St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 3: Foundations of Complexity Theory II
Winter 2025

Anyone who has used a real-world computer can tell you that adding more resources allows you to do more things with that computer. Adding more memory, increased disk space, or a more powerful processor can allow you to solve more resource-intensive problems—or just play the latest and greatest video game.

But what happens if we add more resources to an abstract model of computation like a Turing machine? As a consequence of the linear speedup theorem and the tape compression theorem, we know that we must increase the amount of resources by more than a constant factor if we expect to see any difference, so consider two functions f_1 and f_2 where $f_1 \ll f_2$. Our intuition tells us that machines with more resources should be more powerful than machines with fewer resources, so (for example) we would have that $\mathsf{DTIME}(f_1(n)) \subset \mathsf{DTIME}(f_2(n))$. However, our intuition is only correct if f_1 and f_2 are "nice" functions; that is, if these functions are subject to certain desirable constraints.

If we simply take f_1 and f_2 to be arbitrary functions, then some extremely weird behaviour can arise when we attempt to measure time or space complexity. Therefore, we must impose *some* constraints on the kinds of functions we take as our resource bounds.

1 Gap Theorem

Since we're dealing with Turing machines, we should naturally expect that any function we employ as a resource bound is at least computable. However, computability by itself is not sufficient for us to avoid trouble. In the 1960s, the Russian-Israeli computer scientist Boris Trakhtenbrot and the American-Canadian computer scientist Allan Borodin independently proved a result known as the *gap theorem*, which tells us that if all we know is that our functions are computable, then we can find two functions f_1 and f_2, with $f_1 \ll f_2$, specifying two distinct resource-bounded classes of Turing machines that are both only capable of solving exactly the same set of problems. This sounds completely counterintuitive—and even wrong!—but it stems from the property of computability alone being much too weak.

The name of the gap theorem comes from the fact that, beginning with some appropriate starting value n, we can divide "time" into intervals of increasing exponential length

$$[0, n], [n+1, 2^n], \left[2^n + 1, 2^{2^n}\right], \left[2^{2^n} + 1, 2^{2^{2^n}}\right], \dots$$

and look for a gap within some interval $[m+1, 2^m]$ where, for every finite set of Turing machines and every finite set of input words to these Turing machines, no machine in this set halts after performing some number of computation steps greater than $m+1$ and less than 2^m.

Theorem 1 (Gap theorem)**.** *There exists a computable function $f\colon \mathbb{N} \to \mathbb{N}$ with the property that*

$$\mathsf{DTIME}(f(n)) = \mathsf{DTIME}\big(2^{f(n)}\big).$$

Proof. The idea behind this proof is that we will construct the function f named in the statement of the theorem via diagonalization, and we will purposely construct f to grow extremely quickly. Our definition of f will rely on the value m corresponding to the interval $[m+1, 2^m]$ which contains the aforementioned gap.

Consider an enumeration of all Turing machines $\{\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \dots\}$ in lexicographic order. We define a predicate $\mathrm{gap}_i(a, b)$ for all $i, a, b \geq 0$ and $a \leq b$ such that, given any Turing machine $\mathcal{M}_j$ with $0 \leq j \leq i$ and

any input word w of length i, $\text{gap}_i(a, b)$ is true if $\mathcal{M}_j$ halts on w in (i) fewer than a computation steps, (ii) more than b computation steps, or (iii) not at all.

Observe that we can decide whether $\text{gap}_i(a, b)$ is true by simulating the computation of all Turing machines from $\mathcal{M}_0$ to $\mathcal{M}_i$ in our enumeration on all inputs of length i for at most b computation steps. If some Turing machine in this subset halts before reaching a computation steps, or is still performing its computation upon reaching b computation steps, then we return a positive answer.

Now, take $\text{inp}(n)$ to be the number of inputs of length n given to Turing machines $\mathcal{M}_0$ to $\mathcal{M}_n$ in our enumeration; that is, $\text{inp}(n) = \sum_{j=0}^{n} |\Sigma_j|^n$, where $|\Sigma_j|$ denotes the size of the input alphabet of the Turing machine $\mathcal{M}_j$. We then define a sequence of numbers k_x to be

$$k_x = \begin{cases} 2n, & \text{if } x = 0; \text{ and} \\ 2^{k_{x-1}} & \text{if } x \geq 1. \end{cases}$$

In this way, the interval $[k_0 + 1, k_1]$ is associated with the interval $\left[2n + 1, 2^{2n}\right]$, the interval $[k_1 + 1, k_2]$ is associated with the interval $\left[2^{2n} + 1, 2^{2^{2n}}\right]$, and so on. Finally, we take $f(n)$ to be the least number k_i, $0 \leq i \leq \text{inp}(n)$, such that $\text{gap}_n(k_i + 1, k_{i+1})$ is true.

Next, consider any language $L \in \mathsf{DTIME}\left(2^{f(n)}\right)$, and suppose that some Turing machine $\mathcal{M}_j$ recognizes L. Given any input word w where $|w| \geq j$, $\mathcal{M}_j$ will either halt in fewer than $f(|w|)$ computation steps, halt in more than $2^{f(|w|)}$ computation steps, or not halt at all. This is because the Turing machine $\mathcal{M}_j$ was included in our enumeration and so it was accounted for in our definition of the function f. Since $\mathcal{M}_j$ halts in time *at most* $2^{f(n)}$, it cannot be the case that it halts in *more than* $2^{f(n)}$ computation steps, and so it must halt in time at most $f(n)$ on the input word w. Thus, $L \in \mathsf{DTIME}(f(n))$.

(Note that for input words of length less than j, we do not necessarily know when $\mathcal{M}_j$ halts. However, we can get around this issue by modifying the state set of $\mathcal{M}_j$ so that it handles this finite number of inputs separately.) □

The gap theorem doesn't only apply to deterministic time classes: we can prove similar results for deterministic space classes as well. Moreover, we need not even require an exponential gap like the one between $f(n)$ and $2^{f(n)}$. We can use *any* computable function $g \colon \mathbb{N} \to \mathbb{N}$ with $g(n) \geq n$ in the statement of the gap theorem without affecting the argument.

2 Time and Space Constructibility

In light of this result, we know that we need our functions to be more than just computable—we need to place some additional stronger constraint on them. Instead of focusing on the functions themselves and trying to fit our model of computation to the functions, let's shift our focus to the model of computation directly. We can constrain the kinds of functions f we consider to be "nice" by restricting ourselves to using only those Turing machines for which we can measure and verify that its computation uses time or space bounded by f. If this is possible, then we say that f is *constructible* by that Turing machine, and we gain two definitions depending on whether our focus is on time or on space.

Definition 2 (Time constructibility)**.** A function $f \colon \mathbb{N} \to \mathbb{N}$ is time constructible if there exists a deterministic Turing machine $\mathcal{M}$ which, given any input word of length n, halts after exactly $f(n)$ computation steps.

Definition 3 (Space constructibility)**.** A function $f \colon \mathbb{N} \to \mathbb{N}$ is space constructible if there exists a deterministic Turing machine $\mathcal{M}$ which, given any input of length n, halts in a configuration in which exactly $f(n)$ work tape cells contain symbols and no other work tape cells are visited during the computation.

As it turns out, if we restrict ourselves to considering only time or space constructible functions, then we avoid the anomaly caused by the gap theorem and regain the expected intuitive behaviour that, with more resources, we can do more things. Indeed, we can alternatively define time and space constructibility directly

in terms of how much we can do: a function f is time or space constructible if some Turing machine can compute f using time or space resources bounded by f itself.

Computing Functions. Before we continue further, consider that our Turing machines so far have only been capable of solving decision problems with yes/no answers. So what does it mean for a Turing machine to "compute a function"? To define this notion more formally, we need to introduce the *unary representation* of a number. Where a binary representation of a number uses two symbols (`0` and `1`), a unary representation uses only one symbol (`1`) to write a number n as a length-n string. As an example, if we wanted to write the unary representation of the number 16, we would write `1111111111111111`. We can naturally extend unary representations to functions $f : \mathbb{N} \to \mathbb{N}$: starting with n `1`s, applying the function f would produce $f(n)$ `1`s.

When we say that a function f can be computed in time $t(n)$, we mean that there exists some deterministic Turing machine $\mathcal{M}$ that receives the word $\texttt{1}^n$ on its input tape and writes $\texttt{1}^{f(n)}$ to some work tape designated as an "output tape" in at most $t(n)$ computation steps.

Likewise, when we say that a function f can be computed in space $s(n)$, we mean that there exists some deterministic Turing machine $\mathcal{M}$ that receives the word $\texttt{1}^n$ on its input tape and writes $\texttt{1}^{f(n)}$ to its "output tape" using at most $s(n)$ work tape cells.[1]

You might have noticed that, in Definitions 2 and 3, we could replace the phrase "given any input of length n" with "given the input $\texttt{1}^n$" without affecting the meaning of constructibility. We can formally connect our definitions of time and space constructibility to this notion of computing functions by way of the following two theorems. We will present the space-related theorem first, as its statement and proof are rather straightforward.

Theorem 4. *A function f is space constructible if and only if f can be computed in space $O(f(n))$.*

Proof. ($\Rightarrow$): If f is space constructible, then f can be computed in space $O(f(n))$ naturally by Definition 3.

($\Leftarrow$): If f can be computed in space $O(f(n))$, then f can be computed in space exactly $f(n)$ as a consequence of the tape compression theorem, and so f is space constructible. □

The time-related theorem is more complicated, and its proof is slightly more difficult, but with a little work we can obtain our desired result.

Theorem 5. *Let f be a function where there exists some $\epsilon > 0$ and some $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, $f(n) \geq (1 + \epsilon) \cdot n$. Then f is time constructible if and only if f can be computed in time $O(f(n))$.*

Proof. ($\Rightarrow$): If f is time constructible, then take $\mathcal{M}$ to be the deterministic Turing machine specified in Definition 2. We can compute f using a Turing machine $\mathcal{M}'$ that behaves identically to $\mathcal{M}$ and additionally writes to a separate output tape one copy of the symbol `1` for each computation step of $\mathcal{M}$. Since $\mathcal{M}$ halts after $f(n)$ computation steps, $\mathcal{M}'$ computes f in $O(f(n))$ time.

($\Leftarrow$): If f can be computed in time $O(f(n))$, then take $\mathcal{N}$ to be the deterministic Turing machine that performs this computation, and suppose that $\mathcal{N}$ computes f exactly in time $g(n)$. We then know by the definition of Big-O notation that $g(n) \leq c \cdot f(n)$ for some constant c.

By definition, g is a time constructible function, and we can show that $f + g$ is similarly time constructible by defining a Turing machine that simulates $\mathcal{N}$ to compute the unary representation of $f(n)$ in $g(n)$ computation steps and counts the number of `1`s written to the output tape in $f(n)$ computation steps.

Now, we verify that there exists some $\epsilon > 0$ and some $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$,

$$f(n) \geq \epsilon \cdot g(n) + (1 + \epsilon) \cdot n.$$

Let ϵ_1, ϵ_2, ϵ_3, and ϵ_4 each be positive real numbers satisfying the following properties:

[1] Note that, like the input tape, we designate the output tape to be read-only. Therefore, cells used on the output tape do not count toward our work space measurement.

P1. for all $n \geq n_0$, $f(n) \geq (1 + \epsilon_1) \cdot n$;

P2. $(1 + \epsilon_1) \cdot (1 - \epsilon_2) > 1$;

P3. $\epsilon_3 = (1 + \epsilon_1) \cdot (1 - \epsilon_2) - 1$; and

P4. $\epsilon_4 = \min\{\epsilon_2/c, \epsilon_3\}$.

Then, for all $n \geq n_0$, it is the case that

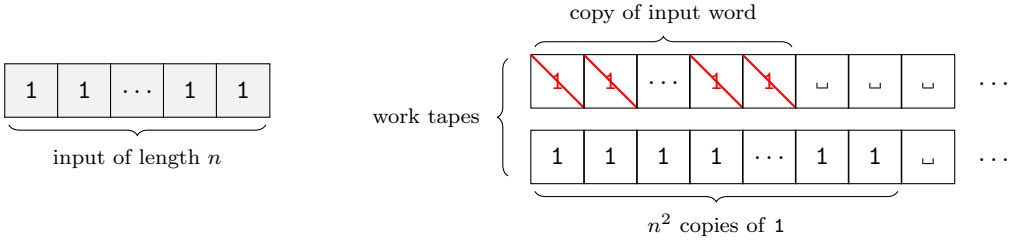
$$\begin{aligned} f(n) &= \epsilon_2 \cdot f(n) + (1 - \epsilon_2) \cdot f(n) && \text{(by definition)} \\ &\geq (\epsilon_2/c) \cdot g(n) + (1 - \epsilon_2) \cdot (1 + \epsilon_1) \cdot n && \text{(since } g(n) \leq c \cdot f(n) \text{ and by P1)} \\ &= (\epsilon_2/c) \cdot g(n) + (1 + \epsilon_3) \cdot n && \text{(by P2 and P3)} \\ &\geq \epsilon_4 \cdot g(n) + (1 + \epsilon_4) \cdot n && \text{(by P4)} \end{aligned}$$

as desired. Via a technical argument omitted here, we can prove that since both g and $f + g$ are time constructible and since $f(n) \geq \epsilon \cdot g(n) + (1 + \epsilon) \cdot n$ for all $n \geq n_0$, f is time constructible as well. □

Examples of Constructible Functions. After all of these definitions, theorems, and proofs, let's now turn our attention toward finding functions that are actually time or space constructible. Obviously, easy functions like n are both time and space constructible, so let's consider a slightly more interesting example.

Example 6. We will show that the function n^2 is time constructible.

Consider a Turing machine $\mathcal{M}$ with the word $\texttt{1}^n$ written to its input tape. We construct the function n^2 by copying all n symbols from the input tape to a work tape. Then, working from left to right, we cross out one `1` and write n `1`s to another work tape designated as the output tape. After all `1`s have been crossed out on the work tape, there will be n^2 `1`s written to the output tape, and this process takes $O(n^2)$ time.

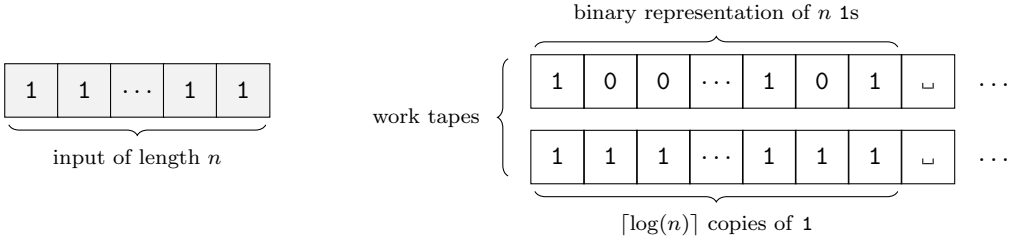


All common functions that grow at least linearly are time constructible, including polynomial, exponential, and factorial functions. What is *not* time constructible, however, is any function that is sublinear, like $\log(n)$. This is because sublinear functions don't give us sufficient time to read the entire input word.

On the other hand, not only are all of the common functions listed in the previous paragraph also space constructible, but so too are logarithmic functions!

Example 7. We will show that the function $\lceil \log(n) \rceil$ is space constructible.

Consider a Turing machine $\mathcal{M}$ with the word $\texttt{1}^n$ written to its input tape. The key observation we will use is that a binary representation of any number n has length exactly $\lceil \log(n) \rceil$. Thus, if we use the input tape as a counter, then we can write the binary representation of the corresponding number n to a work tape. We then write a number of `1`s matching the length of this binary representation to another work tape designated as the output tape. By the end of the computation, exactly $\lceil \log(n) \rceil$ work tape cells will have been used.



Other sublinear functions that grow at least logarithmically, like $\lceil \sqrt{n} \rceil$, are space constructible but not time constructible, while sublogarithmic functions like $\log(\log(n))$ aren't even space constructible.

3 Resource-Bounded Computations

It is possible to show that, in general, it is undecidable for us to determine whether some Turing machine halts within some number of computation steps $f(n)$ on an input of length n. For example, even though we know that the function n^2 is time constructible, the following result is also true.

Proposition 8. *The problem of determining whether or not a deterministic Turing machine halts within time n^2 is undecidable.*

However, this proposition and the existence of time constructible functions do not contradict one another! This is because these two notions work in "different directions", so to speak: the proposition says that we cannot decide whether an *arbitrary* Turing machine halts within time n^2, while our definition of a time constructible function says that we can halt within time n^2, say, by explicitly constructing a *specific* Turing machine that runs in a certain way.

Time Bounds. When we restrict our focus to time constructible resource bounds, we can impose these bounds on arbitrary Turing machines to ensure that they halt within the time that we desire. Knowing that a Turing machine performs its computation within a certain time bound is in fact essential to proving certain results in complexity theory, and the following theorem helps us to obtain such a guarantee.

Theorem 9. *Let t be a time constructible function. Then for any language L in the class $\mathsf{DTIME}(t(n))$, there exists a Turing machine $\mathcal{M}_i'$ that recognizes L within time $t(n)$.*

Proof. Let $\mathcal{M}_t$ be a deterministic Turing machine that, given an input of length n, halts in exactly $t(n)$ computation steps. We know that $\mathcal{M}_t$ exists, since t is time constructible. We will treat $\mathcal{M}_t$ as an *alarm clock* that keeps track of how much time we have spent on a computation; $\mathcal{M}_t$ will forcibly halt another arbitrary Turing machine $\mathcal{M}$ if it takes too much time.

Let us begin by effectively enumerating all deterministic Turing machines; we can do this by encoding Turing machines as binary strings and listing all valid encodings in some order. We will denote this enumeration by $\{\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \dots\}$. Now, for all $i \geq 1$, take $\mathcal{M}_i'$ to be a deterministic Turing machine that runs both $\mathcal{M}_i$ and $\mathcal{M}_t$ in parallel.[2] The Turing machine $\mathcal{M}_i'$ will perform its computation by simulating one computation step of $\mathcal{M}_i$ followed by simulating one computation step of $\mathcal{M}_t$. In this way, we will eventually arrive at one of two outcomes:

- If $\mathcal{M}_i$ halts before $\mathcal{M}_t$, then $\mathcal{M}_i'$ accepts if $\mathcal{M}_i$ would accept or rejects if $\mathcal{M}_i$ would reject.
- If $\mathcal{M}_t$ halts before $\mathcal{M}_i$, then $\mathcal{M}_i'$ rejects.

In the first case, $\mathcal{M}_i$ has "beat the clock" and finished its computation in time at most $t(n)$, while in the second case, $\mathcal{M}_t$ has "run out of time" and cut off the computation of $\mathcal{M}_i$, possibly modifying the outcome of the computation but ensuring that it finishes in time exactly $t(n)$. □

The Turing machines $\mathcal{M}_i'$ we defined are called *$O(t(n))$-clocked Turing machines*, and adding an alarm clock to a Turing machine does not change its language as long as that language belongs to $\mathsf{DTIME}(t(n))$.

We can extend the notion of adding alarm clocks to nondeterministic Turing machines following a nearly identical construction, thus obtaining an analogous result for $\mathsf{NTIME}(t(n))$ which brings with it the added benefit that all computations on the nondeterministic Turing machine are guaranteed to halt.

[2] Running two Turing machines in parallel incurs a factor-of-two slowdown, but this doesn't affect anything because of the linear speedup theorem.

Space Bounds. Now that we have a method of bounding computations by time, we should consider bounding computations by space as well. Let s be a space constructible function, and let $\mathcal{M}_s$ be a deterministic Turing machine that, given an input of length n, uses exactly $s(n)$ work tape cells. Given an arbitrary deterministic Turing machine $\mathcal{M}_i$, we can create a modified Turing machine $\mathcal{M}_i''$ that first uses $\mathcal{M}_s$ to *mark* the available work tape space and then simulates the computation of $\mathcal{M}_i$ within that marked space. As before, if $\mathcal{M}_i$ attempts to use more than $s(n)$ work tape cells, then $\mathcal{M}_i''$ rejects.

However, unlike with time-bounded computations, we encounter a potential issue with space-bounded computations: such computations may run forever! It's possible for a computation to use at most $s(n)$ work tape cells while still getting caught in an infinite loop if, say, the input head of the work tape moves back and forth within those $s(n)$ cells. Thus, we must come up with a method to detect whether our Turing machine has entered an infinite loop.

For this method, we will rely on the fact that if some configuration is repeated during the computation of a deterministic Turing machine, then that machine is in an infinite loop. The reasoning behind this fact is straightforward: since the computation is deterministic, after encountering a configuration for the second time, we are guaranteed to follow the same sequence of steps leading to us encountering the same configuration for the third time (and the fourth time, etc.).

Theorem 10. *Let $\mathcal{M}$ be a k-tape deterministic Turing machine that recognizes a language $L(\mathcal{M})$ using $s(n)$ space, where $s(n) \geq \log(n)$ is a space constructible function. Then there exists a $(k+1)$-tape deterministic Turing machine $\mathcal{N}$ where each of the following properties holds:*

- *$L(\mathcal{N}) = L(\mathcal{M})$;*
- *the computation of $\mathcal{N}$ on any input of length n uses at most $s(n)$ space; and*
- *every computation of $\mathcal{N}$ halts.*

Proof. Given a Turing machine $\mathcal{M}$ as defined in the statement of the theorem, we begin by counting the number of distinct configurations $\mathcal{M}$ could be in over the course of its $s(n)$-space-bounded computation.

An upper bound on the number of distinct configurations is obtained by multiplying together the input length and the number of symbols in the tape alphabet, and then multiplying this product by the number of states of $\mathcal{M}$; that is,

$$\begin{aligned} |Q| \cdot n \cdot |\Gamma| &= |Q| \cdot 2^{\log(n)+\log(|\Gamma|)\cdot s(n)} \\ &\leq |Q| \cdot 2^{c \cdot s(n)} \text{ for some constant } c. \end{aligned}$$

We now construct the Turing machine $\mathcal{N}$. At the start of its computation, $\mathcal{N}$ marks $s(n)$ tape cells on its $(k+1)$st tape; this is possible because s is a space constructible function. Then, on its first through kth tapes, $\mathcal{N}$ simulates the computation of $\mathcal{M}$ on its k tapes. During this simulation, $\mathcal{N}$ uses its $(k+1)$st tape as a base-c counter to measure the length of the computation of $\mathcal{M}$. If this counter surpasses $2^{c \cdot s(n)}$, then $\mathcal{N}$ knows that $\mathcal{M}$ must have repeated some configuration at some earlier stage in its computation, and so $\mathcal{N}$ halts and rejects. □

As before, we can adapt this construction to work for nondeterministic Turing machines as well, although we must be more careful since repeating configurations in a nondeterministic computation does not necessarily imply the computation has entered an infinite loop. Any nondeterministic computation longer than $2^{c \cdot s(n)}$ can be shortened by removing loops within the computation, and so a nondeterministic Turing machine can reduce the length of its computation to at most $2^{c \cdot s(n)}$ without affecting the language it recognizes.

4 Relations Between Basic Complexity Classes

Recall that our motivation behind introducing time and space constructible functions was to avoid anomalies like the gap theorem and to regain our intuition that, for any two functions f_1 and f_2 where $f_1 \ll f_2$,

$\mathsf{DTIME}(f_1(n)) \subset \mathsf{DTIME}(f_2(n))$. (Likewise for space complexity, or for nondeterministic measures.) In this section, we will begin to develop inclusion relationships between time and space complexity classes depending on the amount of resources we allocate, in turn producing a hierarchy of these classes.

To begin, though, we should address one possible criticism: how can we be sure that restricting our resource bounds to use time constructible functions gives an accurate picture of the hierarchy, when the set of computable functions is so much more general? To address this criticism, we will prove one small result which shows that we can find a time constructible function that is capable of growing as fast as any given computable function.

Lemma 11. *For every computable function f, there exists a time constructible function g such that, for all $n \in \mathbb{N}$, $g(n) > f(n)$.*

Proof. Let $\mathcal{M}_f$ be a deterministic Turing machine that receives as input the word $\mathtt{1}^n$ and produces as output the word $\mathtt{1}^{f(n)}$; that is, $\mathcal{M}_f$ computes the function f.

Similarly, let $\mathcal{M}_g$ be a deterministic Turing machine that, given an input word of length n, writes $\mathtt{1}^n$ to a work tape and then simulates the computation of $\mathcal{M}_f$ on the input $\mathtt{1}^n$.

Denote by $g(n)$ the number of computation steps used by $\mathcal{M}_g$ on its input of length n. Then g is time constructible, since it is the running time of $\mathcal{M}_g$, and since $n + f(n)$ time is required to write $\mathtt{1}^n$ to a work tape and simulate the computation of $\mathcal{M}_f$, we have that $g(n) > f(n)$. □

As a consequence of Lemma 11, we obtain our main result: time constructible functions are enough for our purposes.

Theorem 12. *For any computable function f, there exists a time constructible function g such that*

$$\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(g(n)).$$

Proof. Follows from the existence of the Turing machines constructed in the proof of Lemma 11. □

Since any computable resource bound is included in some time constructible resource bound, we are not missing anything from our picture, and so we can move on with producing our hierarchy.

4.1 Space Hierarchy Theorem

The first step in our hierarchy will be to establish strict inclusion relationships between complexity classes involving the same model when we give that model differing amounts of resources. In the deterministic case, these relationships are characterized by two theorems: the *time hierarchy theorem* and the *space hierarchy theorem*. As we did with our constructibility results, we will first consider the space hierarchy theorem, as its formulation is simpler and its proof will motivate our proof of the time hierarchy theorem.

If we give a machine more space, then we would expect it to do more with that additional space as per our intuition. However, if this additional space, say s', only exceeds the original amount of space, say s, by a small amount, then we may not gain any advantage—indeed, we may even need more space than we're given just to compute the small difference itself! Thus, if we restrict ourselves to space constructible functions as we've been doing all along, the space hierarchy theorem tells us that languages exist that can be decided in some amount of space s', but not in space s.

Theorem 13 (Space hierarchy theorem)**.** *For any space constructible function $s(n) \geq \log(n)$, there exists a language $L \in \mathsf{DSPACE}(s(n))$ that cannot be recognized by any deterministic Turing machine using space $o(s(n))$.*

Proof. We want to show that there exists some language L that we can recognize in space $O(s(n))$, but not in space $o(s(n))$.

Consider two functions s_1 and s_2, where $s_1(n) \geq \log(n)$ and $s_1 \in o(s_2(n))$. We construct a deterministic Turing machine $\mathcal{M}$ which takes some input $w \in \{0,1\}^*$ and uses $s_2(n)$ space to perform its computation. Specifically, $\mathcal{M}$ will do the following:

1. Mark $s_2(|w|)$ work tape cells. If the computation tries to use more space, then $\mathcal{M}$ rejects.
2. Write w to the work tape in the form $w = w_1 w_2$, where $w_1 = 1^*0$.
3. Check that w_2 is a correct encoding of a Turing machine $\mathcal{M}_2$, and simulate the computation of $\mathcal{M}_2$ on the input word w. If the simulation halts and rejects, then $\mathcal{M}$ accepts. Otherwise, $\mathcal{M}$ rejects.

Now, assume that the language $L(\mathcal{M})$ is recognized by some deterministic Turing machine $\mathcal{M}'$ using space $s_1(n)$. Let r denote the number of work tape symbols used by $\mathcal{M}'$, and let $\text{bin}(\mathcal{M}')$ denote the binary representation of the Turing machine $\mathcal{M}'$. Choose some $n \geq |\text{bin}(\mathcal{M}')|$ such that

$$\lceil \log(r) \rceil \cdot s_1(n) < s_2(n). \tag{1}$$

Finally, choose some word $w = 1^j 0 \text{bin}(\mathcal{M}')$, where j is such that $|w| = n$.

By Equation 1, we know that $\mathcal{M}$ can simulate $\mathcal{M}'$ in space $s_2(n)$. Thus, running $\mathcal{M}$ on the input word w is equivalent to simulating the computation of $\mathcal{M}'$ on the input word w. However, if $\mathcal{M}'$ accepts w, then $\mathcal{M}$ will reject w, and vice versa! Therefore, we see that $L(\mathcal{M}) \neq L(\mathcal{M}')$. □

The proof of the space hierarchy theorem uses the familiar technique of diagonalization that we've seen before. Furthermore, note the use of little-o notation in the statement of the theorem. This notation effectively says that any deterministic Turing machine using strictly less than $s(n)$ space cannot recognize the language L.

As a consequence of the space hierarchy theorem, we get the following corollary.

Corollary 14. *For any space constructible function s,*

$$\mathsf{DSPACE}(o(s(n)) \subset \mathsf{DSPACE}(s(n)).$$

We can also adapt the space hierarchy theorem to work for nondeterministic Turing machines with not much more effort.

4.2 Time Hierarchy Theorem

We now turn to the time hierarchy theorem, where the diagonalization argument we'll use will be quite similar to that used in the proof of the space hierarchy theorem, but with one additional snag we must account for: the diagonalizing Turing machine will have a fixed number of work tapes k_1, but it must be capable of simulating Turing machines having any number of work tapes k_2 where $k_1 < k_2$. This simulation can't be done without suffering an impact to the time bound, so the diagonalizing Turing machine will incur a logarithmic slowdown in its computation, which in turn leads to a weaker statement for the time hierarchy theorem. Namely, we require that the time bound be increased by a logarithmic factor to account for the slowdown.

Theorem 15 (Time hierarchy theorem)**.** *For any time constructible function $t(n) \geq n \log(n)$, there exists a language $L \in \mathsf{DTIME}(t(n))$ that cannot be recognized by any deterministic Turing machine using time $o(t(n)/\log(t(n)))$.*

Proof. We want to show that there exists some language L that we can recognize in time $O(t(n))$, but not in time $o(t(n)/\log(t(n)))$.

The proof of this theorem goes through in much the same way as in the proof of Theorem 13. We construct a deterministic Turing machine $\mathcal{M}$ which takes some input $w \in \{0,1\}^*$ and uses $t(n)$ time to perform its computation. Specifically, $\mathcal{M}$ will do the following:

1. Compute $t(n)$ and store the binary representation of the value $\lceil t(n)/\log(t(n))\rceil$ on a work tape. For each computation step, decrement this value by one. If the value stored on this work tape ever reaches 0, then $\mathcal{M}$ rejects.
2. Write w to the work tape in the form $w = w_1 w_2$, where $w_1 = \mathtt{1}^*\mathtt{0}$.
3. Check that w_2 is a correct encoding of a Turing machine $\mathcal{M}_2$, and simulate the computation of $\mathcal{M}_2$ on the input word w. If the simulation halts and rejects, then $\mathcal{M}$ accepts. Otherwise, $\mathcal{M}$ rejects.

Regardless of how long the simulation of $\mathcal{M}_2$ takes to compute, it will be time-bounded by the counter on the work tape, so $\mathcal{M}$ will perform at most $\lceil t(n)/\log(t(n))\rceil$ computation steps on the simulation alone. Since the counter value is stored in binary, it has a length of $\log(t(n)/\log(t(n)))$, which is $O(\log(t(n)))$. This means that the process of updating the counter adds a factor of $\log(t(n))$ to the computation time, and so $\mathcal{M}$ recognizes its language in time $O(t(n))$.

Now, assume that the language $L(\mathcal{M})$ is recognized by some deterministic Turing machine $\mathcal{M}'$ using time $f(n) \in o(t(n)/\log(t(n)))$, and consider what happens when we simulate the computation of $\mathcal{M}'$ on $\mathcal{M}$. Recall that the simulation component of $\mathcal{M}$ takes at most $\lceil t(n)/\log(t(n))\rceil$ computation steps to complete. Since $f(n) \in o(t(n)/\log(t(n)))$, there exists some value n_0 where $c \cdot f(n) < t(n)/\log(t(n))$ for all $n \geq n_0$. Therefore, $\mathcal{M}$ will only be able to complete the simulation of $\mathcal{M}'$ if its input word has length at least n_0.

Choose some word $w = \mathtt{1}^{n_0}\mathtt{0}\text{bin}(\mathcal{M}')$, where $\text{bin}(\mathcal{M}')$ is as defined in the proof of Theorem 13. Since this input has sufficient length, $\mathcal{M}$ will complete its simulation. However, if $\mathcal{M}'$ accepts w, then $\mathcal{M}$ will reject w, and vice versa! Therefore, we see that $L(\mathcal{M}) \neq L(\mathcal{M}')$. □

Again, we get the following corollary as a consequence of the time hierarchy theorem.

Corollary 16. *For any time constructible function t,*

$$\mathsf{DTIME}\left(o\left(\frac{t(n)}{\log(t(n))}\right)\right) \subset \mathsf{DTIME}(t(n)).$$

Interestingly, the nondeterministic analogue to the time hierarchy theorem doesn't require the extra logarithmic term! However, we will not consider the nondeterministic version here.

4.3 Savitch's Theorem

Up to now, the relationships we've seen between time and space complexity classes has been either among like classes—such as comparing DTIME and DTIME, or DSPACE and DSPACE—or going from deterministic to nondeterministic classes—such as comparing DTIME and NTIME, or DSPACE and NSPACE. However, it is possible for us to establish further relationships between these classes to get a better picture of how they interact.

For instance, we know that $\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$ for any function $f(n)$, but what about the other way around? Well, if we could prove the same inclusion but in reverse, then we would also have proved that $\mathsf{P} = \mathsf{NP}$. Since nobody has been able to successfully do this (yet...), we will have to settle for a weaker relationship in the other direction. Namely, we can show that any nondeterministic computation taking $f(n)$ time can also be done by a deterministic machine in time exponential in $f(n)$.

Theorem 17. *For any time constructible function f,*

$$\mathsf{NTIME}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))}).$$

Proof. To prove this result, we must first prove that if f is a space constructible function, then $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$. Since we took f to be time constructible, it is also space constructible.

Let $\mathcal{M}$ be a nondeterministic Turing machine that accepts input words of length n in space $f(n)$. We can modify $\mathcal{M}$ so that, if it accepts its input word, then it erases all of its work tapes and moves all input heads to their leftmost tape cells before accepting. This does not affect the language recognized by $\mathcal{M}$, but it does

result in $\mathcal{M}$ having exactly one accepting configuration. Moreover, we know that an $f(n)$-space-bounded nondeterministic Turing machine can be modified not to repeat configurations as a corollary of Theorem 10. Therefore, the number of configurations of the $f(n)$-space-bounded version of $\mathcal{M}$ is $2^{O(f(n))}$.

Next, let $\mathcal{N}$ be a deterministic Turing machine that, given an input word w of length n, constructs a graph where the vertices correspond to configurations of $\mathcal{M}$ on input w and where there exists a directed edge from vertex v_i to vertex v_j if and only if configuration C_j is reachable from configuration C_i in one computation step. The machine $\mathcal{N}$ checks whether there exists a path of length at most n from the vertex corresponding to the initial configuration C_0 to the vertex corresponding to the unique accepting configuration C_{acc}, and if such a path exists, then $\mathcal{N}$ accepts. In other words, $\mathcal{N}$ solves a specific instance of the problem S-T-Connectivity on the configuration graph of $\mathcal{M}$.

Because $\mathcal{N}$ accepts only those words for which an accepting computation of $\mathcal{M}$ exists, $L(\mathcal{N}) = L(\mathcal{M})$, and $\mathcal{N}$ recognizes its language in time $2^{O(f(n))}$.

Finally, recall that $\mathsf{NTIME}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$ for all functions $f(n)$. Altogether, we have that

$$\mathsf{NTIME}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))}). \qquad \square$$

Turning to space complexity, we might reasonably expect that like the relationship between nondeterministic and deterministic time, there is an exponential gap between nondeterministic and deterministic space. After all, we know that $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$ for all functions f, so Theorem 17 also tells us that $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}(2^{O(f(n))})$.

However, an astonishing result published by the American computer scientist Walter Savitch in 1970 shows that we can do *much* better than an exponential gap—we only need a quadratic difference between nondeterministic and deterministic space!

Theorem 18 (Savitch's theorem)**.** *For any space constructible function $f(n) \geq \log(n)$,*

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}(f(n)^2).$$

Proof. Let $\mathcal{M}$ be a nondeterministic Turing machine that accepts input words of length n in space $f(n) \geq \log(n)$. We construct a deterministic Turing machine $\mathcal{N}$ that takes as input two vertices v_i and v_j from some graph $G = (V, E)$ together with some $k \in \mathbb{N}$ and accepts if vertex v_j is reachable from vertex v_i after following at most k edges. In other words, $\mathcal{N}$ solves the general problem S-T-Connectivity, but unlike in the proof of Theorem 17, here $\mathcal{N}$ takes the graph as input instead of constructing it.

The sticking point here is that we cannot solve this connectivity problem by simply running a breadth-first search or depth-first search on the input graph, since this would require $\Omega(n)$ time where $n = |V|$ and would therefore rule out sublinear functions $f(n)$ from consideration. In order to make our connectivity procedure more space-efficient, we must make use of recursion. Our Turing machine $\mathcal{N}$ will therefore run the following steps:

1. If $k = 1$:
   (a) If $v_i = v_j$ or if $(v_i, v_j) \in E$, then accept. Otherwise, reject.
2. If $k > 1$, then for each vertex $u \in V \setminus \{v_i, v_j\}$:
   (a) Run $\mathcal{N}$ on the input v_i, u, and $\lfloor k/2 \rfloor$.
   (b) Run $\mathcal{N}$ on the input u, v_j, and $\lceil k/2 \rceil$.
   (c) If both computations are accepting, then accept. Otherwise, reject.

Before proceeding, we modify $\mathcal{M}$ as in the proof of Theorem 17, where if $\mathcal{M}$ accepts its input word, then it erases all of its work tapes and moves all input heads to their leftmost tape cells before accepting. In doing so, $\mathcal{M}$ has exactly one accepting configuration C_{acc}. Recall also that, since $\mathcal{M}$ is an $f(n)$-space-bounded Turing machine, it has at most $2^{O(f(n))}$ configurations.

Now, we run $\mathcal{N}$ on the configuration graph of $\mathcal{M}$, where the configuration graph is again defined as in the proof of Theorem 17. We give $\mathcal{N}$ the following input: C_0, the initial configuration of the modified $\mathcal{M}$; C_{acc}, the accepting configuration of the modified $\mathcal{M}$; and the value $2^{c \cdot f(n)}$ for some constant c. If $\mathcal{N}$ accepts, then $\mathcal{M}$ accepts as there must exist some path through the configuration graph from C_0 to C_{acc}, and so $\mathcal{N}$ simulates the computation of $\mathcal{M}$.

It remains to show how much space is used by $\mathcal{N}$ over the course of its computation. Each time the computation of $\mathcal{N}$ recurses, it stores the input values v_i, v_j, and k for later retrieval, and storing these values as their binary representations uses logarithmic space, which is $O(f(n))$. Moreover, each recursive call divides the space used by half. Since we start the computation with the value $k = 2^{c \cdot f(n)}$, the depth of the recursion is $O(\log(2^{c \cdot f(n)}))$, or $O(f(n))$. In total, the amount of space used across all levels of the recursive computation is $O(f(n) \cdot f(n))$, or $O(f(n)^2)$ as required. □

So, what makes Savitch's theorem work? It essentially comes down to one key idea that we mentioned previously: unlike time, space is a resource that can be reused. In our proof, we are reusing space by way of recursion, and testing subgraphs of the configuration graph instead of performing a more intensive search. Savitch's theorem is among the earliest results in complexity theory, and the question of whether the quadratic gap it gives can be improved is long-standing.