CMPUT 675: Computational Complexity Theory **Winter 2019**

Lecture 1 (Jan 8): Introduction and Turing machines

Lecturer: Mohammad R. Salavatipour *Scribe: Zachary Friggstad*

Comment: The scribe notes for this lecture begin with more exposition than you will be expected to provide in your scribe notes. See the second set of scribe notes for a better example of what you are expected to submit.

This is also one of the least rigorous lectures. The idea is to impart the impression that we use Turing machines to precisely model computing devices, but many specific details are unimportant when it comes to what can be computed and how efficiently.

1.1 Computational Complexity Theory

In much of your undergraduate studies, your algorithms courses largely focused on designing fast algorithms to solve certain problems. Mostly, you designed polynomial-time algorithms for solving a variety of problems and, in some cases, you discussed ways to further improve the running time. For example, very simple sorting algorithms run in $O(n^2)$ time whereas slightly more involved sorting algorithms run in $O(n \log n)$ time.

Computational complexity moves the discussion away from specific algorithms and more to general questions about what resources are *required* to solve some problems and how different computational resources relate. The most famous example is **P** vs. **NP**.

This is a high-level question about two different *complexity classes*. In computational complexity, we study a variety of complexity classes that are usually defined based on limits to their computational resources, like running time or memory usage, or access to slightly non-standard computational resources like random bits or the ability to interact a more powerful computer in a limited way.

We will see many definitive results, but also many open questions. Most computing scientists are aware of the **P** vs. **NP** problem. But there are other unsolved problems we will discuss that have important implications. For example:

- **P** vs. **BPP**: There are some problems not yet known to be in **P** that can be correctly solved (with high probability) by a polynomial-time randomized algorithm. Do we really need randomization? It is widely conjectured that we do not: that $\mathbf{P} = \mathbf{BPP}$.
- One-way functions: It is conjectured that there is function that is efficiently computable but not efficiently invertible. This is a stronger assumption that $\mathbf{P} \neq \mathbf{NP}$ but is entirely consistent with our current understanding. What could we do with such a function? Essentially, we can design an encryption scheme using short keys that is *provably secure* against every efficient algorithm.

1.2 Turing machines - The Computational Model

To begin exploring the landscape of computing, we should adopt a concrete mathematical model. A first thought might be to somehow formalize a specification for the sort of desktop computers we use. This can be done, but it is not the approach usually taken. Instead, we describe a model that has a very compact description: the **Turing machine**.

1.2.1 The Specification of a Turing Machine

Tape
The memory used by a Turing machine is simply an infinite, bi-directional line of cells where each cell contains a single symbol. This is called the *tape* and is somewhat analogous to RAM on a desktop computer, but a Turing machine will only have sequential access to the tape. We view the tape cells as being indexed by $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

Tape Head
The way a Turing machine will access the contents of the tape is by maintaining a *tape head*, which can read and overwrite the contents of a single tape cell. The tape head moves back and forth across the tape, one cell at a time (*sequential access*).

The proper definition of a Turing machine varies quite a bit across literature. But, in some sense, the definition does not matter. They are essentially equivalent definitions. Here is the one we will use, I chose it for simplicity.

Definition 1 *A (single-tape)* **Turing machine** M *is a tuple* (Γ, Q, δ) *where:*

- Γ *is a finite set of symbols that* M*'s tapes can contain called the* **alphabet** *of* M*. We will always assume* Γ *contains the symbol* $\square$ *which we think of as the* **blank symbol**.
- Q *is a finite set called the* **states** *of* M*. We assume that* Q *always contains three special states: the* **start state** q_{START} *and two* **halting states** q_{ACCEPT} *and* q_{REJECT}.
- *A function* $\delta : (Q - \{q_{\text{ACCEPT}}, q_{\text{REJECT}}\}) \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$ *called the* **transition function** *of* M.

The full description of how to provide input to, initialize, and execute a Turing machine follows. A concrete example will be given later.

Inputs as Strings
Recall that for any set S that S^* denotes the set of all finite-length sequences of elements of S. For example, $\{0, 1\}^*$ is the set of all bitstrings one can form, including the *empty* string with no bits. We often denote members $x \in S^*$ by finite-length concatenations of symbols $x_0 x_1 x_2 x_3 \ldots x_{n-1}$ and let $|x| = n$, the length of the string. The empty string has length 0.

- If $S = \{a, b, c\}$ then $a, abba, bbbbca, ccc \in S^*$. We have $|abba| = 4$ and $|a| = 1$.
- If $S = \{0, 1\}$ then $0, 11, 101, 10000, 000001 \in S^*$. We have $|11| = 2$ and $|101| = 3$.

The input to a Turing machine M is simply a string $x \in (\Gamma - \{\square\})^*$ (i.e. no blanks in the input).

Initializing and Running a Turing machine
To model the execution of a Turing machine $M = (\Gamma, Q, \delta)$, we maintain three things:

- The current contents of the tape, which can be viewed as an infinite sequence of symbols $\mathbf{c} = (\ldots, c_{-2}, c_{-1}, c_0, c_1, c_2, \ldots)$ where each $c_i \in \Gamma$. For example, $(\ldots, \square, \square, 0, 1, 1, 0, 1, 0, \square, \square, \ldots)$
- The current state $q \in Q$ of M.
- The head position $h \in \mathbb{Z}$ indicating which position of the tape is currently being looked at by M.

We can neatly represent this as a tuple $(\mathbf{c}, q, h)$ called a **snapshot**.

The initial snapshot of M when given input $x \in (\Gamma - \square)^*$ is as follows:

- The initial tape $\mathbf{c}$ has $c_i := x_i$ for $0 \leq i \leq |x| - 1$ and $c_j := \square$ for all other indices j.

  That is, the symbols of x appear consecutively from cells 0 through $|x|-1$ on an otherwise blank tape. For example, if $x = 1011$ then the initial tape looks like $(\ldots, \square, \square, \square, 1, 0, 1, 1, \square, \square, \square, \ldots)$ where the leftmost 1 is at index 0 on the tape.

- The initial state is $q := q_{\text{START}}$.

- The initial head position is $h := 0$.

Then M will update this snapshot through a sequence of *steps/transitions* until the current state becomes one of q_{ACCEPT} or q_{ACCEPT}. Note that in a snapshot $(\mathbf{c}, q, h)$, $c_h \in \Gamma$ is the symbol of the cell currently being examined by M.

To update the snapshot after a single step, consider the transition function value $\delta(q, c_h) = (q', \gamma, d)$ for the current state and the contents of the tape being examined.

- The next state is q'.

- The symbol in the tape cell at index h is replaced with γ, all other tape cells remain unchanged.

- The head moves **left** if $d = L$ (i.e. $h' = h - 1$), **right** if $d = R$ (i.e. to $h' = h + 1$)), or **stays** if $d = S$ (i.e. $h' = h$).

This produces a new snapshot $(\mathbf{c}', q', h')$ where $\mathbf{c}'$ is the same as $\mathbf{c}$ except at index h: $c'_h = \gamma$.

The complete execution of a TM on some input x involves creating the initial snapshot for this input, and then constructing the sequence of snapshots iteratively by applying the above rules to go from one snapshot to the next. Once the state becomes either q_{ACCEPT} or q_{ACCEPT}, then M is said to **halt** and no further steps will be made. It could be that M never halts.

1.2.2 An Example

Let us design a Turing machine with $\Gamma = \{0, 1, \square\}$ that, upon input $x \in \{0, 1\}^*$, is guaranteed to halt and will halt in the state q_{ACCEPT} if and only if x is a palindrome (e.g. $110011, 010, 1, 00000$ or even the empty string, but not $110, 10$ or 101001).

This is the only time in the lectures where we will completely specify an entire TM. The point of including this in the lecture is to help you internalize how "programming" a TM is much like programming for a desktop computer.

If we were designing a function to do this in, say, the programming language C then we can imagine using a loop that compares the first and last character, then the second and second last character, and so on ensuring they are all equal. We can do essentially the same thing here.

States and Their Transitions
We use the following states $Q = \{q_{\text{START}}, q_{\text{ACCEPT}}, q_{\text{Reject}}, q_0, q_1, q'_0, q'_1, q_b\}$. Intuitively:

- When we are in q_{START}, the head is at the start of whatever string remains to be inspected. The TM accepts if the string is empty, otherwise it enters state q_0 or q_1, depending on what the leading bit was. It also erases this leading bit when transitioning out of q_{START}.

- In q_0 or q_1, the head is moving to the end of the string. The subscript 0 or 1 indicates we are remembering that the start of the current string had a 0 or 1.

- In q'_0 or q'_1, the head is at the end of the string. The TM checks that the end bit matches the subscript of the state, rejecting if it does not. If there was no end bit, the TM accepts the input as an odd-length palindrome. Otherwise, the end bit matched so we erase it and begin scanning back to the start of the string in state q_b.
- In q_b, the head is moving back to the start of the string.

See the next page for the full specification and a sample execution.

Transition Function

current state	symbol at head	next state	symbol to write at head	direction	comment
q_{START}	0	q_0	$\square$	R	
q_{START}	1	q_1	$\square$	R	
q_{START}	$\square$	q_{ACCEPT}	$\square$	R	even-length palindrome
q_0	0	q_0	0	R	scanning to end
q_0	1	q_0	1	R	scanning to end
q_0	$\square$	q'_0	$\square$	L	end found
q_1	0	q_1	0	R	scanning to end
q_1	1	q_1	1	R	scanning to end
q_1	$\square$	q'_1	$\square$	L	end found
q'_0	0	q_b	$\square$	L	expecting a 0 at the end
q'_0	1	q_{REJECT}	$\square$	L	
q'_0	$\square$	q_{ACCEPT}	$\square$	R	odd-length palindrome
q'_1	0	q_{REJECT}	$\square$	L	
q'_1	1	q_b	$\square$	L	expecting a 1 at the end
q'_1	$\square$	q_{ACCEPT}	$\square$	R	odd-length palindrome
q_b	0	q_b	0	L	scanning to start
q_b	1	q_b	1	L	scanning to start
q_b	$\square$	q_{START}	$\square$	R	now at start of leftover string

Below is a sample of how this Turing machine runs on input 10101. The underlined symbol indicates the head position. The horizontal separator lines are just for presentation.

q_{START}	...	$\square$	$\underline{1}$	0	1	0	1	$\square$	...	initial snapshot
q_1	...	$\square$	$\square$	$\underline{0}$	1	0	1	$\square$	...	
q_1	...	$\square$	$\square$	0	$\underline{1}$	0	1	$\square$	...	
q_1	...	$\square$	$\square$	0	1	$\underline{0}$	1	$\square$	...	going to end
q_1	...	$\square$	$\square$	0	1	0	$\underline{1}$	$\square$	...	
q_1	...	$\square$	$\square$	0	1	0	1	$\underline{\square}$	...	
q'_1	...	$\square$	$\square$	0	1	0	$\underline{1}$	$\square$	...	checking for 1
q_b	...	$\square$	$\square$	0	1	$\underline{0}$	$\square$	$\square$	...	
q_b	...	$\square$	$\square$	0	$\underline{1}$	0	$\square$	$\square$	...	going to start
q_b	...	$\square$	$\square$	$\underline{0}$	1	0	$\square$	$\square$	...	
q_b	...	$\square$	$\underline{\square}$	0	1	0	$\square$	$\square$	...	
q_{START}	...	$\square$	$\square$	$\underline{0}$	1	0	$\square$	$\square$	...	repeat with shorter string
q_0	...	$\square$	$\square$	$\square$	$\underline{1}$	0	$\square$	$\square$	...	
q_0	...	$\square$	$\square$	$\square$	1	$\underline{0}$	$\square$	$\square$	...	going to end
q_0	...	$\square$	$\square$	$\square$	1	0	$\underline{\square}$	$\square$	...	
q'_0	...	$\square$	$\square$	$\square$	1	$\underline{0}$	$\square$	$\square$	...	checking for 0
q_b	...	$\square$	$\square$	$\square$	$\underline{1}$	$\square$	$\square$	$\square$	...	going to start
q_b	...	$\square$	$\square$	$\underline{\square}$	1	$\square$	$\square$	$\square$	...	
q_{START}	...	$\square$	$\square$	$\square$	$\underline{1}$	$\square$	$\square$	$\square$	...	repeat with shorter string
q_1	...	$\square$	$\square$	$\square$	$\square$	$\underline{\square}$	$\square$	$\square$	...	going to end
q'_1	...	$\square$	$\square$	$\square$	$\underline{\square}$	$\square$	$\square$	$\square$	...	checking for 1
q_{ACCEPT}	...	$\square$	$\square$	$\square$	$\square$	$\underline{\square}$	$\square$	$\square$	...	odd-length palindrome

1.2.3 Languages and Decidability

For the sake of simplicity in presentation, we mostly will stick with the alphabet $\Gamma = \{0, 1, \square\}$ unless otherwise stated. Everything here applies naturally to TMs with more general alphabets.

For any input $x \in \{0,1\}^*$, running a TM M on input x produces one of three results:

- **Accept**: The TM halts after entering state q_{ACCEPT}
- **Reject**: The TM halts after entering state q_{REJECT}
- **Did Not Halt**: The TM never enters one of the halting states, the transitions never stop.

In this way, a TM M defines a **partial function** $M : \{0,1\}^* \to \{\text{ACCEPT}, \text{REJECT}\}$ on the inputs x where M halts. We write $M(x)$ for the corresponding ACCEPT or REJECT if M halts on input x. If M does halt on input x, we say that $M(x)$ does not halt (is not defined).

Definition 2 *A* **language** *is a subset $\mathcal{L} \subseteq \{0,1\}^*$.*

Definition 3 *Let $\mathcal{L} \subseteq \{0,1\}^*$ be a language. A Turing machine M is said to* **decide** *$\mathcal{L}$ if M halts on every $x \in \{0,1\}^*$ and $M(x) =$ ACCEPT if and only if $x \in \mathcal{L}$. A language $\mathcal{L}$ that can be decided by some Turing machine is said to be* **decidable***.*

This concept of a language is very broad. For example, we can consider the language of palindromes:

$$\text{PALINDROMES} = \{x \in \{0,1\}^* : x = x^{rev}\}$$

where, of course, x^{rev} denotes the reversal of string x.

If one views $a, b \in \{0,1\}^+$ (same as $\{0,1\}^*$ except excluding empty strings) as being the set of integers written in binary form (with, perhaps, leading 0s) then one could consider the sum $a + b$. In this way, we get another common example.

$$\text{SUMS} = \{a\#b\#c : a, b, c \in \{0,1\}^+, \text{ and } a + b = c\} \subseteq \{0, 1, \#\}^*$$

That is, # is merely a separator and a string in SUMS looks like three binary numbers separated by # such that the third equals the sum of the first two.

One can represent graph problems this way. There are many ways to encode a graph, one is using the *adjacency matrix representation*. Say that $A \in \{0,1\}^*$ is an encoding of an undirected graph if $|A|$ is a perfect square, say n^2, and, viewing A as an $n \times n$ matrix that is just flattened out, $A_{i,j} = A_{j,i}$ for $1 \leq i, j \leq n$ and $A_{i,i} = 0$ for $1 \leq i \leq n$. It is possible to design a TM that recognizes this language which we can call UNDIR-GRAPH.

But now we have the ability to encoding many more interesting problems!

$$\text{CONNECTED} = \{A \in \text{UNDIR-GRAPHS} : A \text{ is the adjacency matrix of a connected graph.}\}$$

$$\text{HAMILTONIAN} = \{A \in \text{UNDIR-GRAPHS} : A \text{ is the adjacency matrix of a graph with a Hamiltonian cycle.}\}$$

Each of the languages listed above is decidable. You can probably think of how to write a computer program to decide such languages. We will soon discuss the connection between Turing machines and desktop computer programs.

1.2.4 Running Time of a Turing machine

The running time of a TM M on input x is the number of steps taken until M halts when given input x. If M does not halt on x, this is ∞.

Definition 4 *Let $T : \mathbb{N} \to \mathbb{N}$. We say that a TM M has running time $T(n)$ if the number of steps taken by M on any input $x \in \{0,1\}^*$ is at most $T(|x|)$.*

This definition only applies to Turing machines that are guaranteed to halt on any input. One could generalize it, but we will only discuss running times when considering halting TMs. Also note the slightly incorrect notation, really we should say the TM M has running time T instead of $T(n)$ but this is a common abuse. It is much like we are used to saying things like $4n^2 = O(n^2)$ when, really, $O(n^2)$ is the *set* of all functions asymptotically bounded by n^2.

Note that if, say, a Turing machine has running time $n+1$ then it also has running time n^2+1. This definition merely places upper bound on the number of steps taken by a TM. We frequently say things like a Turing machine M has running time $O(n^2)$ if there is some function $f \in O(n^2)$ such that M has running time $f(n)$.

1.2.5 Variants on the Definition of Turing Machines

Why this precise model of computation? As stated earlier, we like it because it is very simple to present mathematically. But there are many different variants one could consider.

Multiple Tapes
We could allow a TM to have k tapes instead of just one. This is quite useful in many situations, one tape could be used to store the input, one for scratch space, one for output if we care about the contents of the tape when the TM halts. An input x would initially be written on the first tape and the remaining tapes would initially be empty.

The snapshot would maintain all k head positions and the transition function would take the form

$$(Q - \{q_{\text{ACCEPT}}, q_{\text{REJECT}}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k.$$

Going back to the palindrome problem, the single-tape TM above is runs in $O(n^2)$-time whereas one can easily design a 2-tape TM with running time $O(n)$.

- Put a copy of the input onto the second tape, in reverse order.
- Do a linear scan from start to end of both strings, rejecting if the heads ever look at different characters.

But this quadratic difference is essentially the worst case. We reason about this at a very high level, relying on your "programmer's intuition" that you precisely could implement the ideas with some low-level work.

Claim 1 *If a language $\mathcal{L}$ can be decided on a k-tape TM $M = (\Gamma, Q, \delta)$ with running time $T(n)$, then it can be decided on a single-tape TM M' with running time $c \cdot k \cdot (n^2 + T(n)^2)$ for some absolute constant c.*

We could just say $O(n^2 + T(n)^2)$ because k is a constant that is independent of the size of the input, but it was written this way to reveal the dependence of the running time of M' on the number of tapes of M. Usually, $T(n) \geq n$ because most interesting problems require the entire input to at least be examined. But in some cases it could be that $T(n) < n$ so we cannot just absorb n^2 into $T(n)^2$ by increasing the constant c.

Proof.[sketch] We view the k tapes of M being numbered from 0 to $k-1$. Interleave the k tapes into a single tape. Namely, let cell i of tape $0 \le j \le k-1$ for M correspond to cell $k \cdot i + j$ of the tape for the single-tape TM M' we are designing to simulate M.

For each symbol a of Γ for M, we have both a and $\widehat{a}$ being symbols in M'. Here, a corresponds to the original symbol and $\hat{a}$ means one of the heads in the simulation of M should be looking at a.

When given input $x \in \Gamma$ with $|x| = n$, M' will first move the symbols in x to positions $0, k, 2k, 3k, \ldots, (n-1)k$ in $O(k \cdot n^2)$ time so it looks like it is on the first tape of M'. Cells 0, 1, 2, ..., $k-1$ will then get a "caret" $\widehat{\ }$ added to their symbol to indicate the heads of M' start here. This can be implemented to run in $O(k \cdot n^2)$ steps.

A single step of M will then be simulated in $O(k \cdot T(n))$ time as follows. The head of M' starts from the left of the non-blank cells, scans the cells until all $\widehat{\ }$-symbols are read and stores the symbols read by these k heads of M in the state of M'. Then M' determines the transition that M would have taken, and does another sweep to update the $\widehat{\ }$-symbols by updating the underlying symbol according to the transition of M'. Overall, simulating a single step of M takes $O(k \cdot T(n)$ time because the heads of M will always be at positions with indices in the range $[-T(n), T(n)]$ (they can change by at most one position per step of M).

Note, this may require us to maintain a special symbol $\triangle$ indicating the left-most position the head of M' has ever reached so that we can easily "reset" the head to the left of the non-blank contents of the tape before each sweep. ■

Simple Alphabets
Our definition admits alphabets of arbitrary large (but still finite) size. But in computing science we often use just bits 0 and 1 when programming. The following shows we can restrict the alphabet to be binary (apart from the blanks) without much loss of generality.

Let $L \subseteq \Gamma'^*$ be a language over an arbitrary (nonempty) alphabet Γ' with $|\Gamma'| \ge 2$. Let $\gamma = \lceil \log_2(|\Gamma'^*|) \rceil$. We can encode each $s \in \Gamma'$ with a γ-bit string. For example, if $\Gamma = \{a, b, c, d, e\}$ then we can encode as:

$$a \to 000, \quad b \to 001, \quad c \to 010, \quad d \to 011, \quad e \to 100.$$

Any string $x \in \Gamma'^*$ can then be encoded in binary by simply concatenating the binary representations of the symbols of x. Finally, $\mathcal{L}$ itself can be encoded in binary by letting $\text{bin}(\mathcal{L}) \subseteq \{0,1\}^*$ be the language consisting of the binary encoding of all strings $x \in \mathcal{L}$.

Claim 2 *If $\mathcal{L} \subseteq \Gamma'^*$ can be decided on a TM M with alphabet $\Gamma' \cup \{\square\}$ with running time $T(n)$, then $\text{bin}(\mathcal{L})$ can be decide on a TM M with alphabet $\{0, 1, \square\}$ with running time $c \cdot \log|\Gamma'| \cdot T(n)$ for some absolute constant c.*

Proof.[sketch] To "simulate" a step of M, M' will scan γ bits and store these in the state to determine the original symbol that is encoded. Then it will effect the transition of M by overwriting these γ bits with the encoding of the symbol that would have been written by M. ■

1.2.6 Turing Machines that Compute Functions

So far, we have talked about how Turing machines can decide languages. But we can also use them to compute functions. We allow a TM to have multiple tapes in this definition.

Definition 5 *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. We say that a TM M* **computes** *f in $T(n)$ time if M halts in at most $T(|x|)$ steps for any input x and the last tape of M contains $f(x)$ starting at cell 0 (and all other cells are blank).*

1.2.7 Oblivious Turing machines

Definition 6 *A multi-tape Turing machine M is* **oblivious** *if for any two inputs $x, y \in \{0,1\}^*$ where $|x| = |y|$, the position of each head of M is the same in the i'th step of computing x as it is in the i'th step of computing y for each step i.*

That is, M takes the exact same number of steps for inputs of the same size and the position of the head in the i'th step of computation is only a function of the *length* of the input. Considering oblivious TMs will simplify some future discussions.

Claim 3 *If f can be computed in time $T(n)$ by a TM M, then it can be computed in time $O(T(n)^2)$ by an oblivious TM M'.*

Proof.[sketch] The idea is to use a $\hat{}$-marker for the symbols to indicate the position of the original heads of M. The heads of the oblivious TM will just sweep back and forth across the entire input in big sweeps. One sweep is used to determine all symbols with a $\hat{}$ marker and these will be store in the state of M'. Then M' will simulate the transition of M in two more sweeps: one forward and one reverse. The forward sweep will move all heads of M that should move right (by "moving" the $\hat{}$-marker from one cell to another), the reverse sweep will move all heads of M that should move left.

We can also maintain "endpoint markers" $\triangle$ that will appear to the left and right of the non-blank parts of the tape. After M' simulates a step of M, it moves these markers "outward" one step (the left markers go left one step, the right markers go right one step). ■

1.3 Random Access Machines

One can imagine a model of computing where the memory contents can be accessed by indexing, not sequentially. Such memory is often called **random access memory** (RAM). But we can simulate this with a Turing machine with only polynomial overhead in the running time.

Imagine that the indices into the RAM are arranged sequentially on a **memory tape** and that we have another tape to help with indexing. To access the contents of a specific cell i of RAM, the TM can write this index on the indexing tape, reset the memory tape head to the start, and then increase the memory tape head by 1 while decreasing the index on the index tape by 1 until it reaches 0.

Roughly speaking, if the program running on the computer with RAM takes $T(n)$ time and the maximum index into RAM it addresses is $S(n)$ over all inputs of length n then we can simulate this program on a TM with running time $\text{poly}(T(n), S(n))$. It is reasonable to imagine that $S(n) = \text{poly}(T(n))$ (one can actually ensure this on a computer by implementing some memory management system) so in fact we can simulate any program on a machine with RAM on a Turing machine with only polynomial overhead in the running time.

1.4 The Universal Turing machine

We can even represent a Turing machine as a string in $\{0,1\}^*$. This is really just like how a program on a computer is a binary executable file. We will make some assumptions:

- Every string in $\{0,1\}^*$ represents some Turing machine. If a string is not a valid *canonical* encoding of Turing Machine then we interpret it as a trivial Turing machine that halts and rejects right away.

- Every Turing Machine is represented by infinitely many strings. This can be done by simply padding the canonical encoding of a Turing Machine with arbitrarily many 0s. The real-world analog is that we can add comments to a program. It changes the source code file but represents the same program.

For a string $\alpha \in \{0,1\}^*$ we let M_α denote the Turing Machine encoded by α.

We can also design a **universal Turing machine** that takes, as input, a pair of strings in $\{0,1\}^*$ where the first is some encoding of a Turing Machine M with alphabet $\Gamma = \{0,1,\square\}$ and the second is some possible input x to M. This universal Turing machine will simulate the computation of M in input x. In fact, one can think of our desktop computers as universal computation devices because we can load any program we want and run it on our computer. A universal machine can also be written in software, a PYTHON emulator is software that can run any PYTHON program.

Note: Here we will use the notation $M(x,y)$ for two different strings x, y. This means M is provided, as input, two parameters. There are multiple ways to encode this, we do not worry about those details.

Theorem 1 *There is a Turing machine $\mathcal{U}$ such that for any $x, \alpha \in \{0,1\}^*$ we have $\mathcal{U}(x,\alpha) = M_\alpha(x)$. Furthermore, if M_α has running time $T(n)$ then $\mathcal{U}$ halts in $O(T(|x|)^2)$ steps.*

Proof.[sketch] First have $\mathcal{U}$ transform M_α into an equivalent TM M'_α that uses only a single tape. This can be done efficiently because the reduction from multiple tapes to a single tape is algorithmic.

The machine $\mathcal{U}$ has a tape dedicated to holding the contents of the tape used by M'_α and a tape dedicated to storing the state of M'_α. Then $\mathcal{U}$ can simulate the steps of M'_α by reading the contents of the the tape for M'_α, reading the current state of M'_α, looking up how the transition function for M'_α to determine how to update the tape for M'_α, how to move the head for this tape, and how to update the state for M'_α.

With care, this can all be done so the number of steps in the computation of $\mathcal{U}(x,\alpha)$ is quadratic in the number of steps in the computation $M_\alpha(x)$. ■

In fact, Chapter 1.7 of the text gives a way to perform this simulation in time $O(T(n) \cdot \log T(n))$ where the suppressed constant depends only on the number of tapes and states of M'_α.

1.5 The Church-Turing Thesis

Hopefully by now you are convinced that Turing machines are versatile computing "devices" that can simulate the sort of computation we expect from our computers with only polynomial overhead in the running time.

The Church-Turing Thesis, put forward in the 1930s, proposes that Turing machines can simulate any algorithmic process that is executable on a physical device. It is not a formal statement, which is why it is called a "thesis[1]"

The Strong Church-Turing Thesis further proposes that Turing machines can simulate any algorithmic process with *polynomial overhead*. Our earlier discussion shows how different models of Turing machines can be simulated by a single-tape Turing machine with polynomial overhead. Furthermore, our desktop computers can also be simulated by a Turing machine with polynomial overhead.

Some models of computing challenge this strong-form of the thesis. Perhaps the strongest contender comes from quantum computing. It seems that some quantum algorithms cannot be *efficiently* simulated by a Turing machine. But we do not yet have an explicit refutation of this thesis because we do not have scalable quantum computing devices.

[1] A statement or theory that is put forward as a premise to be maintained or proved.