St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 3: The Fundamental Complexity Hierarchy
Fall 2023

1 Constructible Functions

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.

1.1 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

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

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 $\text{gap}_i(a, b)$ 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 $L \in \mathsf{DTIME}\big(2^{f(n)}\big)$, 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.

1.2 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 $f\colon \mathbb{N} \to \mathbb{N}$: 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 $\mathtt{1}^n$ on its input tape and writes $\mathtt{1}^{f(n)}$ 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 $\mathtt{1}^n$ on its input tape and writes $\mathtt{1}^{f(n)}$ 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 $\mathtt{1}^n$" 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 $n \geq n_0$,

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

[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.

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

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

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

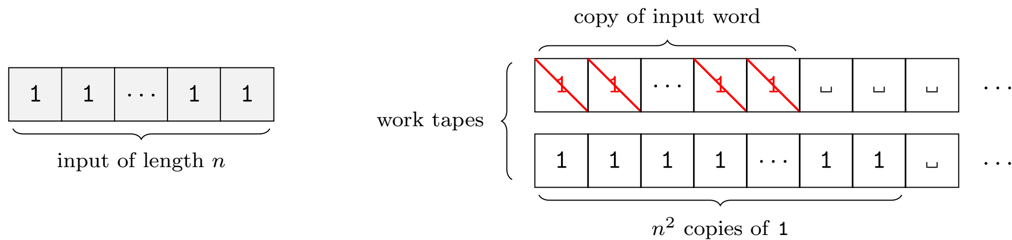
as desired. Via a technical argument omitted here, we can prove that since both g and $f + g$ are time constructible and since $f(n) \geq \epsilon \cdot g(n) + (1 + \epsilon) \cdot n$ for all $n \geq n_0$, f is time constructible as well. □

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.

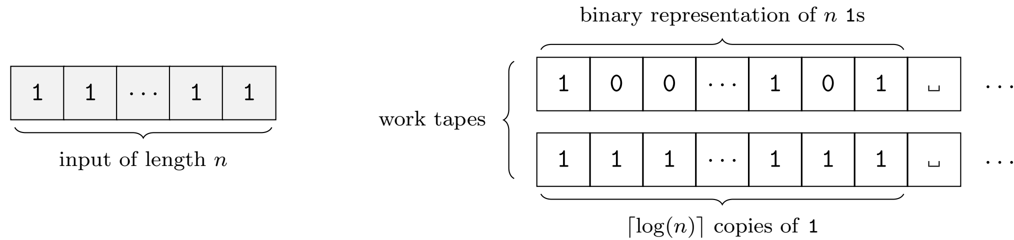


All common functions that grow at least linearly are time constructible, including polynomial, exponential, and factorial functions. What is *not* time constructible, however, is any function that is sublinear, like $\log(n)$. This is because sublinear functions don't give us sufficient time to read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

1.3 Time-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.

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 $\mathsf{DTIME}(t(n))$.

[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.

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.

1.4 Space-Bounded Computations

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. $\square$

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.