we have a finite set of auxiliary characters. A program may assume that initially these do not appear, and must guarantee that finally they don't appear, but in the middle of execution they can be used. For example, suppose the input alphabet is $\{\mathtt{a}, \mathtt{b}\}$ and the auxiliary alphabet is $\{\mathtt{a}', \mathtt{b}'\}$. Then we can write a more straightforward program for copy-reverse, using $\mathtt{a}'$ to indicate a currently being copied, and $\mathtt{b}'$ to indicate b currently being copied (rather than using the blank as previously).

4.2 Auxiliary/Multitape Turing Machines

A two-tape Turing machine has a main tape and an auxiliary tape, with a head on each tape. The input to the two-tape TM is provided on the main tape and a program may assume that initially the auxiliary tape is blank and must ensure that finally it is blank. The original single-tape TM and its reasonable variants all have the same power i.e. they are able to recognize the same class of languages. The available instructions are Write Main x, Write Aux x, Read Main, Read Aux, Left Main, Left Aux, Right Main, Right Aux, No-op and Return v.

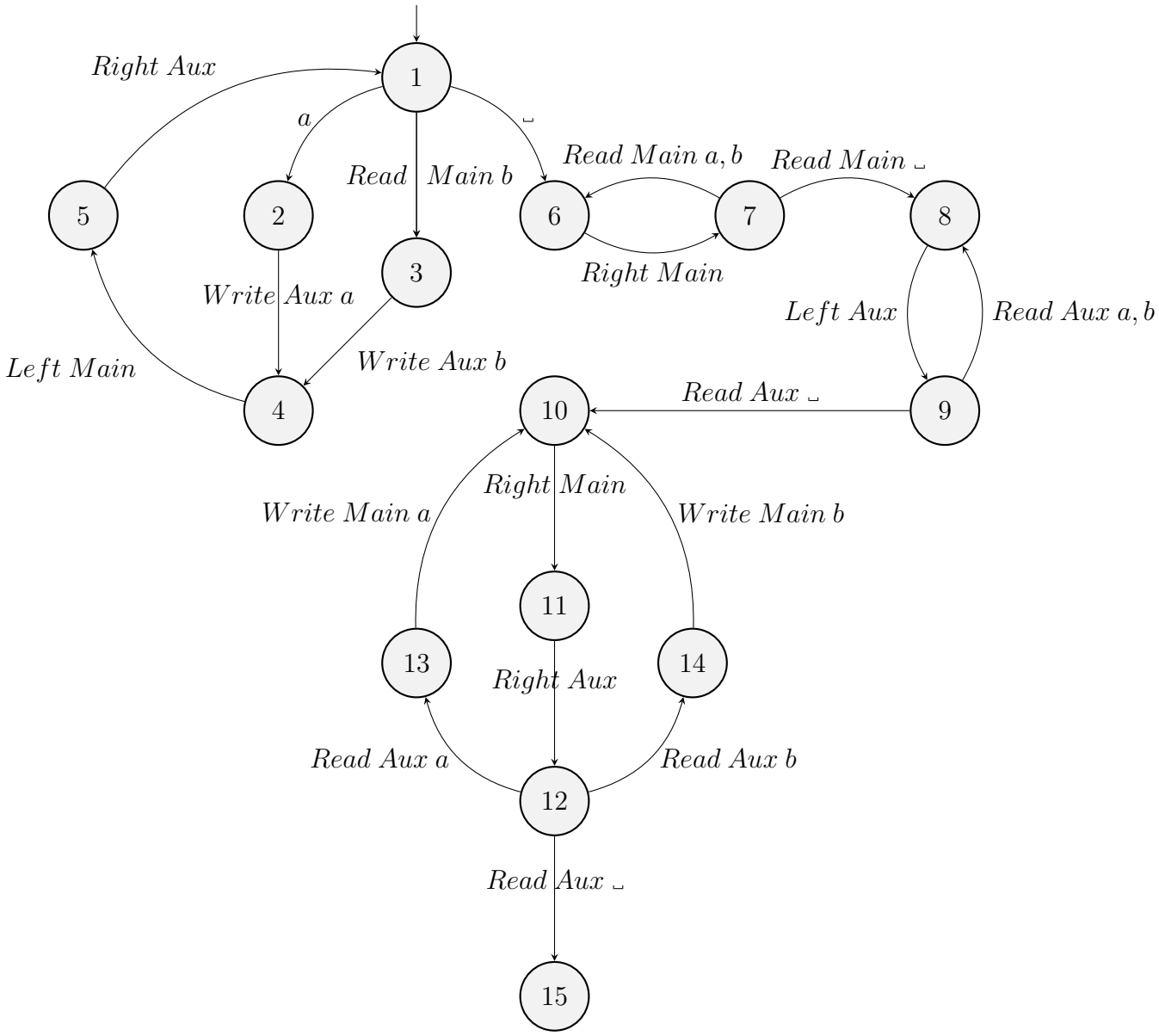


Details on the available instructions are given below:

- Read Main, which may result in a or b or ␣
- Read Aux, which may result in a or b or ␣
- Write Main x (x = a or b or ␣)
- Write Aux x (x = a or b or ␣)
- Left Main
- Left Aux
- Right Main
- Right Aux
- No-op, which does nothing
- Return True (accept)

- Return False (reject)

The following two-tape TM shows how the copy-reverse problem can be solved in linear time:



Move left through 2 blocks on main tape and stop

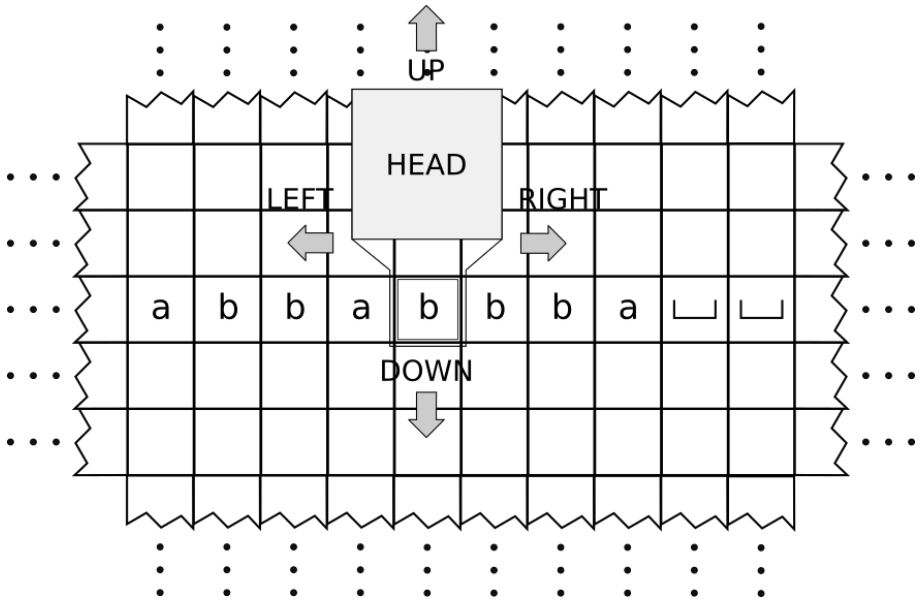
For example, if this two-tape TM starts with: ␣abbab␣$\cdots$ on the main tape (main head at the first blank, after the string) and ␣␣␣␣␣␣␣$\cdots$ on the auxiliary tape (auxiliary head at the first blank, on an overall empty tape). In the first phase, from states 1 to 5, the TM reads from the main tape (a's and b's) and writes them to the auxiliary tape, while moving the main head to the left and the auxiliary head to the right. Once it reads a *blank* from the main tape, it resets both heads, by moving the main head to the right and auxiliary head to the left, until both read a *blank* (states 6 to 9).

In the second phase, from states 10 to 14, it moves both of the heads to the right, and reads from the auxiliary tape (a's and b's, which are now in reverse order) and writes them to the main tape. Once it reads a *blank* from the auxiliary tape, the TM knows that it is done with the program, and can reset its both heads, as desired. We can also erase the auxiliary tape, during the resetting process (details omitted here). We expect to get the following output on the main tape: ␣abbab␣babba

4.3 Two Dimensional Turing Machines

A two-dimensional Turing machine (2D-TM) has a infinite sheet rather than a tape. The available instructions are Write x, Read, Right, Left, Up, Down, No-op and Return v. A program may assume that initially

the sheet is blank except for one row, and must ensure that finally it is blank except for that row. The following figure shows a schematic represenation of a two dimensional turing machine.



5 Summary

In this handout, we have quickly reviewed the complexity notations and understood the need to study Turing machines. We have seen the general model of a Turing machine, which contains a head, an infinite tape, and can move its head left/right while executing a program. We have understood that a TM is a simple formal model of mechanical computation, and it can do everything that a real computer can do. We have studied some examples of Turing machines, including parity checking, macros and hacking. We have also discussed Turing machine variants, including auxiliary characters, multi-tape and two-dimensional TMs. We have understood that the "append reversed copy" problem can be solved in:

- quadratic time $O(n^2)$ on a TM.
- linear time $O(n)$ on 2-tape TM.

Do you think that we can solve this problem on a TM in linear time? The answer is No! Do you think which of these machine models of computation is more appropriate? You could argue that a 2-tape TM is *unrealistic* because it allows for instant communication between the two heads that may be far apart.

We have seen that the same problem can be solved with different complexity on different machines e.g. quadratic on one machine can be linear on another. We haven't however seen that:

- a problem that can be solved in polynomial time on one kind of machine but not on another.
- a problem can be solved in one kind of machine but not on the other.

Actually, these things can't happen for all the kinds of machines we have looked at. So, for Constance, it matters whether we use a TM or a 2D-TM, but for Polly, it doesn't matter because it is still polynomial whether its $O(n)$ or $O(n^2)$.

6 Further Readings / References

- Sipser, M. (2013) *Chapter #3: THE CHURCH – TURING THESIS, Introduction to the Theory of Computation*, 3rd Edition, CENGAGE Learning Custom Publishing, Mason, USA