Recitation 07: The Classes P, NP and Polynomial Equivalence

In this recitation, we’ll begin the discussion of computational complexity. From now on, we’ll only deal with decidable problems. In other words, the Turing Machine halts after a finite number of steps on every input. Within decidable problems, we’ll classify languages according to their computational complexity or “difficulty”. Two of the most important classes will be the class P, the class of problems solvable in polynomial time by a deterministic TM, and the class NP, the class of languages solvable in polynomial time by non-deterministic TM.

The Class P

Before we present the definition of the class P, let’s build some background. We start with the concept of a running time.

**Definition 1** Let $M$ be a deterministic decider. And let $f : \mathbb{N} \to \mathbb{N}$. We say that $M$ runs in time $f(n)$, if $f(n)$ is the maximum number of steps $M$ takes on any input of size $n$.

Let’s take a simple example to digest this definition. Assume your friend gives you a decider $M$ with input alphabet $\Sigma = \{0, 1\}$. Your friend asks you to give them $f(1)$. How would you about this? You go back to the definition above and realize that $n = 1$ here. You would look into every input of size 1 (only 0, 1 in this case) and feed them into the machine. You would then count the number of steps the machine takes on each of them. Say it takes 34 steps on input 0 and 52 on input 1. Then, $f(1) = \max(34, 52) = 52$. Computing $f(2)$ is similar, you would run the machine on 00, 01, 10, 11 and so on...

Since we’re usually interested in the behavior of an algorithm for large inputs, our next concept will be asymptotic $O$-notation.

**Definition 2** Let $f, g : \mathbb{N} \to \mathbb{N}$. We say that $f(n) = O(g(n))$ if there exists constants $C, n_0$ such that for any $n \geq n_0$ we have $f(n) \leq C \cdot g(n)$.
Two things to note about this definition:

1. Given an \( n_0 \) for which this definition works, the behavior of \( f(n), g(n) \)
   is irrelevant for \( n < n_0 \). In other words, a \( \mathcal{O} \) statement says noth-
   ing on the behavior for small inputs.

2. Multiplicative constants can be ignored. If \( f(n) = \mathcal{O}(g(n)) \), then
   \( 2f(n) = \mathcal{O}(g(n)) \). To see this, just change \( C' = 2 \cdot C \) in the above
   definition.

Given these definition, we can start talking about classifying languages
according to the "time it takes to solve them". We start with the follow-
ing definition.

**Definition 3** Let \( t : \mathbb{N} \to \mathbb{N} \) be a function. We define \( \text{TIME}(t(n)) \)
to be the class of languages that can be decided in \( \mathcal{O}(t(n)) \) time. In other words,
\( A \in \text{TIME}(t(n)) \) if there exists a decider \( M \) such that \( L(M) = A \) and \( M \)
runs in \( \mathcal{O}(t(n)) \) time according to definition 1.

As a concrete example, for the language:

\[ A = \{a^k b^k, k \geq 0\} \]

we have demonstrated a decider \( M \). \( M \) first scans the input to check
that it’s of the form \( a^* b^* \). Then \( M \) zig-zags across the input crossing
off \( a \)'s with \( b \)'s in a matching fashion. \( M \) accepts if the number of \( a \)'s
and \( b \)'s matches, and rejects otherwise.

To calculate the computational complexity, we take an arbitrary
input of size \( n \). In our example, the only interesting case would be
\( a^{n/2} b^{n/2} \), otherwise, the TM would reject immediately and remember
that we’re only interested in worst case. The first linear scan would
take \( n \) steps right, then \( n \) steps back. We don’t focus on the constants
so we say \( \mathcal{O}(n) \) steps.

Afterwards, the zig-zagging behavior would move the head \( n/2 \)
steps back and forth (\( \mathcal{O}(n) \) steps) for every match of \( a \) with a \( b \) (\( \mathcal{O}(n) \)
matches) resulting in \( \mathcal{O}(n) \mathcal{O}(n) = \mathcal{O}(n^2) \). Overall \( M \) takes \( \mathcal{O}(n) + \mathcal{O}(n^2) \)
in the worst case scenario on any input of size \( n \).

This proves that \( A \in \text{TIME}(n^2) \). Further thought actually proves
that \( A \in \text{TIME}(n \log(n)) \) using a more efficient algorithm. It’s in the
textbook chapter 7.1. Now we are ready to define the class \( P \)

**Definition 4** \( P = \bigcup_{k \geq 1} \text{TIME}(n^k) \)

Don’t let short definition with an infinite union scare you! All this
is saying is that as long as the running time \( f(n) \) of decider \( M \) is
dominated by some polynomial power \( n^k \), then \( M \) runs in polynomial time. Now, this \( k \) could be arbitrarily large, so \( n^{100}, n^{10000}, n^{10000000} \) are all polynomial running times. To understand this definition better, it is helpful to introduce an additional concept.

**Polynomial Equivalence**

On a high-level, the idea behind polynomial equivalence is simple. Remember when we proved the equivalence of single-tape, multi-tape, and non-deterministic TM recognizers? The story here is similar but there are two crucial differences:

1. As opposed to previously, within the context of computational complexity, all of our machines are deciders. This includes non-deterministic TMs. For a non-deterministic TM to be a decider, it has to halt on every branch.

2. As we’ll see, computational models that we have proved to be equivalent as recognizers can now have vastly different powers in terms of computational efficiency.

Given all this, let’s present the main theorem characterizing polynomial equivalence.

**Theorem 1** Let \( t: \mathbb{N} \to \mathbb{N} \) be function. For every multi-tape TM \( M \) that runs in \( O(t(n)) \) time, there exists an equivalent single-tape TM \( S \) running in \( O(t^2(n)) \) time.

Before we start talking about the proof, let’s talk about the significance of this theorem. It says that if your friend gives you a 6-tape TM that decides a language \( A \) in \( O(n^{42}) \) time, then there exists a single-tape TM that decides \( A \) running in \( O(n^{84}) \) time. Notice that both running times are polynomial. This means that when we try to prove that certain problems are in \( P \), it doesn’t matter which computational model we choose. If we’re able to show that a problem can be solved in polynomial time in one model, then we can show it in the other model.

As for the proof, it’s in the textbook theorem 7.8. But let’s go over the intuition of the proof quickly. To prove the existence of the single tape TM \( S \), we simply construct it. We construct it exactly as we did when we proved that multi-tape recognizers are equivalent to single-tape recognizers. In particular, \( S \) keeps track of \( M \)’s tapes by appending them on its single tape one after the other. \( S \) keeps track of the locations of \( M \)’s heads by placing a special symbol on the corresponding cell.
The basic intuition is this:

1. Since $M$ runs in time $O(t(n))$, no tape of $M$ can be bigger than $O(t(n))$ during any point in $M$’s computation. This is because the tape can grow by at most one cell each step in the worst case scenario if $M$ moves its heads to the right every single time.

2. For $S$ to simulate one step of $M$, it needs to scan/update the entirety of $M$’s tapes one after the other. Say there are $k$ tapes, each one of them is of size $O(t(n))$ by the previous argument, so the scan will take $O(t(n))$ time.

3. Since $S$ takes $O(t(n))$ time for every step of $M$, and $M$ takes $O(t(n))$ steps total, $S$ will take $O(t^2(n))$ steps total.

Please note that while multi-tape TMs are polynomially equivalent to single-tape TMs, we can’t say the same thing (yet!) about non-deterministic TMs. Indeed, if we were to prove such a statement, we would prove that $P=NP$. As a consequence, we’d all win a Fields Medal, go home happy, and eat some cake. Instead, having a non-deterministic machine run in polynomial time $t(n)$ would only give an exponential upper bound $2^{O(t(n))}$ according to theorem 7.11 in the text book. But wait a minute, we haven’t defined what running time is for non-deterministic TMs! Let’s do that in the next section.

**The Class NP**

As promised, let’s begin with a definition of running time for a non-deterministic TMs.

**Definition 5** Given a non-deterministic decider $N$, and a function $f : \mathbb{N} \rightarrow \mathbb{N}$. We say that $N$ runs in time $f(n)$ if $f(n)$ is the maximum number of steps $N$ takes on any input of size $n$ on any computation branch.

Again, suppose your friend, for no good reason, gives you a non-deterministic TM $N$ with input alphabet $\Sigma = \{0, 1\}$. Your friend wants to know $f(1)$. So you go back to your room, run $M$ on all inputs of size 1 (0, 1 in our case). Now, for every input, you’ll have a computation tree. For every computation tree, you measure the longest path from root to leaf. In other words, you measure the depth of the computation tree. Say the depth of the tree was 59 on input 0 and 62 on input 1. You conclude that $f(1) = \max(59, 62) = 62$. You write it on a piece of paper, mail it to your friend, and go to sleep.

We are finally ready to introduce the class NP. The class NP has two characterizations:
1. The class NP is the collection of languages solvable in polynomial time by a non-deterministic TM

2. The class NP is the collection of languages verifiable in polynomial time by a deterministic TM

Let’s now have some intuition about the class NP. Given characterization 1, the class NP includes problems that require a lot of guessing. In particular, a non-deterministic TM has the power to “make guesses” at every step in the computation thereby exploring an exponential number of possibilities in parallel. If any one of these guesses turns out to be correct, the Turing Machine accepts.

A correct guess corresponds to a computational branch in the computation tree. The length of the computational branch will be polynomial since the machine runs in non-deterministic polynomial time. The “nodes” in this branch correspond to the guesses made along the way.

Now, for a deterministic TM, finding the correct sequence of guesses might take a lot of time. In particular, the deterministic TM might try to go through all of the different paths in the computation tree of N corresponding to all the different sequences of guesses sequentially until it finds the right one. This will solve the problem, sure, but it might take an exponential amount of time.

Suppose on the other hand that your friend presents you with a path and tries to convince you that it’s a path of guesses leading to the right answer. At first, you ask your friend “why on earth would you spend your time doing that?” After you’re through with the argument, you use the path your friend gave you as a certificate. In particular, you trace it step by step and verify that it leads to the right solution. Since the path is of polynomial size, it takes you polynomial time to deterministically verify it. This is the idea behind the second characterization.

Combining the two characterizations, you can think about a problem in NP being solved in the following way: A non-deterministic TM first guesses the certificate non-deterministically, then it verifies it deterministically. To make this more concrete, let’s take an example.

**Example 1** Show that

\[ \text{COMPOSITES} = \{x \mid x \text{ is a positive integer such that } x = pq \text{ for some } 1 < p, q < x \} \]

is in NP.
Let’s try to prove this using characterization 2. In other words, what is a certificate that we can use to "easily" verify that a given integer \( x \) is composite? It’s the numbers \( p, q \) themselves! Even though it would be hard for us to come with numbers \( p, q \) that factorize \( x \), if some one came in with them, we can easily verify that they lead to a factorization by checking that \( 1 < p, q < x \) and that \( x = pq \).

On the other hand, if we want to prove this statement using the first characterization, then all we do is "guess" the certificate. In this case, the non-deterministic TM \( N \) guesses \( p, q \). \( N \) can guess \( p, q \) bit-by-bit, for example, and can stop guessing when or before the number of guessed bits reaches that of \( x \). Then \( N \) verifies the certificate by checking that \( x = pq \) and that \( 1 < p, q < x \).