Recitation 08: P, NP, Dynamic Programming

This recitation covers definitions of P and NP, and ways to prove some language is in P and NP.

Definitions of P and NP classes

Let’s recall the definition of classes P and NP. First, we introduce the classes TIME and NTIME.

**Definition 1.** Define $\text{TIME}(t(n))$ to be the set of languages recognized by some 1-tape Turing machine that runs in $O(t(n))$ time on every input with length $n$.

Similarly, define $\text{NTIME}(t(n))$ to be the set of languages recognized by some 1-tape nondeterministic Turing machine that runs in $O(t(n))$ time on every input with length $n$.

With the definitions above, we can define classes P and NP.

**Definition 2.** $P = \bigcup_{i \in \mathbb{N}} \text{TIME}(n^i)$ is the set of languages recognized by some 1-tape Turing machine in polynomial time. Similarly, $NP = \bigcup_{i \in \mathbb{N}} \text{NTIME}(n^i)$ is the set of languages recognized by some 1-tape nondeterministic Turing machine in polynomial time.

Intuitions of P and NP

Once we have the definitions, we can talk about the intuitions.

Intuition of P

From definition 2, we can see that P consists of languages $L$ such that: given an instance $m$, we can check if $m \in L$ quickly. The reason is that $L \in \text{TIME}(n^k)$ for some $k \in \mathbb{N}$, so there is TM $M$ with run time $O(n^k)$ recognizing $L$. We can simply run $M$ on $m$.

Intuition of NP

The intuition of NP is less obvious. It can be formalized as the following theorem.
Theorem 3. NP consists of the languages with short and quickly checkable certificates. Formally, \( L \in \text{NP} \iff \exists \text{ checker } M \in \text{P \ such that } (x \in L \iff \exists |y| = O(poly(x)) \text{ and } (x, y) \in M)). \]

Proof. First, we show that languages in NP have short and quickly checkable certificates. For language \( L \in \text{NP} \), we build TM \( M \) recognizing all accepted computation history of \( L \). The certificate is short since the run time is polynomial, and it’s quickly checkable.

Next, for a language \( L \) with short and quickly checkable certificates, we want to show \( L \in \text{NP} \). Say \( M \in \text{P} \) is the checker, and the certificate length never exceeds \( Cn^C \). We can build NTM \( N \) recognizing \( L \) as follows: on input \( x \), nondeterministically guess certificate \( |y| \leq Cn^C \), accept only if \( (x, y) \in M \). It’s easy to see \( N \) recognizes language \( L \), and the run time is polynomial.

A method of showing something is in NP: certificate

To show some language \( L \in \text{NP} \), by definition, we need to construct an NTM running in polynomial time that recognizes \( L \). Theorem 3 provides us with the following way to build the NTM:

- First, think of a good (short, easy to check) certificate of the language.
- Next, build an NTM as: first, guess the certificate. Then check if it’s valid or not.

Consider the following problem.

Proposition 4. The language \( \text{COMPOSITE} = \{ x \in \{0, 1\}^* \mid x \text{ is composite as a binary number} \} \) is in \( \text{NP} \).

Proof Strategy. A certificate of \( \text{COMPOSITE} \) is a divisor of \( x \). To translate the certificate into an NTM, we can first guess the certificate, then check if it’s valid.

Proof. Build NTM \( M \) as follows:

- On input \( x \), guess \( y \in \{0, 1\}^* \) with \( |y| \leq |x| \).
- Check if \( 1 < y < x \) and \( y \mid x \), accept if true.

It’s easy to see the run time is polynomial in \( |x| \). \( M \) accepts \( x \) if and only if there are proper divisors of \( x \). \( \square \)

A method of showing something is in P: dynamic programming

Dynamic programming is a method to solve some problems in polynomial time. It’s applicable when the problem has the following two properties:

Both "short" and "quickly checkable" mean polynomial in \( n \)

A small detail on guessing: since a TM a fixed number of states, we can’t guess the entire \( y \) all at once (since the number of possibilities may depend on \( n \)). However, we can guess \( y \) bit by bit.
- **Recursive**: The answer for a problem can be computed from answers of smaller subproblems.

- **Memory**: The total number of different subproblems is polynomial.

If a problem has the properties above, it can be solved with dynamic programming. Dynamic programming is essentially a recursion algorithm while memorizing the answers of solved subproblems, so each subproblem is solved once. The recursive approach is referred to as the top-down approach.

An equivalent but often cleaner way to write up DP solutions is the bottom-up approach. Once you figured out all the subproblems to solve, you can sort them by size, then compute the answers one by one. Each time you encounter a subproblem, you can compute the answer directly since you already have the results for all smaller subproblems.

Let’s illustrate the method with the following example.

**Proposition 5.** The class P is closed under *. More precisely, for language $A \in P$, we have $A^* \in P$.

**Proof strategy.** Say $A$ is recognized by TM $M$ in polynomial time. We want to construct TM $N$ recognizing $A^*$ in polynomial time.

On an input instance $x$, checking whether $x \in A^*$ can be done recursively. More precisely, $x \in A^*$ if and only if at least one of the two followings holds:

- $x \in A$ or $x = \epsilon$;
- There exists $1 \leq i < x$, such that $x_{[1..i]} \in A^*$ and $x_{[i+1..|x|]} \in A^*$

With the properties above, we can get the answer of $x$ from the answer of smaller subproblems $x_{[1..i]}$ and $x_{[i+1..|x|]}$. So the problem has the recursive property.

For the memory property, we want to bound the total number of different subproblems. The subproblems $x_{[1..i]}$ can be decomposed further, however, every subproblem we encounter during the recursion process is a substring $x_{[l..r]}$, so there are at most $|x|^2$ many different subproblems, which is polynomial in input length $|x|$.

Thus, we can solve the problem by dynamic programming. We will adapt the bottom-up approach for writing up.

**Proof.** Build TM $N$ recognizing $A^*$ as follows:

- On input $x$, accept if $x = \epsilon$. Otherwise, consider the subproblems $f_{l,r} = \text{“is } x_{[l..r]} \in A^* \text{ or not?”}$ Sort the subproblems by length (i.e., $r - l + 1$), and consider them from smaller length to longer length.

Here, we are using $x_{[l..r]}$ to denote the substring of $x$ from index $l$ to $r$ inclusive.
• To solve the problem $f_{l,r}$, first we check if $x_{[l...r]} \in A$, and set the answer to yes if true. Otherwise, we enumerate all $l \leq i < r$, and set the answer to yes if $x_{[l...i]} \in A^*$ and $x_{[i+1...r]} \in A^*$ for any $i$. Otherwise, set the answer to no.

• If the answer to $f_{1,|x|}$ is yes, accept. Otherwise, reject.

The correctness is obvious, and the run time is obviously polynomial in input length $|x|$.

\[ \square \]

Summary

In this recitation, we first talk about the definitions of classes P and NP. Intuitively, P are the languages that can be solved quickly, while NP are the languages with short and quickly checkable certificates. To show some language is in NP, it suffices to come up with a certificate, then translating it into an NTM guessing the certificate. To prove some language is in P, a useful method is Dynamic Programming.