Big-$\mathcal{O}$ Introduction

Emilie Menard Barnard
Foothill College

Now that we've had practice coding and designing projects, let's consider ways to make our programs as *efficient* as possible. Today we will explore **algorithmic complexity** and **Big-$\mathcal{O}$ growth** in software design.

What is Algorithmic Complexity and Why is it Important?

Consider a simple loop in C++:

```
for (int i = 0; i < n; i++ ) {
     cout << i << endl;
}
```

We know that this loop runs exactly n times. What happens when we add in a nested loop?

```
for (int i = 0; i < n; i++ ) {
     for (int j = 0; j < n; j++ ) {
          cout << i + j << endl;
     }
}
```

Each time the first loop runs, the inner loop will run n times. So we have:

Iteration of "i" loop	$i = 0$	$i = 1$	$i = 2$	$i = 3$	...	$i = n - 1$
Number of times "j" loop runs	n	n	n	n	...	n

In total, the nested loops run $\underbrace{n + n + n + n + \cdots + n}_{n \text{ times}} = n \cdot n = n^2$ times.

When n is small, n^2 is also somewhat small. For example, if n is 10 in the above example, our program will perform 100 additions. This is perfectly reasonable for a computer to do in a limited amount of time. What happens, though, when n starts to grow larger?

Let's say $n = 10,000$. Our program now has to perform $10000 \cdot 10000 = 100$ million additions! Sure, computers calculate quickly, but even computers start to slow down a bit when you ask them to perform this many calculations.

In this example, changing our input n from 10 to 10,000 was only a difference of three significant figures, but it increased the number of calculations from 100 to 100 million, a difference of *six* significant figures.

The **algorithmic complexity** is essentially how fast (or slow) an algorithm runs. It is determined by counting the number of operations needed to perform the full algorithm. Our above algorithm runs n^2 times for an input size of n, as we have already calculated. **As a computer scientist, our job is to not only develop algorithms for code, but to also develop *efficient* algorithms.**

I would argue that the algorithm above for adding all values $0, \ldots, n-1$ to all values $0, \ldots, n-1$ is probably not the most efficient.

For example, when $n = 5$ our algorithm performs the following calculations:

$0 + 0$	$\mathbf{1 + 0}$	$2 + 0$	$3 + 0$	$4 + 0$
$\mathbf{0 + 1}$	$1 + 1$	$2 + 1$	$3 + 1$	$4 + 1$
$0 + 2$	$1 + 2$	$2 + 2$	$3 + 2$	$4 + 2$
$0 + 3$	$1 + 3$	$2 + 3$	$3 + 3$	$4 + 3$
$0 + 4$	$1 + 4$	$2 + 4$	$3 + 4$	$4 + 4$

Using this algorithm, we actually end up adding the same pair of numbers multiple times. This means there is room to improve our algorithm!

Of course, knowing exactly how we can improve our algorithm is a bit tricky and takes some practice. We'll be practicing how to do this over the next few lessons in this course so you'll become a better computer scientist. The most important takeaway now is that you see why this is inefficient. Without seeing the reason why, we have no motivation to improve our algorithm. Plus, identifying the inefficiencies can give you hints for improving the efficiency of your algorithm. **First we need to understand why the algorithm isn't the most efficient. Then we can attempt to make it better.**

STOP: Do not continue until you understand why the above algorithm is not efficient. If you have questions, feel free to e-mail me.

What is Big-$\mathcal{O}$ and How is it Calculated?

As computer scientists, we want to use a standardized, objective technique to measure how efficient our algorithms are so that we may compare efficiencies. This is where Big-$\mathcal{O}$ comes into play.

Big-$\mathcal{O}$ measures how much time [1] a program will take to run.

In, fact, we already calculated the Big-$\mathcal{O}$ runtime of our addition algorithm: $\mathcal{O}(n^2)$. The $\mathcal{O}(\ )$ notation means we're observing the **algorithmic complexity** of our algorithm with respect to the *worst-case* scenario. Let's look at more examples of algorithms and their Big-$\mathcal{O}$ runtimes.

Constant Runtime

Any quick, elementary operation that always takes the same time to execute has a constant runtime, which is denoted as $\mathcal{O}(1)$. Adding two numbers, finding the square of an integer, calculating $(-1)^n$ are all examples of algorithms that are performed in constant time.

Linear Runtime

Any algorithm with a runtime proportional to the input size has a linear runtime, which is denoted as $\mathcal{O}(n)$. A simple loop that iterates n times is $\mathcal{O}(n)$. Here I define a "simple loop" as a loop where only a quick, elementary operation is performed during each iteration.

Runtime of Loops

What happens if each iteration of the loop calculates more than just a simple mathematical operation? We've seen already in the case of two nested loops the runtime is $\mathcal{O}(n^2)$:

```
for (int i = 0; i < n; i++ ) {
     for (int j = 0; j < n; j++ ) {
          cout << i + j << endl;
     }
}
```

In the case of three nested loops:

```
for (int i = 0; i < n; i++ ) {
     for (int j = 0; j < n; j++ ) {
          for (int k = 0; k < n; k++ ) {
               cout << i + j + k << endl;
          }
     }
}
```

the runtime is $\mathcal{O}(n^3)$. For four nested loops, it's $\mathcal{O}(n^4)$. Do you see a pattern? ☺

What if we have two loops that are *not* nested? What do you think the runtime is? Hint: it's *not* $\mathcal{O}(n^2)$.

Consider the following example:

```
for (int i = 0; i < n; i++ ) {
     cout << i << endl;
}
for (int j = 0; j < n; j++ ) {
     cout << i + j << endl;
}
```

[1] Big-$\mathcal{O}$ is also used to measure space, but for now we'll focus on time

This is very similar to what we did before, except here the second loop starts *after* the first loop finishes. In other words, the first loop runs n times, then the second loop runs n times. In Big-$\mathcal{O}$ notation, we can express this as follows: $\mathcal{O}(n+n)$.

Note the difference between nested and un-nested loops. When the loops are nested, the runtime was $\mathcal{O}(n \cdot n)$. When they are not nested, the runtime is $\mathcal{O}(n+n)$. It may seem like a small difference in terms of notation, but the difference in terms of algorithmic complexity is quite substantial.

This addition versus multiplication rule can be generalized to more than just loops.**Runtimes of pieces of code that come one after another (i.e. the first piece of code finishes before the second begins) are *added*. Runtimes of pieces of code that are nested (i.e. the first piece of code is still going before the second begins) are *multiplied*.**

Simplifying Big-$\mathcal{O}$ Notation

In the previous section, we determined that the runtime of two un-nested loops is $\mathcal{O}(n+n)$. We can actually simplify this further.

$$\mathcal{O}(n+n) = \mathcal{O}(2n)$$

We can then use the following rules of Big-$\mathcal{O}$ Notation to simplify this even more:

1. Coefficients should be dropped.
2. Keep only the highest-order or "largest" term.

Let's revisit our previously calculated runtime: $\mathcal{O}(2n)$. By rule (1) above, we can drop the coefficient, so this simply becomes $\mathcal{O}(n)$. Thus, the runtime of the following:

```
for (int i = 0; i < n; i++ ) {
     cout << i << endl;
}
for (int j = 0; j < n; j++ ) {
     cout << i + j << endl;
}
```

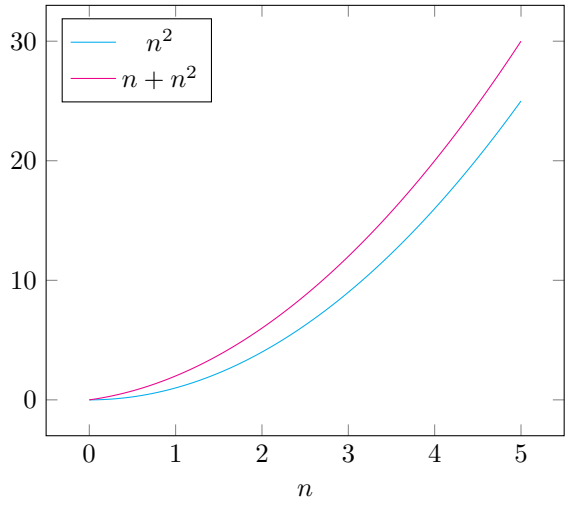
is simply $\mathcal{O}(n)$. Big-$\mathcal{O}$ notation simplifies the runtime as much as it can. Its main concern is what happens for very large values of n. As $n \to \infty$, n and $2n$ are essentially the same (2 times ∞ is still ∞), so we can drop the coefficient.

The same idea holds for a runtime with multiple terms. Consider the following program:

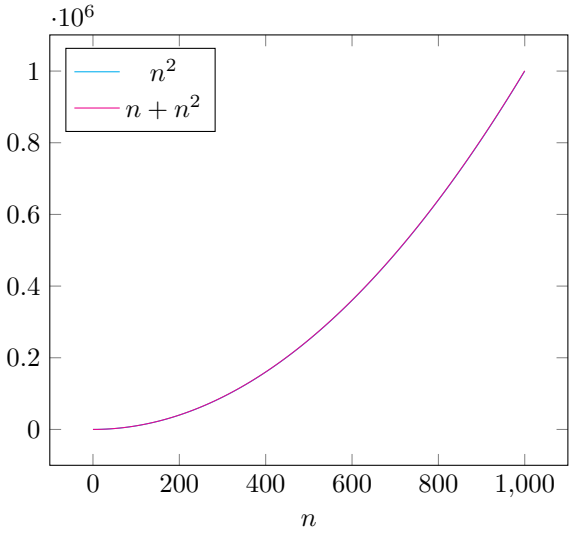
```
for (int i = 0; i < n; i++ ) {
     cout << i << endl;
}
for (int j = 0; j < n; j++ ) {
     for (int k = 0; k < n; k++ )
          cout << j + k << endl;
     }
}
```

The runtime of the first loop (with the variable i) is $\mathcal{O}(n)$, as it runs n times. The next loop has a nested loop, involving j and k. Because these two loops are nested, this has a runtime of $\mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$. Because the first loop finishes completely before the nested loops begin, we add these runtimes to determine the runtime of the entire program: $\mathcal{O}(n) + \mathcal{O}(n^2)$. By rule number (2), we can simplify this to simply $\mathcal{O}(n^2)$.

For small values, it may seem like there is a large difference between $n + n^2$ and n^2:



but as n grows large and approaches ∞, this difference is negligible:



This second chart visualizes why we can simplify Big-$\mathcal{O}$ notation by keeping only the "largest" terms.

Technical Definition

The technical definition of Big-$\mathcal{O}$ is as follows:

Definition. For any functions $f(n)$ and $g(n$, we define $f(n) = \mathcal{O}(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

In other words, the function $g(n)$ is an upper bound for $f(n)$ for all large $n \to \infty$.

A Special Note on Sorting

We will discuss sorting more next week, but I want to briefly introduce it now as it's a commonly-used technique to improve the efficiency of algorithms.

A sorting algorithm takes an unsorted set of values (like a vector of integers):

```
vector<int> unsortedV = {11, 3, 7, 5};
```

and sorts them in increasing order:

```
vector<int> sortedV = {3, 5, 7, 11};
```

Sorting, at best, is $\mathcal{O}(n \log n)$. For now you may generalize a sorting step of your algorithm to have an $\mathcal{O}(n \log n)$ runtime. We will explore this more next week when we learn various sorting techniques.

Try it! Take some time now to determine a way to improve our algorithm above for adding all the values $0, \ldots, n-1$ to all the values $0, \ldots, n-1$. When you have an idea, move on to this week's practice problem.