St. Francis Xavier University
Department of Computer Science

CSCI 356: Theory of Computing
Lecture 6: Time Complexity
Fall 2021

1 Measures of Complexity

In previous lectures, we considered a variety of decision problems for all kinds of models of computation, but the most we established about these problems was whether or not the model could decide (or semidecide) the problem. While this information is useful to quantify the "power" of a model—that is, to determine the problems the model is capable of solving—it tells us nothing about the efficiency of using that model to solve that problem. A model that can solve an extremely difficult problem is close-to-useless if it takes a thousand years to return an answer!

Here, then, we will refine our study of decision problems to consider not just decidability, but also *complexity*. We generally measure the complexity of a decision problem in terms of the amount of computational resources we require to solve the problem: *time complexity* tells us how long (in terms of computational steps) it will take us to solve a problem, while *space complexity* tells us how much space in memory we need to solve a problem.

We will begin by studying time complexity, which appears time and again in various areas of computer science, not least of which in the area of algorithm design and analysis. We will then move on to studying space complexity. In both cases, though, we will only be able to scratch the surface of complexity theory; the complexity classes we will discuss in this course are merely the most well-known inhabitants of the zoo of complexity classes![1]

1.1 Big-O Notation

Often, when we discuss the time complexity of a problem, we don't care about the specific amount of time 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.

More generally, if $\mathcal{M}$ is a deterministic Turing machine that decides its problem (i.e., halts and either accepts or rejects all inputs), then we can define the *running time* of $\mathcal{M}$ as follows.

Definition 1 ($f(n)$-time Turing machine)**.** Given a deterministic Turing machine $\mathcal{M}$ that decides its problem, we say that $\mathcal{M}$ is an $f(n)$-time Turing machine if, on any input instance of length n, $\mathcal{M}$ makes at most $f(n)$ computation steps before halting and either accepting or rejecting.

A fundamental tenet of algorithm analysis is that 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] 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`.

The notion of "on the order of", which we used to simplify the toy analysis of the previous example, is so ubiquitous that it has its own notation. The *order notation*, also known as *Bachmann–Landau notation* in honour of its creators, allows us to denote a relationship between two functions $f(n)$ and $g(n)$ as the value of n grows arbitrarily large.

The most common type of order notation, called *Big-O notation*, gives us a way of establishing upper bounds on the performance of an algorithm. Since an upper bound corresponds to the maximum amount of time an algorithm would need to compute an input instance of a given size, Big-O notation is the most common notation used to analyze and compare algorithms.[2]

We define Big-O notation formally as follows.

Definition 2 (Big-O Notation)**.** Given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is Big-O of $g(n)$ and write $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n \geq n_0$,

$$0 \leq f(n) \leq c \cdot g(n).$$

By this definition, a function $f(n)$ is Big-O of another function $g(n)$ if there exists an appropriate scaling constant c and a large-enough minimal value n_0 such that the value of $f(n)$ is bounded above by the value of $g(n)$ for all values of n beyond n_0. Thus, a Turing machine with a running time of $f(n) \in O(g(n))$ will take no more time to halt than a Turing machine with a running time of $g(n)$, up to some constant factor.

Example 3. Consider the function $f(n) = 2n^2 + 3n + 12$. Observe that $n^2 \geq n$ for all $n \geq 1$ and, likewise, $12n^2 \geq 12n$ for all $n \geq 1$. Therefore,

$$\begin{aligned} 2n^2 + 3n + 12 &\leq 2n^2 + 3n^2 + 12n^2 \\ &\leq 17n^2 \end{aligned}$$

for all $n \geq 1$.

If we choose $c = 17$ and $n_0 = 1$, then we have that $2n^2 + 3n + 12 \leq c \cdot n^2$ for all $n \geq n_0$, and so $f(n) \in O(n^2)$. We can similarly prove that $f(n) \in O(n^3)$, $f(n) \in O(n^4)$, and so on, but these are all weaker upper bounds on $f(n)$.

Finally, we define the names of common functions seen in complexity theory and establish an ordering among these functions. We slightly abuse the notation $\leq$ to denote that functions further to the right grow faster than functions further to the left. We also assume that $k > 1$ in all cases.

$$\underset{\text{constant}}{O(1)} \leq \underset{\text{logarithmic}}{O(\log(n))} \leq \underset{\text{polylogarithmic}}{O(\log^k(n))} \leq \underset{\text{linear}}{O(n)} \leq \underset{\text{linearithmic}}{O(n\log(n))} \leq \underset{\text{quasilinear}}{O(n\log^k(n))} \leq \underset{\text{polynomial}}{O(n^k)} \leq \underset{\text{exponential}}{O(k^n)}$$

1.2 Other Notations

Order notation comprises a variety of different notations beyond Big-O, and each notation indicates a different kind of bound. While our focus in this course will be on Big-O, you may find that knowing the other notations could be helpful in future courses.

We can establish lower bounds in much the same way as we establish upper bounds. *Big-Omega notation* is the "lower bound" analogue to Big-O notation. We say that $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n \geq n_0$, $0 \leq c \cdot g(n) \leq f(n)$.

If some function $f(n)$ is bounded by a function $g(n)$ from both above and below—that is, we have both $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$—then we can use *Big-Theta notation* to denote this tight bound. We say that $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that, for all $n \geq n_0$,

[2] Keep in mind that Big-O notation only establishes an upper bound on something; it says nothing about the *worst-case* or *best-case* performance. We can obtain different bounds for different cases of a given algorithm, but the discussion of this distinction is better left for a course on algorithm analysis.

$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. Note that c_1 is the constant from the Big-Omega lower bound, while c_2 is the constant from the Big-O upper bound.

Finally, there exist *little-o* and *little-omega notations*, which are strict analogues of the Big-O and Big-Omega notations replacing the $\leq$ inequality between $f(n)$ and $g(n)$ with $<$.

1.3 Time Complexity

Now that we're able to reason about the running time of a Turing machine, we can classify languages decided by Turing machines in terms of the amount of time it takes to decide that language. This gives us our first rudimentary complexity class, which we will build upon and use to define further complexity classes.

Definition 4 (The class DTIME)**.** Given a function $f(n)$, the complexity class $\mathsf{DTIME}(f(n))$ is taken to be

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

Let's now consider a couple of examples of Turing machines that recognize certain languages, and determine the running time of each machine.

Example 5. Consider the language $L_{\text{odd1s}} = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ contains an odd number of } \mathtt{1}\text{s}\}$. We construct a deterministic Turing machine $\mathcal{M}_{\text{odd1s}}$ that takes as input a word $w \in \{\mathtt{0},\mathtt{1}\}^*$ and performs the following steps:

1. If the number of 1s read so far is even, get the next symbol.
   (a) If the next symbol is a 0, go to step 1.
   (b) If the next symbol is a 1, go to step 2.
   (c) If there is no more input left to read, reject.
2. If the number of 1s read so far is odd, get the next symbol.
   (a) If the next symbol is a 0, go to step 2.
   (b) If the next symbol is a 1, go to step 1.
   (c) If there is no more input left to read, accept.

Observe that the number of computation steps required for $\mathcal{M}_{\text{odd1s}}$ to accept or reject its input word is linear in the length of the word, since each symbol is read exactly once from left to right. Thus, $L_{\text{odd1s}} \in \mathsf{DTIME}(n)$.

You may have noticed in our previous example that L_{odd1s} is a regular language. That example demonstrates a (nontrivial) result about language classes and time complexity: all regular languages are in $\mathsf{DTIME}(n)$.

Example 6. Consider the language $L_{\text{bal}} = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ contains an equal number of } \mathtt{0}\text{s and } \mathtt{1}\text{s}\}$.[3] We construct a deterministic Turing machine $\mathcal{M}_{\text{bal}}$ that takes as input a word $w \in \{\mathtt{0},\mathtt{1}\}^*$ and performs the following steps:

1. Scan the input tape until an unmarked 0 or an unmarked 1 is read.
   (a) If no such symbols are read, accept.
   (b) If an unmarked 0 (resp., 1) is read, mark the symbol.
   (c) Scan the input tape until a matching unmarked 1 (resp., 0) is read.
      i. If no such symbol is read, reject.
      ii. If such a symbol is read, mark the symbol and return to step 1.

[3] Note that this language is quite similar to the language $L_{\text{a=b}} = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$, but here we don't require all 0s to appear before all 1s.

Observe that step 1 requires at most n computation steps as the Turing machine's input head scans the tape. Similarly, step 1(a) requires 1 computation step, step 1(b) requires 1 computation step, step 1(c) requires at most n computation steps, step 1(c)(i) requires 1 computation step, and step 1(c)(ii) requires at most n computation steps. Moreover, the Turing machine will loop back to step 1 at most $n/2$ times. Altogether then, this computation requires at most $(3n+3)(n/2) = (3n^2+3n)/2$ steps, and so $L_{\text{bal}} \in \mathsf{DTIME}(n^2)$.

Since L_{bal} is a context-free language—in fact, a *deterministic* context-free language—the previous example demonstrates another (nontrivial) connection between language classes and time complexity: all *deterministic* context-free languages are in $\mathsf{DTIME}(n^2)$. Context-free languages that are not deterministic, on the other hand, are in $\mathsf{DTIME}(n^6)$.

Lastly, since we've now seen a couple of examples of $f(n)$-time Turing machines for certain languages, here are two important facts to keep in mind when thinking about time complexity:

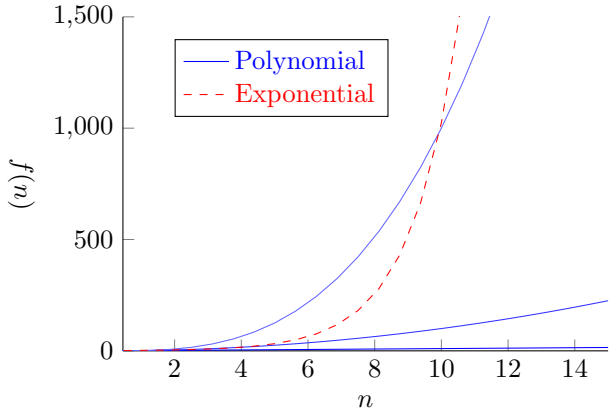
- showing $L \in \mathsf{DTIME}(f(n))$ does not imply that a $f(n)$-time Turing machine is the best possible; and
- failing to find a $f(n)$-time Turing machine for a given language L does not imply that $L \notin \mathsf{DTIME}(f(n))$.

2 P: Polynomial Time

In the previous examples we've seen of $f(n)$-term Turing machines, we saw that each machine decided its language in an amount of time generally considered to be "reasonable": given an input of size n, our first example decided its language in $\mathsf{DTIME}(n)$, while our second example decided its language in $\mathsf{DTIME}(n^2)$. In plain terms, these machines didn't need to do a lot of work per symbol of input they processed.

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 fact, if we plot just a few polynomial functions—n, n^2, and n^3—alongside the exponential function 2^n, we can see that as n increases, the polynomial functions all exhibit reasonable growth, but the exponential function skyrockets even for relatively small values of n. If you had to process an input of size 10, say, would you prefer to use an algorithm with a polynomial running time or an algorithm with an exponential running time?



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.

Cobham–Edmonds Thesis. *A computational problem can be feasibly computed on some model of computation only if the problem can be computed in polynomial time.*

Note that, like the Church–Turing thesis, the Cobham–Edmonds thesis is not so much a statement that we can prove formally. Rather, it's more in line with the general observation we made earlier: algorithms that run in polynomial time are "better", since they take less time to give us an answer.

The observation made by Cobham and Edmonds 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 our first major time complexity class.

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

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

That is, P contains all languages that are decidable by a deterministic Turing machine in polynomial time.

You might think that defining the class P in this way restricts us in a sense, since we explicitly singled-out deterministic Turing machines in the definition. What if we were using a different model of computation? Would we still have some kind of guarantee on the performance of a decision algorithm for the same language on that different model?

One of the great properties of the class P is that it's quite robust, in the sense that small changes to our model or our algorithm don't affect the larger property of "deciding 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.

Now, even though we're working in the world of theory and abstracting away a lot of the finer details, we would be remiss not to mention that 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 decides 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. Thus, we take observations like the Cobham–Edmonds thesis as a rule of thumb, and not as a principle etched in stone.

Having mentioned that caveat, let's now acquaint ourselves with some decision problems that belong to the class P.

s-t Connectivity

The first problem we will consider is a problem on directed graphs. Recall that a graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, where vertices are connected to each other by edges. In a directed graph, each edge has an associated direction of travel, so the existence of an edge from a vertex u to a vertex v does not necessarily imply the existence of an edge from v to u.

Often, when we're given a graph, we care about whether a path (i.e., a sequence of edges) exists that takes us from one vertex to another. We saw this, for example, when we proved that the emptiness problem E_{DFA} was decidable (though, in that case, we were checking whether *no* path existed from the initial state to a final state). If we can follow some sequence of edges from one vertex s to another vertex t, then we say that s and t are *connected*, and this gives rise to the *s-t connectivity problem*.

S-T-Connectivity
Given: a directed graph $G = (V, E)$ and two vertices $s, t \in V$
Determine: whether a path exists from vertex s to vertex t

At first glance, it seems trivial to solve this problem: starting from vertex s, simply check each outgoing edge and follow every path from s until either (a) you reach vertex t, or (b) you run out of edges to check.

However, while this approach does work, it is not at all efficient. Suppose there are a total of k vertices in the graph. In the worst case, we would need to visit all k vertices in the path from s to t, and this means that our naïve approach would need to check at most k^k paths: an exponential running time in the size of the input!

Instead, our approach will use a marking technique, similar to the approach used in our proof of the decidability of E_{DFA}.

Theorem 8. S-T-Connectivity *is in* P.

Proof. We construct a deterministic Turing machine $\mathcal{M}_{\text{st}}$ that takes as input $\langle G, s, t\rangle$, where $G = (V, E)$ is a directed graph and $s, t \in V$ are vertices, and performs the following steps:

1. Mark the vertex s.
2. While there are unmarked vertices remaining:
   (a) Scan every edge $e \in E$.
   (b) If there exists an edge $\{u, v\}$ from a marked vertex u to an unmarked vertex v, then mark vertex v.
3. If vertex t is marked, accept. Otherwise, reject.

Observe that steps 1 and 3 require 1 computation step each, and we only perform these steps once. Step 2(a) requires $|E|$ computation steps, since we must scan each edge in E, and step 2(b) requires 1 computation step. Moreover, we repeat step 2 at most $|V|$ times, since in the worst case we mark exactly one vertex on each iteration.

Altogether, this computation requires $|V|(|E|+1)+2$ steps, which is polynomial in the size of the input. □

Primality Testing

We now move on from graph theory to number theory, where we consider the problem of testing the primality of numbers. Testing primality in an efficient manner is a crucial task for some computer applications such as cryptography, where cryptographic systems often rely on the existence of very large prime numbers to encrypt and secure data.

Recall that an integer $n \geq 2$ is *prime* if its only divisors are 1 and itself. If we're given two integers m and n, then we say that m and n are *relatively prime* if their greatest common divisor is 1. We will first focus on the problem of testing relative primality.

Relative-Primes
Given: two integers m and n
Determine: whether m and n are relatively prime

Like with the s-t connectivity problem, there is a naïve approach to deciding the relative primality problem: simply check all possible divisors of both m and n, and accept if their greatest common divisor is 1. However, the naïve approach is again quite inefficient, since the magnitude of a number represented in base $k \geq 2$ is exponential in the length of its representation. Therefore, searching through every possible divisor would require us to check an exponential number of values, which consequently results in this approach taking exponential time.

Instead of checking every possible divisor, our refined approach will calculate directly the greatest common divisor of both m and n. If the result is 1, then we accept.

Theorem 9. RELATIVE-PRIMES *is in* P.

Proof. To calculate the greatest common divisor of two integers m and n, we use *Euclid's algorithm*. Given an input of the form $\langle m, n \rangle$, where m and n are binary representations of two integers, we construct a deterministic Turing machine $\mathcal{E}$ to run Euclid's algorithm in the following way:

1. While $n = 0$:
   (a) Set $m = m \bmod n$.
   (b) Swap the values of m and n.

At the end of the computation, the value on the tape of $\mathcal{E}$ corresponds to the greatest common divisor of m and n.

We now construct a deterministic Turing machine $\mathcal{M}_{\text{relprimes}}$ that takes as input $\langle m, n \rangle$, where m and n are binary representations of two integers, and performs the following steps:

1. Run $\mathcal{E}$ on input $\langle m, n \rangle$.
2. If $\mathcal{E}$ halts with 1 on its tape, accept. Otherwise, reject.

If $\mathcal{E}$ runs in polynomial time, then $\mathcal{M}_{\text{relprimes}}$ must also run in polynomial time. Therefore, we focus our analysis on $\mathcal{E}$.

Observe that, on each iteration of step 1 of $\mathcal{E}$, the value of m is at least halved. This is because, after step 1(a), we have that $m < n$ by the modulo operation. Then, the swap in step 1(b) results in $m > n$. Thus, if $m/2 \geq n$, then one iteration of step 1 results in $m \bmod n < n \leq m/2$, and if $m/2 < n$, then one iteration of step 1 results in $m \bmod n = m - n < m/2$.

Since the values of m and n are swapped on each iteration of step 1, both m and n are at least halved on every other iteration of step 1. Therefore, $\mathcal{E}$ makes at most $\min\{\log_2(m), \log_2(n)\}$ iterations before halting. Since m and n are represented in binary, the total number of computation steps is $O(n)$, which is polynomial. □

Knowing now that the problem of testing relative primality can be answered in polynomial time, what about the problem of testing primality in general? That is, given an integer n, how quickly can we determine whether n is prime?

PRIMES
Given: an integer n
Determine: whether n is prime

Interestingly, it was not known whether PRIMES was in P until relatively recently. Other primality testing algorithms were known, but none were simultaneously general (i.e., applicable for all integers), deterministic (i.e., not reliant on randomness), polynomial, and whose performance was not conditional on some unproven hypothesis, like the Riemann hypothesis.

In 2002, a team of Indian computer scientists—Manindra Agrawal, Neeraj Kayal, and Nitin Saxena—published the first primality-testing algorithm that satisfied all four of these criteria. Their proof that primality testing could be done deterministically and in polynomial time was widely celebrated, and although we leave out the details of their algorithm here, it was a remarkable achievement in the study of number-theoretic complexity.

Theorem 10. PRIMES *is in* P.

3 NP: Nondeterministic Polynomial Time

Up to now, we've focused on deterministic computations running on deterministic Turing machines. One major problem with adhering strictly to determinism, though, is that it limits the types of languages we're able to decide in polynomial time. For some decision problems, we just aren't able to come up with a clever