St. Francis Xavier University
Department of Computer Science

CSCI 356: Theory of Computing
Lecture 6: Time Complexity
Fall 2022

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:

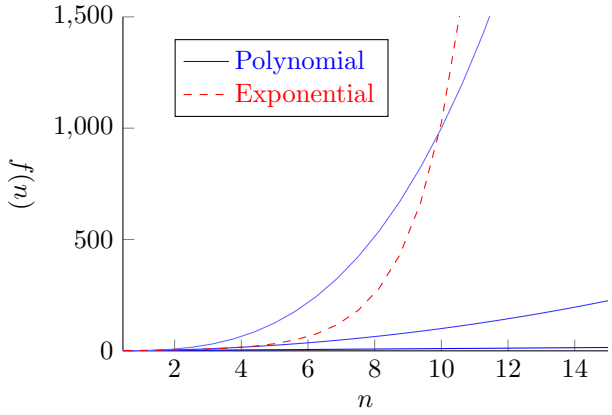
- showing $L \in \mathsf{DTIME}(f(n))$ does not imply that a $f(n)$-time Turing machine is the best possible; and
- failing to find a $f(n)$-time Turing machine for a given language L does not imply that $L \notin \mathsf{DTIME}(f(n))$.

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. Do until $n = 0$:
   (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.

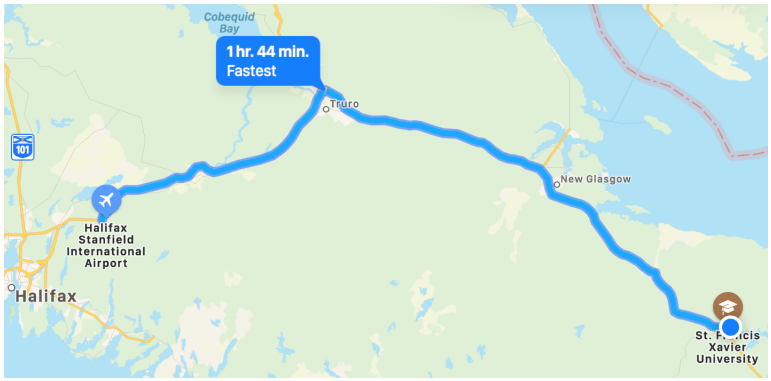
3 NP: Nondeterministic Polynomial Time

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

and efficient decision algorithm as we did for the problems in the previous section. 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.

What we *can* do with some difficult problems, however, is verify in polynomial time that a claimed solution to a problem is indeed valid. We can think of this procedure like an academic presenting the proof of a theorem in a mathematics talk: nobody in the audience may know how to prove the theorem themselves, but the presenter giving their proof of the theorem allows the crowd to verify that it's a correct result.

As another example, if someone asked you to give exact directions from campus to Halifax International Airport right this moment, you likely couldn't recite each turn by memory. However, if that person gave you a copy of the following map, you could easily verify that the route indeed goes from campus to the airport.



In the world of complexity theory, the machine we use to check the validity of a claimed solution to a problem is called a *verifier*. Given a language L, a verifier for L is a Turing machine $\mathcal{V}$ with the property that

$$L = \{w \mid \text{verifier } \mathcal{V} \text{ accepts the input } \langle w, c\rangle \text{ for some } c\}.$$

As usual, the word w is an instance of L that is claimed to be accepted. The value c is a *certificate*, or proof, that $w \in L$. Furthermore, the certificate c has polynomial length in terms of the word w. Given the input $\langle w, c\rangle$, the verifier $\mathcal{V}$ reads the certificate c to conclude that $w \in L$.

Note that the certificate c is auxiliary information that the verifier only uses during its computation; the certificate isn't itself a part of the instance, problem, or language. Thus, when we measure the running time of a verifier $\mathcal{V}$, we measure it only in terms of the word w.

If $\mathcal{V}$ runs in polynomial time, then we say it is a *polynomial-time verifier*. In turn, the language verified by $\mathcal{V}$ is said to be *polynomially verifiable*.

Definition 11 (The class NP—verifier def'n)**.** A language L belongs to the complexity class NP if there exists a verifier $\mathcal{V}$ and two polynomial expressions p_1 and p_2 such that

1. For all inputs $\langle w, c\rangle$, $\mathcal{V}$ has a running time of $p_1(|w|)$;
2. For all instances w of L for which the answer is "yes", there exists a certificate c with $|c| \leq p_2(|w|)$ such that $\mathcal{V}$ on input $\langle w, c\rangle$ accepts; and
3. For all instances w of L for which the answer is "no", and for all certificates c with $|c| \leq p_2(|w|)$, $\mathcal{V}$ on input $\langle w, c\rangle$ rejects.

So, what does the abbreviation NP stand for? Well, hold on... we're not quite done defining NP yet. The definition we just saw was framed in terms of a verifier that took existing solutions and checked whether those solutions were correct. However, we have no such "verifier definition" for P; that class was instead defined in terms of deciders. It would therefore be nice to have a definition of NP that also involved deciders in some way. We know already that deterministic Turing machines won't be enough to decide this class of problems, though, so we'll need to make a slight change to our model of computation.

If you recall the complexity class DTIME we introduced earlier, we took that class to contain all languages that are decided by a deterministic Turing machine in some running time $f(n)$. We can define an analogous class that contains all languages that are decided by a nondeterministic Turing machine in time $f(n)$, and this class is (appropriately) named NTIME.

Definition 12 (The class NTIME)**.** Given a function $f(n)$, the complexity class $\mathsf{NTIME}(f(n))$ is taken to be

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

Now, we can use this class NTIME to define a nondeterministic equivalent to our familiar class P, which gives us an alternative definition of NP framed in terms of deciding a language instead of verifying membership in a language.

Definition 13 (The class NP—decider def'n)**.** The complexity class NP is taken to be

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

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

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

NP does not stand for "non-polynomial"!

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

Now that we have two definitions of NP, it would be prudent of us to show that the second definition is actually equivalent to the first.

Theorem 14. *Let L be a language. Then L is in* NP *(according to the verifier definition) if and only if L is decided by a nondeterministic Turing machine in polynomial time.*

Proof. ($\Rightarrow$): Suppose $L \in \mathsf{NP}$, and let $\mathcal{V}$ be a verifier for L that has a running time of n^k for some $k \geq 0$. We construct a nondeterministic Turing machine $\mathcal{M}$ that decides L that takes as input $\langle w \rangle$, where w is an input word, and performs the following steps:

1. Nondeterministically choose a certificate c of length n^k.
2. Run $\mathcal{V}$ on input $\langle w, c \rangle$.
3. If $\mathcal{V}$ accepts, then accept. Otherwise, reject.

Since $L \in \mathsf{NP}$, we know that $\mathcal{V}$ runs in polynomial time. Moreover, steps 1 and 3 can be performed in a constant number of computation steps. Altogether then, this computation halts in polynomial time.

($\Leftarrow$): Suppose L is decided by a nondeterministic Turing machine $\mathcal{M}$ in polynomial time. We construct a polynomial-time verifier $\mathcal{V}$ that takes as input $\langle w, c \rangle$, where w is an input word and c is a certificate, and performs the following steps:

1. Simulate the computation of $\mathcal{M}$ on input w, reading each symbol of c to determine the nondeterministic choice made by $\mathcal{M}$ at each computation step.
2. If the computation branch of $\mathcal{M}$ corresponding to c accepts, then accept. Otherwise, reject.

Since the computation of $\mathcal{M}$ runs in polynomial time, the simulation performed by $\mathcal{V}$ also runs in polynomial time, and so $\mathcal{V}$ is a polynomial-time verifier. □

The line between problems in P and problems in NP is drawn finely and is often difficult to discern without prior experience in complexity theory. For some problems that are in P, making a seemingly minor adjustment to the definition of the problem can result in something that falls into NP.

For example, while we have efficient (polynomial-time) algorithms to find the shortest path through a graph, the problem of finding the longest path is in NP. Similarly, it's quite easy for us to find an Eulerian circuit in a graph, but the problem of finding a Hamiltonian circuit is not nearly as easy. Even the slightest change can blow up the time complexity of a problem: the 2-satisfiability problem of finding a satisfying assignment of truth values for a Boolean formula with at most two literals per clause can be solved in polynomial time, but the 3-satisfiability problem—exactly the same, but where each clause has at most three literals—is the quintessential example of a problem in NP!

Let's now take a look at a few examples of problems in NP. Note that, in each of the following verifier-based proofs, we will only focus on the polynomial runtime of the verifier and not the polynomial length of the certificate. This is done simply to make the proofs more concise and to communicate the main idea.

Longest Path

In graph theory, the *longest path problem* asks for the path of greatest length in a given graph G. Specifically, the problem asks for a *simple path*, or a path with no repeated edges, to avoid any loopholes where following a cycle in a graph could lead to a path of arbitrary length.

Longest-Path
Given: a graph $G = (V, E)$ and an integer $k \geq 0$
Determine: whether G contains a simple path of length k or greater

The longest path problem is surprisingly difficult to solve in contrast to the shortest path problem, which has a variety of algorithms: Dijkstra's algorithm, the Bellman–Ford algorithm, the Floyd–Warshall algorithm, and Johnson's algorithm among them. So, what accounts for the difficulty of finding long paths? In short, if we were able to solve the longest path problem efficiently, then we would be able to solve some other proven-difficult graph problems efficiently as well. However, since we know those other graph problems are difficult to solve, this suggests that there is no efficient way to solve the longest path problem.

To show that the longest path problem is in NP, we will give two proofs: one proof that uses a verifier, and one proof that uses a decider. As we now know, these two approaches are equivalent.

Theorem 15. Longest-Path *is in* NP.

Proof. We give both a verifier-based proof and a decider-based proof.

Verifier. Given an input of the form $\langle w, c \rangle$, where w is the input graph G and the integer k, and c is a certificate for that input consisting of a path in G of length k, our verifier $\mathcal{V}$ performs the following steps:

1. Check whether the path is simple and of length k.
2. If so, accept. Otherwise, reject.

Since we can check each edge of the path in polynomial time, our verifier runs in polynomial time.

Decider. Given an input graph $G = (V, E)$, a decider nondeterministically guesses a subset of edges of size at least k and at most $|E|$. It then checks whether the subset of edges forms a simple path in G. If so, then the decider accepts. Otherwise, the decider rejects. □

Note that the decider-based proof relies on nondeterminism to guess a subset of edges that forms a simple path. The decider can both make this guess and check whether it is a valid guess in polynomial time. Without nondeterminism, we would have no way of making such a guess, and we would simply have to iterate through all possible combinations of k edges.

Composite Testing

The problem of deciding whether a given number is *composite*—that is, not prime—is closely related to the problem of primality testing that we studied earlier.

COMPOSITES
Given: an integer n
Determine: whether n is composite

From a verification perspective, it's quite easy to decide whether a given number is composite: the input word w is the number to check, and the certificate c provides two smaller numbers m and n where $1 < m \leq n < w$ such that $w = mn$.

Theorem 16. COMPOSITES *is in* NP.

Proof. We give both a verifier-based proof and a decider-based proof.

Verifier. Given an input of the form $\langle w, c \rangle$, where w is the input to verify and c is a certificate for that input consisting of two values m and n, our verifier $\mathcal{V}$ performs the following steps:

1. Calculate mn.
2. If $w = mn$, accept. Otherwise, reject.

Since we can perform integer multiplication in polynomial time, our verifier runs in polynomial time.

Decider. Given an input integer w, run the same procedure as Theorem 10 to decide primality. If the procedure accepts, then reject. If the procedure rejects, then accept. □

You might have noticed at the end of the proof of Theorem 16 that we "cheated" a bit: we used a decider for primality testing to decide compositeness, but we know that primality testing is in P! We didn't do anything wrong here, though. Recall that a problem is in NP if there exists a nondeterministic Turing machine that decides the problem. A deterministic Turing machine is simply a nondeterministic Turing machine that doesn't use nondeterminism. Therefore, the decider for primality testing (and composite testing) is, technically, still a nondeterministic Turing machine!

This brings us to an important result relating the classes P and NP: any problem that can be decided efficiently can also be verified efficiently, since we can just verify the answer that the decider gave us.

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

Proof. Suppose we have some problem $L \in \mathsf{P}$. Then we know that there exists a polynomial-time deterministic Turing machine $\mathcal{M}$ deciding the problem. We use this machine $\mathcal{M}$ as part of a verifier $\mathcal{V}$ for the same problem, where $\mathcal{V}$ is given an input of the form $\langle w, c \rangle$ and performs the following step:

1. Run $\mathcal{M}$ on w.

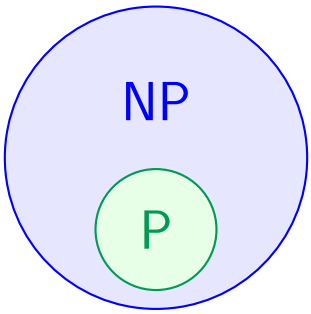
The verifier $\mathcal{V}$ simply ignores the certificate c that is given as input, and instead decides the problem instance w directly. Moreover, since $\mathcal{M}$ runs in polynomial time, so too does $\mathcal{V}$. Therefore, $L \in \mathsf{NP}$. □

Now, what about the other direction? Is $\mathsf{NP} \subseteq \mathsf{P}$, thus making the two classes equal? If you know the answer, please feel free to share it: this P *vs.* NP *problem* is one of the biggest problems in all of computer science. Indeed, finding the solution to this problem will net you $1 million and a wide array of awards and prizes, not to mention eternal fame. Unfortunately, despite the intuitive belief that verifying a solution to a problem should be easier than computing the solution itself, hundreds of researchers and academics have attempted to settle the problem to no avail.[4] That being said, a number of potential approaches have been proven *not* to work, and some computer scientists believe that some entirely new and yet-unknown area of

[4] You can see a list of over 100 attempts to solve this problem on the P-*versus*-NP *page* at `https://www.win.tue.nl/~gwoegi/P-versus-NP.htm`.

theory will need to be developed in order to make any meaningful progress. But then again, who knows what the future will bring?

All in all, here's how we generally think the complexity world looks at the moment (assuming $\mathsf{P} \neq \mathsf{NP}$):



4 NP-Hardness and NP-Completeness

Occasionally, we may have a deep understanding of the complexity or difficulty of one decision problem, and we may come across another decision problem that resembles the first problem in some way. For example, when we introduced the longest path problem, we observed that having an efficient decision procedure for that problem would allow us to solve other difficult graph problems efficiently.

We know from earlier lectures that there is a tool that allows us to model one decision problem in terms of another decision problem: a *reduction*. We're already familiar with the general notion of a mapping reduction, but only to the extent of using it to show a problem is decidable or undecidable. With our newfound knowledge of complexity theory, it becomes important for us to be able to reduce problems while also preserving the overall time complexity measure. Therefore, we would like our reductions now to be efficient; that is, to run in polynomial time.

Fortunately, to get an efficient mapping reduction, we need only modify our original definitions slightly by sprinkling the phrase "polynomial-time" in the appropriate places.

Definition 18 (Polynomial-time computable function)**.** A function $f\colon \Sigma^* \to \Sigma^*$ is polynomial-time computable if there exists some polynomial-time Turing machine that, given an input word w, halts with $f(w)$ on its input tape and nothing else.

Definition 19 (Polynomial-time mapping reduction)**.** Given two decision problems X and Y, problem X is polynomial-time mapping reducible to problem Y if there exists a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ where, for all $w \in \Sigma^*$, $w \in X$ if and only if $f(w) \in Y$.

Just like before, if X is polynomial-time reducible to Y, then we can transform every instance w of X to an instance $f(w)$ of Y in polynomial time. We denote a polynomial-time mapping reduction from X to Y by the notation $X \leq_m^P Y$.

With the notion of a polynomial-time mapping reduction, we can now talk about certain decision problems having certain complexity-theoretic properties.

For starters, if we know that there exists a reduction from a decision problem X to a decision problem Y, then we can conclude that Y is at least as difficult to solve as X. This is because we need to use the "black box" decider for Y as part of our decider for X. With this observation, we can prove two important results, which are essentially the complexity-theoretic analogues of our earlier decidability/undecidability results:

Theorem 20. *If $X \leq_m^P Y$ and $Y \in \mathsf{P}$, then $X \in \mathsf{P}$.*

Proof Sketch. If we are able to both apply the reduction from X to Y in polynomial time and run the decider for Y in polynomial time, then the overall decider for X also runs in polynomial time. □

Theorem 21. *If $X \leq_m^P Y$ and $Y \in \mathsf{NP}$, then $X \in \mathsf{NP}$.*

Proof Sketch. After applying the polynomial-time reduction from X to Y, we are able to use a verifier for instances of Y to verify instances of X. □

These two theorems tell us that, if we know to which complexity class a decision problem Y belongs, and if we have a polynomial-time reduction from another decision problem X to Y, then we can conclude that X belongs to the same complexity class.

4.1 NP-Hardness

Reductions are not symmetric; being able to reduce from one decision problem to another does not necessarily imply that we can reduce the other way around. If we consider a particular complexity class, say NP, then we can use reductions to establish the relative *hardness* of decision problems in that class. We can establish some measure of hardness as follows: if every problem in NP can be reduced to one particular problem X, then we can count X as having a difficulty on par with any other problem in NP. Furthermore, the problem X doesn't have to belong to NP itself.

If we're able to reduce any decision problem in NP to a particular decision problem X in polynomial time, then we say that X is an NP-*hard* decision problem.

Definition 22 (The class NP-hard)**.** A decision problem X is said to be NP-hard if, for every decision problem $Y \in \mathsf{NP}$, there exists a polynomial-time mapping reduction $Y \leq_m^P X$.

Since we can reduce any decision problem in NP to an NP-hard decision problem X, we can informally characterize X as being *at least as difficult* as the most difficult decision problem in NP.

4.2 NP-Completeness

In the same spirit as the notion of hardness, the notion of a decision problem being *complete* for a complexity class indicates that such a decision problem is representative of the entire class with respect to difficulty. Earlier, we noted that an NP-hard decision problem X doesn't necessarily need to belong to the class NP. If, additionally, we know that X is in NP, then we say that X is an NP-*complete* decision problem.[5]

Definition 23 (The class NP-complete)**.** A decision problem X is said to be NP-complete if $X \in \mathsf{NP}$ and X is NP-hard.

The class NP-complete contains all decision problems whose verifiers we can use to verify solutions to any other problem in NP via polynomial-time reductions; in that sense, therefore, NP-complete decision problems are the most difficult problems of any in NP. The class of NP-complete problems is occasionally denoted by NPC.

As we did for P and NP, we can prove a result that allows us to show a decision problem is NP-complete by way of reduction:

Theorem 24. *If $X \leq_m^P Y$, X is NP-complete, and $Y \in \mathsf{NP}$, then Y is NP-complete.*

Proof. For Y to be NP-complete, we require that $Y \in \mathsf{NP}$ and Y is NP-hard. We know that $Y \in \mathsf{NP}$ by our assumption, so all we need is to show that every decision problem Z in NP can be reduced to Y in polynomial time.

Since X is NP-complete by our assumption, every decision problem $Z \in \mathsf{NP}$ can be reduced to X in polynomial time. Moreover, $X \leq_m^P Y$ by our assumption as well. Since reductions are transitive, this implies that every decision problem $Z \in \mathsf{NP}$ can be reduced to Y in polynomial time as desired. □

[5] The terms "hard" and "complete" were settled upon by the theory research community in the mid-1970s. Before the terminology was settled, Donald Knuth mailed out a survey to solicit suggestions from colleagues. The three initial options were "Herculean", "formidable", and "arduous", with write-in options including "impractical", "bad", "heavy", and "hard-ass".

The result of Theorem 24 will be of great use when we prove shortly that a variety of decision problems are NP-complete.

Before we consider examples, however, we will make one more observation about NP-completeness. Since we can reduce every problem in NP to an NP-complete problem, finding an efficient algorithm for that NP-complete problem would change the landscape of complexity theory as we know it. If we were in some way able to decide an NP-complete problem in polynomial time, then we would immediately solve the P vs. NP problem.

Corollary 25. *If X is* NP*-complete and $X \in \mathsf{P}$, then* $\mathsf{P} = \mathsf{NP}$.

Proof. Follows from Definition 19 and Theorem 20. □

Of course, our entire discussion about NP-completeness only matters if we know that there exists some decision problem that is actually NP-complete. Remember that a problem X is NP-complete if both $X \in \mathsf{NP}$ and X is NP-hard. Showing that $X \in \mathsf{NP}$ is straightforward, and we gave two methods of doing so earlier: by using verifiers or by using deciders. Showing that X is NP-hard is, as the name suggests, the hard step. How can we show that every problem in NP is polynomial-time reducible to our problem X?

Fortunately, thanks to Theorem 24, we need only do the hard work of showing that *one* decision problem is NP-complete. Then, if we want to show that another problem is NP-complete, we just need to show that the problem is in NP (easy) and then come up with a reduction from our "original" NP-complete problem (fairly easy).

Satisfiability

In 1971, the American-Canadian computer scientist Stephen Cook and the Soviet mathematician Leonid Levin independently published the same remarkable result, giving the first example of a decision problem that is NP-complete. This decision problem, known as the *Boolean satisfiability problem*, asks whether there exists some assignment of true and false values to Boolean variables that satisfies every clause in a given Boolean formula.

Before we proceed further, let's clarify some terminology. A *Boolean formula* is a logical combination of *Boolean variables*. Variables are arranged in *clauses*; for example, the formula $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ contains four Boolean variables and two clauses. This example formula is also said to be in *conjunctive normal form*, since each clause contains only $\vee$s, and clauses are joined using only $\wedge$s. Lastly, a *satisfying assignment* of a Boolean formula is one that renders the overall formula true; for example, in our previous formula, $x_1 = x_3 =$ true and $x_2 = x_4 =$ false is a satisfying assignment.

The formal statement of the Boolean satisfiability problem, then, is as follows.

SATISFIABILITY
Given: a Boolean formula $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ in conjunctive normal form
Determine: whether there exists an assignment of truth values to Boolean variables satisfying all clauses in the Boolean formula

How did Cook and Levin show that the Boolean satisfiability problem is NP-complete? Well, showing that the problem is in NP is the easy step. The proof of NP-hardness, though, is quite clever: using the fact that the class NP contains all decision problems that can be decided in polynomial time by a nondeterministic Turing machine, one simply constructs a Boolean formula that simulates the computation of such a nondeterministic Turing machine on a given input word. If this simulated machine accepts, then the Boolean formula has a satisfying assignment!

The complete proof of NP-hardness is quite long and technical, and since we don't really require it for anything else, we'll only present a sketch of that part of the proof. The proof of membership in the class NP, however, only takes a few lines.

Theorem 26 (Cook–Levin theorem)**.** Satisfiability *is* NP*-complete.*

Proof Sketch. We begin by showing that Satisfiability is in NP. This step is straightforward: we can construct a polynomial-time nondeterministic Turing machine $\mathcal{M}_{\text{SAT}}$ that takes as input a Boolean formula ϕ and guesses a satisfying assignment of values to variables. If the assignment is in fact satisfying, then $\mathcal{M}_{\text{SAT}}$ accepts.

Next, we sketch the proof that Satisfiability is NP-hard. To do this, consider any decision problem $L \in \mathsf{NP}$. We know that a polynomial-time nondeterministic Turing machine $\mathcal{M}_L$ exists that decides L in time n^k for some $k \geq 0$.

If $n = |w|$, where w is the input word given to $\mathcal{M}_L$, then any computation of $\mathcal{M}_L$ on w has at most n^k configurations. Suppose we take all such configurations and create a computation table of size $n^k \times n^k$. Each index of the computation table contains a symbol from the set $C = Q \cup \Gamma \cup \{\#\}$, where # is a special boundary marker written on the left and right sides of the computation table.

We can represent the contents of each index (i, j) of the computation table by $|C|$ Boolean variables, each of the form $\{x_{i,j,s} \mid s \in C\}$. If $x_{i,j,s} = \text{true}$, then this variable indicates the symbol at index (i, j) of the computation table is s. In total, we require $|C| \cdot n^{2k}$ Boolean variables.

Next, using our set of Boolean variables, we create Boolean formulas to "verify" the computation of $\mathcal{M}_L$. We require our formulas to satisfy four conditions:

$$\begin{aligned}
\phi_{\text{start}} &= \{\text{the first row of the computation table is the start configuration of } \mathcal{M}_L \text{ on } w\};\\
\phi_{\text{acc}} &= \{\text{the last row of the computation table is an accepting configuration of } \mathcal{M}_L \text{ on } w\};\\
\phi_{\text{idx}} &= \{\text{for all indices } (i,j), \text{ there exists exactly one } s \in C \text{ such that } x_{i,j,s} = \text{true}\}; \text{and}\\
\phi_{\text{tran}} &= \{\text{each } 2 \times 3 \text{ subblock of the computation table satisfies the transition function of } \mathcal{M}_L\}.
\end{aligned}$$

For each of these four conditions, we can create a Boolean formula of size $O(n^{2k})$ to express the condition, and we can construct the formula in polynomial time relative to the input word w.

Finally, we claim that $\mathcal{M}_L$ has an accepting configuration on w if and only if the Boolean formula $\phi_{\mathcal{M}_L} = \phi_{\text{start}} \wedge \phi_{\text{acc}} \wedge \phi_{\text{idx}} \wedge \phi_{\text{tran}}$ has a satisfying assignment. In this way, we have developed a polynomial-time reduction $L \leq_m^P$ Satisfiability. Since we can do this for every decision problem $L \in \mathsf{NP}$, we conclude that Satisfiability is NP-hard. □

Having shown that the Boolean satisfiability problem is NP-complete, we can now prove all kinds of other problems are NP-complete using the result of Theorem 24. To show that some new decision problem Y is NP-complete, all we need to do is show both that $Y \in \mathsf{NP}$ and that another NP-complete problem we know (like Boolean satisfiability) reduces to Y in polynomial time.

In the following sections, we will show that three other common decision problems are NP-complete. To show that these problems are NP-complete, we will construct a "chain" of reductions from the Boolean satisfiability problem to our new problems. Diagrammatically, our chain will look like the following, where an arrow $X \to Y$ indicates a reduction $X \leq_m^P Y$:

Satisfiability ⟶ 3-Satisfiability ⟶ Vertex-Cover ⟶ Hamiltonian-Circuit

3-Satisfiability

Our second NP-complete problem, the *3-satisfiability problem*, is quite similar to our first. Indeed, 3-satisfiability is just a restricted form of the Boolean satisfiability problem where each clause of the given Boolean formula contains three variables.

3-SATISFIABILITY
Given: a Boolean formula $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ in conjunctive normal form, where each clause C_i contains three Boolean variables
Determine: whether there exists an assignment of truth values to Boolean variables satisfying all clauses in the Boolean formula

It should come as no surprise that, since Boolean satisfiability is NP-complete, so too is 3-satisfiability.

Theorem 27. 3-SATISFIABILITY *is* NP*-complete.*

Proof Sketch. First, we show that 3-SATISFIABILITY $\in$ NP. For this, we can use the same argument as we did to show SATISFIABILITY $\in$ NP.

Next, we construct a reduction SATISFIABILITY $\leq_m^P$ 3-SATISFIABILITY. Given an instance ϕ of SATISFIABILITY, we construct a Boolean formula ϕ' such that our formula is satisfiable if and only if the original formula ϕ is satisfiable.

Observe that, in our reduction, all we must do is ensure that ϕ' contains three variables per clause.

- For clauses with fewer than three variables, construct a new clause that repeats one of the existing variables in the clause. This does not affect the satisfiability or unsatisfiability of the clause.
- For clauses with more than three variables, split the clause into multiple clauses and add dummy variables y_i that preserve the satisfiability or unsatisfiability of the original clause. For example, given a clause $(x_1 \vee x_2 \vee \cdots \vee x_m)$ where $m > 3$, we construct $m - 2$ clauses

$$(x_1 \vee x_2 \vee y_1) \wedge (\overline{y_1} \vee x_3 \vee y_2) \wedge \cdots \wedge (\overline{y_{m-3}} \vee x_{m-1} \vee x_m).$$

Since the overall satisfiability or unsatisfiability of each clause is unaffected by these modifications, our new formula is satisfiable if and only if the original formula is satisfiable. □

Vertex Cover

Our next decision problem, the *vertex cover problem*, is a graph-theoretic problem that asks us to determine whether there exists a subset of vertices where each vertex in the subset covers, or corresponds to, at least one of the two endpoints of every edge in the graph.

VERTEX-COVER
Given: a graph $G = (V, E)$ and an integer $k \leq |V|$
Determine: whether there exists a subset of vertices $V' \subseteq V$ such that $|V'| \leq k$ and, for all edges $\{u, v\} \in E$, at least one of the vertices u or v belongs to V'

Theorem 28. VERTEX-COVER *is* NP*-complete.*

Proof Sketch. First, we show that VERTEX-COVER $\in$ NP. We can construct a polynomial-time nondeterministic Turing machine $\mathcal{M}_{\text{VC}}$ that takes as input a graph G and an integer k and guesses a subset of vertices of size $k \leq |V|$. If this guessed subset is a cover, then $\mathcal{M}_{\text{VC}}$ accepts.

Next, we construct a reduction 3-SATISFIABILITY $\leq_m^P$ VERTEX-COVER. Given an instance ϕ of 3-SATISFIABILITY, we construct a graph G and an integer k such that our graph has a vertex cover of size $k \leq |V|$ if and only if the given Boolean formula ϕ is satisfiable.

To construct our graph, we create "gadgets" (or small subgraphs) corresponding to each Boolean variable and each clause. Each variable "gadget" consists of two vertices labelled x_i and $\overline{x_i}$ joined by a single edge. Each clause "gadget" consists of three vertices labelled by the three variables in the clause, all of which are joined together.

Variable "gadget" Clause "gadget"

We then construct the overall graph by adding edges between variable "gadgets" and their appearances in corresponding clause "gadgets". Finally, we compute the value k in terms of the number of variables, v, and the number of clauses, c, by taking $k = v + 2c$.

Following this construction, if the given Boolean formula has a satisfying assignment, then we add one of the two vertices from each variable "gadget" to our cover, select one true variable from each clause, and add the remaining two variables of the clause "gadget" to our cover. Conversely, if we can construct such a cover, then it corresponds to a satisfying assignment of the Boolean formula. □

Hamiltonian Circuit

Finally, we consider the *Hamiltonian circuit problem*. This is another problem on graphs, where this time we must determine whether there exists a circuit (that is, a path where the start and end points are the same vertex) that traverses every vertex in a given graph.

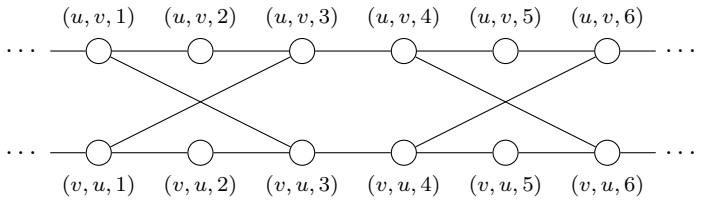
HAMILTONIAN-CIRCUIT
Given: a graph $G = (V, E)$
Determine: whether there exists an ordering of all vertices $\{v_1, v_2, \ldots, v_n\}$ of G such that $\{v_n, v_1\} \in E$ and $\{v_i, v_{i+1}\} \in E$ for all $1 \leq i \leq n - 1$

Theorem 29. HAMILTONIAN-CIRCUIT *is* NP*-complete.*

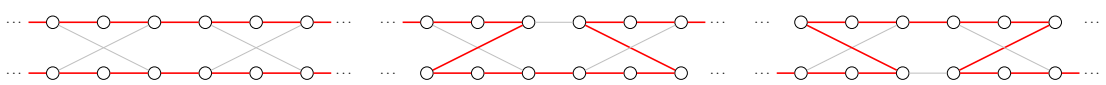
Proof Sketch. First, we show that HAMILTONIAN-CIRCUIT is NP-complete. We can construct a polynomial-time nondeterministic Turing machine $\mathcal{M}_{\text{HC}}$ that takes as input a graph G and guesses an ordering of the vertices in V. Note that this ordering is necessarily "Hamiltonian" since all vertices are included. If this ordering forms a circuit, then $\mathcal{M}_{\text{HC}}$ accepts.

Next, we construct a reduction VERTEX-COVER $\leq_m^P$ HAMILTONIAN-CIRCUIT. Given an instance $\langle G, k \rangle$ of VERTEX-COVER, we construct a graph $G' = (V', E')$ such that our graph contains a Hamiltonian circuit if and only if the given graph has a cover of size $k \leq |V|$.

To perform this reduction, we again use the idea of "gadgets". For each edge $\{u, v\} \in E'$, we construct an edge "gadget" of the form



Note that there exist only three possible ways that a Hamiltonian circuit can pass through this "gadget":



Each of these paths corresponds to one or both of the vertices u and v belonging to the cover.

We also add k "cover vertices", $\{c_1, c_2, \ldots, c_k\}$, to our vertex set V'. Then, we construct the overall graph G' by stringing together each edge "gadget" into a single chain, and connecting the ends of this chain to all of the "cover vertices".

Following this construction, if the given graph has a vertex cover $\{v_1, v_2, \ldots, v_k\}$ of size k, then we can find a corresponding Hamiltonian circuit in G' by starting at c_1, passing through the edge "gadget" for v_1, moving to c_2, passing through the edge "gadget" for v_2, and so on until we return to c_1. Conversely, a Hamiltonian cycle in G' produces a vertex cover in the given graph by following the chain of edge "gadgets" and obtaining the corresponding k vertices. □

Having discussed both NP-hardness and NP-completeness, let's wrap up our discussion of time complexity with one last diagram depicting our current view of the complexity world (again, assuming $\mathsf{P} \neq \mathsf{NP}$):