University of California, Los Angeles
CS 281 Computability and Complexity

*Instructor:* Alexander Sherstov
*Scribe:* Illia Shkirko
*Date:* January 22, 2024

**LECTURE**

# 4

# Universal Computation and Intro to Nondeterminism

In this lecture, we are going to finish our discussion on deterministic computation that we started last time. Today, we will further expand on building a universal computer that operates on 2-tapes. We will build intuition by discussing the challenge problem, which was assigned last week, and step by step reach the desired result. Finally, we will start talking about nondeterminism: we will discuss two different ways of looking at it and explain why they are equivalent. We will finish by providing a solution to one of the earlier challenge problems.

## 4.1 Challenge problem intuition

Last time, we finished our lecture with a thought-provoking challenge problem that is going to help us understand the reasoning behind time complexity analysis for a 2-tape TM.

Problem 4.1. Containers are delivered by ships to a port city. You work at the port's receiving dock and are tasked with handling the containers as they arrive (Figure 4.1). You are strong enough to pick up and carry arbitrarily tall stacks of containers in your arms, but you are required to place each container flat on the dock before the next ship arrives. Suppose that the containers are cubes with side length equal to the length of your step. What is the minimum number of steps needed to handle $n$ containers, where $n$ is arbitrary but known in advance? Prove the best lower and upper bounds that you can. What if $n$ is not known to start with?

To clarify possible confusion, we don't know in advance how many containers each ship might carry. We will also consider the case of an unknown $n$ as the only difference it makes is whether or not we need to go back to the front of the dock after the last ship arrives. Additionally, we are not worried about the wait time between ships as there is no notion of "time" in this problem. All we ultimately care about is a sequence of events and the number of steps we have to make.

While it seems like a "toy problem", in fact, it gives a basis for the Time Hierarchy Theorem, which is one of the most powerful results in Complexity Theory. We will begin

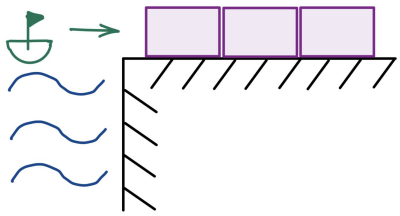

FIGURE 4.1: Port problem sketch.

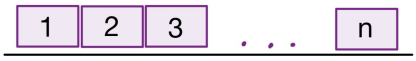


FIGURE 4.2: Placing all containers consecutively.

with some intuition for the bounds. Suppose all containers come together on one ship. Then we can simply place them all on the dock and go back in $2n$ steps. Alternatively, we can say that as we would like to place *every* container on the dock, at least $O(n)$ steps have to be taken. Now suppose, every ship carries just one container, and we take a simple strategy of placing containers one after another along the dock (Figure 4.2). Total number of steps we take is $1 \cdot 2 + 2 \cdot +3 \cdot 2 + \cdots + n \cdot 2 = n(n+1) = O(n^2)$. As one container per ship is a seemingly worst case, $O(n^2)$ may seem like an appropriate upper bound.

Turns out both predictions are wrong. Let's see why.

We propose a different idea of processing the containers. We split the dock in lengths that are increasing exponentially, i.e. section $A_0$ will have length 1, section $A_1$ - length 2, $A_2$ - length 4, and so on. In general, zone $A_i$ will have a length $2^i$ (Figure 4.3). On a high level, our strategy will work like a counter in binary. Our strategy would be to place the incoming package in the closest zone that has room in it. When a zone fills up, we move all packages from that zone to the next empty one. (Remember how we increment numbers in binary: all lower bits become 1 one by one, and once they all are 1's, we reset them all to 0's and make the next most significant bit 1.) Let's see how this strategy works on a small example.

EXAMPLE 4.2. Initially, all zones are empty. When the first package arrives, we place it in zone $A_0$. When the second package arrives, $A_0$ is full, so the new nearest non-full zone is zone $A_1$. We pick up the package in zone $A_0$ and place both new and old packages in zone

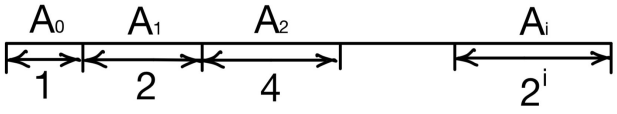


FIGURE 4.3: Separate dock into zones.

| $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | ... | $A_i$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | ... | 0 |
| 0 | 2 | 0 | 0 | 0 | 0 | ... | 0 |
| 1 | 2 | 0 | 0 | 0 | 0 | ... | 0 |
| 0 | 0 | 4 | 0 | 0 | 0 | ... | 0 |
| 1 | 0 | 4 | 0 | 0 | 0 | ... | 0 |
| 0 | 2 | 4 | 0 | 0 | 0 | ... | 0 |
| 1 | 2 | 4 | 0 | 0 | 0 | ... | 0 |
| 0 | 0 | 0 | 8 | 0 | 0 | ... | 0 |

FIGURE 4.4: Evolution of the number of containers in each zone as containers arrive.

$A_1$. As the third package arrives, the nearest non-full zone now is $A_0$, so we place it there. With the next package arriving, both $A_0$ and $A_1$ are full, so we pick up all $3 + 1$ packages we currently have and move them to $A_2$. Now we wait for $A_0$ to fill up, move contents to $A_1$, then wait for $A_0$ to fill up again, and finally move everything to $A_3$ (Figure 4.4).

CLAIM 4.3. *At any given time $A_i$ is either empty or fully occupied.*

Indeed, as we move packages from first $i-1$ zones to zone $i$ (plus newly arrived package), we move $(2^0 + 2^1 + 2^2 + \cdots + 2^{i-1}) + 1 = 2^i$ packages in total, so $A_i$ is now full and $A_j$ for $0 \leq j < i$ are empty.

Let's summarize our strategy. When a container arrives:

- Look for closest zone that has room in it: $A_i$.
- Then $A_0$, $A_1$, ..., $A_{i-1}$ are full and $A_i$ is empty.
- Transfer all containers from $A_0$, $A_1$, ..., $A_{i-1}$ to $A_i$.

We will now analyze the time complexity of this algorithm for $n$ containers, where $n$ is unknown. We are going to show that the number of zones under consideration is finite, and then will find the sum of the number of steps spent in each zone.

FACT 4.4. *With $n$ containers, we will never walk out beyond zone $A_{\log n}$.*

Indeed, zone $A_{\log n}$ holds $2^{\log n} = n$ elements (we are using log base 2), which is the total number of elements.

FACT 4.5. *Each visit to $A_i$ costs less than $2^{i+2}$ steps.*

Indeed, to get to zone $A_i$ will we pass through zones $A_0$ to $A_{i-1}$. Hence, the total number of steps to go to $A_i$ and back is $(2^0 + 2^1 + \cdots + 2^i) \cdot 2 < 2^{i+1} \cdot 2 = 2^{i+2}$

FACT 4.6. *We visit $A_i$ at most $\lceil \frac{n}{2^i} \rceil$ times.*

Finally, let's see if we can calculate how many times we will visit each zone. Notice, that the first time we visit zone $A_i$ is when $2^i$-th element arrives (this is when we move all elements to $A_i$). Once all previous zones fill up, we will access $A_i$ again. This happens after the next $2^i$ elements arrive. In general, we access $A_i$ every $2^i$-th element, hence, total number of visits is $\lceil \frac{n}{2^i} \rceil$.

We conclude that the total number of steps is the sum of *steps* $\cdot$ *# of visits* for each zone:

$$\text{Total} = \sum_i \lfloor \frac{n}{2^i} \rfloor \cdot 2^{i+2} \leq \sum_{i=0}^{\log n} \frac{n}{2^i} \cdot 2^{i+2} = 4n(\log n + 1) = O(n \log n) \tag{4.1}$$

While we won't prove it in this lecture, this is the actual lower bound for the number of steps.

Note: An attentive reader might notice that we are double-counting steps in preceding zones for each zone, however, remember that the geometric series sum is dominated by the largest term in it, hence, our estimate is only off by a factor of 2. This does not impact the big-O analysis.

## 4.2 Beyond 2 tapes: universal computation

Now that we've seen an example solution, the rest of the lecture will be a lot more intuitive than if we just skipped the challenge problem and went straight into the discussion that follows.

In this section we will attack a *harder* task: **building a universal computer**. In other words, we want to simulate any computer (any Turing Machine) that uses an *arbitrary* number of tapes with a TM that uses a *fixed* number of tapes. It turns out that any TM with $k$ tapes is equivalent to a 2-tape TM with $n \log n$ loss in complexity. Proving this will give basis to the Time Hierarchy Theorem - the cornerstone of modern Complexity Theory.

THEOREM 4.7. *If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ on a k-tape TM, then $f$ is computable in time $O(T(n) \log T(n))$ on a 2-tape TM.*

Notice, that we've proven a similar theorem last lecture. We showed that any TM with $k$ tapes is equivalent to a single-tape TM with a quadratic loss in complexity. In our approach, we created new symbols, indicating the locations of heads corresponding to each of the $k$ tapes (Figure 4.5). That approach was slow since we had to "chase" those "heads" to update them one by one. This time we will take a different approach by keeping all heads aligned at all times and shifting tapes relative to each other (Figure 4.6). While, at first glance, this seems expensive (just imagine, if we naively copy and move a lot of stuff!), by utilizing the idea of zones from the challenge problem we will be able to avoid a quadratic slowdown. Just like in the port problem, we will only need to copy over a certain part of the tape instead of copying and moving everything.

*Proof.* Without loss of generality, we can work with bidirectional tapes. We will simulate $k$ tapes by combining a column of $k$ symbols from different tapes into one symbol. Then we will simulate the movement of heads by shifting the corresponding tape (left/right). We will denote a new symbol - *unused cell*. Notice, that *unused cells* $\neq$ *empty cells*. For example,

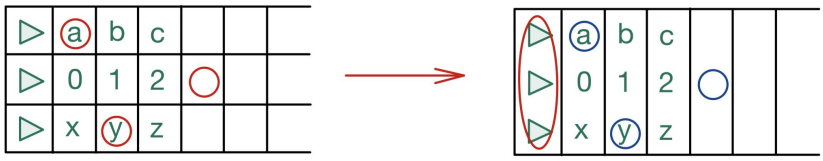


FIGURE 4.5: K-tape transformation used previously.

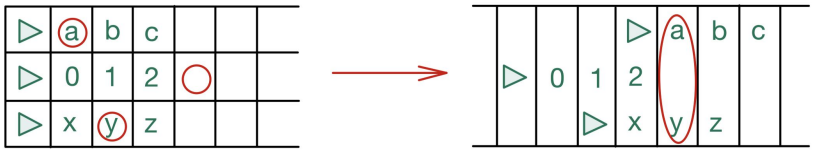


FIGURE 4.6: K-tape transformation with aligned heads.

two patterns in the Figure 4.7 are equivalent. The idea of having *unused cells* will allow us to optimize the process of shifting tapes.

Next, we will divide the entire tape into zones of exponentially increasing length (1, 2, 4, ...) as described in the Figure 4.8. Zones $L_i$ and $R_i$ are $2^i$ cells wide and zone $H$ has width 1. Since we are going to move elements in both directions, we won't be able to guarantee a full/empty scheme like we did in the port problem. Instead, we will agree to have a total of $2^i$ *used* cells in $L_i \cup R_i$. This way, each zone can be either full, empty, or *half-empty*.

ASSUMPTION 4.8.

- $L_i \cup R_i$ has $2^i$ used cells in total.
- Each $L_i$, $R_i$ is either full, empty, or half-empty.

We will show how to shift one row (one of the old $k$ tapes) to the right. Shifting left is done in a symmetrical way.

Just like in the port problem, we will have two steps in the algorithm: finding the closest non-full zone and moving the symbols accordingly (Figure 4.9):

- Find smallest $i$ s.t. $R_i$ has room in it.
  - Note that due to Assumption 4.8 (part 1), this means that $L_i$ is not empty.

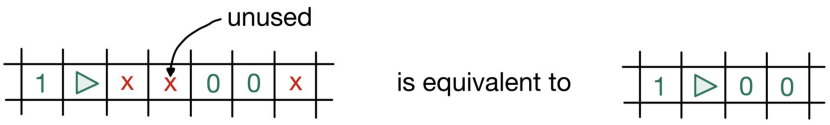


FIGURE 4.7: Example of a tape with unused cells.

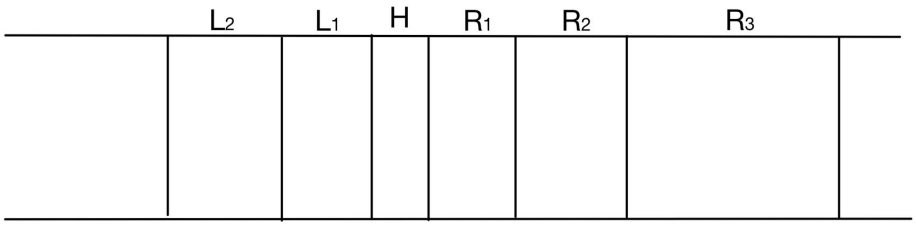


FIGURE 4.8: Tape is separated into zones.

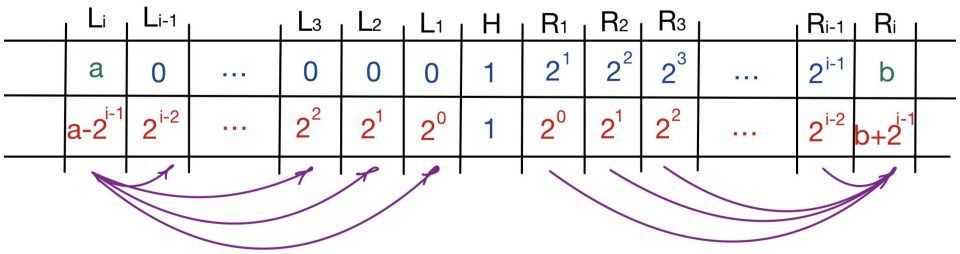


FIGURE 4.9: Tape evolution upon shifting right. The first row shows the number of used cells in each zone before the shift; the second row - after the shift.

  - For the same reason, since $R_1$, ..., $R_{i-1}$ are full, $L_1$, ..., $L_{i-1}$ are empty.

- Re-distribute content among $L_i$, $L_{i-1}$, ..., $L_1$, $H$, $R_1$, $R_2$, ..., $R_i$ by:
  - moving total of $2^{i-1}$ elements from zones $R_1$, .., $R_{i-1}$ (taking one half of elements from each) to $R_i$;
  - moving total of $2^{i-1}$ elements from $L_i$ to zones $L_1$, .., $L_{i-1}$ and filling them to one half.

The reason we have to move elements on the right is to decrease the value of $i$ for future iterations. The reason we have to move elements on the left is to maintain the invariant sum stated in the earlier assumption. Take a minute to confirm that the total count hasn't changed.

Now, we are able to perform state transitioning together for all $k$ rows while we are in the corresponding zone by shifting tapes accordingly.

This approach has a significant advantage: setting zones to be half-full makes visits to one particular zone very infrequent. In fact, we only need to visit zone $L_i$ or $R_i$ every $2^{i-1} - 1$ iterations - once half-full zones 1 to $i - 2$ fill up. Moreover, just like in the port problem, only a small number of zones ($O(\log T(n))$) are actually accessible during the time of execution. Finally, we spend $O(2^i)$ steps (number of used items in zones) in both $L_i$ and $R_i$ together. This is only achievable if we use a separate tape for copying (taking the total number of tapes used to 2). The copying process is time-consuming and without a second tape, we would have gone back to a quadratic slowdown as we did with the no-shifting 1-tape TM approach.

All in all, total number of steps is:

$$\text{Total} = \sum_{i=1}^{\log T(n)} \frac{T(n)}{2^{i-1}-1} \cdot O(2^i) = O(T(n) \log T(n)) \tag{4.2}$$

We have shown that the computation of function $f$, which is calculated on a $k$-tape TM, can be performed on a 2-tape TM with $O(T(n) \log T(n))$ time complexity. □

Notice, that for any realistic computation the value of $\log T(n)$ is relatively small, so we can usually neglect it. (As a reference, the number of atoms in our universe is $10^{82}$ and $\log 10^{82} \approx 328$, which is a very small number). Some books even use the notation $\tilde{O}(n)$, which suppresses log factors.

## 4.3 Nondeterminism

There is not much else to do in the deterministic computational paradigm, so we will expand our reach by going into nondeterminism.

DEFINITION 4.9. $L \subseteq \{0,1\}^*$ is in NP (**n**ondeterministic **p**olynomial time class) if $\exists$ const $c > 0$ and a TM $M$ s.t.:

- $M$ runs in time $n^c$
- $x \in L \Leftrightarrow \exists\ u\ M(x,u) = 1$

In such context, $M$ is called a *verifier*, $u$ is called a *certificate* (of $x$'s membership in $L$). Notice, that the main difference from the definition of P (**p**olynomial time class) is the presence of $\exists$ quantifier. Later, we will see that quantifiers have a powerful ability to expand definitions of classes.

To accustom ourselves with the definition, we provide several examples of languages that belong to NP. Notice, that our definition for NP uses the notion of language as a subset of $\{0,1\}^*$. When in our examples we define languages as discrete math objects, we imply the use of their binary representation.

DEFINITION 4.10. An independent set $S$ is a subset of vertices with no edges between any $u, v \in S$. (An example independent set is provided in Figure 4.10.)

EXAMPLE 4.11. INDSET (independent set) $=\{(G,K) : G$ has an independent set of size $k\}$ is in NP. INDSET is in NP since we can use *candidate* independent set of size $k$ as a certificate $u$ and there exists a polynomial time algorithm to verify if the set is independent.

EXAMPLE 4.12. COMPOSITE $= \{m \in \mathbb{Z} : m$ is composite$\}$ is in NP. We can use 2 different numbers as a certificate and check if $m$ is divisible by both.

EXAMPLE 4.13. SUBSETSUM $= \{(a_1, ..., a_n, t) : \sum_{i \in S} a_i = t$ for some $S \subseteq \{1, ..., n\}\}$ is in NP. We can use the target sum as a certificate.

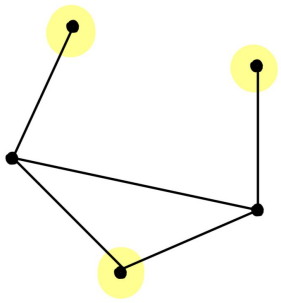

FIGURE 4.10: Independent set highlighted.

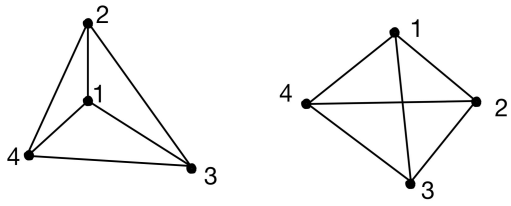


FIGURE 4.11: Example of graph isomorphism.

EXAMPLE 4.14. FACTORING = $\{(m, a, b) : m$ has a prime factor in $[a, b]\}$ is in NP. Checking if the number is prime has polynomial time complexity, hence, we can use the target prime factor as a certificate.

EXAMPLE 4.15. TSP (traveling salesperson problem) = $\{(k, G) : G$ is a weighted graph that has a tour of length $k \in \mathbb{N}\}$ is in NP. (Note: *tour* is when we visit every node exactly once.) TSP is in NP because we can use the tour itself as a certificate. We would need to check that the total length is $k$; that every node is visited; and that there are no repetitions - all can be done in polynomial time.

EXAMPLE 4.16. INTPROG (integer programming) = {system $\mathcal{L}$ of linear inequalities in $x_1, ..., x_n$ : $\mathcal{L}$ has a solution $x \in \{0, 1\}^n\}$ is in NP. The certificate is the solution to the system. The verifier will make sure the solution works.

EXAMPLE 4.17. GI (graph isomorphism) = $\{(G_1, G_2) :$ graphs $G_1, G_2$ are isomorphic$\}$ is in NP. (Note: you can use the pushpins & rubber bands analogy to understand graph isomorphism. For example, the two graphs in the Figure 4.11 are isomorphic.) The certificate can be the one-to-one assignment of the vertices of the second graph to the vertices of the first graph.

EXAMPLE 4.18. STCONN (st-connectivity) = $\{(G, u, v) :$ there is a path from $u$ to $v$ in $G\}$ is in NP. We can use the path from $u$ to $v$ itself as a certificate.

EXAMPLE 4.19. LP (linear programming) = {system $\mathcal{L}$ of linear inqualities in $x_1, ...x_n : \mathcal{L}$ has a solution $x \in \mathbb{Q}^n$} is in NP. Khachiyan has a polynomial time deterministic algorithm for linear programming. Note: Simplex algorithm works in exponential time.

If you are new to complexity theory, the connection between *decision* and *search* versions of the problems in the above examples might not be as obvious. While there are cases when they differ significantly, most of the time the difference is merely syntactical.

To better understand this concept, we suggest a challenge problem:

PROBLEM 4.20. Suppose there is a polynomial-time algorithm $A$ that takes as input a graph $G$ and an integer $k$ and determines whether $G$ has an independent set of size $k$. Use $A$ to construct a polynomial-time algorithm that takes as input a graph and outputs an independent set of the maximum size in it.

## 4.4 Alternative view of nondeterminism: NT

DEFINITION 4.21. A nondeterministic TM $M$ is $(\Gamma, Q, \delta_1, \delta_2)$, where $\delta_1$, $\delta_2$ are transition functions.

- At each step, we are free to choose $\delta_1$ or $\delta_2$ (*note*: no deterministic flow anymore).
- $M$ accepts $x$ if $M$ halts with "1" written on tape for at least one sequence of choices.

As we just defined a new type of TM, we will define a complexity class for such time-bound.

DEFINITION 4.22. $\text{NTIME}(T(n)) = \{L : \text{some TM } M \text{ decides } L \text{ in time } O(T(n))\}$.

Considering polynomial time $T(n)$, a union of NTIME for all such $T(n)$ is, intuitively, the same as NP. However, notice, that in the previous section, we've defined NP to be *something*, but now we claim it to be *something else*.

Intuitively, both models have a nondeterministic component: choice of transition function in TM and choice of certificate for verifier. Let's see why these definitions of NP are equivalent.

PROPOSITION 4.23. $\text{NP} = \bigcup_{i=1}^{\infty} \text{NTIME}(n^i)$.

*Proof.* It will consist of two parts. We will show that each side of the equality is a subset of the other, hence, they must be the same.

LEMMA 4.24. $\text{NP} \supseteq \bigcup_{i=1}^{\infty} \text{NTIME}(n^i)$.

Consider $L \in \bigcup_{i=1}^{\infty} \text{NTIME}(n^i)$. Notice, that for every input $x \in L$ running on the nondeterministic TM, we can use its sequence of choices of transition functions ($\delta_1$ and $\delta_2$) as a certificate for $x$. The verifier would then check that the original TM accepts $x$ with such a sequence of choices. This shows that $L$ is also in NP. Hence, the right side is a subset of the left side.

LEMMA 4.25. $\text{NP} \subseteq \bigcup_{i=1}^{\infty} \text{NTIME}(n^i)$.

Consider $L \in$ NP. For every $x \in L$, there is a polynomial-size certificate $u$ and a polynomial-time verifier $M$. Then, consider a nondeterministic TM that repeatedly guesses the certificate and then feeds it to the verifier $M$ until it accepts. The computation time of such TM is still polynomial since both the certificate and verifier are polynomial bound.

Hence, the left side is a subset of the right side.

This is only possible if both sides are equal. □

## 4.5 Challenge problem solution

Finally, during one of the earlier lectures, we assigned a challenge problem and we will discuss its solution here.

PROBLEM 4.26. Any constant depth circuit for the MAJ function requires exponential size.

One thing to take away from this problem is the importance of taking advantage of previous work. Many challenge problems can be reduced to problems we've seen in class. In this solution, we will use a result we've already proved for the PARITY function. We will see that the existence of a constant depth circuit for MAJ will imply the existence of such a circuit for PARITY - statement, we've proved to be wrong.

Let's play the devil's advocate. Suppose we can compute the majority function MAJ$(x_1, ..., x_n)$ with a constant depth circuit. Then we can also compute ATLEAST(threshold) and ATMOST(threshold) functions:

$$\text{ATLEAST}_k(x) = \begin{cases} 1 & \text{if } x_1 + x_2 + \cdots + x_n \geq k \\ 0 & \end{cases} \tag{4.3}$$

$$\text{ATMOST}_k(x) = \begin{cases} 1 & \text{if } x_1 + x_2 + \cdots + x_n \leq k \\ 0 & \end{cases} \tag{4.4}$$

Indeed, by adding dummy 0's and 1's we can shift the midpoint (proportion of 1's and 0's) such that the new string is decided by the MAJ function. For example, to check that a string of length 5 has at least 4 1's, we would add 2 dummy 0's. Now, if the MAJ = 1 for this new string of length 7, it would mean the original one had 4 1's.

ATMOST is computed by negating the result of ATLEAST.

Now, we have a simple way of checking if a string has exactly $k$ 1's: both ATMOST$_k(x)$ and ATLEAST$_k(x)$ have to be true. Then:

$$\text{PARITY}(x) = \bigvee_{\text{oddk}} \text{ATLEAST}_k(x) \wedge \text{ATMOST}_k(x) \tag{4.5}$$

Therefore, we can compute PARITY with a polynomial-size circuit that has constant depth. We reached contradiction.