Chapter 4

Complexity

To analyze the correctness of a computer program, we reasoned about the flow of variable values throughout that program and verified logical properties using this information. In addition to correctness, another key property of computer programs we wish to understand is their *complexity*. We can break up complexity into two components:

1. *Temporal Complexity*. How long does a program take to execute?
2. *Spacial Complexity*. How much memory does a program take to execute?

To get a handle on how to analyze the complexity of our programs, we'll build up an appropriate mathematical model of our programs that accounts for complexity. First, we'll study how to *identify* and *count* the important operations that our programs perform in terms of their inputs. Then we'll learn how to *characterize* how these amounts change as the size of our inputs grow.

4.1 Real-world Measurement Versus Mathematical Models

To measure the amount of time that a program takes, we can simply use a clock and measure execution of a program from start to finish. In practice, this is only true way get a sense of what the *real* execution time of a program, independent of any mathematical models we develop about our program. Such a mathematical model is necessarily *incomplete*, it cannot capture all the effects that go into the performance of a program, so it is an *approximation* of the true run time of a program. However, even though this is the case, mathematical models are usually preferable to work with than with the physical process of time itself. This is due to a variety of reasons:

- Computers are fast enough that virtually all algorithms on small inputs perform identically.
- At the same time, developing large enough inputs that deviations appear in a program may be impractical for certain problems.
- To see trends in particular programs, we must run performance tests over many sorts of inputs which can be time-consuming, especially if the programs take a long time to run.
- The parts of a program that we want to analyze may not be amendable to real-world timing, *e.g.*, because that portion of the code is deeply intertwined in the system and cannot be isolated.
- Mathematical models can be platform-independent, or to put it another way, the actual run time of a particular program on a particular machine depends on that machine's configuration which may not be easily duplicable.

To this end, we develop and study mathematical models for understanding the complexity of our programs. Even though our focus is on these models rather than real-world measurement, it is important to keep in mind that such models are not the end-all-be-all of understanding program performance. These models are just one additional tool in your toolbox that you should be able to use when appropriate.

4.2 Identifying Relevant Operations

Each of the operations that our programs perform costs time which contribute to the overall run time of the program. Therefore, rather than measuring the actual time a program takes, we can count the number of operations the program performs. However, different operations take different amounts of time, *e.g.*, the cost of function call is (usually) more than accessing an element of an array. Rather than trying to get a more precise characterization of the run time of a program in terms of all the operations it performs, we can *approximate* its behavior by only counting the most "interesting" operations it performs.

For example, consider the following method:

```java
public static boolean contains(int[] arr, int k) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

This method performs a variety of potentially-interesting operations: variable initialization and assignments, comparisons of integers and returning values. In contrast, the variable initializations (the parameters `arr` and `n` and the local `i`) are uninteresting because:

1. They don't directly deal with what the function is computing (whether `n` is contained in `arr`).
2. The number of times they occur does not vary with the size of `arr` and `n`.

In contrast, the array accesses (`arr[i] == n`) are interesting precisely because they are the heart of the `contains` operation, and the number of such operations depends on the size of the input array. Consequently, reporting the number of variable initializations that `contains` performs is *less interesting* than the number of array accesses because it doesn't give us an accurate sense of how the method behaves. Thus, we would build our model of the method around the number of array accesses instead.

In summary, to build a model of the complexity of our program, we must identify which operations we wish to count as relevant. Such relevant operations (usually) directly perform the computation that is the intent of the program and they have dependence on the input in some way, *e.g.*, the size of some input integer or array.

When there are multiple relevant operations, our choice of which operations to count is arbitrary. Therefore, we choose the operation that makes our further calculations easier. For example, consider the following method:

```java
public static int addAdjacentPairs(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length - 1; i++) {
        sum = sum + arr[i] + arr[i + 1];
    }
}
```

Two choices of relevant operations to count are:

1. The number of assignments to `sum`.
2. The number of array accesses.

Clearly there is a relationship between the two operations—for every assignment to `sum` there are two array accesses. However, we ought to choose assignments for the simple reason that there are less of them to count.

4.3 Counting Operations

Once we have identified which operations we wish to count, then we need to go about the business of counting them. For example, consider the following method:

```
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];    // read
    arr[i] = arr[j]       // read, write
    arr[j] = temp;        // write
}
```

If we identify that we wish to count *array accesses*—both *reads* and *writes* to the input array—then this `swap` method performs 4 such operations, annotated in the comments above. Note that because our programs execute in a sequential manner, we simply count up the number of operations that occur in each statement. If we wanted to count the number of array operations of the following code snippet:

```
int[] arr = new int[10];
// ...
swap(arr, 0, 1);    // 4 ops
swap(arr, 10, 20);  // 4 ops
```

It would be the sum of the two `swap` calls—8 array operations overall.

Note that these counts so far have not depended on the size of the array. In contrast, consider the following method:

```
public static int sum(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

The number of array operations `sum` performs is the number of times the for-loop runs times the number of array operations performed in each iteration. One array access is performed during each iteration and we iterate through the loop `arr.length` number of times. Therefore, the `sum` method performs n operations where n is `arr.length`.

We express the number of operations formally as a *mathematical function* $T(n)$ where n is the size of the input that we have identified in our method.[1] This function serves as our *model* of our method's behavior. For example, we may describe the `sum` method as performing $T_1(n) = n$ operations where n is the size of the input array. Analogously, the function describing the run time of the `swap` function is $T_2(n) = 4$ where n is the size of the input array. Note that while `swap` has three parameters—`arr`, `i`, and `j`—we only identify the length of `arr` as the input to our model. Like choosing which operations to count, we must also ensure that we choose appropriate inputs to our mathematical function so that our model is accurate.

4.3.1 Counting Operations in Loops

Because a loop, by design, performs repetitive behavior, we can derive a formula for the number of operations any loop performs:

$$\text{Total Ops} = \text{No. of Iterations} \times \text{No. of Ops in Loop}$$

This is an instance of the *product rule* from a field of a mathematics called *combinatorics*, the study of counting. We saw how to apply this with the `sum` function:

$$\begin{aligned}\text{No. of Iterations} &= n \\ \text{No. of Ops in Loop} &= 1 \\ \text{Total Ops} &= n \times 1 = n\end{aligned}$$

where n is the size of the input array. This formula generalizes to loops with irregular bounds and odd numbers of operations per loop, for example:

```
for (int i = arr.length - 1; i >= 2; i -= 3) {
    arr[i] = arr[i-1] - arr[i-2];
}
```

Here, the total number of iterations is difficult to see at first glance because the termination condition and decrement step are non-standard. If we write a few examples down:

- $n = 0$, no. of iterations $= 0$
- $n = 1$, no. of iterations $= 0$
- $n = 2$, no. of iterations $= 1$
- $n = 3$, no. of iterations $= 1$
- $n = 4$, no. of iterations $= 1$
- $n = 5$, no. of iterations $= 2$
- $n = 6$, no. of iterations $= 2$
- $n = 7$, no. of iterations $= 2$
- $n = 8$, no. of iterations $= 3$

[1] Note that the name of the function T and the input n are arbitrary. We could name them anything, *e.g.*, $f(x)$ is a more traditional notation for an arbitrary function. Throughout, we'll use the convention of T to represent a time complexity function, S to be a space complexity function, and the input of these functions to be n.

We can derive an explicit formula for the number of iterations: $\lfloor \frac{n+1}{3} \rfloor$ where $\lfloor - \rfloor$ is the *floor function* which rounds its argument down to the next whole number. Thus our formula for the total number of operations is $\lfloor \frac{n+1}{3} \rfloor \times 3$ because the loop performs three array accesses—two reads and one write—with each iteration.

The formula also works for nested loops where we simply calculate the total number of iterations of the inner-most loop and work outwards from there. For example, consider the following doubly-nested for-loop:

```java
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr.length; j++) {
        sum = sum + i + j;
    }
}
```

We know that the inner for-loop performs $3n$ additions (two in the update of sum and one to increment j) where n is the length of the input array. Thus, we know that the outer for-loop performs $(3n+1) \times n = 3n^2 + n$ such operations by accounting for the operations done by the inner loop and the increment of i.

The same technique applies when the bounds of the loops are different, for example, the slightly modified example:

```java
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < i; j++) {
        sum = sum + i + j;
    }
}
```

Now the inner loop's range depends on the current value of the outer loop variable. Again, going through examples to get a sense of the pattern of iteration:

- $i = 0$, No. Iterations of Inner Loop = 0
- $i = 1$, No. Iterations of Inner Loop = 1
- $i = 2$, No. Iterations of Inner Loop = 2
- $i = 3$, No. Iterations of Inner Loop = 3

Thus, if the length of the array is n, the inner loop will perform $1 + 2 + \cdots + n$ iterations. We can use summation notation to concisely write down this pattern:

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i$$

Euler gives us an explicit formula for this summation:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Thus, the total number of additions performed by this loop is

$$3 \cdot \frac{n(n+1)}{2} + n$$

where every iteration of the inner loop produces three additions and there are $\frac{n(n+1)}{2}$ such iterations. The additional n comes from the n increments of the outer loop variable.

Indeed, with arbitrary nestings of loops and bounds, we'll sometimes derive complicated nestings of summations. For example, we can express the number of array operations of the following triply-nested loop:

```
for (int i = 0; i < arr.length; i++) {
    for (int j = i / 2; j < arr.length - 1; j++) {
        for (int k = arr.length - 1; k > j; k--) {
            // 1 array operation performed
        }
    }
}
```

with a straightforward translation of loop bounds to summation bounds:

$$\sum_{i=0}^{n-1} \sum_{j=\frac{i}{2}}^{n-2} \sum_{k=n-1}^{k>j} 1$$

Note how the bounds of the for-loops translates directly into the bounds of the summations. The only complication to consider is that our for-loop bounds are typically exclusive on the upper bound whereas a summation is inclusive. In this particular case, we set up our summations to match the values that the iteration variables of the loops take, for example, 0 to $n-1$ in the case of the outermost for-loop. In general, though, we may find it more convenient to choose different, yet equivalent range of numbers, for example, 1 to n in order to match a summation identity that we know of.

Frequently, this results in summations that are not trivial to simplify without summation identities and algebraic manipulation. Luckily, for the purposes of asymptotic analysis which we discuss shortly later, we don't need to be so precise with our counting because we are concerned more about the *trend* in the number of operations performed as the input size increases rather than the exact amount.

Eventually, we will reach the point where we can apply *informal reasoning* to our analysis and note that there are simply three nested for-loops with linear-like behavior and conclude that the number of operations behaves like $T(n) \approx n^3$. However, it is important to understand the fundamental mathematics involved in this analysis so that you can understand its corresponding strengths and weaknesses.

4.4 Cases of Execution

With this machinery in place, let's revisit our `contains` function:

```
public static boolean contains(int[] arr, int k) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

What is our model $T(n)$ of the time complexity of this function? Again, we choose to model the complexity by counting the number of array accesses of `arr` so n is the size of the input array. However, the analysis here is slightly trickier than `swap` and `sum`. This is because the number of array accesses depends not just on n but ultimately *where the contained element is in the array*, if it is in the array at all. How can we reconcile this problem?

To do this, we need to perform *case analysis* on the execution of the function. More specifically, we'll define three interesting cases—the *best case*, the *worst case*, and the *average case*—make assumptions about the input based on these cases, and then proceed to build our model.

The *best case* assumes that our input is set up in a way that leads to the fastest execution of the program possible. In the case of `contains`, this occurs whenever the requested element `k` is at the front of the array. In this situation, our best-case model for the function is $T(n) = 1$ because our for-loop only runs for one iteration.

The *worst case* assumes that our input is set up in the worst way possible, leading the slowest execution of the program. With `contains`, this occurs when the requested element `k` is either the last element in the array or is not in the array at all. In either case case, the for-loop runs n times (where n is the length of the input array), resulting in the model: $T(n) = n$.

The *average case* assumes something between the best and worst case. On average, what does the input look like and thus what sort of performance can we usually expect from the function? For `contains`, if our element has equal probability of being in any position in the array, its expected index is $\frac{n}{2}$. In this situation, the `for`-loop runs $\frac{n}{2}$ times resulting in the model: $T(n) = \frac{n}{2}$.

The case that we want to use depends on the circumstances and the questions that we are asking about our program. Furthermore, analyzing the cases may be trivial in some cases and non-trivial in others. For example, while the average case may be desirable in most circumstances, it may be difficult to assess the "average" case for a particular problem. In these situations, we may have to resort to using the best and worst cases to put a *lower* and *upper* bound on the performance of our program.

Exercise (Operation Count) Consider the following method:

```java
public static int max(int[] arr) {
    if (arr.length == 0) {
        throw new IllegalArgumentException();   // ignore this case
    } else {
        int ret = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (ret < arr[i]) {
                ret = arr[i];
            }
        }
        return ret;
    }
}
```

Build three mathematical model of the `max` method describing its worst, best, and average case performance. Identify the relevant inputs to the method, operations to count, and give functions that describes the number of operations the method performs in terms of the inputs that you identify.

4.5 Big-O Notation

So far we built up intuition about what it means to perform complexity analysis and learned how to model the number relevant operations a program performs using mathematical functions. However, as mentioned previously, computers are fast enough that on small inputs, virtually any set of algorithms that solve the same problem will perform identically. Rather than worry about small inputs, we would like to understand how our program behavior *scales* when it is given larger inputs. The study of the *limiting* behavior of our programs is called the *asymptotic analysis of algorithms* and the main tool we use to express this behavior is *Big-O notation.* In this section we'll study this mathematical formalism and tie it back to the informal reasoning we will need to perform on a day-to-day basis.

4.5.1 Growth of Functions

We model the complexity of computer programs using mathematical functions as we saw in the reading and lab yesterday. We can categorize the different mathematical functions in terms of how they grow as their inputs grow.

- **Constant Functions** are those functions that do not depend on their input. For example $f(x) = 300$ is a constant function that evaluates to 300, irrespective of its input. Our `swap` function from before performed a constant number of array operations.
- **Linear Functions** take the form $f(x) = mx + b$ where m and b are constants. They correspond to lines. For example, walking an array takes linear time.
- **Quadratic Functions** take the form $f(x) = ax^2 + bx + c$ where a, b, and c are constants. They correspond to curves. Functions with quadratic complexity arise, for example, when we must perform an operation involving all possible pairs of a collection of objects. If there are n objects, then there are $n \times n = n^2$ operations that must be performed.
- **Cubic Functions** take the form $f(x) = ax^3 + bx^2 + cx + d$ where a, b, c, and d. They correspond to curves with an inflection point and have a slope greater than a quadratic function. Functions with cubic complexity arise, for example, when we must perform an operation involving all possible triples of a collection of objects. Like the quadratic case, if there are n objects, then there are $n \times n \times n = n^3$ operations to be performed.
- **Polynomial Functions** generalizes all of the previous functions discussed so far. A polynomial has the form $f(x) = \sum_i a_i x^i + c$ where each a_i and c are constants. We'll usually lump quadratic and cubic functions under the "polynomial" functions and be more specific when we want to talk about linear and constant functions.
- **Exponential Functions** take the form $f(x) = ab^x$ where a and b are constants. They also correspond to curves but with a steeper slope. Exponential functions arise, for example, when we have to consider all possible subsets of a collection of objects. For a collection of n objects, there are 2^n possible such subsets.
- **Factorial**, $f(x) = x!$, corresponds to the number of possible orderings or *permutations* of x elements. If our program needs to generate or consider all of the permutations of a collection of n elements, then its runtime will be $n!$.
- **Logarithmic Functions** take the form $f(x) = \log x$. When using log we usually assume the base of the logarithm is 10 (so that $\log 10^3 = 3$). However, in computer science, we usually assume log is

base 2. It will turn out the base of the logarithm is usually irrelevant for our purposes of asymptotic analysis because via the change-of-base rule—$\log_a x = \frac{\log_b x}{\log_a b}$—logarithms of different bases only differ by a constant amount (the term $\log_a b$ in the rule). Logarithmic functions arise when we are able to divide a problem into sub-problems whose size is reduced by some factor, *e.g.*, by half. When these problems are smaller versions the original problem, we call them "divide-and-conquer" problems and frequently use recursive design to solve them.

- **Linearithmic Functions** are "linear-like" functions by some logarithmic factor, *i.e.*, have the form $f(x) = x \log x$. Linearithmic functions arise when a divide-and-conquer sub-problems requires a linear amount of work. For example, the most efficient general-purpose sorting algorithms have this runtime.

4.5.2 Big-O Notation

When we perform complexity analysis, we would like to classify the growth behavior of our program according to one of the classes of functions listed above. We use *Big-O* notation to capture this fact. When we write $\mathcal{O}(f)$ for some function f, we refer to the set of functions that are all in the same growth class as f. For example, $\mathcal{O}(f)$ where $f(n) = n$, refers to the class of linear functions such as:

$$f_1(n) = 3n f_2(n) = 5n - 3 f_3(n) = 1 - n$$

If we therefore think of $\mathcal{O}(f)$ as a set of functions, we can write $g \in \mathcal{O}(f)$ to mean that function g belongs to the same class of functions that f belongs to (*i.e.*, the class denoted by $\mathcal{O}(f)$. The functions f, f_1, f_2, f_3 above are all in the same complexity class so $f \in \mathcal{O}(f_1)$, $f_1 \in \mathcal{O}(f)$, $f \in \mathcal{O}(f_2)$, $f_2 \in \mathcal{O}(f_1)$, *etc.*.

We can categorizing the complexity of our functions by using Big-O notation in tandem with the mathematical models we build to count the functions' relevant operations. For example:

- The number of array operations performed by the `swap` method is $T(n) = 4$ where n is the size of the input array. We can say that $T \in \mathcal{O}(1)$, declaring that the runtime of `swap` is in the constant complexity class.
- The number of array operations performed by the `sum` method is $T(n) = n$ where n is the size of the input array. We can say that $T \in \mathcal{O}(n)$, declaring that the runtime of `sum` is in the linear complexity class.

Note that when describing the complexity class, we use the simplest function in that class, *e.g.*, n instead of $2n$ or $3n + 5$. While using more complicated classes is technically accurate, such usage loses sight of the fact that we are referring to the class of functions that are linear rather than a particular such function. In particular, with the constant complexity class we write $\mathcal{O}(1)$ since it relates together all constant functions together.

4.5.3 The Formal Definition of Big-O

So far, we've developed an informal idea of Big-O—a classification of the growth rate of mathematical functions. Now let's unveil the specifics. The formal definition of Big-O is:

$$f \in \mathcal{O}(g) \iff \exists c > 0, n_0. \forall n \leq n_0. |f(n)| \leq c|g(n)|$$

What does this mean? First of all, for some function of interest f, we say that $f \in \mathcal{O}(()g)$, pronounced "f is oh-of-g" or "f is order g". This is true whenever there exists ($\exists$) two constants c and n_0 such that for all ($\forall$) n

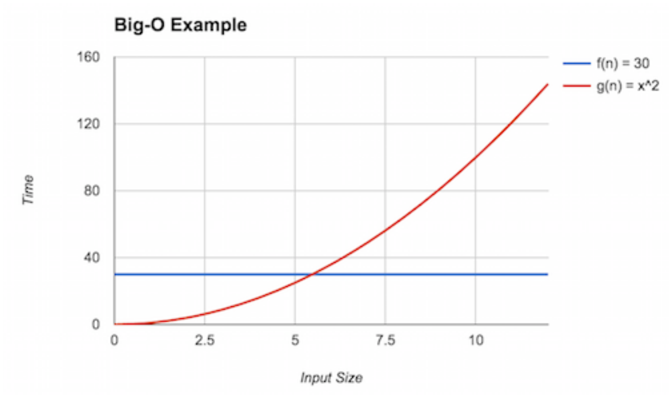
where $n \leq n_0$ the following inequality holds: $|f(n) \leq c|g(n)|$. That is $g(n)$ dominates $f(n)$ by some constant factor c after some starting input n_0.

$f \in \mathcal{O}(()g)$ captures the idea that f is *bounded above* by g.[2] To prove this fact, we must provide the two integers demanded by the existential:

- c, a constant factor that g is multiplied by and
- n_0, the minimum input size to consider.

such that for all input sizes greater than or equal to n_0, $f(n)$ is less than or equal to $c \cdot g(n)$. That is, from n_0 on, f is also smaller (or equal) to g within a constant.

For example, let's show that $f \in \mathcal{O}(g)$ where $f(n) = 30$ and $g(n) = x^2$. First let's examine a graph of the two functions: We can see that eventually g (the red line) dominates f (the blue line), but what is that



point? This is the point where $g(n) = x^2 = 30$. Solving for n yields $n = 30^{1/2} \approx 5.48$. Thus, we can claim that $n_0 = 6$ (rounding up to be safe) and $c = 1$. Here, we see the inequality holds because $f(6) = 30 \leq g(6) = 36$. With this, we can conclude that $f \in \mathcal{O}(g)$.

Note that Big-O provides an *upper bound* on the asymptotic complexity of a function. For example, in the above example $f \in \mathcal{O}(n!)$ as well. To see this, note that for $n \geq 5$, $f(n) \leq g(n)$. However, this is a *weak* upper bound because many other classes of functions are "smaller" than factorial, for example, polynomials and linear functions.

We always want to provide the *tightest* bound possible. However, because we are not analyzing pure mathematical functions but computer programs with arbitrary behavior, we will sometimes be unable to give a tight bound because it will be impractical, infeasible, or impossible to give a precise model. We will therefore resort to a less tight bound in these situations. For example, you may suspect that the program has quadratic complexity but have difficulty proving it, so instead, you may claim a cubic bound instead which may be easier to show.

4.5.4 Additional Asymptotic Bounds

Big-O notation gives us a way of describing the behavior of a function as its input grows. In mathematics, we formally define this as taking the limit of the function (typically to infinity). Thus, our techniques above are an example of *asymptotic analysis* where we analyze the limiting behavior of mathematical functions.

[2]The traditional way of writing $f \in \mathcal{O}(()g)$ is $f(x) = \mathcal{O}(g(x))$. However, equality notation here obscures the idea that $\mathcal{O}(g(x))$ denotes a *set* of functions—all functions that differ by $g(x)$ by at most a constant.

The definition of Big-O described previously has many parts to it. By changing this definition we obtain other useful notations for describing the asymptotic behavior of functions. For example, consider flipping the inequality relating f and g. We obtain the following definition:

$$f \in \Omega(g) \iff \exists c > 0, n_0. \forall n \leq n_0. |f(n)| \geq c|g(n)|$$

When it is f that dominates g, we write $f \in \Omega(g)$ (pronounced "f is big-omega of g"). Intuitively, this definition states that g is a *lower bound* of f. That is, f performs no better than g. For example, it is known that any comparison-based sorting algorithm is $\Omega(n \log n)$, *i.e.*, no such algorithm can perform better than $\mathcal{O}(n \log n)$.

If g serves both as an upper- and lower-bound for f, we write $f \in \Theta(g)$ (pronounced "f is big-theta of g").

$$f \in \Theta(g) \iff \exists c_1, c_2 > 0, n_0. \forall n \leq n_0. c_1|g(n) \leq |f(n)| \leq c_2|g(n)|$$

Here, g describes *exactly* the family of functions in which f resides. Often we would like to characterize our models with big-theta bounds. However, it is frequently onerous, difficult, or both to prove that g is a lower bound, so we frequently stick with big-O bounds instead of big-theta.

Finally, note in the statements above that we posit that *there exists* a constant that inflates g appropriately. This implies that there may exist some constants where the desired inequality—less than for big-O and greater than for big-omega—does not hold. We can make a stronger claim by asserting that the inequality holds *for all* constants rather than at least one:

$$f \in o(g) \iff \forall c > 0. \exists n_0. \forall n \leq n_0. |f(n)| \leq c|g(n)|$$
$$f \in \omega(g) \iff \forall c > 0. \exists n_0. \forall n \leq n_0. |f(n)| \geq c|g(n)|$$

These "little" versions of big-O and big-omega ("little-o" and "little-omega", respectively) make a strictly stronger claim than their big counterparts by asserting that the desired inequality holds, irrespective of the chosen constant. In practice, this means that f and g are in strictly different families of functions.

Exercise (Big-O) For each of the three mathematical models of the `max` method that you gave in the previous reading exercises, give a big-O bound on the asymptotic complexity of those models. Give the tightest bounds possible for each.

4.6 Recurrence Relations

So far, we have analyzed the complexity of programs that contain loops by straightforward counting, *e.g.*, if a loop executes n times and performs k operations on each iteration, the total number of operations performed is $k \times k$. However, what about recursive programs? How do we account for recursive calls within these programs? To build mathematical models for these programs, we introduce *recurrence relations*, a particular sort of mathematical function defined in terms of itself, just like recursive programs!

4.6.1 An Example: Factorial

Consider the standard recursive definition of factorial:

```
public static long factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
}
```

How do we analyze the complexity of this function? First we must choose what operations we will track and the corresponding ”input” to our model. We compute factorial by ”stripping” off one unit from the input, n, performing a multiplication in the process. Therefore, we should analyze this function by counting multiplications; our input should be n itself.

Note that our recursive definition of `factorial` is defined by a conditional on n and in general, you should recognize this as an instance of the general recursive program *skeleton*:

```
if (<base case condition) {
    // <base case>
} else {
    // <recursive case>
}
```

Our model will have similar structure. We will define it in terms of the number of operations performed at the base case (when $n = 0$) and at the recursive case (when $n \neq 0$):

$$T(0) = \ldots$$
$$T(n) = \ldots$$

And now we need to give the number of operations that occur in each of these cases. $T(0)$ is straightforward: in the base case, we perform no multiplications, so $T(0) = 0$. However, what about $T(n)$? We perform one multiplication and then perform a recursive call where the input is reduced by one. Our definition for $T(n)$ reflects this exactly:

$$T(n) = 1 + T(n - 1)$$

Thus, our complete *recurrence relation* that models `factorial` is

$$T(0) = 0$$
$$T(n) = 1 + T(n - 1)$$

4.6.2 From Recurrences to Big-O

So far, we have used Big-O notation to “summarize” the behavior of our model as its input grows in size. We cannot immediately use Big-O notation to summarize our recurrence relations because they are not in the form of one of our standard function classes. We therefore need to either (a) *solve* our recurrence relation and derive an explicit formula or (b) derive an *approximate* formula for our relation. Some relations are solvable (*i.e.*, have a closed-form solution), but many others are not, and thus we have to resort to approximations.

It turns out that our recurrence above has an easy closed-form solution that we can derive using the *substitutive method* in the follow manner:

1. First, let's expand the recurrence a few steps to get a feel for the pattern of its behavior:

$$T(n) = 1 + T(n-1) = 1 + (1 + T(n-2)) = 1 + (1 + 1 + T(n-3))$$

2. Next, let's try to rewrite $T(n)$ in terms of the number of expansions or *unfoldings* that we perform of the function. Call this number k.

$$T(n) = 1 + T(n-1) = k + T(n-k)$$

   We arrive at this formula by noting that after k unfoldings, we add k 1s and make a recursive call to $T(n-k)$ in the final unfolding.

3. Finally, we note that the base case of our recursion is when $n = 0$. So we consider the case where the recursive call in the formula above results in the base case. This is when $n = k$ so $T(n-k) = T(n-n) = T(0)$. Plugging in for k gives us an explicit formula for the overall recurrence:

$$T(n) = k + T(n-k) = n + T(0) = n + 0 = n.$$

Thus, the explicit formula for the number of multiplications that `factorial` performs in terms of its input n is $T(n) = n$. In terms of big-O notation, we say that $T \in \mathcal{O}(n)$—that is, `factorial` takes *linear* time with respect to its input.

Exercise (Recurrence Relations) Solve the following recurrence, slightly modified from the reading:

$$T'(0) = 0$$
$$T'(n) = 1 + T(n-2)$$

and give big-O bound on the behavior of T'. In your analysis, assume that n is even and $n \geq 2$. Follow the outline of the proof in the reading as a guide. Does your big-O bound change if, instead of subtracting 1 or 2, you subtract any constant k (assuming $n \geq k$ and k divides n evenly)?