Computational Complexity

This last topic of our course is one that lies at the very heart of computer science and mathematics. Until now, we have explored what it means to describe an algorithm and to define its runtime. We have shown in great detail why it is that we describe runtimes in Big-oh notation, noting that this description captures the performance of the algorithm without relying on the specifics of hardware, speed of computers, multiplicative constants, etc. However, for almost every problem we have come across, we have been able to find a polynomial-time algorithm: one that runs in $O(n^k)$ for some k. A polynomial-time runtime is considered "reasonable". It may seem as though we can solve anything in polynomial time since our examples so far have found polynomial-time solutions. Unfortunately, this is not the case.

In this lecture we provide a brief overview of **computational complexity theory**, which focuses on classifying problems according to their "difficulty". What is interesting, and fundamental to many current applications today, is that there are problems for which we don't yet *know* if they can be solved in polynomial time. This does not mean that one doesn't exist! It simply means that we are currently unable to find one. In fact, this great unknown has a huge influence in many real-world applications, and in particular in security.

Although it may seem quite extraordinary that there are many problems for which do not yet have polynomial time algorithms - having a polynomial-time algorithm is quite rare. There are many problems we know for sure *cannot* be solved in polynomial time. Many well-known games fall into this category: $n \times n$ chess for example is one. Problems in this category are very difficult - so difficult in fact that scientists have been able to prove we will *never* find a polynomial-time algorithm that solves these problems.

That may seem as bad as it gets, and yet, even worse, there are problems for which we know can't be solved *at all*, in any finite amount of time. This is one of the most powerful results in computer science, dating back to the beginning of the theory on this topic, in the early 1930s.

This brief lecture can certainly not define and compare the many classes of computational complexity. Instead we begin with a brief history on the topic, and then define some of the most important complexity classes.

1 What does it mean to be able to compute something?

There are many periods in history where a particular area of science explodes with a new discovery, often fuelled by the results of a remarkable researcher, or a particular need in society, and other times merely due to a bit good luck. The world of physics was redefined in 1911 when Ernest Rutherford discovered the nucleus of an atom. This simple fact (which we have most likely all learned in our school days) is a relatively recent discovery, and it completely changed our understanding of matter and the universe. In the 1920s the field of biology was changed forever by Alexander Fleming's discovery of penicillium, from which we now have the benefits of modern antibiotics. In the 1930s there was a huge race in the field of mathematics to prove what it means for something to be "computable". At this point in history, we did not yet have actual computers as we know them today. Nevertheless, mathematicians were hard at work imagining that they could potentially codify axioms of mathematics, and then "procedures" (which we know call algorithms) could be used to determine if statements were true or false. Looking back today, it is quite extraordinary that the concept of being able to use procedures or "algorithms" to solve problems was being explored well before any notion of a physical computer existed. At this particular point in history, there were many scientists hard at work sharing ideas, offering different models of computation, and deciding what each model could compute. Some of the names from here on you may recognize, others perhaps not, yet hopefully upon completion of this lecture you will take with you at least a small token from the great minds who brought us towards our modern notion of analyzing an *algorithm*.

One of the most recognizable names in the field is Alonzo Church, an American mathematician who, in the 1930s, defined the *Lambda Calculus*. This was one of the first attempts to formally express "computation". One of his students, Stephen Kleene, also became a great pioneer in the field. Kleene is well-known for defining *Recursion theory*, also known as *Computability theory*, and his work aimed at defining "computable functions". In 1936, a Polish-born mathematician named Emil Post defined a model called the *Post machine* which is a model of computation, not an actual physical machine. Post hoped to show that he could build more and more powerful computing models. And also in 1936, Alan Turing wrote a now famous paper on an abstract machine called the *Turing machine*. The 1930s was a buzz of mathematicians trying to describe some theoretical definition of what it means to be able to *compute* something. There are many more names and models that can be added to this list, and today it is really these great minds that pioneered the entire field of theoretical computer science. What is very surprising, however, is that after all their competing work, it was proven that **all of their models are equivalent**. In other words, anything that the Turing machine could do, so could the Post machine, etc. This led scientists to believe that there was a basic notion of "computable" that unified all these models. This is what is known today as the *Church-Turing thesis*. It is called a "thesis" because it has never been proved, yet it is widely accepted as true. The simplified version of the Church-Turing thesis is that any computation that can be effectively carried out by a procedure, can be computed by a Turing machine.

Once models of computation were defined, the next big question was quite simple: **"what can we compute?"** Can these models compute *anything*? Is there something that can't be computed? At this time, the types of problems in question were **decision problems**: those for which the answer is just "yes" or "no". So the BIG question at the time was the very famous "Entsheidungsproblem", which is german for "The problem of making a decision". A very basic way of explaining the entsheidungsproblem is as follows: given any decision problem, is there a procedure that will eventually decide if it is true or false? This was the probably one of the biggest and most exciting mysteries in mathematics in the 1930s.

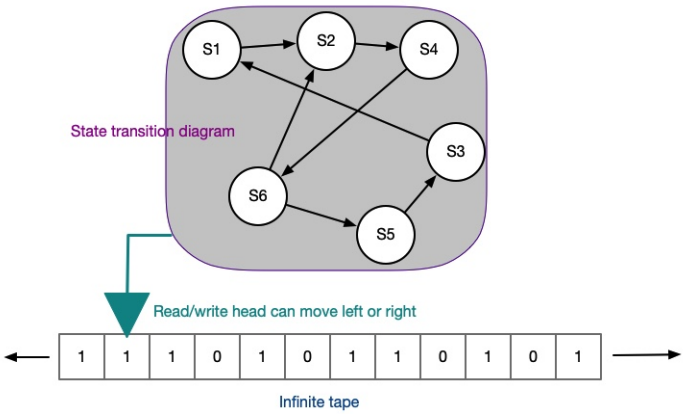
During your education as a computer scientist, you may have heard the name **Alan Turing** before, most likely in reference to his work on the construction of the *Bombe* at Bletchley Park during the second world war. However, Turing's most significant work occurred many years earlier. In 1936 he published his world-famous paper entitled "On computable numbers with application to the Entscheidungs problem".

Alan Turing. King's College Library, Cambridge. AMT/K/7/11. By kind permission of the Provost and Fellows of King's College, Cambridge.

This paper remains today one of the most brilliant contributions to the field, if not the most. So what was so extraordinary about his work? In the next section we briefly summarize the results of Turing's work, what it means for computer science, and how this discovery opened the doors for the world to begin thinking about what we *can* and *can't* compute.

1.1 The Turing machine

Alan Turing's "machine" that was introduced in his paper in 1936 is not actually a machine at all - it is a theoretical concept that defines what it means for something to be computable. His abstract machine called the *"Turing machine"* is a beautifully simple model of computation.



The model consists of:

- a **tape**, which is assumed to be infinite,
- a set of **states**
- a **scanning head** that can move left and right and can read/write onto the tape
- a **transition function** which decides what to in a certain state upon reading a certain symbol on the table.

What is remarkable about Turing's 1936 paper is that he argued that his machine was as powerful as *any possible computing device*. Quite a remarkable fact, years before computers even exist! He went on to define the *Universal machine* which can run the program of any other Turing machine, meaning that no matter what computational tasks are necessary, one machine can run them all. This is obvious to us in modern day - we certainly don't go out and buy a new computer each time we need to run a new program! But in 1936, before computers even existed, it was Alan Turing who proved that this concept existed.

Once Turing had carefully defined his machine, he went on to prove that the *Entscheidungsproblem* is unsolvable. In other words, he proved that some problems *cannot* be decided by his computation machine. The problem for which he proved that no decision can be made is the **Halting problem**. The great repercurssions today mean that we know there are some problems that are uncomputable: we simply cannot compute an answer in a finite amount of time. Turing's Halting problem was the first such problem, and shockingly, these problems are not rare. In fact, *most* problems are unsolvable - we cannot compute a yes/no answer in a finite amount of time.

1.2 What happened next

After the theory of which problems can and cannot be computed, research moved next towards categorizing the relative difficulty of problems. The notion of how to classify problems according to difficulty relies on some concept of how we measure how *complex* an algorithm is. What is the right measure of difficulty? Obvious choices are things like **time** or **space**, but these are not necessarily the only options. Certainly time should be one of the most important measures, and runtime was quickly isolated as an important measure. Once we focus on the time of an algorithm, there is the secondary problem of defining what is a computational "step". A step certainly depends on what computer model you are using: a Turing machine, a multi-tape Turing machine, a real computer with random access memory, etc. Each model would carry out different types of steps, and then have potentially different runtimes. Further ambiguity arises when we try to define steps with simple notions such as : writing an integer, reading an integer etc. Wouldn't the size of the integer make a difference? Should there be an upper bound how how big they could become? Throughout the 1960s researches attempted to present many different models of computation that clearly defined how to count steps, and how to define runtime. There is no single clear definition of *step* unfortunately. However most computation models used today are based on some random-access

machine where each operation involving an integer has unit cost, and we simply state that the integers can't become unreasonably large.

Finally, around 1964, researchers began to create "categories" that encompassed problems of certain difficulties. The first was the class P which was defined by Cobham in 1964. The next section gives a brief overview of some of the more important complexity classes, and we conclude this lecture with a discussion of problems that are *NP-complete*.

2 Complexity Classes and decision problems

A complexity class contains a set of problems that take a similar range of space and/or time to solve. The different classes help computer scientists group similar problems together. For example, suppose we wish to compare the relatively difficulty of determining whether or not a number is prime, versus the problem of determining if two vertices are connected in a graph. The descriptions of each of these problems are vastly different. Nevertheless, they both belong to one of the most important complexity classes: the class P. Knowing that these problems both belong to P allows us to conclude that they are somewhat similar - we know that the number of steps required to compute each problem is bounded by some polynomial.

The complexity classes that we look at in this section are for **decision problems**: problems for which the goal is to determine if the input is *true or false*. Many of the problems that we have looked at so far have decision *versions* of the same problem. For example, we saw an algorithm for computing the minimum spanning tree of a weighted undirected graph, and we studied algorithms that solve this problem in time $O(E \log V)$. The decision version of the MST problem is the following has only a yes/no answer:

Decision MST: Given a weighted, undirected graph and an integer k, is there a spanning tree whose total weight is at most k?

The decision version of MST is at least as easy as the optimal problem. In other words, if we can solve the optimal problem with a certain runtime, then we can also solve the decision problem in the same runtime. Therefore whenever we have a certain runtime for the "optimal" version of a problem, we can assume the same runtime applies to the decision version.

The complexity classes that we introduce in the remaining of this section are for decision problems. There are several measures of complexity that could be used to classify problems, two of the most natural are described below:

- The **time complexity** of an algorithm is used when describing the number of steps it takes to solve the problem, and these steps depend on the particular model that we use (Turing maching, RAM model, Quantum computer, etc).
- The **space complexity** of an algorithm describes how much memory the algorithm needs in order to operate.

In this lecture we focus on classes of *time* complexity. In order to count the number of steps, we must decide on what **model of computation** is used. For example, do we count steps on a Turing machine? Or using the RAM model that we defined earlier in the course? Does it make a difference? Most time complexity classes actually count the number of steps on a Turing machine. However, in this course we have analyzed runtimes using the RAM model, which is a simplified version of a modern computer. The good news is that a Turing machine is not that much different from the RAM model. In fact, the number of steps an algorithm would take on a Turing machine is not exactly the same as the number of steps it would take using the RAM model, however, the number of steps on the Turing machine is only *polynomially* larger. In other words, a $O(n)$ algorithm on the RAM model *may* take $O(n^k)$ time on a Turing machine, which is actually pretty reasonable difference. This means that whether or not we use a Turing machine, or the RAM model, this does not change the overall *complexity class* of the problem.

2.1 The class EXP

The first class that we look at is often referred to as EXP or $EXPTIME$ and it consists of all decision problems that can be solved in an **exponential** number of steps. Formally, if the input size is n, the problem can be solved by an algorithm in $O(2^{p(n)})$ steps where $p(n)$ is a polynomial. Any decision problem for which there is an exponential-time algorithm belongs to this class. We have seen some algorithms in this class that run in exponential time: in particular the brute-force algorithms we saw for Longest common subsequence. Therefore the *decision* version of this problem is in the class EXPTIME.

Other examples of problems that are known to be solvable in exponential time are:

- Given an $n \times n$ chess board and any positions of the players pieces, determine which player can win from that position
- Given an $n \times n$ Go board and any positions of the players pieces, determine which player can win from that position.
- Given an algorithm, determine if it halts after k steps
- Given a large integer n and an integer k, does n have a prime factor smaller than k.

Most algorithms that run in exponential time are infeasible on our modern day computers, even when the input size is not that large. We usually focus on finding algorithms that run in polynomial time, as defined in the next section.

2.2 The class P:

Decision problems that are in the class P are those for which a polynomial-time algorithm is known. Until now, almost all algorithms we have seen in this course run in polynomial time. Most of them have decision versions that are in the class P. Some notable examples are given below:

- Given a graph G and two vertices s and t, is there a path from s to t in the graph?
- Given a string s, is the string a palindrome?
- Is the number p a prime?
- Decision MST: given a weighted graph G, is there a spanning tree with weight $\leq k$?

Any problem in P must have an algorithm that solves the problem in time $O(n^k)$ for some integer k. Many algorithms that we studied have runtimes that are upper bounded by a polynomial. For example, Kruskal's algorithm runs in time $O(E \log V)$. The input size is the total size of the edges and the vertices, which means that $O(E \log V)$ is $O(n^k)$ for any $k > 2$.

Since polynomial functions are upper bounded by exponential functions, we have the following fact:

Theorem 1. $P \subseteq EXP$

All decision problems that can be solve in polynomial time can also be solved in exponential time.

2.3 The class NP:

The class NP is slightly more abstract that the class P. The class NP consists of all decision problems which can be **verified in polynomial time**. This means that given a solution, we can verify that it is indeed correct in polynomial time. This class of problem is referred to as NP.

Example 1. *Given a set of n items each with a specific weight $w_1, \ldots, w_n$ and each with a specific value $v_1, \ldots, v_n$, consider the problem of determining if it is possible to select a set of items whose total weight is less that T and whose total value is at least V. Explain why this problem is in NP*

Solution: This is actually a program we have seen previously in our section on dynamic programming. Here we show that the problem is in NP. Our job is to describe how to *verify* a solution in polynomial time. This is actually a rather trivial task. If you are given a set of items, you can easily verify their total weights by summing their weights and verifying that the total is at most T. You can also sum their values and determine if the total value is at least V. Therefore in time $O(n)$ one can easily verify if the given set is indeed a solution.

Notice that it seems as though there are "more" problems in NP, since we don't have to actually solve the problem and find the solution, we only need to verify it.

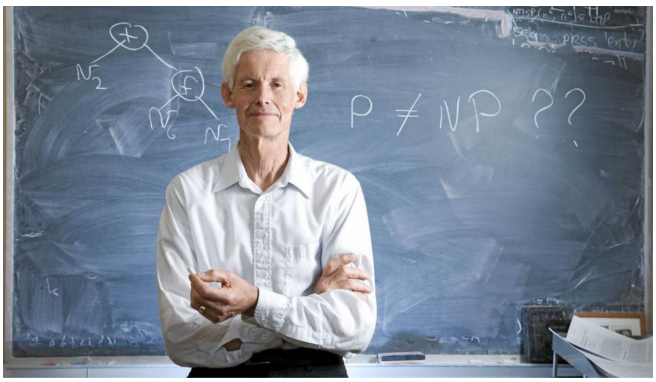
Theorem 2. *If a problem is in P, then it is also in NP*

Proof: If a problem is in P, then it can be solved in polynomial time. If it can be *solved* in polynomial time, then certainly we can also verify a solution in polynomial time. Therefore anything in P is also in NP.

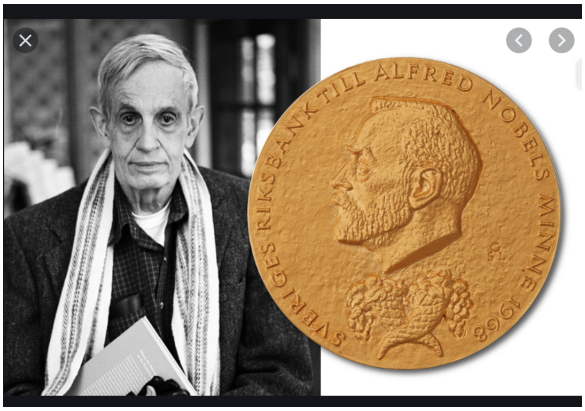
The class of problems in NP includes many problems that we can solve efficiently and many more that we would very much *like* to solve efficiently but so far don't know how. If we could solve *all* of them in polynomial time, then in fact the class NP and P would be the same.

2.4 The big question: is $P = NP$?

Although the above two definitions are rather uncomplicated, it turns out we don't actually know if these two classes are the same or different. Is it true that any problem that we can *verify* in polynomial time (those in NP) can also be *solved* in polynomial time (P)? This is one of the great mysteries of modern mathematics and computer science. Answering this question is one of the *Millennium Prize Problems* for which there is a 1 million dollar prize. The original question was posed by *Stephen Cook* in 1971.



Stephen Cook, professor of computer science and mathematics at the University of Toronto.



However, even earlier than that, other mathematicians were beginning to realize how important this question was. The famous mathematician John Nash (whose biography is featured in the film *A Beautiful Mind*) speculated in the 50's that cracking codes could not be done in polynomial time, and therefore he already had an inkling some problems were in NP and not in P. John Nash is a 1994 Nobel prize winner.

If one day we discover that $P = NP$, the consequences may be rather drastic to our modern applications. Many security and cryptography protocols *rely* on the fact that we do *not* know of any polynomial-time algorithms to decrypt or uncrack them. If suddenly we learn that these decryption algorithms (which are in NP) are *also* in P, then it opens the door to the possibility that many of our secure systems can be decrypted by a polynomial-time algorithm.

In our final section we define a subset of problems in NP, those that are considered "just as hard" as any other problem in NP.

3 NP-complete

Informally, the problems that are NP-complete are considered the "hardest" problems in NP. These problems are in NP but no polynomial-time algorithm is known to solve them, nor do we know if one exists. As mentioned in the previous section, some of these problems are extremely relevant to our modern systems. Consider the following two scenarios:

Scenario 1: Imagine for a moment that you build a system that can be accessed only with a correct password. Certainly your system is secure if you *knew absolutely* that no algorithm could efficiently solve the problem of finding/guessing the correct password. In such a case, it is reasonable to conclude that your system is safe. This is the scenario if we *knew absolutely* that *no* decryption algorithm runs in polynomial time.

Scenario 2: On the other hand, suppose that you build a secure system which encrypts a bank code. Your system is secure as long as no one has a polynomial-time algorithm to decrypt your bank code. But what if the existence of such an algorithm was *unkown*? What if you *assumed* that no such algorithm exited, and then built your entire system hoping no one would come up with a way to unlawfully decrypt it. Is it still reasonable to assume that your system is secure? In fact, much of modern security and cryptography is based on this assumption.

Why is it then that these NP-complete problems are so mysterious? Let's look first at the formal definition of an NP-complete problem, and then move onto some basic examples.

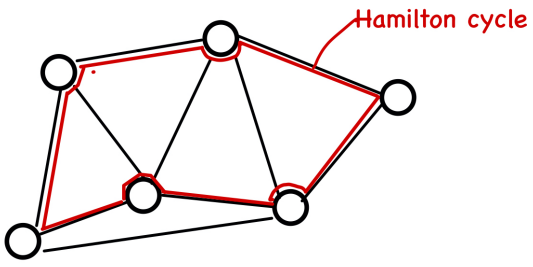
A problem p is NP-complete if both of the following are true:

- The problem p is in NP: there is a polynomial time algorithm that *verifies* a solution to p.
- The problem p is just as hard as any other problem in NP. In other words, **IF** you can solve p in polynomial time, then you can solve **ALL** the other problems in NP!

Many NP-complete problems are surprisingly simple, and often very similar to problems that *do* have a polynomial-time solution. Here are some examples of known NP-complete problems:

Hamilton Cycle

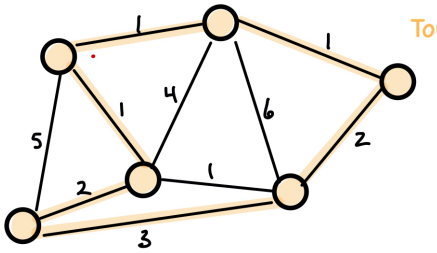
Given a graph G, the Hamilton cycle problem is the problem of deciding if there is path through the graph that starts at a vertex, visits each other vertex exactly once, and comes back to where it started. Determining if this exists is NP-complete! In other words, no polynomial time algorithm is known.



A Hamilton *Path* is a simple path in G that visits every vertex exactly once. Like the Hamilton Cycle problem, determining whether or not a graph has a Hamilton path is also NP-complete.

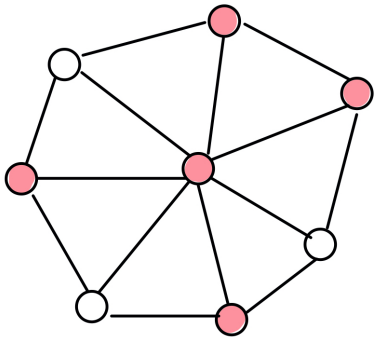
Travelling salesman problem

Given a list of cities and the distance between each city, is it possible visit each city exactly once and come back to the starting city using a total distance less than L?



Vertex cover

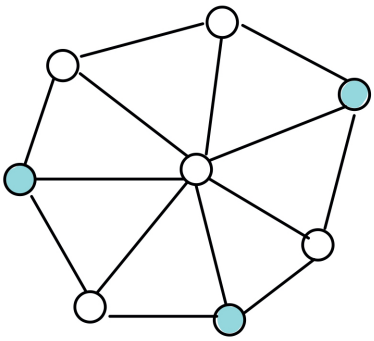
Given an undirected graph G we would like to select a set of **at most** k vertices such that each edge has at least one selected endpoint. This is a typical problem that seems so easy, and in fact no polynomial-time algorithm is known! In the example below, the graph requires at least 5 vertices in order to have a vertex cover. Note that every edge in the graph below has at least one selected endpoint.



On the other hand, a simple variation of the problem, where we instead ask to select a set of *edges* in G so that each vertex is adjacent to some selected edge, is *not* NP-complete. This algorithm is called *Edge-cover* and has a polynomial time algorithm.

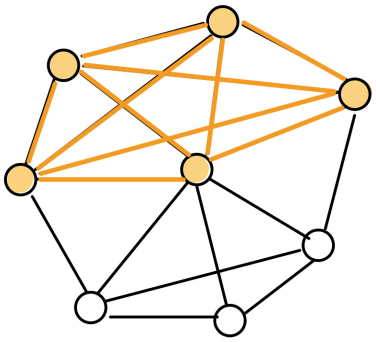
Independent Set

Given an undirected graph G, suppose we are asked to find a set of **at least** k vertices in the graph such that no two are adjacent. This decision problem has no polynomial-time algorithm.



Finding Cliques

Given an undirected graph G on n vertices, the Clique problem is the problem of determining if there are k vertices in the graph that are *all* pair-wise connected. For example, imagine a set of n people, some pairs of which are friends. The simple problem of determining if there is a subset of k people in which everyone is friends with everyone else, is the Clique problem. This problem is NP-complete. In the example below, the graph does contain a clique of size 5.



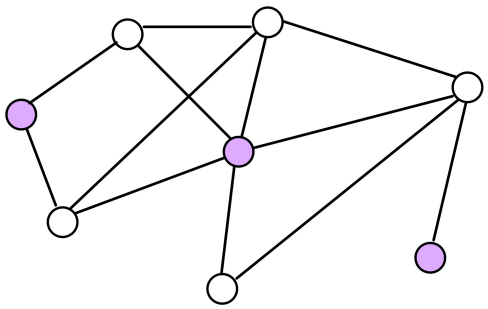
Subset Sum

Given a set of n numbers (not necessarily distinct) and a target T, is there a subset of of the numbers whose total is T? Ex. $S = \{3, 2, 4, 5, 3, 6, 4, 2\}$ and target $T = 8$, then a subset exists with this sum, namely $\{5, 3\}$.

Dominating Set

Given an undirected graph G on n vertices, determine if there is a way to select at most k vertices such that every vertex in the graph is either selected, or is adjacent to a vertex that is selected. In the graph below, the highlighted vertices represent a dominating set of size 3. Notice that every vertex in G is either a vertex in the dominating set, or adjacent to a vertex in the dominating set. The same set of vertices **do not represent a vertex cover of the same graph**. Dominating sets and vertex covers are not always the same.

The problem of determining if a general graph has a dominating set of size k is NP-complete.



3.1 How to show a problem is NP complete:

Deciding if a problem is NP-complete is extremely relevant because it gives us an idea of how difficult the problem is. If the problem *is* NP-complete, then it is extremely unlikely that you will be able to find a polynomial-time algorithm for it. The problems that are NP-complete are *so hard* that no one has yet found a polynomial-time algorithm for any of them. Therefore something that is NP-complete is generally assumed to be too unreasonable to solve.

Using the above definition of NP-complete, there are two steps to showing that a problem is NP-complete.

Steps to show NP-completeness for a problem p:
1. Show that a solution to p can be **verified** in polynomial time
2. Show that if you can solve p in polynomial time, then *all* other problems in NP can be solved in polynomial time. This is done using a **reduction** from another NP complete problem to problem p.

The second step above is solved using what is referred to as a **reduction**: take a known NP-complete problem, and show that your problem can be used to solve that problem. The example below illustrates how to work through the above two steps.

Example 2. *Show that the* ***Clique Problem*** *above is NP complete.*

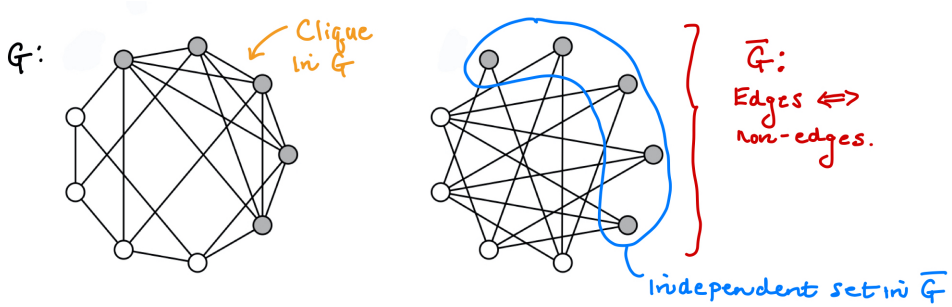
Solution:

Step 1: The verification

Our first step is to show that a solution to the Clique problem can be verified in polynomial time. Suppose you are given a graph with n vertices and m edges, and are asked if there is a clique of size k in your graph. A proposed solution consists of a set of k vertices. In order to verify if indeed this is a valid solution, one simply needs to check if an edge exists in G for every pair of vertices in the set. That is exactly $k(k-1)/2$ verifications, and therefore this can be done in polynomial time.

Step 2: The reduction

Next, we need to carry out the reduction: show that the Clique problem can be *used* to solve some other known NP-complete problem. Our goal is to pick a known NP-complete problem that is similar to the Clique problem. Although it may not seem like it at first glance, the Independent Set problem is almost identical to Clique, except that it is asking for the reverse property. Notice that if we select a set of vertices in G that represent a clique, then those exact same vertices represent an independent set in the complement of G. Therefore it seems reasonable to attempt a reduction from independent set. That means that we just need to show that we could *solve* the independent set problem by *using* a solution to the Clique problem.



The concept of a Reduction:

- Assume you are given an instance to the Independent Set problem. Convert it to an instance of the Clique problem in such a way that the Clique problem "solves" the independent set problem. The time it takes you do to this conversion must be polynomial.
- Show that the solution to the clique problem corresponds *exactly* to the solution of the independent set problem. If the independent set problem was a "yes", then the clique conversion must be a "yes". Similarly, if the independent set problem was a "no", then the clique problem was also a "no".

We now work through the details of the reduction:

An instance to the independent set problem consists of a graph G and a target k, and we are asked if it is possible to select k non-adjacent vertices in G. Convert this into a new graph G' where every edge in G is removed, and every non-edge in G becomes an actual edge in G'. This new graph G' is the input to the clique problem.

If the input, G, to the independent set were a YES instance, then there is an independent set of size k in G. Suppose these vertices are called set S. This means in the converted graph G' there are edges between all pairs of vertices in S. Therefore G' has a clique of size k, and thus G' is a YES instance. On the other hand, suppose the converted graph G' had a clique of size k. Then in the original graph G there were no edges between any of these vertices. So the original graph has an independent set of size k.

The above three steps use the Clique problem to solve the independent set problem. This means that any problem in NP can be reduced (i.e. solved) using the Clique problem, and therefore it is NP-complete.