St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 2: Foundations of Complexity Theory I
Winter 2025

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.

1 Running Time and Work Space

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.

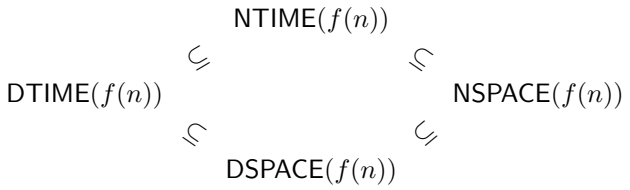


Figure 1: Relationships between basic time and space complexity classes.

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 diagram in Figure 1 depicts the same relationships as in Theorem 7.

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.

3.1 Linear Speedup Theorem

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.

Theorem 8 (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. This process is depicted in Figure 2.

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.

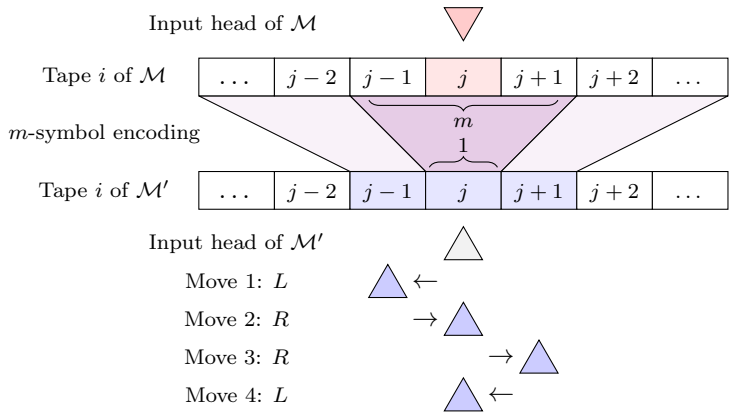
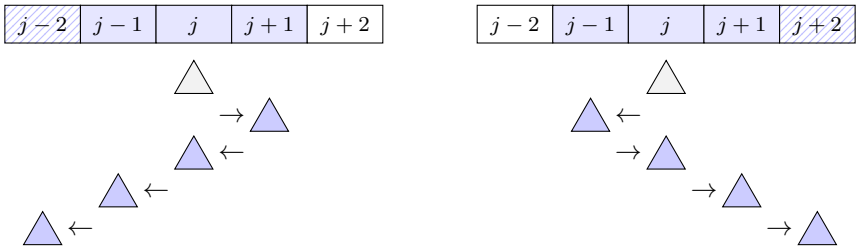


Figure 2: The neighbourhood-scanning process by which the Turing machine $\mathcal{M}'$ simulates the computation of $\mathcal{M}$ in the proof of Theorem 8.

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 $\mathcal{M}'$.

3.2 Tape Compression Theorem

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 9 (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 $1 \leq j \leq k-1$. This encoding process is illustrated in Figure 3.

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.

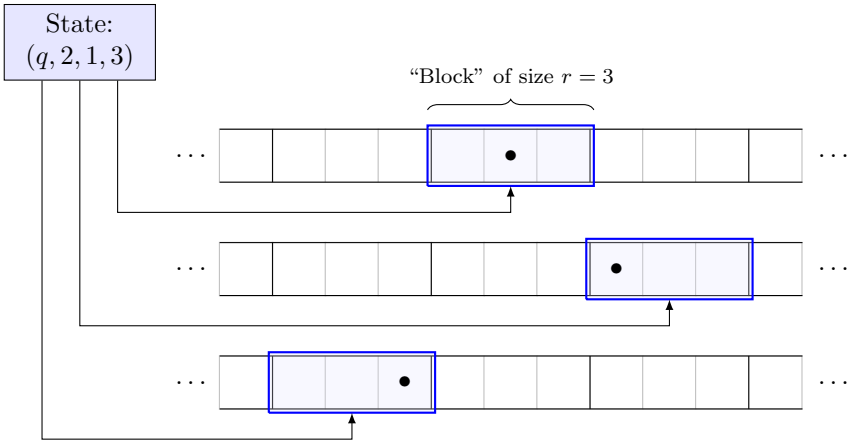


Figure 3: The block encoding process by which the Turing machine $\mathcal{M}'$ simulates the computation of $\mathcal{M}$ in the proof of Theorem 9.