Chapter 34 (II)
On $NP$-completeness

We have discussed many problems in this course and the others. Most of them are “easy” in the sense that we can solve them in polynomial time, such as the sorting problem; while others are “hard”, i.e., its complexity is $2^n$, e.g., the Hanoi Tower problem.

We have also seen the following implementation for the Fibonacci sequence:

```c
fib(n){
    if n=0 return 0;
    if n=1 return 1;
    return fib(n-1)+fib(n-2);
}
```

Let $F(n)$ be the run-time of the above code, it is easy to see that

$$F(n) \geq fib(n) \approx 1.6^n.$$  

(Cf. page 24, Parallel algorithm chapter notes.)
Another perspective

If you send in $n = 200$, it takes at least $1.6^{200}$, about $2^{138}$ time units to calculate.

It would take the *NEC Earth Simulator*, with a clock speed of 40 trillion instructions per second, about $2^{92}$ seconds to finish, when the earth has already turned into a red giant star.

On the other hand, since $F(n + 1) = 1.6^{n+1} = 1.6F(n)$, it takes a computer 1.6 times longer to calculate $fib(n + 1)$ as compared to $fib(n)$.

By Moore’s law, computers has been doubling their processing speed every 18 month, roughly 1.6 times every year. Let $n_0$ be the largest number the fastest machine can do this year for $fib(n)$, the fastest machine next year can only solve $fib(n)$ for $n_0 + 1$.

In the upcoming Theory course, we will see some problems are not solvable. Stay tuned....
Not a clear cut

We also have seen problems that lie somewhere in between, such as the *Traveling Salesman problem*. (Check out the course page)

Another “simpler” example is the Hamiltonian problem, i.e., whether, in a given graph G, there is a simple cycle that contains all the vertices exactly once. It is related to, but different from, the Könisburg problem, where we want to have a simple cycle that includes all the edges exactly once. (Check out the course page)

For these problems, the best algorithm as we know of is exponential, but we have yet to prove that its complexity is indeed exponential.

There are actually many real-life problems for which no efficient algorithm is known, but whose complexity has yet to be proved to be exponential, either. The hardest ones are called *NP-Complete problems*. 
What to do?

It may be the case that efficient algorithms do exist for them, but none is found yet.

It is also possible that the problems are intrinsically hard, but we don’t yet have the techniques to prove that is the case.

In this part, we are going to show a remarkable result: if we find an efficient algorithm for any one of these NP-Complete problems, all of them can be solved efficiently.

Since it is too good to be true, the common sense is that none of the NP-Complete problems is easy, i.e., can be solved in polynomial time.
The class $\mathcal{P}$

**Question:** What should be considered to be efficiently solvable?

**Answer:** If, for some polynomial $p(n)$, there exists an algorithm which can solve any instance of the problem of size $n$ in $p(n)$.

Although it is reasonable to regard a program that requires $\Theta(n^{100})$ *intractable*, only few problems in reality needs this much.

It is also often the case that once a polynomial time algorithm is found, a more efficient version will follow.
Why do we love them?

It is often the case that if a problem can be solved in one computer model, e.g., one processor with RAM, in polynomial time, it can be so solved in another model.

Finally, the problems in $\mathcal{P}$ enjoy nice closure properties, since polynomials are closed under addition, multiplication, and composition.

Thus, if the output of a polynomial-time algorithm is sent into another one as the input, then the composition of the two also takes polynomial time.

In this sense, most of the problems that we have been working with fall into $\mathcal{P}$.

Do they not?
Decision problems

We are particularly interested in *decision problems*, for which the answer is either “yes” or “no”. For example, both “Is graph $G$ is Hamiltonian?” and “Is the shortest path in graph $G$ between two vertices less than 10?” are decision problems.

A decision problem can be thought of defining a set $X$, of instances of the problems on which the correct answer is “yes”.

For example, the above decision problem for Hamiltonian cycles defines a set of graphs $H = \{G_1, G_2, \ldots, \}$, for each of such graphs some Hamiltonian cycles do exist. These instances are referred to as the “yes”-instances of this problem.

All the other graphs are the “no”-instances.
What is \( \mathcal{P} \)?

We further say that a correct algorithm that solves a decision problem accepts “yes”-instance and rejects “no”-instances.

For example, an algorithm for the Hamiltonian cycles problem is correct, if it accepts all the graphs in \( H \) and rejects all the other graphs.

**Definition:** \( \mathcal{P} \) is the class of decision problems that can be solved by a polynomial-time algorithm.
Too much a restriction?

A decision problem can only send out a binary answer of “yes” or “no”, which seems to be rather restricted.

The idea is that a problem can often be converted to a decision problem by setting up a bound. For example, once we have a solution to the shortest path problem, we immediately have one for the corresponding decision problem.

Thus, if a problem can be easily solved, so is the associated decision problem. On the other hand, if the decision problem is “hard”, then the associated problem has to be “hard”, too.

Thus, when we study how hard a problem is, it suffices to study the decision problems.
Some are, some aren’t.

The above decision problem for the shortest path belongs to $\mathcal{P}$, since, for any graph, we can run, e.g., the Bellman-Ford algorithm, to find out the shortest path between those two vertices, in $O(|V||E|)$, then compare its length with 10 in $\Theta(1)$.

On the other hand, we don’t know yet if the decision problem for the Hamiltonian cycles belongs to $\mathcal{P}$.

But, if somebody claims that a given graph is Hamiltonian and offers a piece of evidence: a sequence of nodes, we can than trivially verify if it is indeed a Hamiltonian cycle for that graph.
How to do it?

We simply check if the given input is a permutation of the vertices of the graph, i.e., if it contains exactly those vertices in the graph; and, if it is, proceed to check whether each of the consecutive edges along the cycle exists in the graph.

It takes $\Theta(|V| + |E|)$ to do the checking....

Thus, we have a polynomial algorithm to verify whether a graph is Hamiltonian with a given evidence, although we haven’t found a polynomial algorithm to construct a Hamiltonian cycle for a general graph.
What do we get out of it?

If the verification goes through, we are convinced that this graph is Hamiltonian. On the other hand, if it does not check out, we only know that this evidence does not work.

Since there might be other evidences to show that this graph is Hamiltonian, we are still unsure about whether this graph is indeed Hamiltonian.

Intuitively, this sort of verification is much easier than the actual construction.

Is it?

No body knows... 😞

Why don’t we wrap up here... 😊
What do you need to know?

1. Although many problems can be efficiently solved, some of them are not solvable, while others can be solved, but it takes too long a time.

2. There exist a collection of problems for which all the existing solutions take exponential time, but no one has proved that is the case. On the other hand, it takes only polynomial time to verify whether something is a solution to such a problem. This set of problems is referred to as the NP class.

3. In contrast, the P class refers to the class of problems that a solution can be constructed in polynomial time.

4. Although $P \subset NP$, whether $NP \subset P$ is a major problem remains to be solved in Computer Science.