Time Complexity – the P Class

**Review**

So far we have learned about the classes of decidable, undecidable but Turing-recognizable, and unrecognizable problems. In general the class of decidable problems is the class of problems thought to be solvable by computers. An algorithm is a process for determining membership in a language that can be implemented in a Turing machine that halts on all inputs – a decider.

Now we will focus our attention to the decidable problems, which are in principle solvable. We will ask which problems are solvable in a reasonable amount of time. As we will see, not all decidable problems can be solved in practice. Some are intractable. There are problems that require a lot of time (or space) – more time than plausibly available in practice.

**Measuring Complexity**

Let’s start with an example:

\[ L = \{ 0^n 1^n \mid n \geq 0 \} \]

Here is a TM accepting this language:

\[ T = \text{“On input } w, \]
  \[ 1. \text{ Scan the input across from left to right, and syntax check.} \]
  \[ 2. \text{ Keep doing:} \]
    \[ a. \text{ Go back to the first uncrossed 0, and cross it.} \]
    \[ b. \text{ Go forward to the first uncrossed 1, and cross it.} \]
    \[ \text{Until either } 0\text{s or } 1\text{s run out.} \]
  \[ 3. \text{ If } 0\text{s and } 1\text{s run out at the same time, accept. Otherwise reject.”} \]

How many steps does T take, on an input \( 0^i 1^j \) of length \( n = i+j \)?

- Step 1 takes \( n \) steps.
- Step 2 happens at most \( n/2 \) times, and each time:
  - \( n \) steps – go back to first 0 and go to first uncrossed 1.
- Step 3: takes about \( n \) steps.
**Total:** At worst it takes
\[ O(n) + O(n^2) + O(n) = O(n^2) + 2O(n) \]

Now we define a way to talk about the order of magnitude of time a computation takes. For example, \(2n^2+3n\) steps is roughly the same as \(n^2\) steps because (i) the \(3n\) term is too small compared to \(2n^2\), as \(n\) becomes large, and (ii) the \(2n^2\) term is on the order of magnitude of \(n^2\), just a multiplicative constant times that. We want to have time estimates of the order of magnitude, so that we don’t bother with too much detail when analyzing an algorithm.

**Definition:** “Big-O” notation. Let \(f\) and \(g\) be two functions \(f, g: \mathbb{N} \to \mathbb{R}^+ = \{1, 2, \ldots\}\).

We say that \(f(n) = O(g(n))\) if:

There is a constant \(c\), and a constant \(n_0\) such that for every \(n \geq n_0\), \(f(n) \leq cg(n)\).

That is, there is a constant \(c\) such that for \(n\) sufficiently large, \(f\) is at most as big as \(c\) times \(g\). We say that \(g(n)\) is an asymptotic upper bound on \(f(n)\).

**Example.** Let \(f(n) = 5n^3 + 2n^2 + 2000n + 6\).

Then, \(f(n) = O(n^3)\). So \(g(n) = n^3\), and \(f(n) = O(g(n))\)

**Proof:** Let \(c = 6\), and \(n_0 = 100\).
For any \( n \geq n_0 \),
\[
    cg(n) - f(n) = (6-5)n^3 - 2n^2 - 2000n - 6
    = n^3 - 2n^2 - 2000n - 6
    \geq 100^3 - 2\times100^2 - 2000\times100 - 6
    > 0.
\]

We can also say that \( f(n) = O(n^4) \). This is even easier to see because it is a
looser statement. However, \( f(n) \) is not \( O(n^2) \). No matter how big a constant \( c \)
one chooses, one cannot make \( n^2 \) bigger than \( f(n) \) for all \( n \) greater than some \( n_0 \).

**Polynomial bounds** – bounds of the form \( n^c \) for \( c \) greater than 0

**Exponential bounds** – bounds of the form \( 2^{cn} \) where \( c \) is a real number
greater than 0

**Definition.** Little-o notation. We say that \( f(n) = o(g(n)) \) if:

We say that \( f(n) = o(g(n)) \) if:

There is a constant \( c \), and a constant \( n_0 \) such that for every \( n \geq n_0 \), \( f(n) < cg(n) \).

**Example:** \( 100n^2 = o(n^3) \)

**Example:** \( n^2 \) is **not** \( o(1000n^2 + 100n) \).

1. Proof – counter-example: \( c = 1/2000 \). This would mean that \( cg(n) = 1/2n^2 + 1/20n \)

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

\[
    \text{TIME}(t(n)) = \{ L | \text{L is decided by a TM running in } O(t(n)) \text{ time } \}
\]

**Note:** \( \text{TIME}(t(n)) \) includes all languages that are decided in less time than
\( t(n) \), i.e. in \( o(t(n)) \) time. If \( t_1(n) = O(t_2(n)) \), then \( \text{TIME}(t_1(n)) \subseteq \text{TIME}(t_2(n)) \),
because anything decided by a TM running within time \( t_1(n) \) can also be
decided by a TM allowed to run for longer.
Recall the TM for recognizing \( L = \{ 0^n 1^n \mid n \geq 0 \} \). This language was decided by a TM running in time \( n^2 + 2n = O(n^2) \), therefore \( L \in \text{TIME}(n^2) \).

However, we can create an algorithm which is more efficient for \( L \). Consider the following TM:

\[
M = \text{“On input } w, \\
1. \text{ Scan across for syntax check.} \\
2. \text{ Keep doing:} \\
   a. \text{ Scan across the tape to check if the number of 0s + the number of 1s is odd. Reject if it is.} \\
   b. \text{ Scan across the tape, crossing out every second 0.} \\
   c. \text{ Scan across the tape, crossing out every second 1.} \\
3. \text{ If 0s and 1s run out at the same time, accept. Otherwise reject.”}
\]

Running time:
Step 1: Syntax checking takes \( n \) steps.
Step 2: The loop is repeated \( O(\log n) \) times, because each time the number of uncrossed 0s and 1s is cut in half. In each pass of the loop, at most \( 2n \) steps are used (\( n \) to check odd and \( n \) to cross out).

Total: \( O(n) + O(\log n \times 2n) \) Therefore, \( L \in \text{TIME}(n \log n) \).

**Nondeterministic time**

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

\[
\text{NTIME}(t(n)) = \{ L \mid L \text{ is decided by a NTM whose maximum running time of a nondeterministic branch of computation is } O(t(n)) \}
\]
Theorem. Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.

Proof. Let $N$ be a NTM running in time $t(n)$. We build a deterministic TM $D$, simulating $N$ by searching through the nondeterministic tree of computation of $N$. This is similar to the proof that NTMs and TMs are equivalent.

If $b$ is the maximum number of children each node of the tree of nondeterministic computation of $N$ has, then the total number of nodes of the tree is at most $O(b^{t(n)})$.

$D$ will explore the tree breadth-first, by going from the root of the tree to each node, then back to the root.

Visiting a node and going back to the root takes $O(t(n))$ time, so the total time is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. 

\[ \begin{array}{cc}
\text{Deterministic} & \text{Nondeterministic} \\
\uparrow & \uparrow \\
\vdots & \vdots \\
\text{accept/reject} & \text{accept/reject} \\
\downarrow & \downarrow \\
\text{reject} & \text{reject} \\
\end{array} \]
The Class P

A problem solved in time $O(n^2)$ is easier than one solved in time $O(n^4)$, or even worse $O(n^6)$. However, all such problems are extremely easy compared to one that requires time $2^n$. Consider $n = 100$. Then $n^6 = 10^{12}$ which is a large number of operations, but manageable for modern computers. However, $2^{100}$ is somewhat larger than $10^{30}$, which is an overwhelming number.

**Theorem** All reasonable deterministic models of computation are polynomially equivalent. That is, anyone model can simulate another with only a polynomial increase in running time.

**Example.** One can prove that if a language $L$ is decidable in time $O(t(n))$, where $t(n) > n$, on a multitape TM, then $L$ is decided in time $O(t(n)^2)$ on a single-tape TM.

Therefore it is reasonable to define the class of problems that are decidable by some deterministic TM in polynomial time:

**Definition.** $P$ is the class of languages that are decidable in polynomial time on a deterministic TM:

$$ P = \bigcup_{k=1}^{\infty} \text{TIME}(n^k) $$

Two observations

- $P$ is the same for all deterministic models of computation that are equivalent to TMs
- $P$ corresponds reasonably well to the class of problems that are realistically solvable on a computer.

**Example.** PATH problem

$$ \text{PATH} = \{ (G, s, t) \mid G \text{ is an directed graph, } s \text{ and } t \text{ are nodes of } G, \text{ and there is a path between } s \text{ and } t \text{ in } G \} $$
**Theorem.** $\text{PATH} \in \mathbb{P}$.

**Proof.** We describe an algorithm to find a path $s \rightarrow t$ in $G$, if such a path exists.

$M =$ “On input $(G, s, t)$,
1. Mark node $s$.
2. Repeat until no additional nodes are marked:
   Scan all edges in $G$. If $(u, v)$ is found, going from some marked node $u$, to some unmarked node $v$, mark $v$.
3. If $t$ is marked, accept. Otherwise reject.”

Step 2 is repeated $\leq n$ times, where $n = \text{number of nodes} + \text{number of edges}$. Each repetition of step 2 requires $\leq O(n)$ work. Therefore the above is an $O(n^2)$, therefore a polynomial algorithm.

**Find the shortest path:** The above algorithm somewhat modified, can find the shortest path between $s$ and $t$, where each edge has a distance associated with it. To do this, each node is marked with a number corresponding to a distance estimate from $s$ to that node:
1. Initially all nodes are marked with infinity, except s is marked with 0.
2. Do the following n times where n is the number of nodes:
   For each edge (u, v) update the distance associated with v, \( d_v \) to
   \( \min(d_v, d_u + \text{weight}((u,v))) \).

In the end of the algorithm each node \( v \) has a distance \( d_v \), which is the shortest distance from \( s \) to \( v \).

Example: find the shortest path from 1 to 9.

So, finding a path, or indeed a shortest path, between two nodes in a graph is indeed easy (it is in P).

Two numbers are relatively prime if 1 is the largest integer that evenly divides them both.

10 and 21 are relatively prime.

\( RELPRIME \in P \)

Proof.
The Euclidean algorithm \( E \) finds the greatest common divisor between 2 numbers. If that number is 1 then accept, else reject.

It can be shown that the total running time of \( E \) is polynomial

Theorem 7.16 – Every context-free language is a member of P.