University of California, Los Angeles
CS 281 Computability and Complexity

*Instructor:* Alexander Sherstov
*Scribe:* Nakul Khambhati
*Date:* January 18, 2023

**LECTURE**
**3**

# Time Bounds and Robustness

In the previous lecture, we completed a proof that demonstrates that constant depth circuits cannot compute parity. We also reviewed Turing Machines (TMs) and defined them formally. Finally, we discussed an $O(n)$ algorithm for computing `PALINDROME`.

In this lecture, we will explore efficiency by defining time bounded TMs. We argue that TMs are robust to most minor modifications. However, replacing a $k$-tape TM with a single tape TM results in a quadratic slowdown for some languages. Finally, we revisit `PALINDROME` and introduce crossing sequences to exhibit a lower bound on the slowdown caused by this restriction.

## 3.1 Efficiency

We will now narrow our focus to TMs that efficiently solve problems. Before we can define efficiency, we need to introduce some notation. Consider a function $f : \{0,1\}^* \to \{0,1\}^*$. Intuitively, this function $f$ is solving some problem where the input and output are both expressed in binary. Technically, however, this can be any arbitrary mapping of binary strings. Also, let $T : \mathbb{N} \to \mathbb{N}$ be a natural-valued function. This function represents the time-bound that we impose upon our Turing Machine.

Definition 3.1. A TM $M$ computes a function $f$ in time $T(n)$ if when started with arbitrary input $x \in \{0,1\}^*$, $M$ halts within $T(|x|)$ steps with $f(x)$ written on the output tape. Here, $|x|$ denotes the bit-length of $x$.

Example 3.2. The TM $M$ that we constructed last lecture to solve `PALINDROME` runs in time $O(n)$. Here, and several times later, $n$ is used as a placeholder for $|x|$.

Let's restrict our attention to functions that output 0 or 1 instead of arbitrarily long strings. These functions are called languages or decision problems.

Definition 3.3. A language is a subset $L \subset \{0,1\}^*$. $M$ decides a language $L$ if $M$ computes the associated characteristic function $f_L(x) : \{0,1\}^* \to \{0,1\}$.

$$f_L(x) = \left\{ \begin{array}{cc} 1, & \text{if } x \in L \\ 0, & \text{otherwise} \end{array} \right\} \tag{3.1}$$

This equips us with the tools necessary to introduce certain classes of languages.

Definition 3.4. $\mathbf{DTIME}(T(n))$ is the family of all languages $L$ that are decidable within time $cT(n)$ for some constant $c > 0$.

Definition 3.5. The class $\mathbf{P} = \bigcup_{k=1}^{\infty} \mathbf{DTIME}(n^k)$ contains all languages $L$ that can be solved by some TM $M$ running in polynomial time. These are considered to be the languages that can be solved *efficiently*.

Remark 3.6. Note that while the class $\mathbf{DTIME}(T(n))$ is defined in terms of TMs, the objects being classified are languages and not TMs.

It is worth discussing the appearance of the constant $c$ in Definition 3.4. According to this definition, a language $L$, which is decidable by $M$ that runs in $100n$, lies in the same class as a language $L'$ that some $M'$ decides in $0.01n$. Is this because constant factors don't matter in complexity theory? Or is there a deeper reason? As it turns out, both are true. For starters, the exact specifications of a machine may increase the runtime of an algorithm by a constant factor. However, the same algorithm run on different types of classical[1] machines will belong to the same class $\mathbf{DTIME}(T(n))$. At a deeper level, the linear speedup theorem [2] states that given any real $c > 0$ and any $k$-tape Turing machine solving a problem in time $T(n)$, there is another $k$-tape machine that solves the same problem in time at most $T(n)/c+2n+3$, where $k > 1$. This brings us to our next topic of discussion − the robustness of Turing Machines.

## 3.2 Robustness of TMs

As it turns out, our model of a TM is robust in the sense that the theory is largely unaffected by minor modifications. Since most of the details in our definition of a TM are arbitrary, we want a model that doesn't change significantly with these parameters. Some variations we can make to our TM are restricting its alphabet to the bare minimum, allowing the tape to be infinitely long in both directions, and insisting that our TM uses a single tape.

1. **Restricting the alphabet**: In our definition of a TM, we allow the alphabet $\Gamma$ to be arbitrarily long. However, we can restrict ourselves to the alphabet $\Gamma = \{0, 1, \triangleright, \Box\}$ and increase the running time by only a constant factor.

   Theorem 3.7. *If $f : \{0,1\}^*$ is computable in time $T(n)$ with alphabet $\Gamma$, then $f$ is computable in time $O(T(n)\log\Gamma) = O(T(n))$ with alphabet $\{0, 1, \triangleright, \Box\}$.*

   Matan provides the intuition here by explaining that we can create a mapping from the original alphabet to the set $\{0,1\}^*$.

   Then, we replace every instance of a symbol from the original alphabet with its corresponding binary representation and provide the TM with instructions to interpret it accordingly. Assuming that our original alphabet $\Gamma$ had length $|\Gamma|$, we need $\log\Gamma$ bits to uniquely identify each symbol. Therefore, this results in a constant (with respect

[1]Quantum computers can increase the speed of certain algorithms by more than constant factor but these have not yet been physically realized.

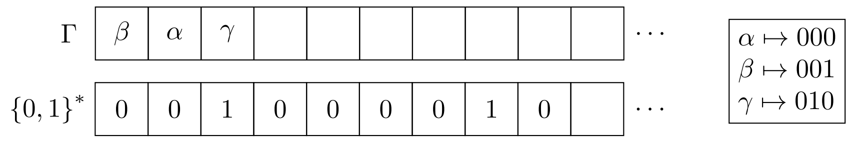

FIGURE 3.1: Translating from Γ to $\{0,1\}^*$

to input length $n$) increase in the running time. Note that there will be some other constant factors involved in this slowdown as well. For example, the modified TM will need additional states to store the symbols being read.

2. **Bidirectional tape**: By definition, we allow the TM to extend infinitely to the right. We can instead allow it to extend infinitely in both directions. In practice, this is very useful while solving certain problems. Surprisingly, though, this does not make the TM any more powerful.

THEOREM 3.8. *If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ with bidirectional tapes, then $f$ is computable in time $O(T(n))$ using standard tapes.*

*Proof.* Assume that $f$ is computable by a bidirectional TM $M$. Let's denote $M$'s alphabet as Γ. We will construct a TM $\tilde{M}$ with alphabet $\Gamma^2$. Now, in each cell, $\tilde{M}$ can represent a pair $(\sigma, \tau)$ of symbols in Γ. We construct $\tilde{M}$ by *folding* $M$'s tapes [1] at some point from left to right. This reduces the tapes from bidirectional to standard.

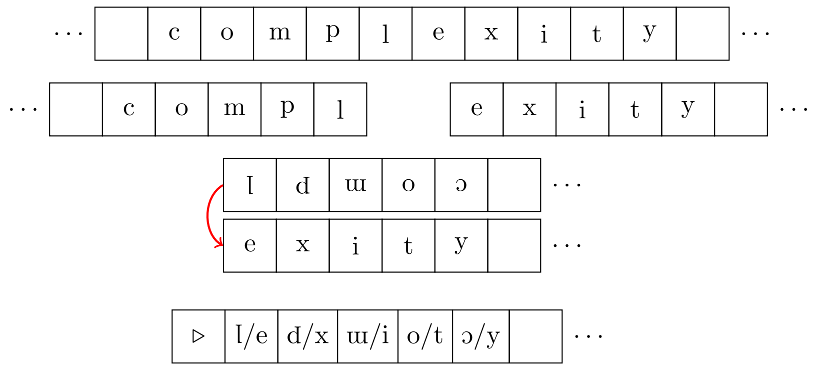

FIGURE 3.2: Reducing a bidirectional tape to a standard tape

Now every cell holds two values: its original value before the fold and the symbol in the cell with which it overlaps after the fold. $\tilde{M}$ also has an additional state for each tape, called FLIP which takes on two values. As the head moves across the tape, this state instructs it to only read one of these symbols (say the first one). This corresponds to the head moving to the right in the original TM $M$. When any of $M$'s heads crosses over to the left of its starting position, $\tilde{M}$ changes its state and starts reading the second symbol instead. In this way, changing the state FLIP corresponds to one of $M$'s heads moving across the edge of the fold in the original tape. The additional states and extended alphabet only produce a constant increase in runtime. ∎

Remark 3.9. A corollary of this theorem is that we can determine whether a certain language $L$ is in **DTIME**$(T(n))$ by constructing a bidirectional tape to solve the problem in $T(n)$. Then, by the theorem, the language is computable in $O(T(n))$ with standard tapes.

3. **Single tape**: Now we consider the case where we allow our TM $M$ to have only one tape. This means that $M$ uses the same tape to read input, perform intermediate calculations and write output. This case is particularly interesting because this restriction results in a quadratic slowdown.

Theorem 3.10. *If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ on a $k$-tape TM then $f$ is computable in time $O(T(n)^2)$ on a single tape TM.*

*Proof.* Let $M$ be a $k$-tape TM that computes $f$ in time $T(n)$. As illustrated, it starts with $k$ tapes and $k$ heads. At every computational step the heads scan the $k$ cells directly underneath and use that information, along with the $M$'s states, to decide which symbols to write in the cells and which direction to move in afterwards. This parallelism (for $k \geq 2$) comes in very handy. For example, in `PALINDROME`, two heads move simultaneously, working together to test equality between the $i^{th}$ and $(n-i)^{th}$ symbols. We wish to simulate this behaviour with only 1 tape.

One simple approach here, which proves to be optimal, is to use a richer alphabet $\tilde{\Gamma} = (\Gamma \times \{0,1\})^k$ in $\tilde{M}$ to keep track of the position of the $k$ heads of $M$. This way, there is a dedicated extended alphabet $\tilde{\Gamma}_i$ for the $i^{th}$ tape. Also, each $\tilde{\Gamma}_i$ is twice as large as the original $\Gamma$ to allow us to mark the location of that tape's head. For this purpose, think of each $\tilde{\Gamma}_i$ as two copies of $\Gamma$ – one with regular symbols $\sigma$ and one with circled symbols $\textcircled{\sigma}$. A circled symbol will indicate that in $M$ a tape's head is above that cell. Now, as illustrated below, we can pretend like $\tilde{M}$ has one giant head that simultaneously scans through all $k$ tapes together from left to right till it reaches the last symbol.

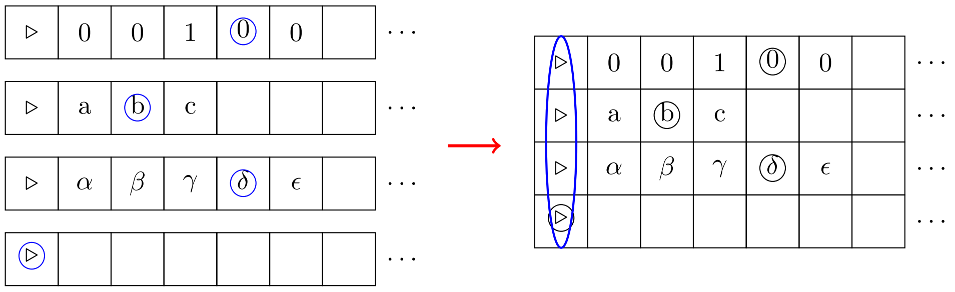

Figure 3.3: Restricting the TM to a single tape

While doing this scan, it memorizes the circled symbols by updating its state registers. It cannot memorize the location of each head as it only has fixed memory. Therefore, it only stores the symbols under each head. It then uses the transition function $\delta$ of $M$ to decide which changes to make. Finally, it scans back from right to left and updates the required cells. An entire scan is required before any updates can be made since $\delta$ is a function of all cells being read by $M$'s head and only an entire scan can ensure that all circled symbols have been read. As a result, every computational step of $M$ takes $O(n)$ time to run on $\tilde{M}$ which causes a quadratic slowdown to $O(T(n)^2)$ total steps. ∎

With the previous modifications, we saw constant factors appear and we didn't mind them. However, the quadratic slowdown in the single tape TM is quite significant. This begs the question: Is this necessary? Is it just that we have not yet found a better solution? Or that it's in the nature of things for a single tape TM to be inheritently weaker than a multi-tape TM? Surprisingly, the latter is true. We will devote the next section to proving this result.

Remark 3.11. This is a good opportunity for us to get into the habit of proving lower bounds, a recurring theme in complexity theory. One of the many thrills of complexity theory is proving such impossibility theorems. Since we are already equipped with the tools to prove this result, it is better to prove it now than to delay it.

## 3.3 Deciding palindrome on a single tape TM

The lower bound we are trying to prove is that there is a language $L$ such that it can be decided within $O(T(n))$ time on a $k$-tape Turing Machine where $k \geq 2$ but takes at least $O(T(n)^2)$ times on a single tape TM. We will exhibit this lower bound on `PALINDROME`. We have seen that with 2-tapes we can solve `PALINDROME` in $O(n)$ time. So, the theorem we need to prove is as follows.

Theorem 3.12. *Deciding* `PALINDROME` *requires* $\Omega(n^2)$ *time on a single tape TM.*

We have not yet discussed many sophisticated techniques for proving lower bounds. Still, this result is rather surprising and certainly non-trivial. Hence, proving it requires taking a fundamental approach. In the process, we will find ourselves pondering over the very nature of information exchange.

To aid this process, we will introduce some notiation. But first, it's important to develop some intuition regarding why this result should be true.

By restricting ourselves to one tape, we only have one *source* of information i.e. the one head that moves across the tape. We are asking the TM to verify that the string looks the same when read forwards or backwards. Most importantly, we are asking it to do this without allowing it to *copy* information from one place to another.[2] As a result, any time the TM needs to make a comparison between two characters, the head needs to *cross* over to the other side. In general, each crossing can take linear time and, since the comparison needs to be made about $n/2$ times, the total runtime becomes quadratic. It seems as though asking for any better is unreasonable. But how can we know this for sure?

We recognized that the head needs to cross over to the second half of the word every time it makes a comparison. So, one approach would be to show that in sub-quadratic time, there are simply not enough crossings that can be made to accurately distinguish all inputs $x \in \{0,1\}^n$. We now formalize this idea.

[2]At least to a place that can be conveniently accessed regularly, such as another tape.

Definition 3.13 (crossing sequence). A crossing sequence $C_i(x)$ is the sequence of states $(q_1, q_2 \dots, q_k)$ of $M$, run on input $x$, where $q_k$ is the state of $M$ immediately after crossing the $i^{th}$ border of the tape (illustrated in Figure 3.4) for the $k^{th}$ time.

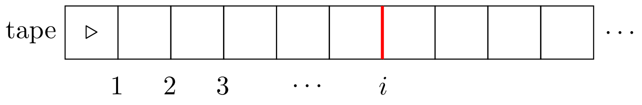

Figure 3.4: The $i^{th}$ border of a tape

Remark 3.14. If the head of a TM never crosses $i$ for a certain input $x \in \{0,1\}^n$ then $C_i(x)$ is empty. A crossing sequence is analagous to a customs officer that makes a certain observation about you (say the color of your t-shirt) and notes it down every time you cross the border.

Next, we introduce and prove a lemma that provides the core of the theorem's proof.

Lemma 3.15. *Let $M$ be a TM that decides some language $L$. Let $x, y$ be inputs such that $M(x) = M(y)$. Assume that we partition the input $x$ at some point $i$ calling the left half $x'$ and the right half $x''$ such that $x = x'x''$. We do the same to $y$ at $j$ such that $y = y'y''$.*

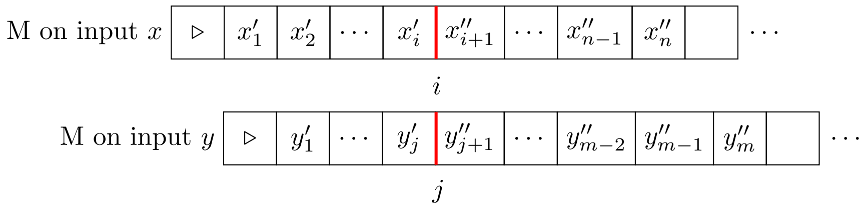

Figure 3.5: Partitioning $x$ at $i$ and $y$ at $j$

*Further, assume that $C_i(x) = C_j(y)$. Then $M(x'y'') = M(x) = M(y)$.*

*Proof.* The proof is best understood by visualizing running $M$ on the hybrid input $x'y''$. We will compare this to $M$ running on $x$ and $y$ illustrated in Figure 3.6. Let's simulate $M$ on $x'y''$. Clearly, until the $i^{th}$ border is crossed for the first time $M(x'y'')$ runs in the exact same way as $M(x)$. Once $i$ is crossed, $M(x'y'')$ runs like $M(y)$ until $i$ is crossed again. In this way, we can partition the computation period on $x'y''$ into multiple intervals, each separated by a crossing of $i$. These intervals can be classified into two types: one, colored in green, where the head stays to the left of $i$ and one, colored in blue, where the head stays to the right of $i$. Each left (right) interval corresponds to some sequence of computations on $x$ ($y$) where the head stays in $x'$ ($y''$).

Further, we are given that $C_i(x) = C_j(y)$. This means that $\forall k \in \{1, 2, \dots, |C_i(x)|\}$ the state of $M$ running on $x$ when the head crosses the $i^{th}$ border for the $k^{th}$ time is the same as the state of $M$ running on $y$ when the head crosses the $j^{th}$ border for the $k^{th}$ time. Matthew summarizes this by observing that when $M$ runs on $x'y''$, it thinks that it is running on either $x$ or $y$ (depending on the computational interval) and preserves its state when crossing

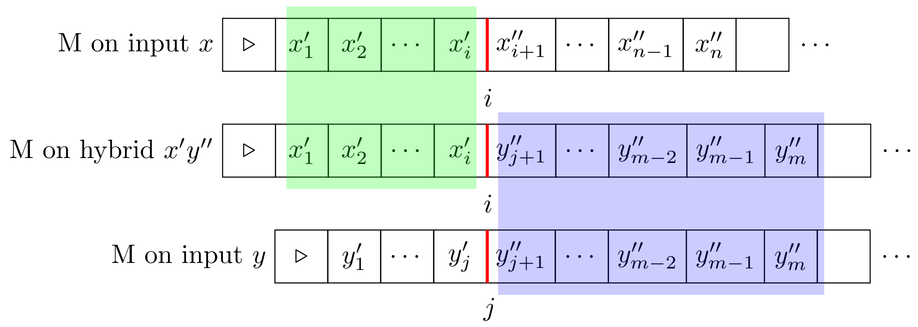

FIGURE 3.6: Comparing $M$ on hybrid input with $M$ on $x$ and $y$

the border between the two. As a result, $M$ decides on the hybrid input the same way it decides on $x$ and $y$. ∎

Now we will prove the theorem. Informally, will we show that for a certain class of inputs, there must be many distinct crossing sequences. As a result, there must be some input $x$ of this type that has long (in the order of $O(n)$) crossing sequences at $n$ different borders. Therefore, the $TM$ must take at least $O(n^2)$ time to compute it.

*Proof of theorem.* We have shown that if $M$ decides the same for two inputs $x$ and $y$ and they have the same crossing sequences $C_i(x)$ and $C_j(y)$ for any $i \in \{1, 2, \ldots, |x|\}, j \in \{1, 2, \ldots, |y|\}$, then we can create a hybrid input $x'y''$ for which $M(x'y'') = M(x) = M(y)$.

We use the contrapositive of this lemma by fixing two inputs of the form $x0^nx^R$ and $y0^ny^R$ where $x \neq y$. Here $x^R$ denotes $x$ reversed i.e. read from right to left.

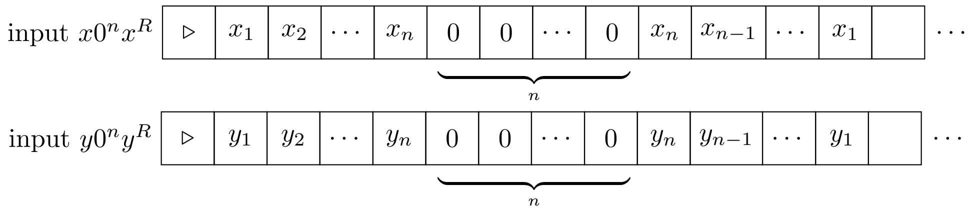

FIGURE 3.7: Constructing inputs that are forced to be palidromes

Clearly, both are palindromes so $M(x) = M(y) = 1$ but for any $i, j \in \{n+1, n+2, \ldots, 2n\}$ we have $C_i(x0^nx^R) \neq C_j(y0^ny^R)$ since the hybrids corresponding to these indices cannot be palidromes (by the assumption $x \neq y$).

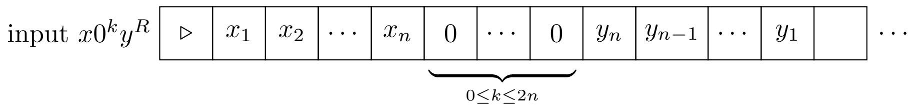

FIGURE 3.8: Example of $2n$ different hybrids that are not palidromes

Therefore, if we restrict ourselves to inputs of the form $x0^nx^R, x \in \{0,1\}^n$ we observe that for for all indices $i \in \{n+1, n+2, \ldots, 2n\}$, as we range over all possible $x \in \{0,1\}^n$, the crossing sequences $C_i(x0^nx^R)$ must be distinct.

Remark 3.16. The previous sentence had multiple quantifiers so it is worth re-reading it to appreciate just how many distinct crossing sequences we have found. For each $i$, we have found $2^n$ distinct crossing sequences, each corresponding to some $x \in \{0,1\}^n$.

To accomodate all these distinct crossing sequences, we inevitably need some sequences that are long. Specifically, if the longest sequence has length $l$ and the number of states that the TM can represent is $|Q|$, then the total number of distinct crossing sequences is $\sum\limits_{k=1}^{l} |Q|^k \leq (|Q|+1)^l$ which is relatively small in size unless $l$ is large. So for each $i$, if we have to accomodate $2^n$ distinct sequences, then by the pigeonhole principle we need at least one sequence with large length $l$ where $(|Q|+1)^l \geq 2^n$. Writing this with quantifiers and rearranging terms

$$\forall i \in \{n+1, n+2, \ldots, 2n\}\ \exists x \in \{0,1\}^n : |C_i(x0^nx^R)| \geq \frac{n}{\log(|Q|+1)} = \Omega(n) \tag{3.2}$$

We now simply switch the order of the quantifiers.

$$\exists x \in \{0,1\}^n\ \forall i \in \{n+1, n+2, \ldots, 2n\} : |C_i(x0^nx^R)| \geq \frac{n}{\log(|Q|+1)} = \Omega(n) \tag{3.3}$$

In words, there is some input $x0^nx^R$ such that for $n$ different indices $i$, the $i^{th}$ border is crossed at least $n$ times. In total, there are $n$ different borders each of which is crossed at least $n$ times. So, $M$ takes at least $n^2$ steps to halt on the input $x0^nx^R$. This concludes our proof that the running time of $M$ is $\Omega(n^2)$. ∎

Remark 3.17. The hybrid approach used in this proof is common in theoretical computer science. It is used when there are two interesting features exhibited in two different cases. We then try to construct a hybrid scenario where both phenomena can be seen together. Matan adds that this approach is commonly used in proving lower bounds in cryptography.

Problem 3.18. Prove that computing $f : x \mapsto xx$ requires $\Omega(n^2)$ time on a single tape TM.

Problem 3.19. We are given a turing machine $M$ that decides some language $L$ in $O(n)$. What speed-up can be achieved with a $k$-tape machine?

Problem 3.20. Prove that if $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ with $k$ tapes, then it can computed in time $O(T(n)\log T(n))$ using 2 tapes.

Remark 3.21. In fact, a logarithmic slowdown is so insigniciant that we use the notation $\tilde{O}(T(n)) = O(T(n)\log T(n))$.

Problem 3.22. From the Problem 3.20, a $k$-tape TM can be replaced with a 2-tape TM without a significant slowdown. What additional *functionality* do we get by using two tapes over one?

## References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Pearson Addison-Wesley, 3rd edition, 2006.