Chapter 1

Asymptotic Analysis

In algorithms, as in life,
persistence usually pays off.

Steven S. Skiena

1.1 Introduction

What is an algorithm?

Definition 1.1.1 Algorithm
A recipe/description of a process that accomplishes a goal

Why study algorithms? First, you will gain tools to *compare* algorithms. Second, studying algorithms will help you to creatively make *new* algorithms.

Why study algorithms in discrete math? Computers are discrete. Computers run programs, which are based on algorithms. Hence, algorithms must be discrete.

1.2 $\mathcal{O}$, Ω, and Θ Notations

Our motivation here is to describe the *growth-rate* of functions. The growth-rate class gives us key insights to what the actual function does on large inputs.

Definition 1.2.1 Big-Oh
An upper-bound on the growth-rate of a function. Formally:

$$f(n) \in \mathcal{O}(g(n)) \Leftrightarrow (\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+, \forall n \in \mathbb{N}^{\geq n_0})[f(n) \leq c \cdot g(n)]$$

Definition 1.2.2 Big-Omega
A lower-bound on the growth-rate of a function. Formally:

$$f(n) \in \Omega(g(n)) \Leftrightarrow (\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+, \forall n \in \mathbb{N}^{\geq n_0})[f(n) \geq c \cdot g(n)]$$

Definition 1.2.3 Big-Theta
An exact growth-rate of a function. Formally:

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$$

Informally, $f(n) \in \mathcal{O}(g(n))$ just means that **eventually** (for big enough n) we have that (some constant-multiple of) g overtakes f. For example, $2^n \in \mathcal{O}(n!)$ since $2^n < n!$ ($n!$ overtakes 2^n) for $n \geq 4$ ($n \in \mathbb{N}$). So the growth-rate of 2^n is bounded *above* by $n!$. Similarly, $n! \in \Omega(2^n)$ since $n! > 2^n$ for $n \geq 4$. So $n!$ is bounded *below* by 2^n. Oftentimes in algorithm analysis we aim to show an algorithm has exact run-time (Θ), and we do this by showing it has both $\mathcal{O}$ and Ω run-time. Sometimes, though, this is hard to do.

Remark 1.2.4
You may see $f(n) = \mathcal{O}(g(n))$ instead of $f(n) \in \mathcal{O}(g(n))$. Both notations represent the same thing.

1.3 Analyzing Algorithms

We can express algorithms in terms of functions. The function input is typically n, the number of elements inputted into, or input size for, the algorithm, and the function output is typically time. Usually *time* will be the number of certain operations performed during the algorithm.

Example: Come up with a function $T(n)$ that describes the run-time of the following algorithm. Define run-time as the number of conditional evaluations. In the following algorithm, L is a list of integers of length n.

```
function GetMax($L$, $n$)
    $M \leftarrow -\infty$
    $i \leftarrow 0$
    while $i < n$ do
        if $L[i] > M$ then
            $M \leftarrow L[i]$
        end if
        $i \leftarrow i + 1$
    end while
    return $M$
end function
```

Solution: We evaluate the if-statement in line 5 exactly n times, and we evaluate the while-loop condition exactly $n+1$ times (the last time is when we break out of the loop). Thus, $T(n) = n + n + 1 = 2n + 1$

Example: Now come up with a function $U(n)$ that outputs the number of $\leftarrow$ operations. Compare $T(n)$ to $U(n)$.

Solution: We have two initial stores from lines 1 and 2. Each while-loop iteration has a store command in line 8. The while-loop runs exactly n times, so this gives us n stores. Line 6 has a store command which is run $\leq n$ times. We can model this using a natural number $c \leq n$ where c is the number of times line 5 evaluates to true. Our function is thus $U(n) = n + c + 2$
Since $c \leq n$ we have that $U(n) \leq T(n)$ for all $n > 1$. Note that when $n = 1$ we have $T(n) = 3$ and $U(n) = 4$ (since $c = 1$). In any case, we have that the growth-rates of both functions are solely dependent on n and are hence the same. Both functions are in $\Theta(n)$

Definition 1.3.1 The Loop-Heuristic
(algorithm analysis) For each nested loop in an algorithm, multiply n by the number of nested levels. This can be helpful in an initial analysis, however it will only take you so far.

Remark 1.3.2
We state the loop-heuristic as a definition, however it can be rephrased as a provable theorem.

It is important to think about what is going on during the algorithm instead of blindly following the loop-heuristic. We present a few related examples.

Example: Analyze the following algorithm:

```
1: function AnalyzeList($L$, $n$)
2:     $M \leftarrow 0$
3:     for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:         $k \leftarrow L[i]$
5:         for $j \leftarrow i$; $j < n$; $j \leftarrow j + 1$ do
6:             $k \leftarrow k * L[j]$
7:         end for
```

```
        $M \leftarrow M + k$
    end for
    return $M$
end function
```

Solution: The algorithm contains 2 nested loops, so by the loop-heuristic our run-time is n^2. We examine further. The outer loop goes through the list n times. For each iteration i, we have a loop that goes through the list $n - i$ times. The actual *work* is done in the inner loop. Thus, the amount of work equals $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$ by the Gaussian sum. This has a growth rate of n^2, so we see that the loop-heuristic worked and the algorithm runs in $\mathcal{O}(n^2)$. In fact, the amount of work does not change with respect to different input ordering, so the algorithm runs in $\Theta(n^2)$

Example: Examine the following algorithm:

```
function AnalyzeGraph($G = (V, E)$)
    $c \leftarrow 0$
    for $v \in V$ do
        for $w \in$ UnvisitedNeighbors($v$) do
            Visit $w$
            $c \leftarrow c + 1$
        end for
    end for
    return $c$
end function
```

Solution: G represents a graph, where V is the set of vertices and E is the set of edges. We usually denote $|V| = n$ and $|E| = m$.
Let us first think about what the algorithm is *doing*. We instantiate $c = 0$, then go through each unvisited vertex we increment c. We consider the amount of work in the inner loop $\Theta(1)$, and we do this exactly $|V| = n$ times. So our algorithm runs in $\Theta(V)$ (or $\Theta(n)$).

Remark 1.3.3
How would this change had we instead iterated through the *neighbors*, instead of *unvisited neighbors*? We will come back to this

Okay, so how does one determine whether an algorithm is *good*? Well, it depends on how you define *good*. Typically we define *good* as having a low run-time.

Unfortunately, an algorithm's run-time can change depending on the input. Remember: when writing algorithms, you cannot predetermine the input! This heeds way to the idea of *best-case*, *worst-case*, and *average-case* analysis.

Definition 1.3.4 Best-case Analysis
Analyzing an algorithm based on an optimal input that minimizes its run-time.

Best-case analysis is akin to bounding an algorithm's run-time function below, which is just determining the tightest Big-Omega bound. So, Ω essentially tells you how fast the algorithm will run in the best case.

Definition 1.3.5 Worst-case Analysis
Analyzing an algorithm based on a pathological input that maximizes its run-time.

By contrast, worst-case analysis is akin to bounding an algorithm's run-time function above, which is just determining the tightest Big-Oh bound. So, $\mathcal{O}$ essentially tells you how fast the algorithm will run in the worst case.

Definition 1.3.6 Average-case Analysis
Analyzing an algorithm's run-time based on typical, expected, input.

Remark 1.3.7
You may be tempted to think that, as the analogy would imply, average-case analysis is akin to Θ. This is not true. Θ tells you the *exact* run-time, which is when $\mathcal{O}$ and Ω agree. If you know your algorithm run-time has a Θ bound, then the average-case analysis will be this bound. The converse is not true though. If you have the average-case bound, this does not necessarily mean the worst-case or best-case is the same. An important example of this is the Quicksort algorithm.

We present a pathological example to explain why these different analysis types are important.

Example: Examine the following algorithm:

```
1: function PATHOLOGICAL($L$, $n$)
2:     if $n \equiv 0 \pmod{2}$ then
3:
4:     else
5:         return $L[0]$
6:     end if
7:     $c \leftarrow 0$
8:     for $v \in V$ do
```

```
        **for** $w \in$ UnvisitedNeighbors($v$) **do**
            Visit $w$
            $c \leftarrow c + 1$
        **end for**
    **end for**
    **return** $c$
**end function**
```

Solution:

1.3.1 Solving Summations

1.3.2 Solving Recurrences

1.4 Common Complexities

There are a handful of common run-time complexities that you should be familiar with. In this book, we give them to you in ascending order of growth rate in Big Theta notation.

Definition 1.4.1 Constant

$$\Theta(1)$$

Definition 1.4.2 Logarithmic

$$\Theta(\log n)$$

Definition 1.4.3 Linear

$$\Theta(n)$$

Definition 1.4.4 Polynomial

$$\Theta(n^c),\ c > 1$$

We call $\Theta(n^2)$ Quadratic

Definition 1.4.5 Exponential

$$\Theta(c^n),\ c > 1$$

1.5 P vs NP

There exist many YouTube videos discussing the famous P vs. NP problem (ex 1, ex 2). Moreover, if you solve it, the Clay Mathematics Institute has $1,000,000 waiting for you. So what is it?

Definition 1.5.1 P
The class of computational problems that can be solved in *P*olynomial time.

Definition 1.5.2 NP
The class of computational problems where a given solution can be *verified* in polynomial time.

Remark 1.5.3
NP stands for *Nondeterministic Polynomial.* It does **not** stand for *non-polynomial* as some people think.

Definition 1.5.4 Reductions

Definition 1.5.5 NP-Complete
A computational problem is NP-Complete if it satisfies two conditions:

1. The problem is in NP
2. Every problem in NP is reducible to it in polynomial time

Remark 1.5.6
Problems that satisfy the second condition above, and not necessarily the first, are called **NP-Hard**.

The second condition is tricky

Theorem 1.5.1 The Cook-Levin Theorem
The Boolean satisfiability problem is NP-Complete.

Remark 1.5.7
While at this point in the course you likely can understand the proof, we omit it for scope.

We leave this discussion with one final remark. P = NP will render most (all) cryptographic algorithms and services broken. This is because most modern cryptography is built on the observation that factoring very large primes is *hard*. If you are interested, look into RSA and/or the Diffie-Hellman key exchange.

1.6 The Halting Problem

Infinite loops are a software developer's worst nightmare. They are hard to find, debug, and (sometimes) fix. If only there were a piece of software that could examine your code and tell you whether there are any infinite loops. More abstractly, if it could tell you whether your program continues forever or eventually stops.

Well, we are sorry to break the news to you, but this piece of software can never exist; sort-of.

The halting problem is most commonly used as an introduction to computability theory. What problems can we compute? What does it even mean to be computable? These questions are out of scope of this book, but if the following problem interests you, consider taking a course on theory of computation.

Definition 1.6.1 The Halting Problem
A computational problem of determining whether an arbitrary computer program, with input, will stop or run forever.

Think to yourself whether you can come up with a solution program. It is a difficult problem. Alan Turing proved in 1936 that no such algorithm could exist *for all* pairs of programs and inputs. This proof led to (created) the Turing machine, and arguably all of modern computer science. The proof sketch follows. We call it a sketch because the actual proof is significantly more in-depth.

Proof. Assume that such an algorithm (code description, function, method, whatever you want to call it) h exists. The function $h : P \times I \to B$ given by $h : (p, i) \mapsto b$ tells us whether a given input program $p \in P$ (the set of all programs) with input $i \in I$ (the set of all program inputs) will halt (stop) or not: $b \in B = \{\text{True}, \text{False}\}$.
Then the following procedure $\textsc{G} \in P$:

```
1: function G(·)
2:     if h(G, ·) then
3:         while True do                    ▷ loop forever
4:         end while
5:     end if
6: end function
```

We have two cases: $h(\textsc{G}, \cdot)$ returns True, or $h(\textsc{G}, \cdot)$ returns False.
Case 1: $h(\textsc{G}, \cdot) = \text{True}$. Then G stops. But since h returned True, the body of the if-statement will be executed. Then G will loop forever. But this contradicts the output of h.
Case 2: $h(\textsc{G}, \cdot) = \text{False}$. Then G does not stop. But since h returned False, the body of the if-statement is *not* executed, so G will stop. But this contradicts the output of h.
Thus, the halting problem is undecidable (impossible to answer for all

inputs). □

1.7 Summary

-
-
-

1.8 Practice

1. Analyze the following algorithm:
   ```
   1: function SomeFunc(n)
   2:     something
   3: end function
   ```

2. Order the following complexities: $\Theta(n \log n)$, $\Theta(n \log(n^2))$, $\Theta(n)$, $\Theta(n^2)$. Explain why your ordering is correct.

3. Order the following complexities: $\Theta(\log(3n))$, $\Theta(n^n)$, $\Theta(n!)$, $\Theta(\log(\log n))$.