Turing Machines

▼ *Turing Machines*

Introduction to 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.

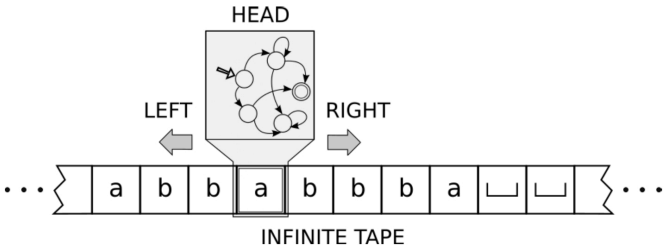
This kind of analysis is widely used and convenient. But how do we get the basic step counts in each part of the code? They are just assumptions (or guesses). For example, many people analyze sorting algorithms by assuming that each comparison of two values is "one step". That's certainly a helpful assumption 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 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 $T = \{\texttt{a}, \texttt{b}, ␣\}$. The set of non-blank characters $\{\texttt{a}, \texttt{b}\}$ is called the *input alphabet* (Σ).
- The set V of *Return Values*.

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

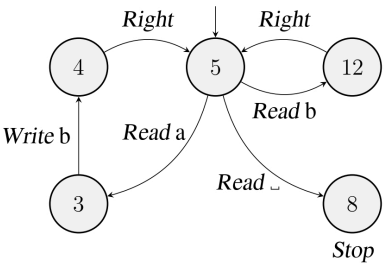
- *Read*, which may result in `a` or `b` or ␣
- *Write* `a`
- *Write* `b`
- *Write* ␣
- *Move Left*
- *Move Right*
- *No-op*, which does nothing

- *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?

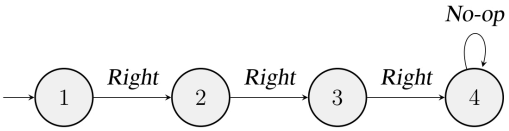
The following machine starts on the leftmost cell of an a, b-block on an otherwise blank tape. It moves to the right, converting every a to b, and halts on the cell to the right of the block.



For example:

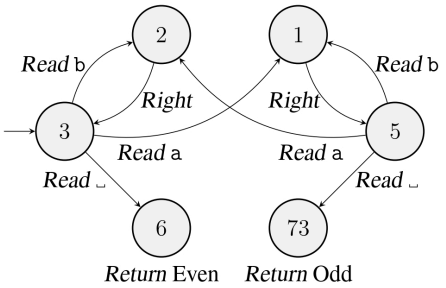
$\dot{\texttt{b}}$aba	5	*Read* b
$\dot{\texttt{b}}$aba	12	*Right*
b$\dot{\texttt{a}}$ba	5	*Read* a
b$\dot{\texttt{a}}$ba	3	*Write* b
bb$\dot{\texttt{b}}$a	4	*Right*
bb$\dot{\texttt{b}}$a	5	*Read* b
...		

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



2.1 Parity Checking Example

The following "parity checking" machine has:
Tape Alphabet: $T = \{␣, \texttt{a}, \texttt{b}\}$, 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

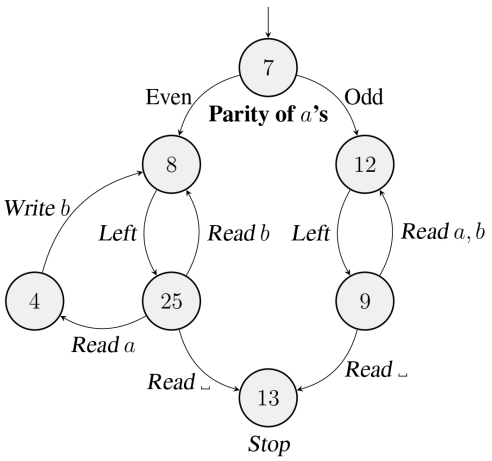
$$\begin{aligned}
&\{\textit{Read}(f) \mid f \in X^T\} && \text{(Read instructions and change state)}\\
\cup\, &\{\textit{Write}(l, s) \mid (l, s) \in T \times X\} && \text{(Write instructions and change state)}\\
\cup\, &\{\textit{Left}\ s \mid s \in X\} && \text{(Move head left and change state)}\\
\cup\, &\{\textit{Right}\ s \mid s \in X\} && \text{(Move head right and change state)}\\
\cup\, &\{\textit{No-op}\ s \mid s \in X\} && \text{(No-op and change state)}\\
\cup\, &\{\textit{Return}\ v \mid v \in V\} && \text{(Return a value from set } V\text{)}
\end{aligned}$$

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

$$\begin{aligned}
X \triangleright\ & (\{3, 2, 6, 73, 5, 1\},\\
p \triangleright\ & \{3\},\\
\delta \triangleright\ & \{3 \longmapsto \textit{Read}(a \mapsto 1, b \mapsto 2, \text{␣} \mapsto 6)\\
& \ \ 2 \longmapsto \textit{Right}\ 3\\
& \ \ 6 \longmapsto \textit{Return}\ \text{Even}\\
& \ \ 5 \longmapsto \textit{Read}(a \mapsto 2, b \mapsto 1, \text{␣} \mapsto 73)\\
& \ \ 73 \longmapsto \textit{Return}\ \text{Odd}\\
& \ \ 1 \longmapsto \textit{Right}\ 5\\
& \})
\end{aligned}$$

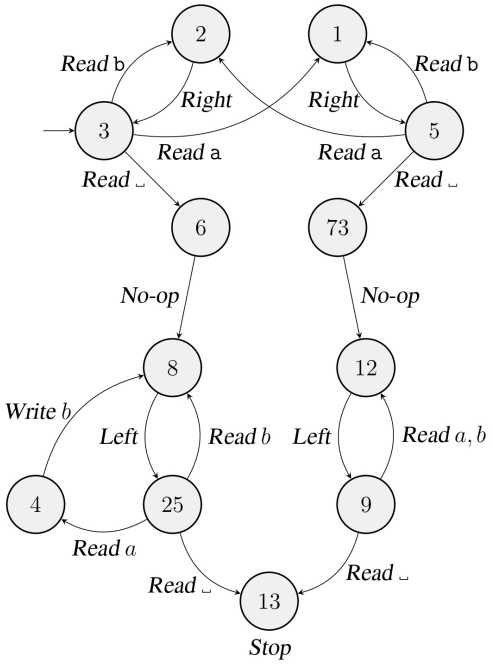
2.2 Macros

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.



2.3 Example: Reverse copy on a single tape machine

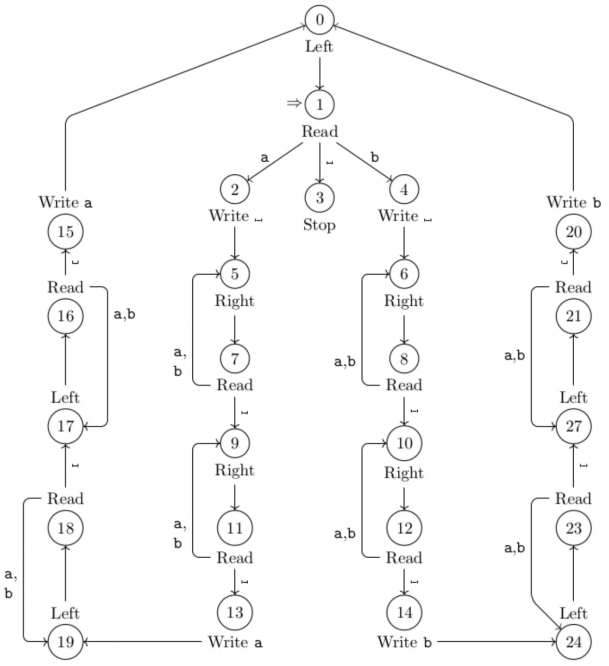
For example, we would like to build a Turing machine that starts at the rightmost character of an a, b-block on an otherwise blank tape and places a reversed copy of the input string to the right, with a $blank$ character in between the original and reversed strings. The machine should halt with the head on the blank cell to the left of the original block. For example, at the start:

abbab

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 ␣ character to record the position currently being copied whether the copied character is a or b is included in the state. This is somewhat an abuse of the ␣ character which would normally denote the start/end of the input — *hacking!*

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 ␣ (*hacking!*)
- Move two steps to the right and write this character
- Then go back and replace the ␣ 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 ␣ (*hacking!*)
  - Move right to the central ␣ character
  - Move right to the next ␣ and write this character
  - Move left to the central ␣ character
  - Move left to the character we just blacked-out
  - Replace the ␣ with a or b (whatever was there before)
  - Move one step to the left, if its ␣ 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.

▼ *Fancy Turing Machines Part 1*

Fancy Turing Machines

1 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.

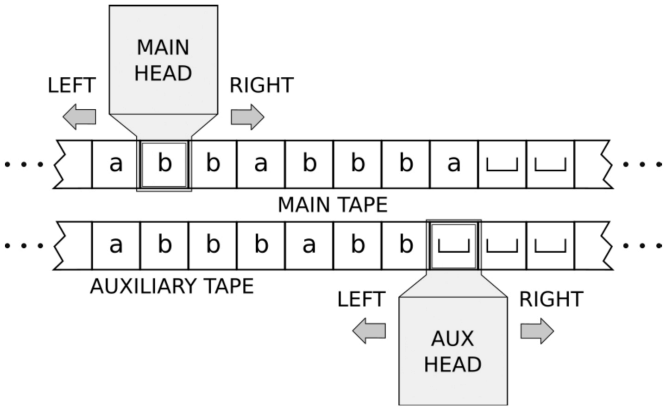
1.1 Auxiliary Characters

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 $\mathtt{a}$ currently being copied, and $\mathtt{b}'$ to indicate $\mathtt{b}$ currently being copied (rather than using the blank as previously).

1.2 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.



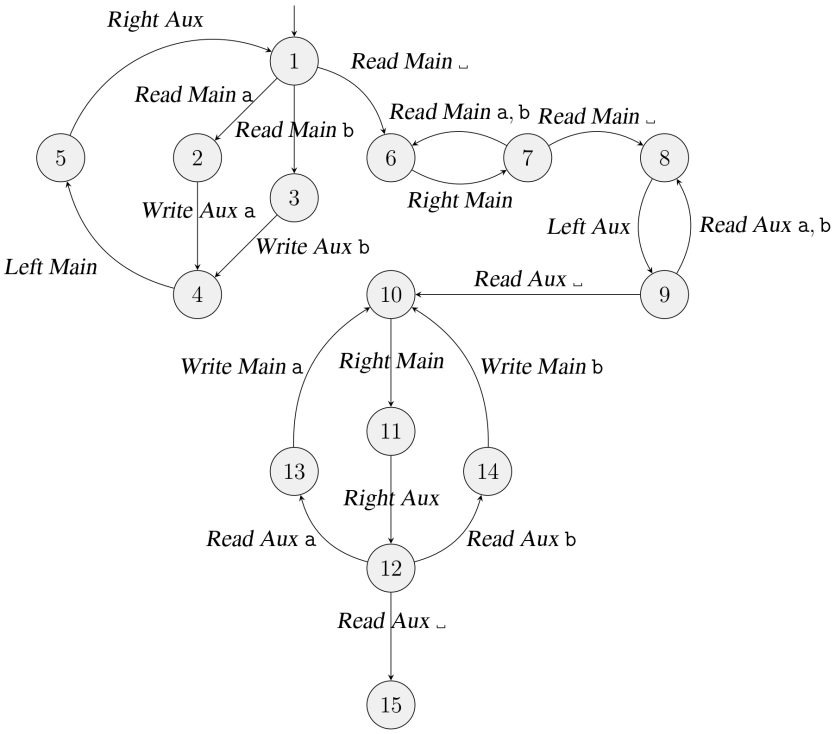
Details on the available instructions are given below:

- *Read Main*, which may result in $\mathtt{a}$ or $\mathtt{b}$ or $\sqcup$
- *Read Aux*, which may result in $\mathtt{a}$ or $\mathtt{b}$ or $\sqcup$
- *Write Main* x (x = $\mathtt{a}$ or $\mathtt{b}$ or $\sqcup$)
- *Write Aux* x (x = $\mathtt{a}$ or $\mathtt{b}$ or $\sqcup$)

- *Left Main*
- *Left Aux*
- *Right Main*
- *Right Aux*
- *No-op*, which does nothing
- *Return* `true` (accept)
- *Return* `false` (reject)

1.2.1 Example: Reverse copy on a two-tape machine

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: ␣abbaḃ␣$\cdots$ on the main tape (main head at the rightmost non-blank character) 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 should 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

2 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.

For a more famous example that has been analysed in the literature, consider the problem of testing palindromicity. This can be accomplished in

- quadratic time $\theta(n^2)$ on a TM
- $\theta(n^2/\log n)$ on a two-dimensional TM
- linear time $\theta(n)$ on a two-tape TM.

Recall that θ means that these are lower bounds as well as upper bounds. So the problems cannot be solved any faster.

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, for all the kinds of machines we have looked at, these things can't happen. 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 $O(n^2)$.

3 Further Readings / References

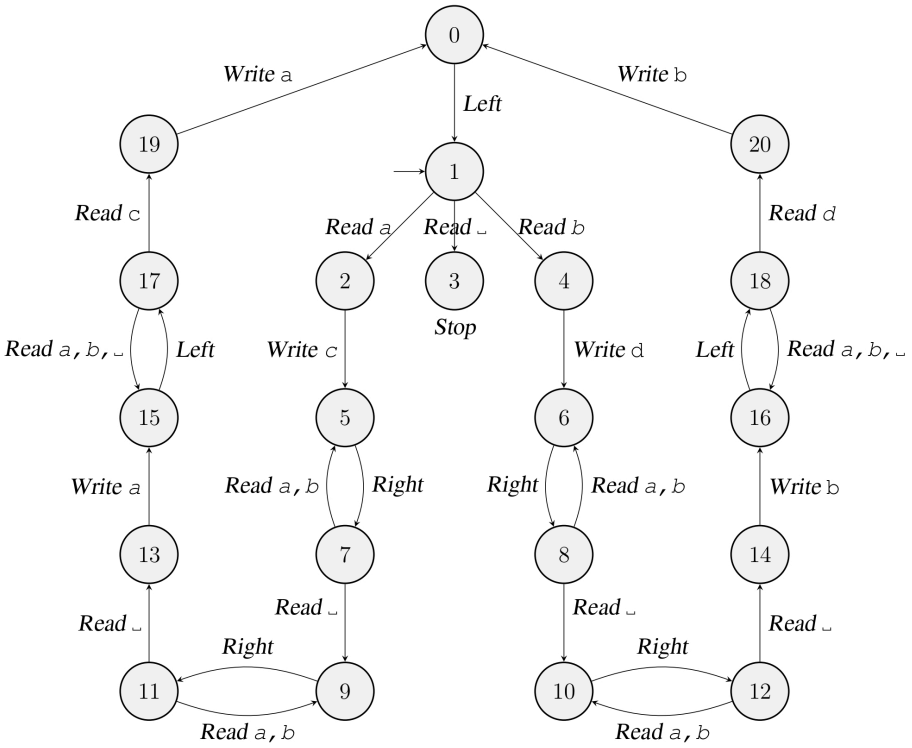
- Sipser, M. (2013) *Chapter #3: The Church–Turing Thesis, Introduction to the Theory of Computation*, 3rd Edition, CENGAGE Learning Custom Publishing, Mason, USA

▼ *Fancy Turing Machines Part 2*

Converting Fancy Turing Machines to Simple Machines

1 Turing Machines using Auxiliary Characters

Suppose that, in addition to the input alphabet and the blank, we have a finite set of auxiliary characters. A program assumes 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 {a, b} and the auxiliary alphabet is {c, d}. This means that we can have instructions *Write* a, *Write* b, *Write* c, *Write* d and *Write* ␣, and each *Read* instruction has 5 possible outcomes (a, b, c, d and ␣). We can now write a more straightforward (but still quadratic time) program for copy-reverse problem, using c to indicate that character a is currently being copied, and d to indicate that character b is currently being copied (rather than using the blank, as previously). For example, here is a fancy TM (a TM including auxiliary characters) for the copy-reverse problem discussed earlier.



We're going to learn a general method for converting a fancy TM to an ordinary TM. This general method will involve the following steps:

1. Give a relation between the tape configurations of fancy TM and the tape configuration of the simple TM. In simple words, define a way to represent the tape configuration (i.e. tape contents plus head position) of the fancy TM as a configuration of the simple TM. This is a *creative step!*
2. Give a program to convert the initial configuration of the fancy TM into a corresponding configuration of the simple TM (The "Setting-up" program).
3. For each instruction of the fancy TM, show how to simulate it on a simple TM. In other words, show how to "perform" each step of the fancy TM on a simple TM (The "Simulating" programs).
4. Give a program to convert the result of simple TM to the fancy TM result (The "Finishing" program).

Typically all of the above programs are polynomial time.

Key Point: *A polynomial time program on a Fancy Turing machine can be converted into a polynomial time program on a Simple Turing machine.*

We will now show the details of the above steps in the following sections.

1.1 Defining Relation between Fancy & Simple Tape Configurations

Let's say we have a fancy TM using the input alphabet {a, b}, and auxiliary characters {c, d}. It means that we **assume** only a, b and ␣ at the start of TM, but we are **allowed** to use c and d in the middle of execution. However, we must **ensure** that only a, b, ␣ are present in the output of the fancy TM at the end. When defining the relationship between fancy and simple tapes, the key point will be to assume that the fancy and simple tape configurations are related at the start of each simulation step, we may violate this relationship in the middle of the step, but we must ensure it at the end.

In this step, we shall define a relation between the fancy and simple TMs' tapes configurations. Lets consider the following fancy tape configuration:

$$\texttt{acbccda}$$

Note: *Obviously, the above configuration is not the initial configuration of the fancy TM, as the auxiliary symbols can only appear on the tape during the execution, but not at the start or at the end.*

In order to represent the fancy TM's tape configuration as a simple TM's tape configuration, we define the following relation (which is a *creative suggestion*):

Character on Fancy tape	Represented on Simple tape
a	aa
b	bb
c	ab
d	ba
␣	␣␣

The simple TM's head position will be the leftmost of the two characters representing the fancy TM's head position. For example, the fancy tape configuration:

$$\texttt{acbccda} \tag{1}$$

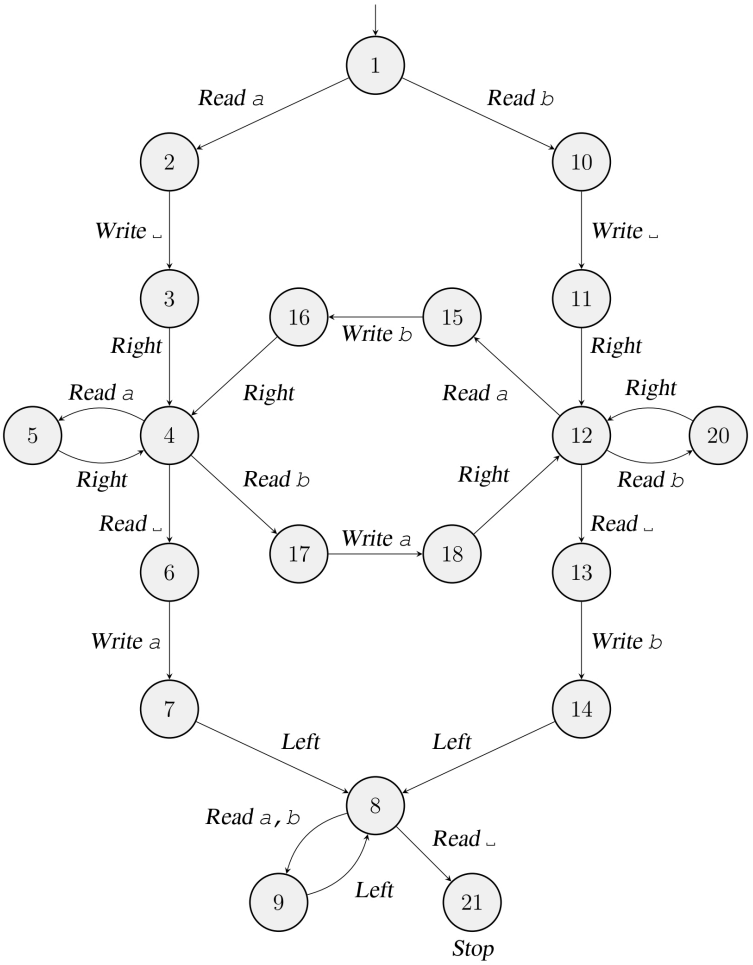
is represented as the simple tape configuration:

$$\texttt{aaabbbababbaaa} \tag{2}$$

1.2 Converting Initial Configurations – The Setting-up Program

In this step, we will setup the simple TM to represent the fancy TM *i.e.* we want a program that stretches a fancy *input* tape — recall that this will contain only a, b, ␣ — into a corresponding simple tape. For example, if the input is initially ȧbbab, the simple TM will start by *stretching* the given input to ȧabbbbaabb.

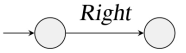
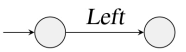
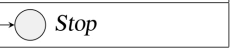
One way to write a stretching program is to add one character at a time, which would require $O(n)$ steps, and then repeat it for each of the characters in the input, requiring a total of $O(n^2)$ steps. As an example, consider the modified version of the TM seen during last week, which makes a space at the current head position, e.g. given the input $\dot{a}bab$ it will result in the output $\dot{\sqcup}abab$. We can further modify this TM to achieve the stretching program (left as an exercise for you).



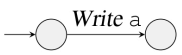
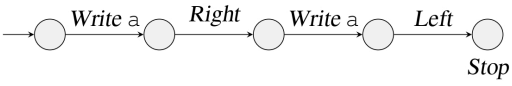
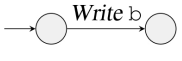
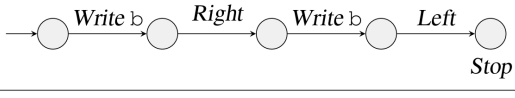
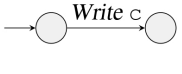
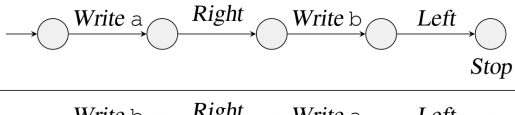
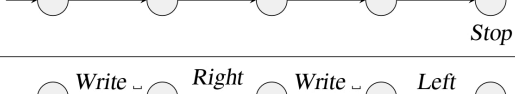
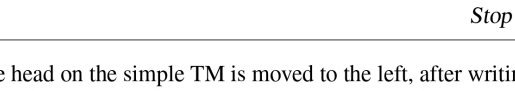
You can easily see that the above TM takes $O(n)$ steps for creating a space; we will have a similar complexity for the modified version to duplicate a single character on the tape. The full stretching program will repeat the above steps for each of the characters on the tape, therefore, it will take $O(n^2)$ steps.

1.3 Performing Fancy TM Instructions – The Simulating Programs

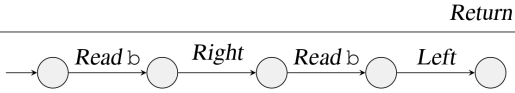
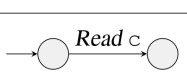
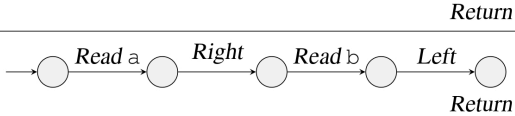
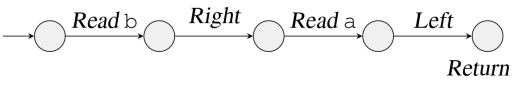
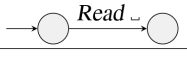
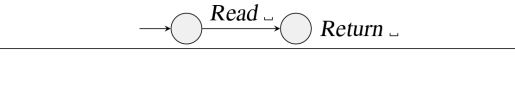
In this step, we simulate each instruction of the fancy TM by a program of the simple TM. For example, we simulate the fancy *Right*, *Left* and *Stop* as the following simple programs:

Fancy TM Steps	Simple Simulating TMs
	
	
	

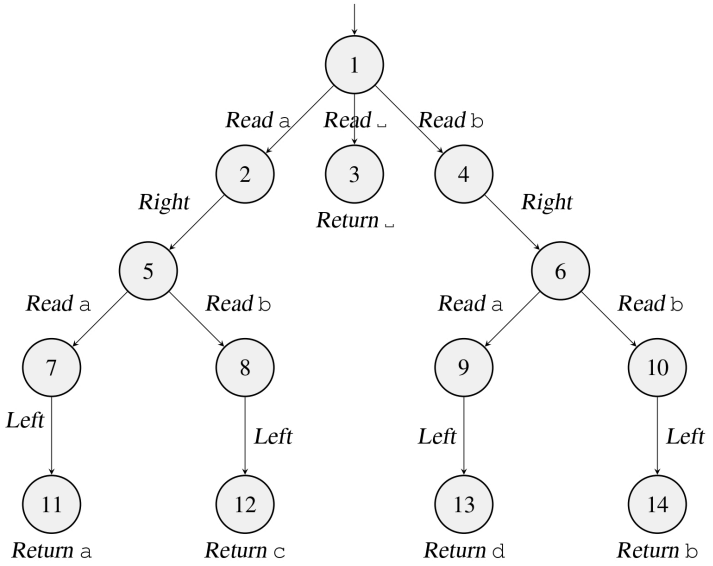
Similarly, we can create the following simple TM programs for the *Write* operations.

Fancy TM Steps	Simple Simulating TMs
	
	
	
	
	

It is important to note that the tape head on the simple TM is moved to the left, after writing the second character. Similarly, we can create the following simple TM programs for the *Read* operations.

Fancy TM Steps	Simple Simulating TMs
	
	
	
	

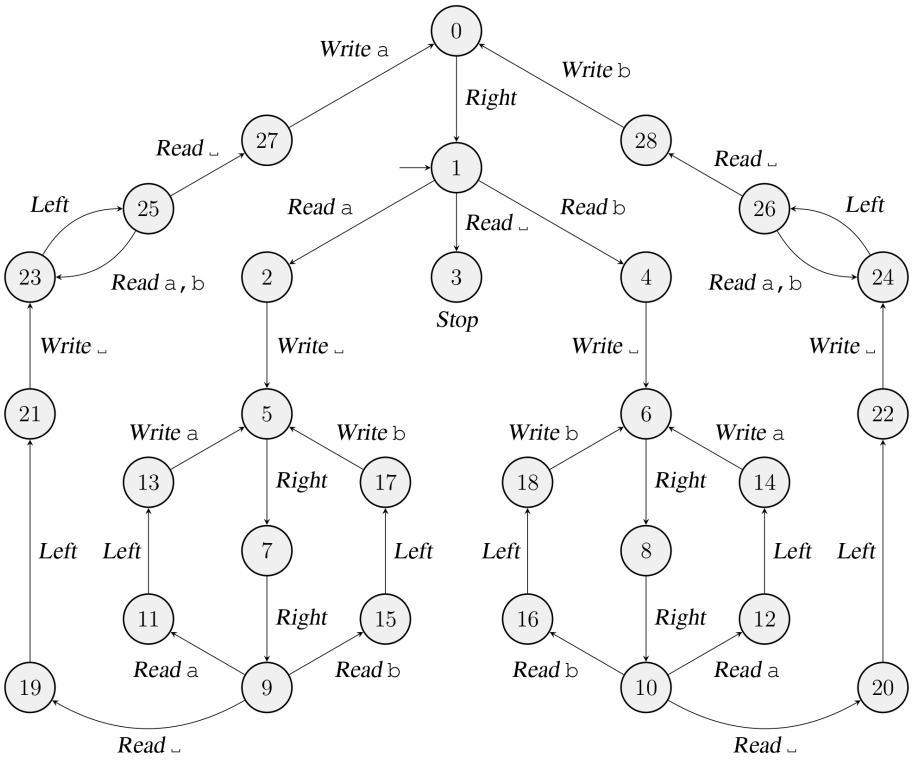
We can combine the simple simulating TMs for the read operations as shown below:



All of these simple TM programs are constant times e.g. each *Read/Write* operation takes 4 steps, therefore, all of these have polynomial time complexity.

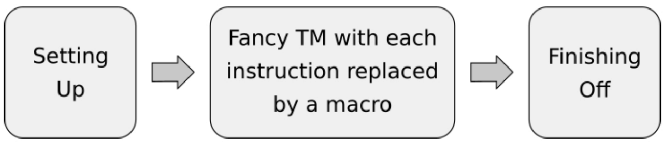
1.4 Converting Simple to Fancy TM's Output – The Finishing Program

In the last step, we want a program that *squashes* a simple tape back into a fancy one that serves as the output. For example, it would squash ȧabbaabbbb into ababb␣. Similar to the *stretching* program, the squashing program will do it one character at a time and its complexity will be quadratic i.e. $O(n^2)$. Here is one possible program for squashing the output tape of a simple (simulating) TM back to the fancy tape.



1.5 Fancy to Simple TM Conversion – Discussion

Given a fancy TM, the corresponding simple TM (expressed using macros) is as follows:



Suppose that the fancy program runs in polynomial time, then consider the following arguments:

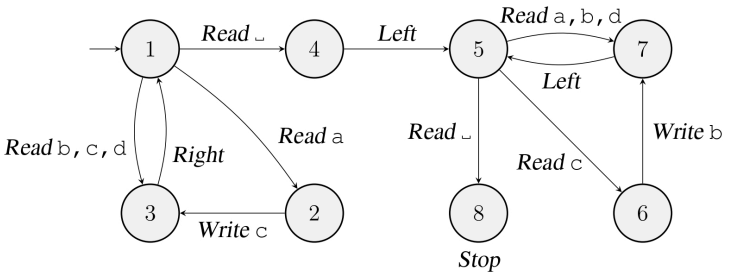
- The stretching program is quadratic time, as discussed earlier.
- Each simulating instruction is constant time, and polynomially many of them happen.
- The squashing program is also quadratic time, and gets applied to the simple tape contents that's twice the size of the corresponding fancy tape contents.

Therefore, the simple TM runs in polynomial time as well.

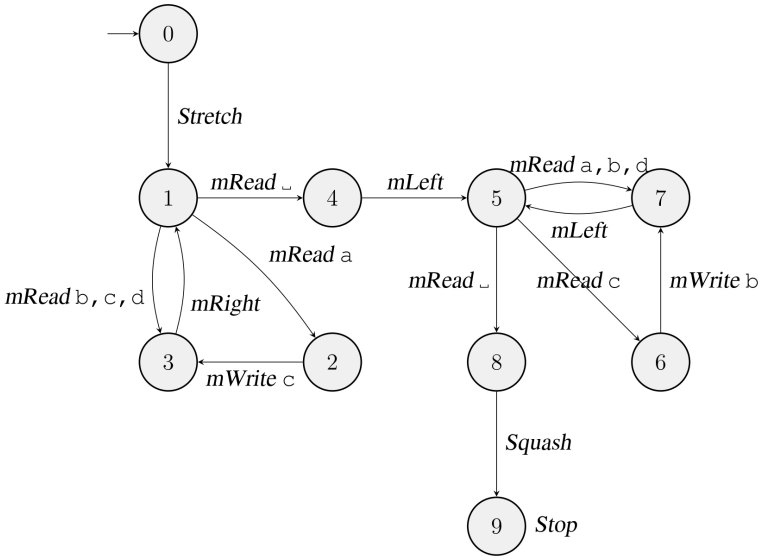
1.5.1 Fancy to Simple TM Conversion – Example

Now lets consider how we can convert a fancy program into a simple program. Let's assume that we have a fancy TM that starts on the leftmost character, changes all `a`'s to `b`'s and ends-up on the cell to the left of input block. Lets assume that the program accomplishes this in two steps:

1. When going to the right, the fancy TM changes all `a`'s to `c`'s
2. When going to the left, the fancy TM changes all `c`'s to `b`'s.



This becomes the following simple TM written using macros (that you can expand, if interested). Each macro name starts with m, in order to differentiate it from the fancy machine instruction, e.g. *mRead* is a macro that replaces the *Read* instruction of the fancy TM.



From the above discussion and examples, we can conclude the following:

1. We can convert a fancy TM using auxialiary characters into a simple TM, so allowing the extra characters does not give us any more power to express functions.
2. If the fancy TM is polynomial time (setting-up, simulating and finishing programs are all polynomial), then the resulting simple TM is also polynomial time.