Complexity Theory

Investigate time, memory, or other resources required to solve a decidable problem.
Chapter 7: Time Complexity

Definition 7.1 Let $M$ be a deterministic TM that halts on all inputs. The \textit{running time} or \textit{time complexity} of $M$ is a function $f : \mathcal{N} \rightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that $M$ uses on any input of length $n$. If $f(n)$ is the running time of $M$, we say that $M$ runs in time $f(n)$ and that $M$ is an $f(n)$ time Turing machine.

Note that we’re doing \textit{worst-case analysis}. 
Worst-case vs average-case

In average case analysis we consider the time taken by the algorithm to be the average of the times taken on inputs of size $n$.

Most often, algorithms are analyzed by their worst case running times — the reasons for this are:

- This is the only “safe” analysis that provides a guaranteed upper bound on the time taken by the algorithm.
- Often, the average case is as bad as the worst case.
- Average case analysis requires making some assumptions about the probability distribution of the inputs.
- Average case analysis is much harder to do.
Asymptotic Analysis

Coming up with the exact running time of a TM is often quite complicated. Generally, it is enough to estimate the running time using asymptotic analysis.

Consider two Turing machines with the following running times:

- $2n^2$
- $50n + 1000$

In order to compare these running times, the most important is the rate-of-growth ($n^2$ vs $n$).
Definition 7.2: Let $f$ and $g$ be two functions from $\mathbb{N}$ to $\mathbb{R}^+$. We say that $f(n) = O(g(n))$ ($f$ is Big-Oh of $g$) iff there exist positive constants $c$ and $n_0$ such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$ 

$g(n)$ is an asymptotic upper bound for $f(n)$. (You can think of $f(n) = O(g(n))$ as “$f \leq g$ if we ignore multiplicative constants.”)
Examples

The following are all true.

- \( n = O(n) \)
- \( 0.0001n = O(n) \)
- \( 1000n = O(n) \)
- \( 100 \log n = O(n) \)
- \( 100n + 1000 \log n = O(n) \)
- \( n = O(n^2) \)
- \( 10n^2 + 2367n + 12 = O(n^2) \).
- \( n \log n = O(n^2) \)
- \( n^2 + 0.00000001n^3 \neq O(n^2) \)
- \( 0.00000001n^3 \neq O(n^2) \)
- \( \log_{10} n = O(\log_2 n) \)
Let \( f \) and \( g \) be two functions from \( \mathcal{N} \) to \( \mathcal{R}^+ \). We say that \( f(n) = \Theta(g(n)) \) (\( f \) is Theta of \( g \)) iff there exist positive constants \( c, d, \) and \( n_0 \) such that

\[
 cg(n) \leq f(n) \leq dg(n) \quad \text{for all } n \geq n_0.
\]

\( g(n) \) is an asymptotically tight bound for \( f(n) \).
(You can think of \( f(n) = \Theta(g(n)) \) as "\( f = g \) if we ignore multiplicative constants.")

Note that \( f(n) = \Theta(g(n)) \) iff \( f(n) = O(g(n)) \) and \( g(n) = O(f(n)) \).

Go through the list from the previous slide. Replace all \( O \)'s by \( \Theta \)'s. Which of the statements are still true?
Other asymptotic notations

$o$ denotes an upper bound that is not asymptotically tight.

**Definition 7.5** Let $f$ and $g$ be two functions from $\mathcal{N}$ to $\mathcal{R}^+$. We say that $f(n) = o(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$ 

In other words, for any $c > 0$ there exists a positive constant $n_0$ such that

$$f(n) < cg(n) \quad \forall n \geq n_0$$

(You can think of $f(n) = o(g(n))$ as “$f < g$ if we ignore multiplicative constants.”)
Time Complexity Classes

**Definition 7.7** Let \( t : \mathbb{N} \rightarrow \mathbb{N} \) be a function. Define the *time complexity class* \( \text{TIME}(t(n)) \), to be

\[
\text{TIME}(t(n)) = \{ L \mid L \text{ is a language decided by an } O(t(n)) \text{ TM} \}.
\]

**Examples**

- Let \( B = \{ w \# w \mid w \in \{0, 1\}^* \} \) (see page 167[139]).
- Let \( A = \{ 0^{2^n} \mid n \geq 0 \} \) (see page 171[143]).

\( B \in \text{TIME}(n^2) \). Why? Can you do better?

\( A \in \text{TIME}(n \log n) \). Why? We cannot do better, since any language that can be decided by a single-tape TM that runs in time \( o(n \log n) \) is regular. (We won’t prove this).
Single-tape versus multitape

Note that we can decide $B$ in time $O(n)$ (linear time) on a 2-tape TM.

Multitape TMs can speed up computation, but not arbitrarily much:

**Theorem 7.8** Let $t : \mathbb{N} \to \mathbb{N}$ be a function such that $t(n) \geq n$ for all $n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.

How would you prove this?

Is the assumption that $t(n) \geq n$ for all $n$ very restrictive?
Nondeterministic Time

Definition 7.9 Let $N$ be a nondeterministic TM that is a decider. The running time of $N$ is a function $f : \mathcal{N} \rightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that $N$ uses on any branch of its computation on any input of length $n$.

Definition 7.21 Let $t : \mathcal{N} \rightarrow \mathcal{N}$ be a function. Define the nondeterministic time complexity class $\text{NTIME}(t(n))$, to be

$$\text{NTIME}(t(n)) = \{ L \mid L \text{ is a language decided by a } O(t(n)) \text{ NTM} \}.$$

How much can nondeterminism speed up computation time? Formulate a nondeterministic analog of Theorem 7.8. [see Theorem 7.11.]
P: polynomial time

**Definition 7.12** \( P \) is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. That is,

\[
P = \bigcup_k \text{TIME}(n^k).
\]

The class \( P \) is *robust* in that it is the same class for all reasonable computational models and all reasonable encodings.

Thus, to show that a problem is in \( P \), it suffices to analyze a high-level description.

\( P \) roughly corresponds to the class of problems that are realistically solvable on a computer.

In complexity theory, we often disregard polynomial differences. This does not mean that those differences are not important!
Do we know any problems in P?

Yes! Most or all of the algorithms that you have seen, like sorting algorithms, shortest path on a graph ... are in P. (Of course these problems are usually not decision problems, but we can formulate analogous decision problems.)

Are all problems in P?
No! For example, the halting problem is not in P, because it is not even decidable.

Are there problems that are decidable problems but not in P? Yes! There are problems that provably need exponential time, doubly-exponential time, ...

There are also a large number of problems for which we don’t know whether they are in P or not......
Encodings

Reasonable encodings and decodings for graphs take polynomial time in the number of vertices of the graph. Showing that a graph algorithm runs in polynomial time is equivalent to showing that the algorithm runs in polynomial time in the number of vertices.

Can DFAs and NFAs be encoded and decoded in polynomial time in the number of states?

A reasonable encoding for a number would be binary or decimal. In both cases, the length of natural number \( m \) is \( \Theta(\log m) \).

Unary notation for encoding numbers is not reasonable.
\( E_{\text{DFA}} \text{ is in } P \)

Use Chapter 4’s definition of a TM \( T \) that decides \( E_{\text{DFA}} \) (the emptiness problem for DFAs):

\[ T = \text{“On input } \langle A \rangle, \text{ where } A \text{ is a DFA:} \]

1. Mark the start state of \( A \).
2. Repeat step 3 until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, accept; otherwise reject.”

Let \( m \) be the number of states of \( A \). Show that this algorithm works in polynomial time in \( m \):
- Give an upper bound on the number of stages that the algorithm uses.
- Show that every stage can be implemented in polynomial time.
Use Chapter 4’s definition of a TM \( T \) that decides \( E_{\text{CFG}} \) (the emptiness problem for CFGs):

\[
R = \text{"On input } \langle G \rangle \text{, where } G \text{ is a CFG:}\]

1. Mark all terminal symbols in \( G \).
2. Repeat step 3 until no new variables get marked:
3. Mark any variable \( A \) where \( G \) has a rule \( A \rightarrow U_1 U_2 \cdots U_k \) and each symbol \( U_1, \ldots, U_k \) has already been marked.
4. If the start symbol is not marked, accept; otherwise reject.”

Show that this algorithm runs in polynomial time.
RELPRIME is in P

\[
RELPRIME = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \}
\]

Two numbers are relatively prime if the largest integer that evenly divides them is 1. TM \( M \) decides RELPRIME:

\[
M = \text{"On input } \langle x, y \rangle, \text{ where } x \text{ and } y \text{ are natural numbers:}\n\]

1. Repeat for \( d = 2, 3, \ldots, \min(x, y) \)
2. if \( d \) divides \( x \) and \( y \), reject.
3. Accept

This algorithm runs in time polynomial in \( x + y \), but it is not a polynomial-time algorithm. Why not?

Still, RELPRIME is in P. We just need to come up with a better algorithm than trying all possibilities (this is often called brute-force search).
Dynamic Programming

Dynamic programming is a powerful technique for solving problems with the following properties:

- A solution to the problem can be built from solutions to smaller subproblems. (This gives a recursive algorithm.)
- There are a small number of relevant subproblems.
- The recursive algorithm revisits the same subproblems over and over again.
Dynamic Programming Algorithms

Use *memoization*: Store the solutions of subproblems when they are computed for the first time. The next time(s) you need the solution to the same subproblem, just look it up.

Or write a bottom-up algorithm.

**A simple example**

*Fibonacci numbers* are defined as follows

\[
\begin{align*}
F_0 &= 0, \quad F_1 = 1 \\
F_i &= F_{i-1} + F_{i-2} \text{ for } i \geq 2
\end{align*}
\]
Naive recursive algorithm for Fibonacci numbers

Note: $F$ is not really a description of a TM, since it uses recursion.

$F =$ "On input $0^i$, where $i$ is a non-negative integer:

1. if $i = 0$, output 0 and halt
2. if $i = 1$, output 1 and halt
3. output $F(0^{i-1}) + F(0^{i-2})$

What's the running time of $F$?

What is the number of subproblems to compute $F(0^i)$?"
Memoized Version

\[ F = \text{"On input } 0^i, \text{ where } i \text{ is a non-negative integer:} \]

1. If \( i = 0 \), output 0 and halt
2. If \( i = 1 \), output 1 and halt
3. If \( f[i - 1] = \infty \), \( f[i - 1] := F(0^{i-1}) \)
4. If \( f[i - 2] = \infty \), \( f[i - 2] := F(0^{i-2}) \)
5. Output \( f[i - 1] + f[i - 2] \)

What is the running time?
Naive Bottom-up version

\[ F = \text{"On input } 0^i, \text{ where is } i \text{ a non-negative integer:\}
\]

1. if \( i = 0 \), output 0 and halt
2. if \( i = 1 \), output 1 and halt
3. \( f[0] = 0 \)
4. \( f[1] = 1 \)
5. Repeat for \( j = 2, \ldots, i \)
6. \( f[j] \leftarrow f[j - 1] + f[j - 2] \)
7. output \( f[i] \)

What is the running time?

What’s better: memoized or bottom-up?

In the book (page 290/291) dynamic programming is used to show that every context-free language is in P.