C Computational Complexity

When discussing various algorithms, it is useful to analyze their *computational complexity*, which refers to the resources required to run them to completion.[1] We are generally interested in either time or space complexity. This appendix reviews asymptotic notation, which is what is generally used to characterize complexity. We then review a few of the complexity classes that are relevant to the algorithms in the book and discuss the problem of decidability.

[1] The analysis of algorithms represents a large field within computer science. For an introductory textbook, see O. Goldreich, *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008. A rigorous treatment requires the introduction of concepts and computational models, such as *Turing machines*, which we will bypass in our discussion here.

C.1 Asymptotic Notation

Asymptotic notation is often used to characterize the growth of a function. This notation is sometimes called *big-Oh notation*, since the letter O is used because the growth rate of a function is often called its *order*. This notation can be used to describe the error associated with a numerical method or the time or space complexity of an algorithm. This notation provides an upper bound on a function as its argument approaches a certain value.

Mathematically, if $f(x) = O(g(x))$ as $x \to a$, then the absolute value of $f(x)$ is bounded by the absolute value of $g(x)$ times some positive and finite c for values of x sufficiently close to a:

$$|f(x)| \leq c|g(x)| \quad \text{for } x \to a \tag{C.1}$$

Writing $f(x) = O(g(x))$ is a common abuse of the equal sign. For example, $x^2 = O(x^2)$ and $2x^2 = O(x^2)$, but, of course, $x^2 \neq 2x^2$. In some mathematical texts, $O(g(x))$ represents the set of all functions that do not grow faster than $g(x)$. For example, $5x^2 \in O(x^2)$. Example C.1 demonstrates asymptotic notation.

If $f(x)$ is a *linear combination*[2] of terms, then $O(f)$ corresponds to the order of the fastest-growing term. Example C.2 compares the orders of several terms.

[2] A linear combination is a weighted sum of terms. If the terms are in a vector $\mathbf{x}$, then the linear combination is $w_1x_1 + w_2x_2 + \cdots = \mathbf{w}^\top\mathbf{x}$.

Example C.1. Asymptotic notation for a constant times a function.

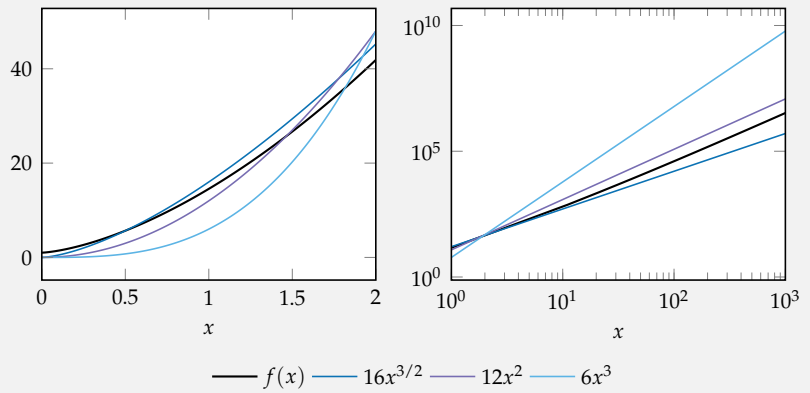
Consider $f(x) = 10^6 e^x$ as $x \to \infty$. Here, f is a product of the constant 10^6 and e^x. The constant can simply be incorporated into the bounding constant c as follows:

$$\begin{aligned} |f(x)| &\leq c|g(x)| \\ 10^6|e^x| &\leq c|g(x)| \\ |e^x| &\leq c|g(x)| \end{aligned}$$

Thus, $f = O(e^x)$ as $x \to \infty$.

Example C.2. Finding the order of a linear combination of terms.

Consider $f(x) = \cos(x) + x + 10x^{3/2} + 3x^2$. Here, f is a linear combination of terms. The terms $\cos(x)$, x, $x^{3/2}$, x^2 are arranged in order of increasing value as x approaches infinity. We plot $f(x)$ along with $c|g(x)|$, where c has been chosen for each term such that $c|g(x = 2)|$ exceeds $f(x = 2)$.



There is no constant c such that $f(x)$ is always less than $c|x^{3/2}|$ for sufficiently large values of x. The same is true for $\cos(x)$ and x.

We find that $f(x) = O(x^3)$, and in general, $f(x) = O(x^m)$ for $m \geq 2$, along with other function classes like $f(x) = e^x$. We typically discuss the order that provides the tightest upper bound. Thus, $f = O(x^2)$ as $x \to \infty$.

C.2 Time Complexity Classes

The difficulty of solving certain problems can be grouped into different time complexity classes. Important classes that appear frequently throughout this book include

- *P*: problems that can be solved in polynomial time,
- *NP*: problems whose solutions can be verified in polynomial time,
- *NP-hard*: problems that are at least as hard as the hardest problems in NP, and
- *NP-complete*: problems that are both NP-hard and in NP.

The formal definitions of these complexity classes are rather involved. It is generally believed that P $\neq$ NP, but it has not been proven and remains one of the most important open problems in mathematics. In fact, modern cryptography depends on the fact that there are no known efficient (i.e., polynomial time) algorithms for solving NP-hard problems. Figure C.1 illustrates the relationships among the complexity classes, under the assumption that P $\neq$ NP.

A common approach to proving whether a particular problem Q is NP-hard is to come up with a polynomial transformation from a known *NP-complete* problem[3] Q' to an instance of Q. The *3SAT* problem is the first known NP-complete problem and is discussed in example C.3.

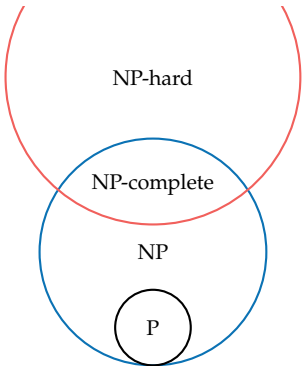


Figure C.1. Complexity classes.

[3] There are many well-known NP-complete problems, as surveyed by R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum, 1972, pp. 85–103.

C.3 Space Complexity Classes

Another set of complexity classes pertain to space, referring to the amount of memory required to execute an algorithm to completion. The complexity class *PSPACE* contains the set of all problems that can be solved with a polynomial amount of space, without any considerations about time. There is a fundamental difference between time and space complexity, in that time cannot be reused, but space can be. We know that P and NP are subsets of PSPACE. It is not yet known, but it is suspected, that PSPACE includes problems not in NP. Through polynomial time transformations, we can define *PSPACE-hard* and *PSPACE-complete* classes, just as we did with NP-hard and NP-complete classes.

Example C.3. The 3SAT problem, which is the first known NP-complete problem.

The problem of *Boolean satisfiability* involves determining whether a Boolean formula is *satisfiable*. The Boolean formula consists of conjunctions ($\wedge$), disjunctions ($\vee$), and negations ($\neg$) involving n Boolean variables $x_1, \ldots x_n$. A literal is a variable x_i or its negation $\neg x_i$. A 3SAT clause is a disjunction of up to three literals (e.g., $x_3 \vee \neg x_5 \vee x_6$). A 3SAT formula is a conjunction of 3SAT clauses like

$$\begin{aligned} F(x_1, x_2, x_3, x_4) = &(\quad x_1 \quad \vee \quad x_2 \quad \vee \quad x_3 \quad) \quad \wedge \\ &(\quad \neg x_1 \quad \vee \quad \neg x_2 \quad \vee \quad x_3 \quad) \quad \wedge \\ &(\quad x_2 \quad \vee \quad \neg x_3 \quad \vee \quad x_4 \quad) \end{aligned}$$

The challenge in 3SAT is to determine whether a possible assignment of truth values to variables exists that makes the formula true. In the formula above,

$$F(\text{true}, \text{false}, \text{false}, \text{true}) = \text{true}$$

Hence, the formula is satisfiable. Although a satisfying assignment can be easily found for some 3SAT problems, sometimes just by quick inspection, they are difficult to solve in general. One way to determine whether a satisfying assignment can be made is to enumerate the 2^n possible truth values of all the variables. Although determining whether a satisfying truth assignment exists is difficult, verification of whether a truth assignment leads to satisfaction can be done in linear time.

C.4 Decidability

An *undecidable* problem cannot always be solved in finite time. Perhaps one of the most famous undecidable problems is the *halting problem*, which involves taking any program written in a sufficiently expressive language[4] as input and deciding whether it will terminate. It was proved that there is no algorithm that can perform such an analysis in general. Although algorithms exist that can correctly determine whether some programs terminate, there is no algorithm that can determine whether any arbitrary program will terminate.

[4] The technical requirement is that the language is *Turing complete* or *computationally universal*, meaning that it can be used to simulate any Turing machine.