St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 4: Time and Space Complexity
Winter 2026

While our four complexity classes are serviceable for talking about time and space complexity, it would become rather clumsy for us to reason about the complexities of problems at the broad level of, say, "deterministic time" or "nondeterministic space". For example, we might not care about specific functions f, but rather about all functions that grow according to some polynomial. If we stuck to our four complexity classes, then we might have to write something like $\mathsf{DTIME}(n) \cup \mathsf{DTIME}(n^2) \cup \mathsf{DTIME}(n^3) \cup \ldots$. For this reason, theoretical computer scientists have devised special notations for common time and space complexity classes.

1 Time Complexity Classes

We begin by considering classifications of running times. Here and in the following space complexity section, we will progress "upwards", starting with complexity classes that are generally considered to be the most efficient and building on these with larger and more difficult-to-solve complexity classes. We will also see that, for each complexity class, there is a deterministic version and a nondeterministic version.

In both time and space, however, we will only be able to scratch the surface of complexity theory. The complexity classes we will discuss here are merely the most well-known inhabitants of the complexity zoo,[1] which is filled with dozens—if not hundreds—of examples of precisely defined classes for every type of problem and solution we might come across.

1.1 Polynomial Time

Our first two time complexity classes are also arguably the most famous of all the classes, in no small part due to the status and popularity of a very famous Millennium Prize problem that has resisted efforts to solve it by computer scientists and the public alike.

With what we know about the growth rates of various functions, we can intuit that polynomial growth rates are "good", while anything above polynomial (such as exponential) is "bad". A polynomial function $f(n)$ doesn't grow particularly quickly as n grows large, which is good if we interpret n to be the size of the input to an algorithm; in this case, the value $f(n)$ then specifies how many operations our algorithm must perform on each symbol of the input, and fewer is always better.

In 1965, the American computer scientist Alan Cobham and the American-Canadian computer scientist Jack Edmonds each made the same observation in separate papers: problems that can be solved in time polynomial in the size of the input can be solved efficiently. This observation spurred a major focus on the study of problems for which there exist efficient (i.e., polynomial-time) algorithms, and this focus led to the formalization of the class of polynomial-time problems.

Definition 1 (The class P)**.** The complexity class P is taken to be

$$\mathsf{P} = \bigcup_{k \geq 0} \mathsf{DTIME}(n^k).$$

One of the great properties of the class P is that it's robust, in the sense that small changes to our model or our algorithm don't affect the larger property of "recognizing in polynomial time". In general, any deterministic

[1] On that note, if you're interested in learning about the vast assortment of time and space complexity classes that have been studied in the literature, you may wish to visit the actual *Complexity Zoo* at `https://complexityzoo.net`.

model of computation that (very broadly speaking) acts like a computer is *polynomially equivalent* to a deterministic Turing machine; that is, we can simulate the computation of that model with a deterministic Turing machine, and this simulation requires only a polynomial amount of additional resources. Thus, we can safely ignore any such differences, since their impact on the running time will be swept up in the overall polynomial aspect of the computation.

However, a Turing machine running in polynomial time does *not* always make that machine the best choice for a given problem. For example, showing that a language is in $\mathsf{DTIME}(n^{100})$ means that there exists a Turing machine that technically recognizes the language in "polynomial time", but running that machine on large inputs would lead to a truly painful wait for an answer. On the other hand, showing that the same language is in $\mathsf{DTIME}(2^{0.01n})$ doesn't give a polynomial-time decision procedure, but it is much better than the alternative.

Another problem arises by us adhering strictly to determinism: it limits the types of languages we're able to recognize in polynomial time. For some problems, we just aren't able to come up with a clever and efficient algorithm. The "naïve approach", or the approach that takes a long (i.e., superpolynomial) amount of time to return an answer, is the best we've got at the moment for such problems.

Fortunately, we can use nondeterminism and the power of guessing to obtain efficient algorithms for some problems, and this produces the nondeterministic version of the class P.

Definition 2 (The class NP)**.** The complexity class NP is taken to be

$$\mathsf{NP} = \bigcup_{k \geq 0} \mathsf{NTIME}(n^k).$$

If you take away one thing from these notes, let it be this:

NP does not mean "non-polynomial"!

Problems in NP can be recognized by a nondeterministic Turing machine in polynomial time, so the abbreviation NP stands for "nondeterministic polynomial time".

Note that our definition of NP is sometimes referred to as the "decider" definition, since we've defined the class in terms of a nondeterministic Turing machine that *finds* (or *decides*) an answer to its problem. We could alternatively frame the class NP in terms of a machine that *verifies* the validity of a claimed solution, and it turns out that these two definitions are equivalent.

1.2 Linear Exponential Time

Taking one step up from the class of polynomial-time problems brings us to our first example of *superpolynomial*-time problems; specifically, problems that require exponential time. This first step doesn't give us the full class of exponential-time problems, but rather a sort-of "baby exponential" class where the exponent is bounded by some linear expression in n.

Definition 3 (The class E)**.** The complexity class E is taken to be

$$\mathsf{E} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{cn}).$$

Naturally, we can also define the nondeterministic version of this class.

Definition 4 (The class NE)**.** The complexity class NE is taken to be

$$\mathsf{NE} = \bigcup_{c \geq 1} \mathsf{NTIME}(2^{cn}).$$

While the classes E and NE are admittedly not as exciting as the more general class of exponential-time problems we're about to define, they do play their own important roles in theory and come with their own special properties, and they allow us to make a fuzzy distinction between "problems we might be able to solve feasibly in some cases" and "problems that seem fairly hopeless".

1.3 Exponential Time

If we remove the restriction that the exponential running time of some problem has a linear-bounded exponent, then we obtain the aforementioned class of exponential-time problems. Here, the exponent may now be bounded by any polynomial expression in n, meaning that the running time has the potential to grow incredibly fast.

Definition 5 (The class EXP)**.** The complexity class EXP is taken to be

$$\mathsf{EXP} = \bigcup_{c \geq 1} \mathsf{DTIME}\left(2^{n^c}\right).$$

Again, we define the nondeterministic version of this class in the usual way.

Definition 6 (The class NEXP)**.** The complexity class NEXP is taken to be

$$\mathsf{NEXP} = \bigcup_{c \geq 1} \mathsf{NTIME}\left(2^{n^c}\right).$$

Just by their definitions, the classes EXP and NEXP are very large indeed, and many real-world problems that are second nature to humans—such as playing checkers or chess—are known to belong to EXP.

Interestingly, although EXP and NEXP seem like they would perhaps be at the top of the complexity hierarchy, this is not the case! We can define even larger classes containing problems with even more lengthy runtimes via tetration: the class 2EXP corresponds to doubly exponential running times ($2^{2^{n^c}}$), the class 3EXP corresponds to triply exponential running times ($2^{2^{2^{n^c}}}$), and so on.

From this infinite hierarchy of exponential-time classes, we can obtain an entirely new class called ELEMENTARY by taking the union of all kEXP classes for $k \geq 1$. Despite the name, problems contained in the class ELEMENTARY are far from elementary for us to solve. And incredibly, despite how huge this class is, it doesn't even come close to encompassing *all* computable problems! All of this hopefully serves to illustrate exactly how large the study of time complexity is on its own, and why theoretical computer scientists take such an intense interest in studying and classifying problems.

2 Space Complexity Classes

We now turn to classifications of work space, where one important observation sets space complexity apart from time complexity: work space can be reused! While expending one computation step means it's gone forever, we can write and rewrite to the same tape cell as many times as we please. Thus, while some problems might take a very long amount of time to solve, those same problems could be very efficient in space, and so we must take care to consider both measures when we tackle such problems.

2.1 Logarithmic Space

Recall that, in our definition of work space, we only count the number of cells used on work tapes. Even though we read the entire input of length n on the input tape, these n cells do not count toward our work space measure. Therefore, it's possible for us in some cases to achieve sublinear space complexity bounds, and our first space complexity class reflects this: the class of logarithmic-space problems.

Definition 7 (The class L)**.** The complexity class L is taken to be

$$\mathsf{L} = \mathsf{DSPACE}(\log(n)).$$

Of course, we can also define the same class, but for nondeterministic computations.

Definition 8 (The class NL)**.** The complexity class NL is taken to be

$$\mathsf{NL} = \mathsf{NSPACE}(\log(n)).$$

The classes L and NL are unique to the study of space complexity. Naturally, we couldn't define analogues of these classes when we were talking about time complexity, since there's no way for us to read the entire input in less than linear time.

Additionally, the classes of problems that use a logarithmic amount of space are of great interest to theoretical and practical researchers alike, since a variety of commonplace problems fall into this class. Indeed, logarithmic space grants us *just* enough space to perform common tasks, such as remembering the position of an input head at a particular tape cell. A position can be represented by a number, which can be encoded in binary as a bit string with a length logarithmic to the magnitude of the number itself. Using this same encoding idea, we can even implement techniques to perform basic arithmetic operations on numbers in logarithmic space.

2.2 Polynomial Space

One level above the class of logarithmic-space problems is the space complexity analogues of our classes P and NP, which we obtain by restricting the amount of work tape space available to some amount polynomial in the size of the input.

Definition 9 (The class PSPACE)**.** The complexity class PSPACE is taken to be

$$\mathsf{PSPACE} = \bigcup_{k \geq 0} \mathsf{DSPACE}(n^k).$$

In keeping with our other definitions, we will define the nondeterministic version of the PSPACE class here. However, we won't pay too much mind to this complexity class, as we will later see an interesting result that will reveal we don't really need to define this class independently.

Definition 10 (The class NPSPACE)**.** The complexity class NPSPACE is taken to be

$$\mathsf{NPSPACE} = \bigcup_{k \geq 0} \mathsf{NSPACE}(n^k).$$

The class PSPACE may seem trivial, since most efficient computations use a polynomial amount of space simply by virtue of the fact that they run in polynomial time. However, moving beyond those problems that take polynomial time, PSPACE remains of interest because it serves as a refinement of sorts between difficult-to-solve (i.e., exponential-time) problems that may take up a lesser amount of space and those problems that require an exponential amount of both time and space.

For example, the Boolean satisfiability problem (or Satisfiability) is a well-known problem that most people believe can only be solved in superpolynomial time; that is, Satisfiability is generally believed not to be in P. However, the brute-force algorithm to solve instances of Satisfiability only requires a linear amount of space to check all possible truth value assignments, and so Satisfiability is in PSPACE!

A wide variety of board games, such as tic-tac-toe, reversi, and Connect Four, are in PSPACE as well. Even the video game Super Mario Bros. falls into this class!

2.3 Exponential Space

Speaking of exponential amounts of space, we arrive at our final pair of complexity classes that we will focus on in this course. If we allow our model to use an amount of space exponential in the size of the input, then we obtain the space complexity analogues of the classes EXP and NEXP.

Definition 11 (The class EXPSPACE)**.** The complexity class EXPSPACE is taken to be

$$\mathsf{EXPSPACE} = \bigcup_{c \geq 1} \mathsf{DSPACE}\left(2^{n^c}\right).$$

As you might expect by this point, we define the nondeterministic version of EXPSPACE as we did previously. However, the future "interesting result" we alluded to when we introduced NPSPACE applies here as well, meaning that we don't strictly need to define this class on its own either.

Definition 12 (The class NEXPSPACE)**.** The complexity class NEXPSPACE is taken to be

$$\mathsf{NEXPSPACE} = \bigcup_{c \geq 1} \mathsf{NSPACE}\left(2^{n^c}\right).$$

The class EXPSPACE contains some truly tough, seemingly impossible-to-solve problems, but it also contains some problems that are very easy for humans to solve; for example, deciding whether or not an arithmetic statement involving real numbers, addition ($+$), and comparison ($<$) is true requires an exponential amount of space. But like EXP, the class EXPSPACE is far from the top of our complexity hierarchy!

3 The Fundamental Complexity Hierarchy

Using what we have learned from past lectures, we now have enough machinery to prove all known relations between the time and space complexity classes we have defined and, in doing so, to establish the fundamental complexity hierarchy. We begin by proving the most elementary relations between complexity classes, which follow immediately from what we know about DTIME and NTIME and about DSPACE and NSPACE.

Theorem 13. $\mathsf{L} \subseteq \mathsf{NL}$.

Proof. Follows immediately from the fact that $\mathsf{DSPACE}(\log(n)) \subseteq \mathsf{NSPACE}(\log(n))$. □

Theorem 14. *The following inclusions hold:*

1. $\mathsf{P} \subseteq \mathsf{NP}$.
2. $\mathsf{EXP} \subseteq \mathsf{NEXP}$.

Proof. Follows immediately from the fact that $\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$. □

Next, we will draw connections between nondeterministic time complexity classes and deterministic space complexity classes. To obtain these particular inclusions, we must prove an intermediate result relating the classes NTIME and DSPACE.

We already know that $\mathsf{NTIME}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$, and we further proved that $\mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(2^{O(f(n))})$. However, we can greatly narrow this exponential gap between NTIME and DSPACE to obtain a much-improved relationship.

Theorem 15. *The following inclusions hold:*

1. $\mathsf{NP} \subseteq \mathsf{PSPACE}$.
2. $\mathsf{NEXP} \subseteq \mathsf{EXPSPACE}$.

Proof. Let f be a time constructible function and let $\mathcal{M}$ be a nondeterministic Turing machine that accepts words of length n in time $f(n)$. Since f is time constructible, we can define a deterministic Turing machine $\mathcal{N}$ that simulates $f(n)$ steps of each computation path of $\mathcal{M}$ and accepts if $\mathcal{M}$ accepts. If no accepting computation path exists, then $\mathcal{N}$ rejects. Each of these simulated computations is performed using the same space, and we can keep track of which computation path we're simulating by maintaining an $f(n)$-bit

counter on a work tape. Therefore, $\mathcal{N}$ simulates the computation of $\mathcal{M}$ in space $f(n)$, and so $\mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$. Both of the inclusions listed in the statement of the theorem follow directly. □

We turn now to an important consequence of Savitch's theorem. Recall that, when we defined space complexity classes like NPSPACE and NEXPSPACE, we mentioned that we wouldn't pay too much mind to such classes because of an "interesting result" that meant we didn't need to define such classes independently. That interesting result, as we now know, is none other than Savitch's theorem.

Since we know that the class DSPACE is contained within the class NSPACE, and since Savitch's theorem tells us that NSPACE is contained within "DSPACE squared", we obtain equality between deterministic and nondeterministic space complexity classes that use at least polynomial space!

Theorem 16. *The following equalities hold:*

1. $\mathsf{PSPACE} = \mathsf{NPSPACE}$.
2. $\mathsf{EXPSPACE} = \mathsf{NEXPSPACE}$.

Proof. Follows immediately from Savitch's theorem and the fact that $\mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$. □

These results are effectively the answers to the space complexity equivalent of the P vs. NP problem. As a consequence, we will speak no further of the classes NPSPACE or NEXPSPACE on their own.

Note, however, that Savitch's theorem does not also allow us to cast away the class NL, since it does not establish that $\mathsf{L} = \mathsf{NL}$. This is because if f is at least a polynomial function, then f^2 is also at least polynomial, but the square of a logarithmic function is not polynomial. Thus, the only thing Savitch's theorem can tell us is that $\mathsf{NL} \subseteq \mathsf{DSPACE}(\log(n)^2)$, and clearly we have that $\mathsf{L} \neq \mathsf{DSPACE}(\log(n)^2)$. Indeed, the question of whether $\mathsf{L} = \mathsf{NL}$ is another long-standing problem in complexity theory.

It remains for us to prove two more inclusions individually, but fortunately, both of these relationships follow as a consequence of a prior result we investigated.

Theorem 17. $\mathsf{NL} \subseteq \mathsf{P}$.

Proof. Follows from the fact that $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$ for any space constructible function f. Namely, if we take $f(n) = \log(n)$, then we have that

$$\mathsf{NSPACE}(\log(n)) \subseteq \mathsf{DTIME}(2^{\log(n)}) = \mathsf{DTIME}(n) \in \mathsf{P}.$$ □

Theorem 18. $\mathsf{PSPACE} \subseteq \mathsf{EXP}$.

Proof. Follows from the fact that $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$ for any space constructible function f, together with the fact that $\mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$ for any function f. Namely, if we take $f(n) = n^k$ for any $k \geq 0$, then we have that

$$\mathsf{DSPACE}(n^k) \subseteq \mathsf{NSPACE}(n^k) \subseteq \mathsf{DTIME}(2^{n^k}) \in \mathsf{EXP}.$$ □

Going one step further, we can establish a handful of strict inclusion results between certain complexity classes, and the strictness of each of these results is due to the time and space hierarchy theorems we studied in an earlier section.

Theorem 19. $\mathsf{P} \subset \mathsf{EXP}$.

Proof. Follows immediately from the time hierarchy theorem. □

Theorem 20. $\mathsf{NL} \subset \mathsf{PSPACE} \subset \mathsf{EXPSPACE}$.

Proof. Follows immediately from the space hierarchy theorem. □

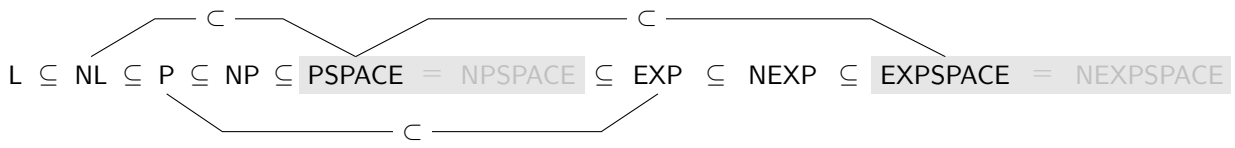

Figure 1: The fundamental complexity hierarchy.

All told, our fundamental complexity hierarchy is depicted in Figure 1.

As a consequence of Theorems 19 and 20, we know that in the chains of inclusions $\mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$ and $\mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP} \subseteq \mathsf{EXPSPACE}$, *at least* one of the inclusions in each chain must be strict. However, nobody knows which one—or indeed, which ones—should be! The general consensus among complexity theorists is that *all* of the inclusions in these two chains are strict.