

Chapter 4

Algorithm Analysis

The term “algorithm analysis” refers to mathematical analysis of algorithms for the purposes of determining their consumption of resources such as the amount of total work they perform, the energy they consume, the time to execute, and the memory or storage space that they require. When analyzing algorithms, it is important to be precise enough so that we can compare different algorithms to assess for example their suitability for our purposes or to select the better one, *and* to be abstract enough so that we don’t have to look at minute details of compilers and computer architectures.

To find the right balance between precision and abstraction, we rely on two levels of abstraction: asymptotic analysis and cost models. Asymptotic analysis enables abstracting over small factors contributing to the resource consumption of an algorithm such as the exact time a particular operation may require. Cost models make precise the cost of operations performed by the algorithm but usually only up to the precision of the asymptotic analysis. Of the two forms of cost models, machine-based models and language-based models, in this course, we use a language-based cost model. Perhaps the most important reason for this is that when using a machine-based cost model, the complexity of both the analysis and the specification of the algorithm increases because of the need to reason about the mapping parallel algorithm to actual parallel hardware, which usually involves scheduling of parallel computations over multiple processors.

In the rest of this chapter, we present a brief overview of asymptotic notation, and then discuss cost models and define the cost models used in this course. We finish with recurrence relations and how to solve them.

4.1 Asymptotic Complexity

If we analyze an algorithm precisely, we usually end up with an equation in terms of a variable characterizing the input. For example, by analyzing the work of the algorithm A for problem P in terms of its input size, we may obtain the equation: $W_A(n) = 2n \log n + 3n + 4 \log n + 5$. By

applying the analysis method to another algorithm, algorithm B , we may derive the equation: $W_B(n) = 6n + 7 \log^2 n + 8 \log n + 9$.

When given such equations, how should we interpret them? For example, which one of the two algorithms should we prefer? It is not easy to tell by simply looking at the two equations. But what we can do is to calculate the two equations for varying values of n and pick the algorithm that does the least amount of work for the values of n that we are interested in.

In the common case, in computer science, what we care most about is how the cost of an algorithm behaves for large values of n —the input size. Asymptotic analysis offers a technique for comparing algorithms at such large input sizes. For example, for the two algorithms that we considered in our example, via asymptotic analysis, we would derive $W_A(n) = \Theta(n \log n)$ and $W_B(n) = \Theta(n)$. Since the first function $n \log n$ grows faster than the second n , we would prefer the second algorithm (for large inputs). The difference between the exact work expressions and the “asymptotic bounds” written in terms of the “Theta” functions is that the latter ignores so called *constant factors*, which are the constants in front of the variables, and *lower-order terms*, which are the terms such as $3n$ and $4 \log n$ that diminish in growth with respect to $n \log n$ as n increases.

In addition to enabling us to compare algorithms, asymptotic analysis also allows us to ignore certain details such as the exact time an operation may require to complete on a particular architecture. Specifically, when designing our cost model, we take advantage of this to assign most operations unit costs even if they require more than that unit work.

Question 4.1. *Do you know of an algorithm that compared to other algorithms for the same problem, performs asymptotically better at large inputs but poorly at smaller inputs.*

Compared to other algorithms solving the same problem, some algorithm may perform better on larger inputs than on smaller ones. A classical example is the merge-sort algorithm that performs $\Theta(n \log n)$ work but performs much worse on smaller inputs than the asymptotically less efficient $\Theta(n^2)$ -work insertion sort. Note that we may not be able to tell that insertion-sort performs better at small input sizes by just comparing their work asymptotically. To do that, we will need to compare their actual work equations which include the constant factors and lower-order terms that asymptotic notation omits.

We now consider the three most important asymptotic functions, the “Big-Oh”, “Theta”, and “Omega.” We also discuss some important conventions that we will follow when doing analysis and using these notations. All of these asymptotic functions are defined based on the notion of asymptotic dominance, which we define below. Throughout this chapter and more generally in this course, the cost functions that we consider must be defined as functions whose domains are natural numbers and whose range is real numbers. Such functions are sometimes called *numeric functions*.

Definition 4.2 (Asymptotic dominance). Let $f(\cdot)$ and $g(\cdot)$ be two (numeric) functions, we say that $f(\cdot)$ asymptotically dominates $g(\cdot)$ if there exists positive constants c and n_0 such that for all $n \geq n_0$,

$$|g(n)| \leq c \cdot f(n),$$

When a function $f(\cdot)$ asymptotically dominates another $g(\cdot)$, we say that $f(\cdot)$ grows faster than $g(\cdot)$: the absolute value of $g(\cdot)$ does not exceed a constant multiple of $f(\cdot)$ for sufficiently large values.

Big-Oh: $O(\cdot)$. The asymptotic expression $O(f(n))$ is the set of all functions that are asymptotically dominated by the function $f(n)$. Intuitively this means that the set consists of the functions that grow at the same or slower rate than $f(n)$. We write $g(n) \in O(f(n))$ to refer to a function $g(n)$ that is in the set $O(f(n))$. We often think of $f(n)$ being an *upper bound* for $g(n)$ because $f(n)$ grows faster than $g(n)$ as n increases.

Definition 4.3. For a function $g(n)$, we say that $g(n) \in O(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $g(n) \leq c \cdot f(n)$.

If $g(n)$ is a finite function ($g(n)$ is finite for all n), then it follows that *there exist constants k_1 and k_2 such that for all $n \geq 1$,*

$$g(n) \leq k_1 \cdot f(n) + k_2,$$

where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{n_0} |g(i)|$.

Remark 4.4. Make sure to become very comfortable with asymptotic analysis. Also its different versions such as the $\Theta(\cdot)$ and $\Omega(\cdot)$.

Exercise 4.5. Can you illustrate graphically when $g(n) \in O(f(n))$? Show different cases by considering different functions, to hone your understanding.

Omega notation $\Omega(\cdot)$. The “big-oh” notation gives us a way to upper bound a function but it says nothing about lower bounds. The asymptotic expression $\Omega(f(n))$ is the set of all functions that asymptotically dominate the function $f(n)$. Intuitively this means that the set consists of the functions that grow faster than $f(n)$. We write $g(n) \in \Omega(f(n))$ to refer to a function $g(n)$ that is in the set $\Omega(f(n))$. We often think of $f(n)$ being a *lower bound* for $g(n)$.

Definition 4.6. For a function $g(n)$, we say that $g(n) \in \Omega(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $0 \leq c \cdot f(n) \leq g(n)$.

Theta notation: $\Theta(\cdot)$. The asymptotic expression $\Theta(f(n))$ is the set of all functions that grow at the same rate as $f(n)$. In other words, the set $\Theta(f(n))$ is the set of functions that are both in $O(f(n))$ and $\Omega(f(n))$. We write $g(n) \in \Theta(f(n))$ to refer to a function $g(n)$ that is in the set $\Theta(f(n))$. We often think of $f(n)$ being a *tight bound* for $g(n)$.

Definition 4.7. For a function $g(n)$, we say that $g(n) \in \Theta(f(n))$ if there exist positive constants n_0 , c_1 , and c_2 such that for all $n \geq n_0$, we have $0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$.

Important conventions. Even though the asymptotic notations $O(\cdot)$, $\Theta(\cdot)$, $\Omega(\cdot)$ all denote sets, we use the equality relation instead of set membership to state that a function belongs to an asymptotic class, e.g., $g(n) = O(f(n))$ instead of $g(n) \in O(f(n))$. This notation makes it easier to use the asymptotic notation. For example, in an expression such as $4W(n/2) + O(n)$, the $O(n)$ refers to some function $g(n) \in O(n)$ that we care not to specify. Be careful, if there are asymptotics are used multiple times an expression, especially in equalities or other relations. For example, in $4W(n/2) + O(n) + \Theta(n^2)$, the $O(n)$ and $\Theta(n^2)$ refer to functions $g(n) \in O(n)$ and $h(n) \in \Theta(n^2)$ that we care not to specify. But in $4W(n/2) + O(n) = \Theta(n^2)$, we mean to say that the equality is satisfied such that for *any* function $g(n) = O(n)$ and we can find some function $h(n) = \Theta(n^2)$ to satisfy the equality.

4.2 Cost Models: Machine and Language Based

Essentially any analysis must assume a *cost model* that specifies the resource cost of the operations that can be performed by an algorithm. Over time, two ways to define cost models have emerged: machine-based and language-based models.

A machine-based model defines the cost of each (kind of) instruction that can be executed by the machine. When using a machine-based model for analysis, we study the instructions executed by the machine when running an algorithm to bound the resources of interest. A language-based model defines cost as a function from the expressions of the language to cost metric. Such a function is usually defined as a recursive function over the different forms of expressions in the language. When using a language-based model for analysis, we analyze the algorithm by using the cost function provided by the model.

Since we are usually interested in performing asymptotic analysis, we can usually simplify our cost functions in both models by ignoring “constant factors” that depend on the specifics of the actual practical hardware our algorithms may execute on. For example, in a machine model, we can assign unit costs to many different kinds of instructions, even though some may be more expensive than others. Similarly, in a language-based model, we can assign unit costs to all primitive operations on numbers, even though the costs of such operations usually vary.