Space Complexity: Diving into New Dimensions

Justin Zhang

A 15251 Project

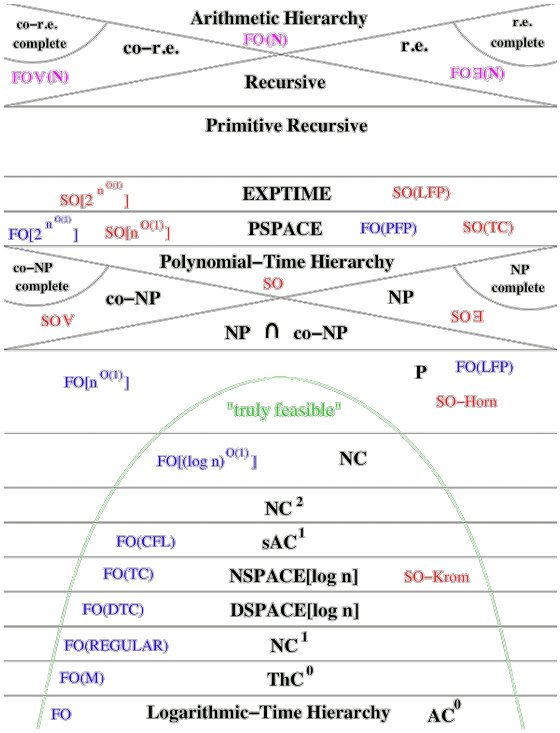

Figure 1. A Headache Visualized.

INTRODUCTION

In 251, we were introduced to complexity theory through time complexity, which we further used to analyze all sorts of algorithms. Additionally, time complexity allowed us to develop classes of languages, like P, NP, and EXP that can be solved by some algorithm in the specified time bound.

But what if I told you that there was more to it than time, that there was another piece to the puzzle of understanding the universe around us? I hope I didn't get your hopes too high for something grandiose and revolutionizing because what I want to discuss in this project is **space complexity**. Space and time are both fundamentals of our existence (some may even say they are the same), and it turns out, just like with analyzing time complexity, space complexity has has a lot of interesting ideas which complement our understanding of time complexity.

In this project, I hope to introduce space complexity and showcase some interesting theorems. I hope you will enjoy them as much as I have! :)

TABLE OF CONTENTS

1 BASIC DEFINITIONS

Figure 2. A contemporary take

Let's hit the ground running.

---

definition (Space Complexity)

The **space complexity** of a decider Turing Machine $\mathbb{M}$ is the function:

$S : \mathbb{N} \to \mathbb{N}$ where $S(n)$ is the maximum number of tape cells that M visits for all inputs of length n

If the space complexity of $\mathbb{M}$ is $S(n)$, then we also say that $\mathbb{M}$ runs in space $S(n)$.

In time complexity, we used a computation model closely related to the Random Access Machine (RAM) model. When analyzing space complexity, we use a modified Turing Machine model loosely defined as follows:

---

definition (STM Model: Space Turing Machine Model)

An STM is a TM model with two tapes: a **read only tape** for the input, and a **work tape**.

There is a tape head for both tape, and we assume that it can move left, right, or stay. We cannot modify the read only tape. We can only modify the work tape, and our output is whatever is on the work tape after reaching a success state.

---

remark I coined the term STM. This is my contribution to the sciences.

double remark There are different variations of the STM model such as those with multiple work tapes. For simplicity, we will keep the work tape singular, but note that any constant amount of worktapes is negligible in space complexity. We can simply append the tapes and still keep the same space bounds (multiple tapes: S space $\to$ one tape: $O(S)$ space)

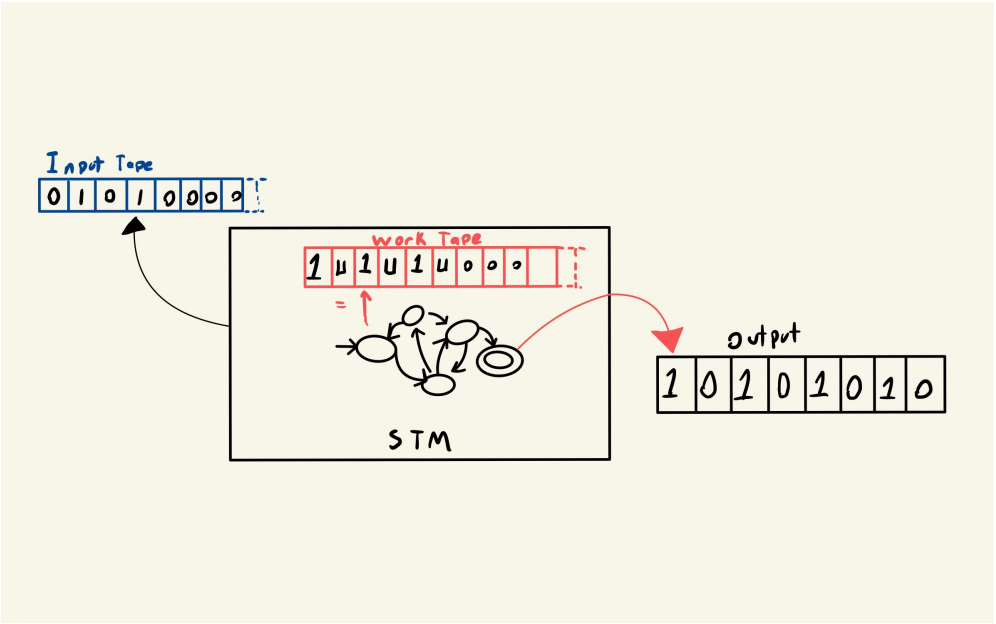

Figure 3. An illustration of an STM

In the STM model, we analyze only the space of used by the working tape. In most decidable languages, their corresponding TMs will read through the entire input. Thus, the time and space in the conventional TM model would both be linearly lower-bounded. The STM model opens the possibility of sub-linear space use as we only have to worry about work done on the work tape. We will discuss a certain sublinear space class that we choose to draw the line in the sand at because of its nice properties.

2 SPACE CLASSES

Figure 4. This may or may not reflect my (nondeterministic) opinion on P = NP.

Similarly with time complexity, we can define complexity classes for space.

---

definition

Let SPACE($f(n)$) = $\{A \subset \Sigma* :$ there exists an STM for A with space cost $O(f(n))\}$
Then, we denote the following space classes:

$$\mathbf{L} = \text{SPACE}(logn)$$

$$\mathbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \text{SPACE}(n^k)$$

We only denote these two specific classes for now since we will require further insights to create more space classes. However, L and PSPACE cover a wide variety of problems, and you might be surprised by their relationships with their time complexity counterparts

2.1 The Space Class L

Generally, the smallest space class we are concerned with is L. As it turns out, a large amount of problems we are concerned with fall into L. Let's explore a simple one below for some intuition:

Exercise (length of a string):

Show that a STM on input x can compute $n = |x|$ in $logn$ space.

SOLUTION:

Our STM is described as follows: We use our STM work tape as a bit counter. As we move the head on the read only tape to the right, we increment our bit counter by 1 for each cell read that is not the empty cell. After we reach an empty cell, we should reach an accepting state, and we directly return the bit counter on our work tape as the length of the input in binary.

It is obvious that this machine will return the correct answer. Let the length of the input string be n. Then, we will need $logn$ binary bits to represent n. Since this is exactly what we do in our machine, we use $logn$ space to compute $n = |x|$.

Exercise (0^n1^n)**:**
Show that $\{0^n1^n : n \in \mathbb{Z}\} \in L$

SOLUTION:
We can use two bit counter and a flag bit.

```
def A(x):
    let n = |x|
    bitCounter0 = 0
    bitCounter1 = 0
    flag = 0

    for i in range(n):
        let curr = x[i]

        if curr == 0:
            if flag == 1: reject
            bitCounter0++

        if curr == 1:
            if flag == 0: set flag = 1
            bitCounter1++
    if bitCounter0 == bitCounter1: accept
```

Figure 5. the algorithm for $\{0^n1^n : n \in \mathbb{Z}\}$ that uses $logn$ space

We have a constant amount of bit counters (each using $logn$ space) and a single bit used for the flag (constant space).

For each iteration we keep track of our index i, which since i is bounded by n, uses $logn$ space. We also obtain the character $x[i]$ which we can store in $logn$ space. The only caveat is that since we have $logn$ loops, and we assign the $logn$ space variable curr in each iteration, one may argue that we ultimately use log^2n space. However, observe that curr is only used in the context of its iteration, so we can REUSE space. We reuse the tape cells used to store curr per iteration, and thus we only use $logn$ space for the algorithm.

---

remark These exercises highlight conveniences that we may take advantage of when create algorithms for problems in L:

1. We can store a constant number of variables, namely integers in $O(n)$
2. Reusing space is a powerful and sometimes necessary technique
3. Basic arithmetic operations of our variables are allowed

As you can see, the class L defines many problems that can be solved through some form of counting but has strong enough restrictions that bar more complex problems. Consider problems where it is necessary to back trace, such as search algorithms. In order to correctly implement something like DFS or BFS, we would need to keep track of the nodes visited. In previous implementations, we did this by ”marking” the node. since the input is read only in our model, we must mark these nodes in the work tape, implying that we use a linear amount of space. Thus, search problems would fall into PSPACE, where we will make an interesting observation of a complexity class we are well acquainted with!

2.2 The Space Class PSPACE

It might be slightly unintuitive that PSPACE represents much more problems than, say P, or NP, or even the union of the two spaces! This is clearly illustrated by an example:

Exercise (SAT $\in$ PSPACE):
Show that SAT $\in$ PSPACE.

SOLUTION:
What is the most obvious algorithm for SAT? Brute force!

```
def SAT(phi):
    for each possible combination of true and false literals in phi:
        if the combination satisfies phi:
            accept
    reject
```

Figure 6. Brute Force SAT.

A combination of assignment of literals would take linear space. Since we would only have to keep track of any single assignment once per iteration, we can reuse space for that. Thus, since we have an algorithm that uses linear space, SAT $\in$ PSPACE.

---

remark As you might have seen with the algorithms presented for problems in L, many of these algorithms that focus on minimizing space use don't have great time complexity. This observation will be formalized in the hierarchy section.

A nice observation you might have about this exercise is that an NP-Complete language falls into PSPACE. It turns out that NP is actually a subset of PSPACE! On a high level, this makes sense; for a language to be in NP, it needs to a polynomial length certificate that is verifiable in polynomail time. We would just have to prove that the work done never exceeds linear bounds.

HOWEVER, to prove this observation, we must first prove another nice (and simple) observation.

Proposition (Time bounded by Space):
Claim: For any function $f : \mathbb{N} \to \mathbb{N}$.

$$\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$$

proof
TIME$(f(n))$ implies that we visit at most $O(n)$ tape cells. We can just write all this down into our work tape implying SPACE$(f(n))$.

Sorry for the delay, let's get on with the proof.

Proposition (NP $\subseteq$ PSPACE):
Claim: NP $\subseteq$ PSPACE.

Consider an arbitrary problem θ in NP. There exists some poly time verifier V which accepts any poly length certificate u for θ and rejects otherwise. Then, V runs in $O(sd)$, where $s = |u|$ and d is some constant term.

proof:

We know that the space used by V is at most $O(sd)$ by our previous proposition, so this implies that all certificates it accepts are of at most $O(sd)$ length. Thus, for any NP problem, we can try all combinations of strings of length at most $O(sd)$ and use them as certificates in V. If V accepts, then we know a certificate exists, and we can accept. Since there are a finite number of $O(sd)$ length strings, we will eventually loop through all of them, in which we will reject since there is no poly length certificate. V uses at most $O(sd)$ space as well, so our algorithm would use linear space. Thus $\theta \in PSPACE$, and therefore NP $\subseteq$ PSPACE.

A natural followup would be: is PSPACE $\subseteq$ NP, and the answer is..... maybe! We don't know if this is true, so we also don't know if PSPACE $=$ NP either. Aaaaaand you might be able to deduce that we don't know if P $=$ PSPACE either. Life is unfair.

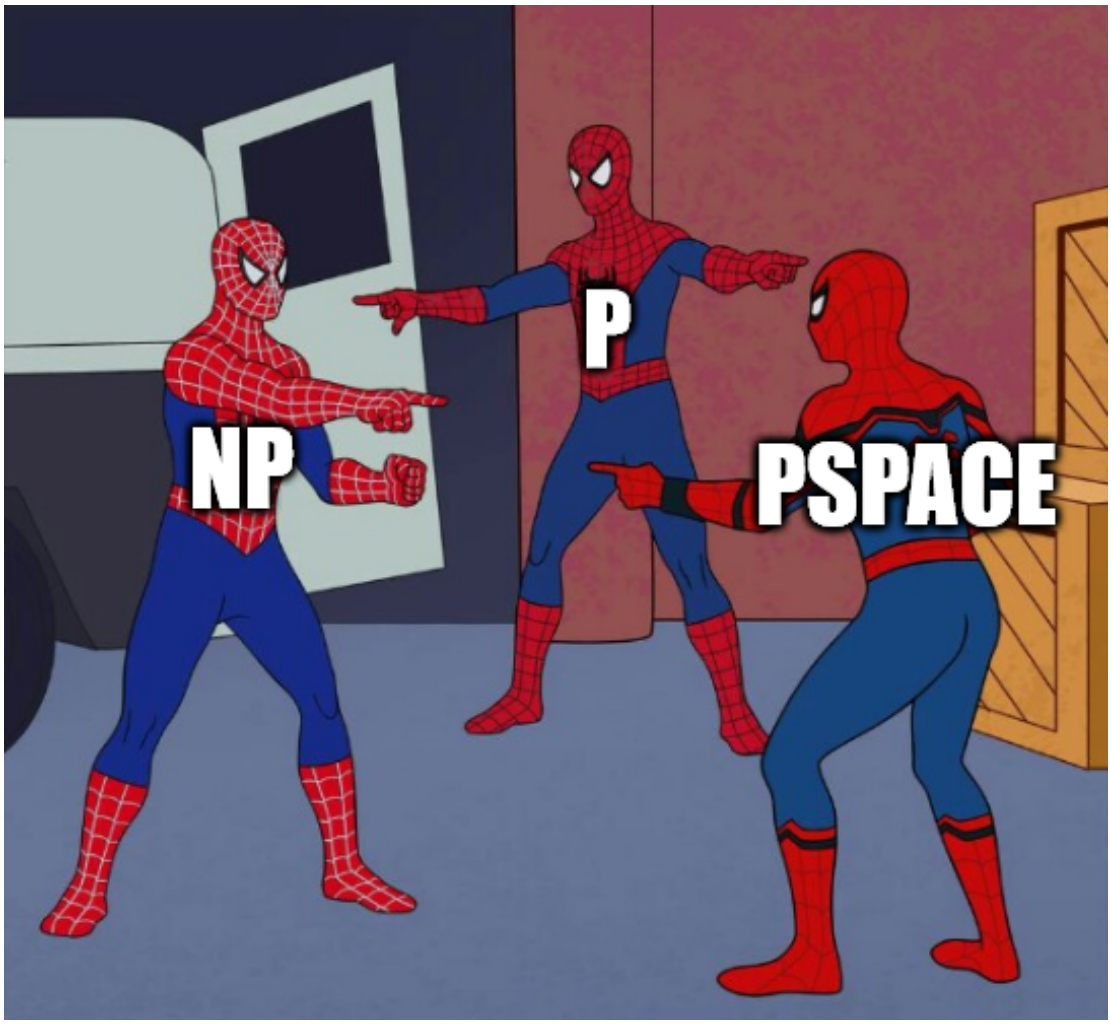

Figure 7. Complexity classes and the multiverse of madness

NONDETERMINISTIC SPACE CLASSES

In 251 we defined the class NP via a proof system with a poly time verifier. We could define the classes of problems that have algorithms with nondeterministic space used in a similar fashion, but this makes it harder to prove some things about space complexity that are especially interesting. Instead, we use the notion of a **nondeterministic turing machine**

---

definition (NTM)

A nondeterministic turing machine is formally defined as a 6-tuple $(Q, \tau, \Sigma, \delta, q_0, Q_{accept})$ where,

1. Q: the set of states
2. $\tau = \Sigma \cup \{blank\}$: the tape alphabet
3. Σ: the alphabet
4. $\delta : Q \times \tau \to \mathbb{P}(Q \times \tau \times \{L, R\})$: the **nondeterministic** transition function
5. q_0: the starting state
6. $Q_{accept} \subseteq Q$: the set of accepting states

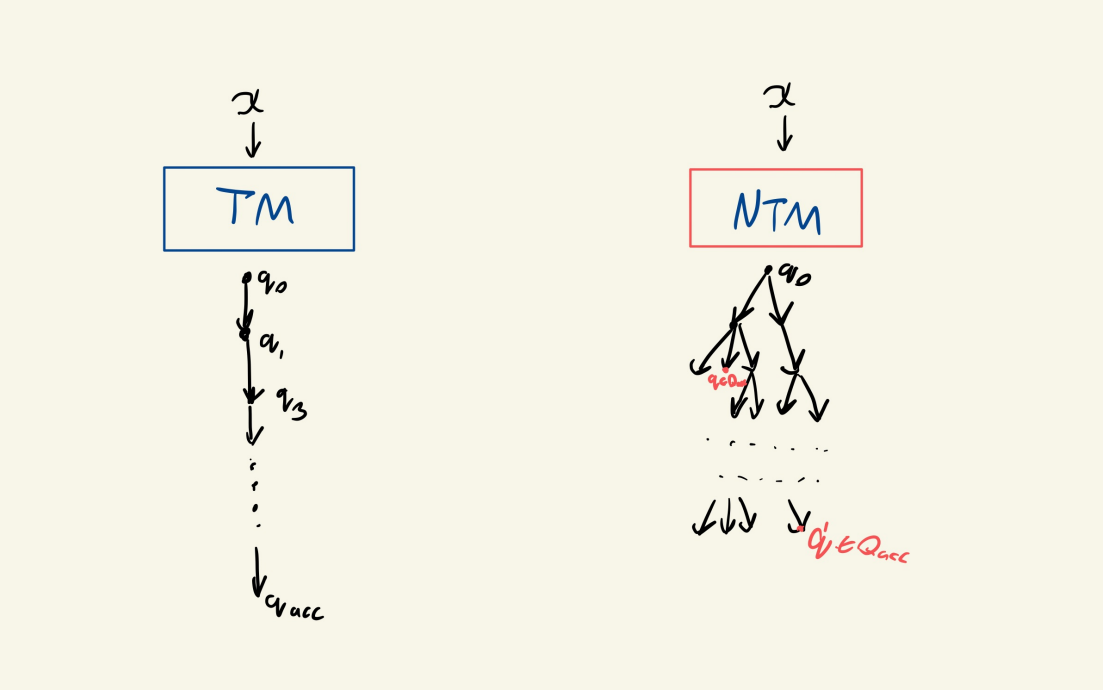

Figure 8. TMs vs NTMs visualized as directed graphs

Note the differences between NTMs and TMs. First off, the transition function δ of NTMs returns a set of possible transitions, unlike TMs that have a single transition for each provided input. Thus, our transition function chooses one possible combination from its set nondeterministically as a transition. This creates a number of possible results represented by the illustration above. Secondly, there may be multiple accepting states. An NTM is a decider if at least one branch leads to an accepting state for a specific input.

With this, we can now define the concept of nondeterministic space.

definition (NSPACE)

$$\textbf{NSPACE}(f(n)) =$$

$$\{L \in \Sigma* : \exists \text{ a decider NTM for } L \text{ using } O(f(n)) \text{ space on its work tape per each of its branches}\}$$

Now we can define more classes!

$$\textbf{NL} = \text{NSPACE}(logn)$$

$$\textbf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k)$$

We will see something very interesting in the next section, something that might almost seem miraculous if you view it from far away enough. Get your glasses off and your contacts out because we're diving into Savitch's Theorem.

SAVITCH'S THEOREM

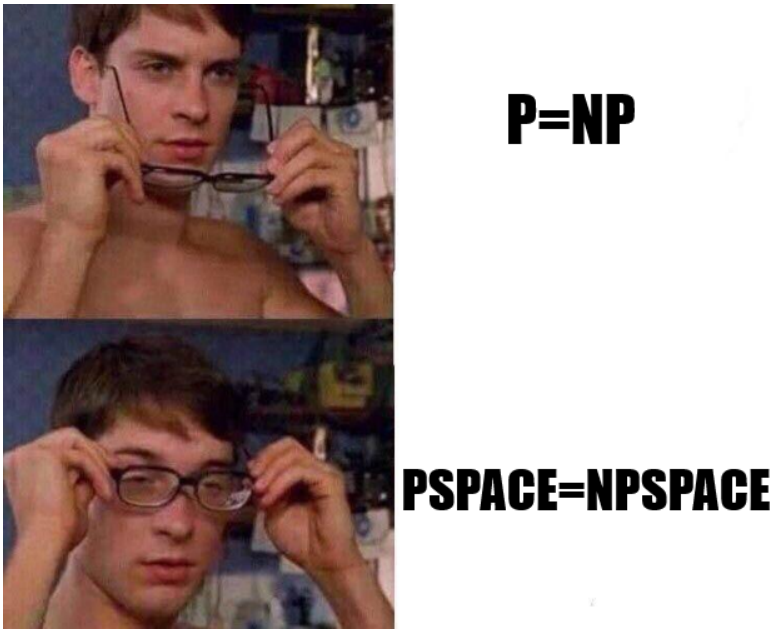

Figure 9. WHA- oh

On paper, it might seem that the relationship between P and NP should directly correlate with PSPACE and NPSPACE. However, as we have shown earlier with things like NP being a subset of PSPACE, space class relations may not be as obvious as they seem.
I believe nothing encapsulates this more than **Savitch's theorem**

Savitch's theorem For any function $f : \mathbb{N} \to \mathbb{N}$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$$

On a high level, what we need to do is to simulate each $f(n)$ space of a NTM's branch on a deterministic TM. We could try to just try all the branches, but this would require us to keep track of which branches we have visited already. Any branch uses $f(n)$ space, and branches off into some constant number of different branches. The number of possible branches is thus $2^{O(f(n))}$. Therefore, this approach would use $2^{O(f(n))}$ space, which is too much. Instead, we take a different approach, one that professor O'donnell called the "middle first search".

Consider the configurations c_i of an NTM on a specific input, where c_0 is the starting configuration and c_{acc} is the accepting configuration. As we saw earlier, we can create a directed graph for the NTM where each node is a configuration and an edge between two nodes implies a transition from the transition function between the two configurations.

Now, we want to show that we can find an algorithm that accepts whenever there exists an accepting configuration in the NTM with $f^2(n)$ space.

proof
We define a configuration c_i more specifically as the string $XqY;\omega$, where X is some portion of the work tape starting from the 0th index onward, q is the current state, Y is the rest of the work tape, and

ω is the position of the tapehead on the input tape. Note that the placement of q between X and Y also denote the placement of the tape head on the work tape. This encapsulates all the information needed about any configuration.

Additionally, the number of configurations possible is bounded by $2^{O(f(n))}$.

1. The worktape tapehead has at most $f(n)$ positions on the worktape
2. per tape cell, we have $|\tau|^{f(n)}$ different possible symbols layouts on the work tape
3. There are $|Q|$ possible states for q
4. There are n possible input tapehead positions

Thus, # configurations $= |\tau|^{f(n)} \times |Q| \times f(n) \times n$. Since $|\tau|$ is constant, $|\tau|^{f(n)} = 2^{O(f(n))}$. $|Q|$ is also constant, and $f(n), n \leq 2^{O(f(n))}$. Thus,

$$\text{\# configurations} = 2^{O(f(n))}$$

We define a deterministic recursive algorithm as follows:

```
#M defines a specific NTM
def PATH_M(s,t,n):

    #bc
    if n == 1:
        if s = t or s transitions to t in a single step:
            accept

        else: reject

    for each configuration c of M:
        run PATH_M(s,c,n//2)
        run PATH_M(c,t,n//2)
        if both accept, accept
    reject
```

Figure 10. A recursive path finding algorithm

We can now define a deterministic TM by having it return the output of PATHM$(c_0, c_{acc}, 2^{df(n)})$ for some constant d. Note that $2^{df(n)}$ gives an upper bound for the maximum number of steps possible because that is the number of configurations possible. Observe that the NTM must be a decider by definition, so we never infinite loop ie we never repeat a configuration. Thus the number of steps is bounded by the number of unique configurations, $2^{df(n)}$.

It is obvious that if a path does exist from q_0 and q_{acc}, our algorithm would find it. We want to show that it takes $f^2(n)$ space.

Whenever PATHM is called recursively, it stores its inputs s, t, m so that it may restore its state after the recursive call (backtracking). Thus, each level of recursion uses $f(n)$ space. Since each level of recursion divides m by 2 and we finish recursing once $m = 1$, we will have $log(2^{df(n)}) = f(n)$ number of recursions. Therefore, in total, we use $f^2(n)$ space to simulate the NTM M, and thus NSPACE$(f(n)) \subseteq$ SPACE$(f^2(n))$

Hooray! So, what does this mean? Well since exponentiated a polynomial by another factor of 2 still makes it a polynomial, by Savitch's theorem, it follows that PSPACE= NPSPACE. Oh my god! . . . So what does this mean?

To me, Savitch's theorem is a testament to the power of space over time. The biggest advantage space has over time is that it can be reused. Savitch's theorem is able to sort of imply that through its definition and proof, while also being able to make concrete and useful claims about the relations of deterministic space classes and their nondeterministic counterparts. I think that's pretty beautiful.

HIERARCHIES

Before we start mashing together all our knowledge of complexity classes, there is one last relation I want to make. Recall the proposition that TIME($f(n)$) $\subseteq$ SPACE($f(n)$). This tells us what time complexity classes are subsets of space complexity classes, like P $\subseteq$ PSPACE, but what about the other way around. Do we have an idea of what space complexity class fall into time complexity classes? Thanfully, the answer is yes, and we actually proved this indirectly in the Savitch's theorem section:

Proposition (Space bounded by Time):
Claim: For any function $f : \mathbb{N} \to \mathbb{N}$.

$$\text{SPACE}(f(n)) \subseteq \text{TIME}(2^{O(f(n))})$$

proof

Recall that there are a maximum of $2^{O(f(n))}$ configurations for any STM. The definition of SPACE($f(n)$) also requires that our STM be a decider, so we may not visit any same configuration twice, else we fall into a non terminating loop. Thus, the maximum number of steps we can take are the number of unique configurations, $2^{O(f(n))}$. Therefore, the language in SPACE($f(n)$) is also in TIME($2^{O(f(n))}$), and SPACE($f(n)$) $\subseteq$ TIME($2^{O(f(n))}$).

Now, lets start applying our theorems and getting a picture of where we stand with all our complexity classes. I will denote proposition TIME($f(n)$) $\subseteq$ SPACE($f(n)$) as p1, SPACE($f(n)$) $\subseteq$ TIME($2^{O(f(n))}$) as p2, and Savitch's theorem as the super awesome theorem:
Some cool things about L/NL

1. L $\subseteq$ P by p1
2. L $\subseteq$ NL by super awesome theorem
3. NL $\subseteq$ SPACE($log^2 n$) (by p1) $\Rightarrow$ NL $\subseteq$ P (by p2)

Some cool things about PSPACE

1. PSPACE = NPSPACE by super awesome theorem
2. P $\subseteq$ PSPACE by p1
3. PSPACE $\subseteq$ EXP ($log^2 n$) by p2
4. NP $\subseteq$ NPSPACE (by p1) $\Rightarrow$ NP $\subseteq$ PSPACE (by PSPACE = NPSPACE)

Compiling this new information together, here is a rough picture to how the complexity classes look (assuming classes we don't know are equal or not as subsets, as per expert opinion):

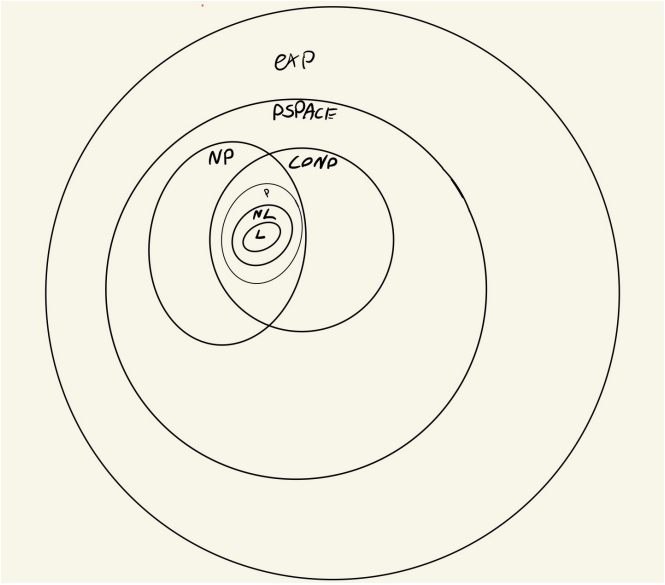

Figure 11. As beautiful as the sunset

3 CONCLUSION

Space complexity is an eye opening extension of time complexity, and what I have covered is only the tip of the iceberg; there are randomized space classes and complete space classes, applications of space complexity found in streaming algorithms, hardness theorems for specific complexity classes, and much more. As we grow our understanding these complexity classes, their implications will lead to fundamental changes in the way we live our lives and the technologies we may see.

ACKNOWLEDGMENTS

Figure 1's source is the cover of the Descriptive Complexity Textbook authored by Neil Immerman.

All illustrations and meme pictures were my own creations.

The sources of my information were from Sipser's Introduction to the Theory of Compution Space Complexity chapter, CMU 15455 S20 lecture 18, and Professor Ryan O'Donnel's UCT lectures 16,17 on Youtube.