St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 2: Time and Space Complexity
Fall 2023

1 Resource-Bounded Computations

Often, when we discuss the complexity of a problem, we don't care about the specific amount of resources a Turing machine needs to solve that problem. If we have two machines that solve the same problem of size n, where Machine 1 makes $3n$ computation steps and Machine 2 makes $2n$ computation steps, both machines perform on par as the value of n grows very large. That is, the difference between the constants 2 and 3 becomes negligible if n is much larger than either of those constants.

This same idea is a fundamental tenet of algorithm analysis: we want to simplify and abstract away as much as possible, until all we're left with is a general comparison between the input size of the problem and the performance of an algorithm for that problem. Often, this simplification results in us focusing only on the highest-order term in the running time of the algorithm, resulting in a process known as *asymptotic analysis*. Given an input instance of size n, we can intuit that a Turing machine making "on the order of" n computation steps will finish faster than a Turing machine making "on the order of" n^2 computation steps. It doesn't matter if the first machine makes exactly $10n + 50$ steps while the second machine makes exactly $0.001n^2 + 20n + 5$ steps; as n grows larger, the quadratic term is guaranteed to outpace the linear term.

We can evaluate the complexity of a problem in two ways: by measuring the amount of time required for a Turing machine to solve the problem, and by measuring the amount of space used by a Turing machine over the course of solving the problem. In both cases, the principles of asymptotic analysis allow us to simplify the evaluation greatly, measuring the complexity in terms of only the highest-order term while ignoring lower-order and constant terms. Additionally, we often focus on the amount of time or space used in the worst case on an input of a given length n, as this gives us a nice way of obtaining upper bounds on the problem we're considering. (Lower bounds are a very different kind of beast that we won't consider here.)

1.1 Deterministic Running Time and Work Space

If $\mathcal{M}$ is a deterministic Turing machine, then we can define the *running time* of $\mathcal{M}$ as follows.

Definition 1 ($t(n)$-time deterministic Turing machine)**.** Given a k-tape deterministic Turing machine $\mathcal{M}$, we say that $\mathcal{M}$ is a $t(n)$-time Turing machine if, on any input instance of length n, $\mathcal{M}$ makes at most $t(n)$ computation steps before halting.

For the sake of convenience, we will assume that all of our computations scan the entire input of length n; that is, we will not consider computations that only partially read their input. We will also simplify the function t in the case where it is real-valued by taking the floor of the function to act as the bound. Thus, any time bound $t(n)$ is more properly written as the function $\max\{n, \lfloor t(n) \rfloor\}$.

We can define the *work space* of $\mathcal{M}$ in a similar way as before.

Definition 2 ($s(n)$-space deterministic Turing machine)**.** Given a k-tape deterministic Turing machine $\mathcal{M}$, we say that $\mathcal{M}$ is an $s(n)$-space Turing machine if, on any input instance of length n, $\mathcal{M}$ reads at most $s(n)$ work tape cells.

With this definition, though, we have two points that are worth discussing. First, comparing Definition 1 to Definition 2, we see that we no longer need the condition that $\mathcal{M}$ must halt in order for us to discuss work space. This is because it's possible for a non-halting computation to read only a finite number of tape cells,

and so we may still speak of work space bounds even if a computation doesn't halt. On the other hand, it doesn't make sense to speak of the "running time" of a non-halting computation, since such a computation naturally uses infinite time.

Second, observe that Definition 2 only requires us to count the number of work tape cells read by the Turing machine. This is an interesting restriction—why should we focus only on the work tapes and not on the input tape? Well, recall our assumption that every computation scans the entire input of length n. If we included the input tape in our work space measurement, then reading the entire input would impose an artificial minimum on all of our measurements of at least n tape cells, and we could never achieve sublinear space complexity bounds! Since sublinear space complexity results in very efficient algorithms, we want to hold onto this desirable property, and so we only care about the usage of our work tapes.[1]

Finally, of course, we want our work space measurements to make sense even if our computation doesn't use the work tapes at all. If we have a k-tape Turing machine, then we have $k-1$ work tapes at our disposal. We will require that each of these work tapes have at least one tape cell available to be read, giving us a total of at least $k-1$ work tape cells to use. Thus, any space bound $s(n)$ is more properly written as the function $\max\{k-1, s(n)\}$. Note that we don't need to take the floor of the function as we did with time bounds, since $s(n)$ counts discrete tape cells and can't be real-valued.

1.2 Nondeterministic Running Time and Work Space

We can adapt our definitions of running time and work space to work for nondeterministic Turing machines as well. However, here we must take care to account for the fact that nondeterministic Turing machines have computations that branch whenever the model makes a "guess", and multiple branches may correspond to multiple accepting computations. Thus, we can no longer talk about the time or space used in *the* accepting computation, but rather in the *best* accepting computation, where we measure "best" according to two different metrics depending on whether we're talking about time or space.

When we're dealing with a nondeterministic Turing machine, we measure the amount of time used in exactly the same way as with the deterministic model. However, some computation branches may take differing amounts of time, while others may run for an infinite amount of time. Thus, the nondeterministic running time is taken to be the upper bound of the *shortest* accepting computation across all input words of a certain length.

Definition 3 ($t(n)$-time nondeterministic Turing machine)**.** Given an input word w, the computation time of a k-tape nondeterministic Turing machine $\mathcal{M}$ is the number of computation steps in the shortest accepting computation of $\mathcal{M}$ if it accepts w, or 1 otherwise. We then say that $\mathcal{M}$ is a $t(n)$-time Turing machine where $t(n)$ is the maximum computation time of $\mathcal{M}$ on any input of length n.

We likewise measure space usage in the same way for both deterministic and nondeterministic machines, but due to us again needing to account for differences across computation branches, nondeterministic work space is taken to be the upper bound of the *smallest* number of work tape cells used in an accepting computation on any input word of a certain length.

Definition 4 ($s(n)$-space nondeterministic Turing machine)**.** Given an input word w, the computation space of a k-tape nondeterministic Turing machine $\mathcal{M}$ is the smallest number of work tape cells read in an accepting computation of $\mathcal{M}$ if it accepts w, or 1 otherwise. We then say that $\mathcal{M}$ is an $s(n)$-space Turing machine where $s(n)$ is the maximum computation space of $\mathcal{M}$ on any input of length n.

2 Basic Time and Space Complexity Classes

Now that we're able to reason about the running time and work space of a Turing machine, we can classify languages recognized by Turing machines in terms of the amount of time or space it takes to recognize that

[1] If we were considering single-tape Turing machines instead of k-tape Turing machines, we would need to adjust our definition of work space to suit the model; say, by counting the number of distinct tape cells read during the computation. In doing so, we could also no longer assume that the entire input is read from the single tape.

language. This gives us our first set of rudimentary complexity classes, which we will build upon and use to define further complexity classes.

Definition 5 (The classes DTIME and DSPACE)**.** Given a function f, the complexity class $\mathsf{DTIME}(f(n))$ is taken to be

$$\mathsf{DTIME}(f(n)) = \{L \mid L \text{ is a language recognized by a } O(f(n))\text{-time deterministic Turing machine}\}$$

and the complexity class $\mathsf{DSPACE}(f(n))$ is taken to be

$$\mathsf{DSPACE}(f(n)) = \{L \mid L \text{ is a language recognized by a } O(f(n))\text{-space deterministic Turing machine}\}.$$

Definition 6 (The classes NTIME and NSPACE)**.** Given a function f, the complexity class $\mathsf{NTIME}(f(n))$ is taken to be

$$\mathsf{NTIME}(f(n)) = \{L \mid L \text{ is a language recognized by a } O(f(n))\text{-time nondeterministic Turing machine}\}$$

and the complexity class $\mathsf{NSPACE}(f(n))$ is taken to be

$$\mathsf{NSPACE}(f(n)) = \{L \mid L \text{ is a language recognized by a } O(f(n))\text{-space nondeterministic Turing machine}\}.$$

Note that, in the definitions of each of these rudimentary complexity classes, we use Big-O notation to indicate that the function f acts as an asymptotic upper bound on the running time or work space.

Even with just these four complexity classes, we can begin to construct a basic hierarchy of relationships showing how each class relates to the others.

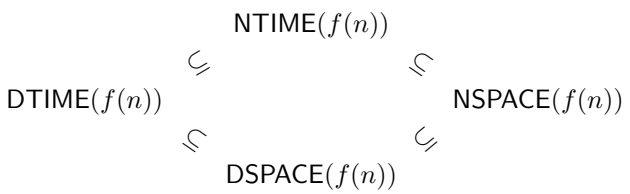
Theorem 7. *The following statements hold for any function f:*

1. $\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$*;*
2. $\mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$*;*
3. $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$*; and*
4. $\mathsf{NTIME}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$.

Proof. Statements 1 and 2 follow immediately from the fact that any language recognized by a deterministic Turing machine is also recognized by a nondeterministic Turing machine that doesn't use nondeterminism.

Statements 3 and 4 follow immediately from the observation that, in one computation step, a Turing machine can read only one cell of its input tape. Therefore, in $f(n)$ computation steps, a Turing machine can visit at most $f(n)$ input tape cells. □

If you prefer to think visually, the following diagram depicts the same relationships as in Theorem 7:



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.

2.1 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,[2] 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.

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 8 (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 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.

[2] 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`.

Definition 9 (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.

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 10 (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 11 (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".

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 12 (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 13 (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.

2.2 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.

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 14 (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 15 (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.

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 16 (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 17 (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!

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 18 (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 19 (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!

2.3 Complement Classes

Let's recall for a moment how a Turing machine solves a problem. Every accepted input instance belongs to the language of the Turing machine, and this intuitively corresponds to the Turing machine giving us a "yes" answer for that instance. For example, if we have a Turing machine $\mathcal{M}$ whose language is all even-length binary words and we give the word `01101001` to $\mathcal{M}$, it will accept it. Effectively, $\mathcal{M}$ has told us "yes, this input word is binary and it has even length". We can then go one step further and classify our problems (and languages) into the complexity classes we defined earlier.

But what of the "no" answers? If we gave the word `010` as input to our even-length-checking Turing machine, it would not accept it. Indeed, we wouldn't get any information about the word from $\mathcal{M}$ other than "no, this input word is either not binary or not of even length".

If we wanted to check specifically whether some input instance was *not* an even-length binary word, we would need to construct a brand new Turing machine recognizing the *complement* language. Fortunately, in this example, we can simply swap the accepting and rejecting states of our original Turing machine $\mathcal{M}$ so that original "no"-answers become "yes"-answers and vice versa.

Where things become interesting from a complexity-theoretic perspective is when we classify these complement problems (and complement languages) into complexity classes. Placing a complement language into a complexity class is not as simple as simply taking the complement of the complexity class itself; for example, the complement of the class P contains a huge number of languages, not all of which are complements of languages within P. Instead, we must take a more refined approach: for each complexity class C, we define a *complement class* coC containing all and only those languages that are complements of anything already in the class C.

Definition 20 (Complement class)**.** For any complexity class C, the class of complements of languages in C is denoted coC and is taken to be

$$\mathsf{coC} = \{\Sigma^* \setminus L \mid L \in \mathsf{C}\}.$$

Note that, in our definition of a complement class, we implicitly assume that each language $L \in \mathsf{C}$ shares a common alphabet Σ with which we can define the complement language $\overline{L} = \Sigma^* \setminus L$. This is just a technicality; we can take as our alphabet, for instance, the set of all symbols that appear in some word of L.

Immediately from our definition, we get all kinds of new complexity classes like coP, coNP, coL, coNL, and so on, and we can relate all of these complement classes to our original set of classes. However, we need not concern ourselves with *all* of the complement classes, and we can actually narrow down our focus to just the nondeterministic classes. This is because of one easy-to-prove fact: deterministic models of computation are closed under complement.

Theorem 21. *For any deterministic complexity class C, we have that $\mathsf{C} = \mathsf{coC}$.*

Proof. Let $\mathcal{M}$ be a deterministic Turing machine recognizing some language L that belongs to a complexity class C. We can create a deterministic Turing machine $\mathcal{M}'$ recognizing the complement language $\overline{L}$ by swapping the accepting and rejecting states of $\mathcal{M}$, and since the running time and work space of $\mathcal{M}'$ is unaffected, this language necessarily belongs to the complexity class coC. □

As consequences of Theorem 21, we get that $\mathsf{P} = \mathsf{coP}$, $\mathsf{E} = \mathsf{coE}$, $\mathsf{EXP} = \mathsf{coEXP}$, $\mathsf{L} = \mathsf{coL}$, $\mathsf{PSPACE} = \mathsf{coPSPACE}$, and $\mathsf{EXPSPACE} = \mathsf{coEXPSPACE}$. Therefore, all we really need to focus on is the nondeterministic complexity classes. Unfortunately, this is where things get a bit more difficult.

Since nondeterministic computations make "guesses", they are not always guaranteed to be symmetric in the same way that deterministic computations are; that is, we cannot simply swap the accepting and rejecting states of a nondeterministic Turing machine, since some nondeterministic computation *branches* may be accepting while others may not be. Thus, nondeterministic models of computation—and nondeterministic complexity classes—aren't necessarily closed under complement.

What's more, if we knew whether some nondeterministic complexity classes were closed under complement, then we could solve some huge open problems in computer science. For instance, if $\mathsf{P} = \mathsf{NP}$, then we would

know that $\mathsf{NP} = \mathsf{coNP}$ as this follows from $\mathsf{P} = \mathsf{coP}$. Therefore, if we showed that $\mathsf{NP} \neq \mathsf{coNP}$, then we would prove that $\mathsf{P} \neq \mathsf{NP}$!

Luckily for us, not all is lost. As it turns out, these difficulties only pertain to nondeterministic *time* complexity classes. If we turn our attention to nondeterministic *space* complexity classes, a groundbreaking result due to the American computer scientist Neil Immerman and the Slovak computer scientist Róbert Szelepcsényi shows that we always have closure under complement.

Theorem 22 (Immerman–Szelepcsényi theorem)**.** $\mathsf{NL} = \mathsf{coNL}$.

Proof. Omitted. □

The Immerman–Szelepcsényi theorem resolves what was known as the *second LBA problem*, where "LBA" refers to the *linear bounded automaton* model—essentially, a restricted form of a Turing machine. The *first LBA problem* asks whether $\mathsf{DSPACE}(n) = \mathsf{NSPACE}(n)$; this problem has remained open ever since it was introduced by the Japanese linguist Sige-Yuki Kuroda in 1964, so it seems quite difficult to answer. Indeed, when Immerman and Szelepcsényi independently announced their proofs in 1987, the outcome took the complexity world by surprise, as many people had believed that NL and coNL were not equal.

Even though the statement of the theorem as we present it here is framed in terms of nondeterministic logarithmic space, we can use a *padding argument* to show that any nondeterministic space complexity class using at least $\log(n)$ space is equal to its complement. A padding argument allows us to establish a relationship between some complexity class C and some other[3] complexity class C' by taking a language $L \in \mathsf{C}$ and padding each word in L with a certain number of extraneous symbols, thus making the padded language "fit" into the complexity class C'. By such an argument (which we won't get into here), we can similarly show that $\mathsf{NPSPACE} = \mathsf{coNPSPACE}$ and $\mathsf{NEXPSPACE} = \mathsf{coNEXPSPACE}$.

3 Speedup and Compression

When we defined our basic complexity classes DTIME, DSPACE, NTIME, and NSPACE, we said that these classes consisted of all (non)deterministic Turing machines that used $O(f(n))$ time or space for some function f. Our use of Big-O notation highlighted that, in practice, we only care about obtaining upper bounds on the amount of time or space being used over the course of some computation—we need not pay any mind to constants or lower-order terms.

We know that using Big-O notation in our definitions has no impact on our measurement of time or space complexity by virtue of the fact that, given any Turing machine, we can change its tape alphabet to pack more symbols into individual tape cells. In doing so, we effectively reduce the time or space being used by the Turing machine by some constant, which allows us to safely disregard such factors.

This idea of packing more symbols into tape cells is formalized in two different ways by the *linear speedup theorem* and the *tape compression theorem*, whether our focus is on time or on space. The proofs of both theorems are similar, though the linear speedup theorem for time complexity is somewhat more technical due to the fact that we need to properly encode multiple tape symbols and handle the behaviour of the input head as it reads our new tape alphabet. Therefore, let's get the harder proof out of the way first.

The linear speedup theorem tells us that if the input head of a Turing machine does not move very far during some part of its computation, then we can condense many moves across few tape cells into a single computation step on a new Turing machine. This can be achieved by focusing on the *neighbourhood* of symbols being read by the new Turing machine; that is, the current symbol being read itself, together with the symbols to the left and to the right of the symbol being read.

[3]Note that the meaning of the word "other" here depends on whether the relationship we're establishing is an equality or an inequality. Typically, equalities (or collapses) between complexity classes go upward, and hence we would take our "other" class to be larger, while inequalities (or separations) between complexity classes go downward, and hence we would take our "other" class to be smaller.

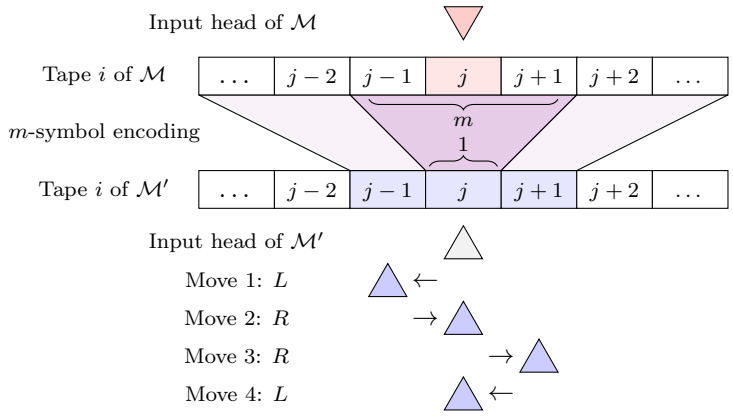
Theorem 23 (Linear speedup theorem)**.** *Suppose that a language L is recognized by a $t(n)$-time k-tape deterministic Turing machine $\mathcal{M}$, where $k \geq 2$ and $n \in o(t(n))$. Then, for any constant $0 < c < 1$, there exists a $c \cdot t(n)$-time k-tape deterministic Turing machine $\mathcal{M}'$ also recognizing L.*

Proof. Let n and c be as defined in the statement of the theorem, and choose a value m such that $mc \geq 16$. The idea behind the proof is that the ith tape of our new Turing machine $\mathcal{M}'$ will encode the contents of the ith tape of $\mathcal{M}$ using a larger tape alphabet in such a way that eight moves of the input head of $\mathcal{M}'$ will simulate at least m moves of the input head of $\mathcal{M}$.

First, $\mathcal{M}'$ scans the input tape of $\mathcal{M}$ and copies the contents onto one of its work tapes in such a way that m symbols on the input tape of $\mathcal{M}$ are written as one symbol on the work tape of $\mathcal{M}'$. After the copying process is complete, $\mathcal{M}'$ moves its input head back to the leftmost cell of this work tape. Overall, this requires $\mathcal{M}'$ to make $n + \lceil n/m \rceil$ input head moves.

(Note that, for the remainder of the proof, $\mathcal{M}'$ will use this work tape as its "input tape" and will use its original input tape as a "work tape".)

The Turing machine $\mathcal{M}'$ now simulates the computation of $\mathcal{M}$ in the following way. Suppose that the input head of tape i of $\mathcal{M}$ is currently reading cell j. The machine $\mathcal{M}'$ scans the neighbourhood of cell j by making a sequence of four moves: L, R, R, and L. In this way, $\mathcal{M}'$ now knows the contents of cells $j-1$, j, and $j+1$ on its tape.

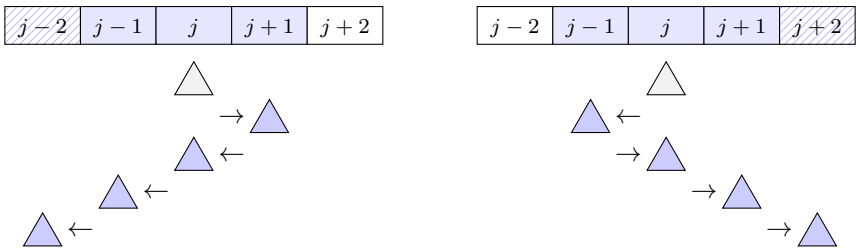


Then, $\mathcal{M}'$ uses its finite state control to determine what the contents of this neighbourhood will be at the moment the input head moves out of the neighbourhood, which occurs after $\mathcal{M}$ makes at least m input head moves due to our tape alphabet encoding. This results in one of two cases:

- If $\mathcal{M}$ halts or accepts its input word before any of its tape heads move out of the tape cells making up the neighbourhood, then $\mathcal{M}'$ similarly halts or accepts.
- Otherwise, $\mathcal{M}'$ makes four more input head moves to update the contents of the tape cells within the neighbourhood, and then positions its input head over the tape cell corresponding to the neighbourhood of the new tape cell being read by $\mathcal{M}$ at the next computation step.

In total, $\mathcal{M}'$ makes eight input head moves to simulate at least m input head moves of $\mathcal{M}$.

The final four of these eight input head moves of $\mathcal{M}'$ are depicted in the following figures, where the left figure shows the input head moving out of the neighbourhood to the left and the right figure shows it moving out to the right.



The running time of $\mathcal{M}'$ is given by the expression

$$T_{\mathcal{M}'} \leq (n + \lceil n/m \rceil) + 8\lceil t(n)/m \rceil,$$

where the first additive term comes from the tape alphabet encoding step and the second term comes from the simulation step. Since $\lceil x \rceil \leq x + 1$ for all x, we can rewrite this expression as

$$T_{\mathcal{M}'} \leq (n + n/m) + 8t(n)/m + 9.$$

Since we have that $n \in o(t(n))$, by the definition of little-o notation, we know that for all constants d, there exists some n_0 such that for all values $n \geq n_0$, $dn \leq t(n)$. Rewriting this expression to isolate n, we get that $n < t(n)/d$. Moreover, observe that for all $n \geq 9$, we have that $n + 9 \leq 2n$. Therefore, for all $d > 0$ and for all $n \geq \max\{9, n_0\}$, we have that

$$\begin{aligned} T_{\mathcal{M}'} &\leq 2t(n)/d + t(n)/md + 8tn/m \\ &= t(n)\,(2/d + 1/md + 8/m) \\ &\leq t(n)\,(32/16d + c/16d + 8c/16) \quad \text{(because } mc \geq 16\text{)} \end{aligned}$$

Finally, we want to choose the constant d such that $32/16d + c/16d + 8c/16 \leq c$. We can rewrite this inequality in the following way:

$$\begin{aligned} 32/16d + c/16d + 8c/16 &\leq c \\ 32 + c + 8cd &\leq 16cd \\ 32 + c &\leq 8cd \\ (32 + c)/8c &\leq d \\ 4/c + 1/8 &\leq d. \end{aligned}$$

Therefore, we may take the constant d to be such that $d \geq 4/c + 1/8$. Altogether, for all $n \geq \max\{9, n_0\}$, the Turing machine $\mathcal{M}'$ makes at most $c \cdot t(n)$ input head moves, and so the running time of $\mathcal{M}'$ is at most $c \cdot t(n)$. □

Note that we needed the assumption $n \in o(t(n))$ in the statement of the linear speedup theorem in order to ensure that $c \cdot t(n) \geq n$ for all but a finite number of values of n. Specifically, this finite number of values corresponds to those input words having a length shorter than $\max\{9, n_0\}$, and we can handle these input words by encoding them directly into the finite state control of $\mathcal{M}'$.

We now turn to applying a similar idea to space complexity and, fortunately, the proof in this case is more straightforward. The tape compression theorem uses the same notion of scanning neighbourhoods of tape cells, but goes about defining these neighbourhoods in a slightly different way: it takes advantage of the fact that our definition of space complexity does not rely on any properties of the tape alphabet, thereby allowing us to create and use a larger tape alphabet to attain an improvement on our work space measure.

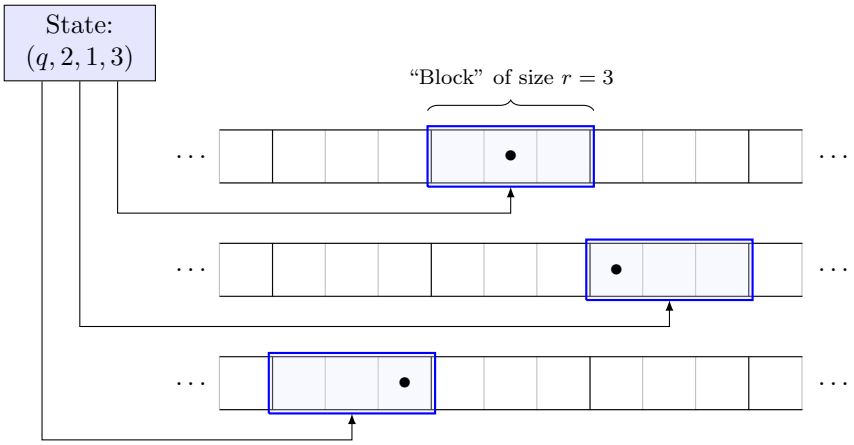
Theorem 24 (Tape compression theorem)**.** *Suppose that a language L is recognized by an $s(n)$-space k-tape deterministic Turing machine $\mathcal{M}$, where $k \geq 2$. Then, for any constant $0 < c < 1$, there exists a $c \cdot s(n)$-space k-tape deterministic Turing machine $\mathcal{M}'$ also recognizing L.*

Proof. Let c be as defined in the statement of the theorem, and choose a value r such that $rc \geq 2$. The idea behind the proof is that, using a modified larger tape alphabet, our new Turing machine $\mathcal{M}'$ will use less space on its work tapes to store the same data in "blocks".

We encode each symbol in the tape alphabet of our new Turing machine $\mathcal{M}'$ as a tuple of r symbols from the tape alphabet of $\mathcal{M}$. We also define the state set of $\mathcal{M}'$ in a particular way: each state of $\mathcal{M}'$ is itself a tuple

$$(q, i_1, \ldots, i_{k-1}),$$

where q is a state of $\mathcal{M}$ and $1 \leq i_j \leq r$ is the individual symbol in the "block" on the jth work tape of $\mathcal{M}'$ currently being read in the simulation of the computation of $\mathcal{M}$, where $1 \leq j \leq k-1$.



Encoding the contents of the tapes of $\mathcal{M}$ in this way, we can simulate one computation step of $\mathcal{M}$ and update the components of the "block" currently being read all in one computation step of $\mathcal{M}'$. Therefore, while the running time remains the same, the work space used by $\mathcal{M}'$ is at most $c \cdot s(n)$. □

Lastly, note that although our statements of both theorems specified deterministic Turing machines, we can make the appropriate modifications to each proof so that the same results hold for nondeterministic Turing machines.