CS 530: Theory of Computation

Time Complexity

The number of steps that an algorithm uses on a particular input may depend on several parameters. For simplicity we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters. In **worst-case analysis**, the form we consider here, we consider the longest running time of all inputs of a particular length. In **average-case analysis** we consider the average of all the running times of inputs of a particular length.

Definition 1 (Definition 7.1, page 248) *Let M be a deterministic Turing machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : \mathcal{N} \to \mathcal{N}$, 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.*

Because the exact running time of an algorithm often is a complex expression, we usually just estimate it. In one convenient form of estimation, called **asymptotic analysis**, we seek to understand the running time of the algorithm when it is run on large inputs. We do so by considering only the highest order term of the expression for the running time of the algorithm, disregarding both the co-efficient of that term and any lower order terms. For example, the function $f(n) = 6n^3 + 2n^2 + 20n + 45$ has four terms. The **asymptotic notation** or **big-O notation** for describing this relationship is $f(n) = O(n^3)$. Let $\mathcal{R}^+$ be the set of real numbers greater than 0.

Definition 2 (Definition 7.2, page 249) *Let f and g be two functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist so that for every integer $n \geq n_0$*

$$f(n) \leq cg(n).$$

*When $f(n) = O(g(n))$ we say that $g(n)$ is an **upper bound** for $f(n)$, or more precisely, that $g(n)$ is an **asymptotic upper bound** for $f(n)$, to emphasize that we are suppressing constant factors.*

Frequently we derive bounds of the form n^c for c greater than 0. Such bounds are called **polynomial bounds**. Bounds of the form $2^{(n^\delta)}$ are called **exponential bounds** when δ is a real number greater than 0.

Big-O notation has a companion called **small-o notation**. Big-O notation says that one function is asymptotically *no more than* another. To say that one function is asymptotically *less than* another we use small-o notation.

Definition 3 (Definition 7.5, page 250) *Let f and g be two functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $f(n) = o(g(n))$ if*

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

In other words, $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$.

Definition 4 (Definition 7.7, page 251) *Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function. Define the **time complexity class**, $TIME(t(n))$, to be $TIME(t(n)) = \{L | L$ is a language decided by an $O(t(n))$ time Turing machine $\}$.*

Theorem 5 (Theorem 7.8, page 254) *Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.*

Definition 6 (Definition 7.9, page 255) *Let N be a nondeterministic Turing machine that is a decider. The **running time** of N is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.*

Theorem 7 (Theorem 7.11, page 256) *Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.*

Definition 8 (Definition 7.12, page 258) *P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,*

$$P = \bigcup_k TIME(n^k).$$

The class P plays a central role in our theory and is important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

A **Hamilton path** in a directed graph G is a directed path that goes through each node exactly once. We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes. Let $HAMPATH = \{\langle G\rangle, \langle s\rangle, \langle t\rangle | G$ is a directed graph with a Hamiltonian path from s to $t\}$. No one knows whether $HAMPATH$ is solvable in polynomial time.

The $HAMPATH$ problem does have a feature called **polynomial verifiability**. Even though we don't know of a fast (i.e., polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using the exponential time algorithm), we could easily convince someone else of its existence, simply by presenting it. In other words, *verifying* the existence of a Hamiltonian path may be much easier than *determining* its existence.

Another polynomially verifiable problem is compositeness. Recall that a natural number is **composite** if it is the product of two integers greater

than 1 (i.e., a composite number is one that is not a prime number). Let $COMPOSITES = \{\langle x \rangle \mid x = pq \text{ for integers } p, q > 1\}$. Here the representation of an integer is that integer written in binary. We can easily verify that a number is composite — all that is needed is a divisor of that number.

Definition 9 (Definition 7.18, page 265) *A **verifier** for a language A over alphabet Σ is an algorithm V, where*

$$A = \{w \mid V \text{ accepts } w@c \text{ for some string } c\}.$$

*Here the symbol $@ \notin \Sigma$. We measure the time of a verifier only in terms of the length of w, so a **polynomial time verifier** runs in polynomial time in the length of w. A language A is **polynomially verifiable** if it has a polynomial time verifier.*

A verifier uses additional information, represented by the symbol c in Definition 7.18, to verify that a string w is a member of A. This information is called a **certificate**, or **proof**, of membership in A.

Definition 10 (Definition 7.19, page 266) *NP is the class of languages that have polynomial time verifiers.*

The term NP comes from **nondeterministic polynomial time** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines.

Theorem 11 (Theorem 7.20, page 266) *A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.*

Definition 12 (Definition 7.21, page 267) $NTIME(O(t(n))) = \{L | L$ *is a language decided by a $O(t(n))$ time nondeterministic Turing machine* $\}$.

Corollary 13 (Corollary 7.22, page 267) $NP = \bigcup_k NTIME(n^k)$.