Extracted from:

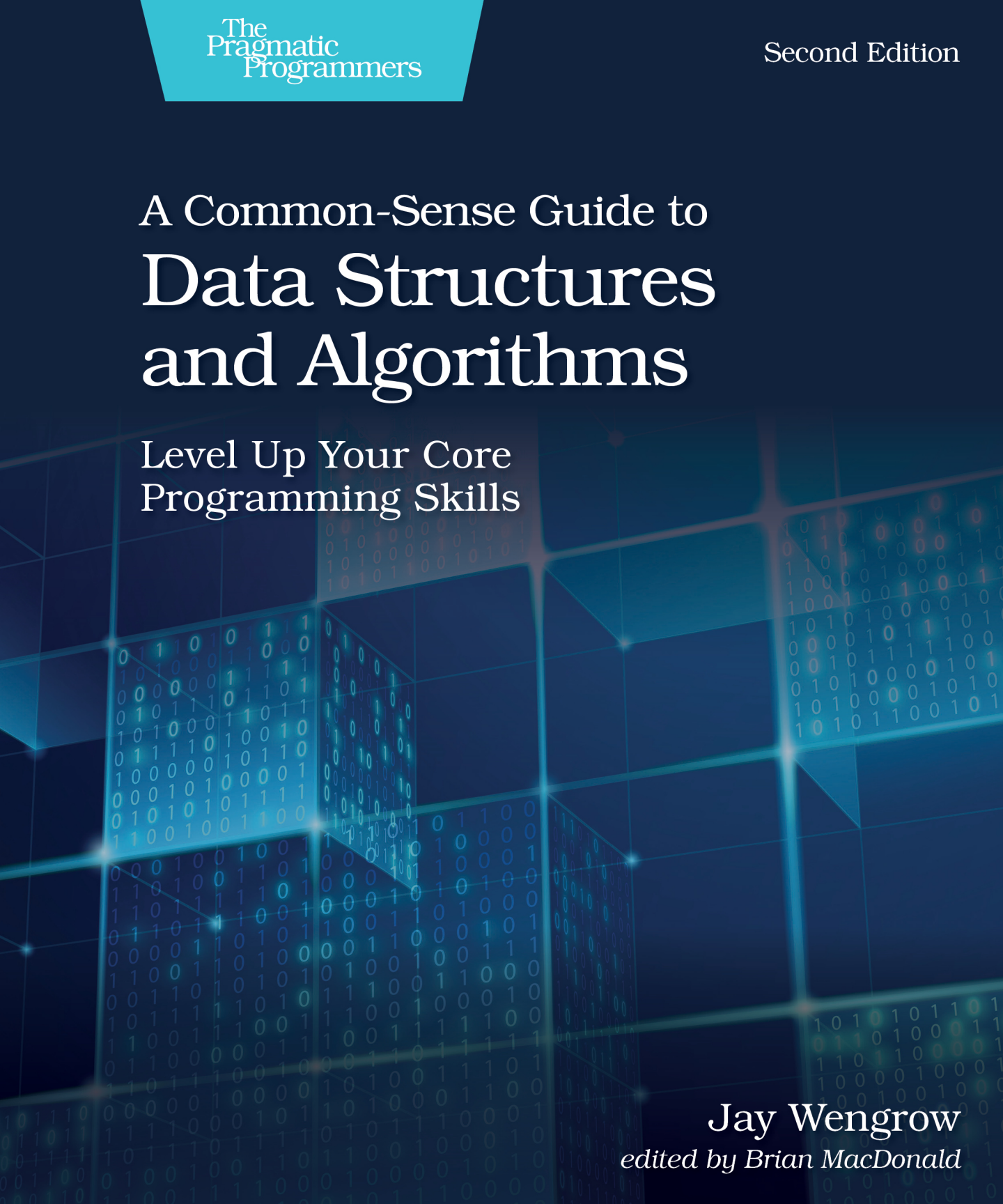
# A Common-Sense Guide to Data Structures and Algorithms, Second Edition

## Level Up Your Core Programming Skills



The Pragmatic Bookshelf

Raleigh, North Carolina

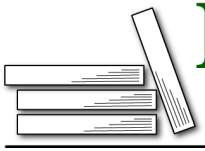

The Pragmatic Programmers
Second Edition
A Common-Sense Guide to
Data Structures and Algorithms
Level Up Your Core Programming Skills
Jay Wengrow
edited by Brian MacDonald

# A Common-Sense Guide to Data Structures and Algorithms, Second Edition

## Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf
Raleigh, North Carolina





Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Brian MacDonald
Copy Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



ISBN-13: 978-1-68050-722-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2020

CHAPTER 7

# Big O in Everyday Code

In the previous chapters, you learned how to use Big O notation to express the time complexity of code. As you've seen, there are quite a few details that go into Big O analysis. In this chapter, we'll use everything you've learned so far to analyze the efficiency of practical code samples that might be found in real-world codebases.

Determining the efficiency of our code is the first step in optimizing it. After all, if we don't know how fast our code is, how would we know if our modifications would make it faster?

Additionally, once we know how our code is categorized in terms of Big O Notation, we can make a judgment call as to whether it may need optimization in the first place. For example, an algorithm that is $O(N^2)$ is generally considered to be a "slow" algorithm. So, if we've determined that our algorithm falls into such a category, we should take pause and wonder if there are ways to optimize it.

Of course, $O(N^2)$ may be the best we can do for a given problem. However, knowing that our algorithm is considered slow can signal to us to dig deeper and analyze whether faster alternatives are available.

In the future chapters of this book, you're going to learn many techniques for optimizing our code for speed. But the first step of optimization is being able to determine how fast our code currently is.

So, let's begin.

## Mean Average of Even Numbers

The following Ruby method accepts an array of numbers and returns the mean average of all its *even* numbers. How would we express its efficiency in terms of Big O?

```ruby
def average_of_even_numbers(array)

  # The mean average of even numbers will be defined as the sum of
  # the even numbers divided by the count of even numbers, so we
  # keep track of both the sum and the count:

  sum = 0.0
  count_of_even_numbers = 0

  # We iterate through each number in the array, and if we encounter
  # an even number, we modify the sum and the count:

  array.each do |number|
    if number.even?
      sum += number
      count_of_even_numbers += 1
    end
  end

  # We return the mean average:

  return sum / count_of_even_numbers
end
```

Here's how to break the code down to determine its efficiency.

Remember that Big O is all about answering the key question: if there are N data elements, how many steps will the algorithm take? Therefore, the first thing we want to do is determine what the "N" data elements are.

In this case, the algorithm is processing the array of numbers passed into this method. These, then, would be the "N" data elements, with N being the size of the array.

Next, we have to determine how many steps the algorithm takes to process these N values.

We can see that the guts of the algorithm is the loop that iterates over each number inside the array, so we'll want to analyze that first. Since the loop iterates over each of the N elements, we know that the algorithm takes at least N steps.

Looking *inside* the loop, though, we can see that a varying number of steps occur within each round of the loop. For each and every number, we check whether the number is even. Then, if the number is even, we perform two more steps: we modify the `sum` variable, and we modify the `count_of_even_numbers` variable. So, we execute three more steps for even numbers than we do for odd numbers.

As you've learned, Big O focuses primarily on worst-case scenarios. In our case, the worst case is when all the numbers are even, in which case we

perform three steps during each round of the loop. Because of this, we can say that for N data elements, our algorithm takes 3N steps. That is, for each of the N numbers, our algorithm executes three steps.

Now, our method performs a few other steps outside of the loop as well. Before the loop, we initialize the two variables and set them to 0. Technically, these are two steps. After the loop, we perform another step: the division of `sum / count_of_even_numbers`. Technically, then, our algorithm takes three extra steps in addition to the 3N steps, so the total number of steps is 3N + 3.

However, you also learned that Big O notation ignores constant numbers, so instead of calling our algorithm O(3N + 3), we simply call it O(N).

## Word Builder

The next example is an algorithm that collects every combination of two-character strings built from an array of single characters. For example, given the array: `["a", "b", "c", "d"]`, we'd return a new array containing the following string combinations:

```
[
  'ab', 'ac', 'ad', 'ba', 'bc', 'bd',
  'ca', 'cb', 'cd', 'da', 'db', 'dc'
]
```

Following is a JavaScript implementation of this algorithm. Let's see if we can figure out its Big O efficiency:

```
function wordBuilder(array) {
  let collection = [];

  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      if (i !== j) {
        collection.push(array[i] + array[j]);
      }
    }
  }

  return collection;
}
```

Here we're running one loop nested inside another. The outer loop iterates over each character in the array, keeping track of the index `i`. For each index `i`, we run an inner loop that iterates again over each character in the same array using the index `j`. Within this inner loop, we concatenate the characters at `i` and `j`, with the exception of when `i` and `j` are pointing to the same index.

To determine the efficiency of our algorithm, we once again need to determine what the N data elements are. In our case, as in the previous example, N is the number of items inside the array passed to the function.

The next step is to determine the number of steps our algorithm takes relative to the N data elements. In our case, the outer loop iterates over all N elements, and for each element, the inner loop iterates again over all N elements, which amounts to N steps multiplied by N steps. This is the classic case of $O(N^2)$, and is often what nested-loop algorithms turn out to be.

Now, what would happen if we modified our algorithm to compute each combination of *three-character* strings? That is, for our example array of `["a", "b", "c", "d"]`, our function would return the array:

```
[
  'abc', 'abd', 'acb',
  'acd', 'adb', 'adc',
  'bac', 'bad', 'bca',
  'bcd', 'bda', 'bdc',
  'cab', 'cad', 'cba',
  'cbd', 'cda', 'cdb',
  'dab', 'dac', 'dba',
  'dbc', 'dca', 'dcb'
]
```

Here's an implementation that uses three nested loops. What is its time complexity?

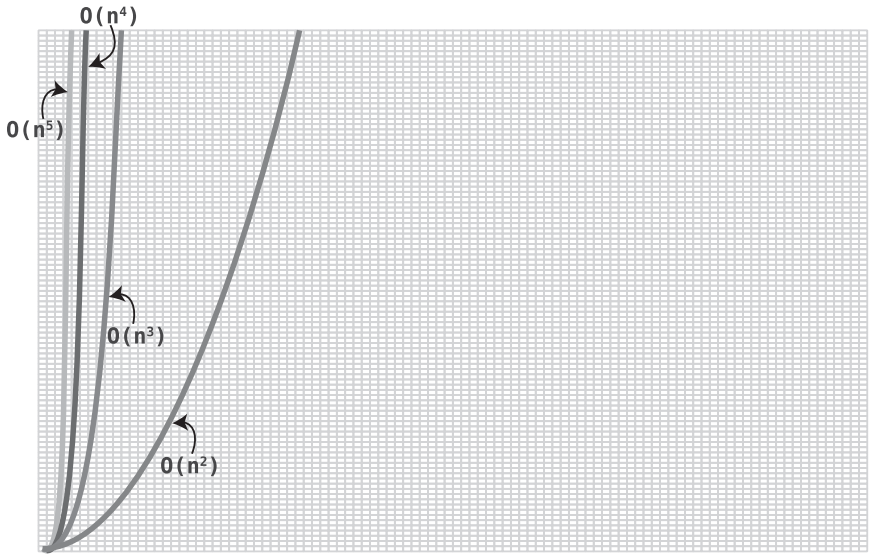
```
function wordBuilder(array) {
  let collection = [];

  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      for(let k = 0; k < array.length; k++) {
        if (i !== j && j !== k && i !== k) {
          collection.push(array[i] + array[j] + array[k]);
        }
      }
    }
  }

  return collection;
}
```

In this algorithm, for N data elements, we have N steps of the `i` loop multiplied by the N steps of the `j` loop multiplied by the N steps of the `k` loop. This is N * N * N, which is $N^3$ steps, which is described as $O(N^3)$.

If we had four or five nested loops, we'd have algorithms that are $O(N^4)$ and $O(N^5)$, respectively. Let's see how these all appear on a graph on page 9.




Optimizing code from a speed of $O(N^3)$ to $O(N^2)$ would be a big win, since the code becomes exponentially faster.

## Array Sample

In the next example, we create a function that takes a small sample of an array. We expect to have very large arrays, so our sample is just the first, middlemost, and last value from the array.

Here is a Python implementation of this function. See if you can identify its efficiency in Big O:

```
def sample(array):
  first = array[0]
  middle = array[int(len(array) / 2)]
  last = array[-1]

  return [first, middle, last]
```

In this case again, the array passed into this function is the primary data, so we can say that N is the number of elements in this array.

However, our function ends up taking the same number of steps no matter what N is. Reading from the beginning, midpoint, and last indexes of an array each takes one step no matter the size of the array. Similarly, finding the array's length and dividing it by 2 also takes one step.

Since the number of steps is constant—that is, it remains the same no matter what N is—this algorithm is considered O(1).

## Average Celsius Reading

Here's another example that involves mean averages. Let's say we're building weather-forecasting software. To determine the temperature of a city, we take temperature readings from across many thermometers across the city, and we calculate the mean average of those temperatures.

We'd also like to display the temperatures in both Fahrenheit and Celsius, but our readings are initially only provided to us in Fahrenheit.

To get the average Celsius temperature, our algorithm does two things: first, it converts all the readings from Fahrenheit to Celsius. Then, it calculates the mean average of all the Celsius numbers.

Following is some Ruby code that accomplishes this. What is its Big O?

```
def average_celsius(fahrenheit_readings)

  # Collect Celsius numbers here:
  celsius_numbers = []

  # Convert each reading to Celsius and add to array:
  fahrenheit_readings.each do |fahrenheit_reading|
    celsius_conversion = (fahrenheit_reading - 32) / 1.8
    celsius_numbers.push(celsius_conversion)
  end

  # Get sum of all Celsius numbers:
  sum = 0.0

  celsius_numbers.each do |celsius_number|
    sum += celsius_number
  end

  # Return mean average:
  return sum / celsius_numbers.length
end
```

First, we can say that N is the number of `fahrenheit_readings` passed into our method.

Inside the method, we run two loops. The first converts the readings to Celsius, and the second sums all the Celsius numbers. Since we have two loops that each iterate over all N elements, we have N + N, which is 2N (plus a few constant steps). Because Big O Notation drops the constants, this gets reduced to O(N).

Don't get thrown off by the fact that in the earlier Word Builder example, two loops led to an efficiency of $O(N^2)$. There, the loops were *nested,* which led to N steps *multiplied* by N steps. In our case, however, we simply have two loops, one after the other. This is N steps *plus* N steps (2N), which is a mere O(N).

## Clothing Labels

Suppose we're writing software for a clothing manufacturer. Our code accepts an array of newly produced clothing items (stored as strings), and creates text for every possible label we'll need.

Specifically, our labels should contain the item name plus its size, ranging from 1 to 5. For example, if we have the array, ["Purple Shirt", "Green Shirt"], we want to produce label text for those shirts like this:

```
[
"Purple Shirt Size: 1",
"Purple Shirt Size: 2",
"Purple Shirt Size: 3",
"Purple Shirt Size: 4",
"Purple Shirt Size: 5",
"Green Shirt Size: 1",
"Green Shirt Size: 2",
"Green Shirt Size: 3",
"Green Shirt Size: 4",
"Green Shirt Size: 5"
]
```

Here is Python code that will create this text for an entire array of clothing items:

```
def mark_inventory(clothing_items):
  clothing_options = []

  for item in clothing_items:
    # For sizes 1 through 5 (Python ranges go UP TO second
    # number, but do not include it):
    for size in range(1, 6):
      clothing_options.append(item + " Size: " + str(size))

  return clothing_options
```

Let's determine this algorithm's efficiency. The clothing_items are the primary data being processed, so N is the number of clothing_items.

This code contains nested loops, so it's tempting to declare this algorithm to be $O(N^2)$. However, we need to analyze this case a little more carefully. While code containing nested loops often is $O(N^2)$, in this case, it's not.

Nested loops that result in $O(N^2)$ occur when each loop revolves around N. In our case, however, while our outer loop runs N times, our inner loop runs a constant five times. That is, this inner loop will always run five times no matter what N is.

So, while our outer loop runs N times, the inner loop runs five times for each of the N strings. While this means our algorithm runs 5N times, this is reduced to O(N), since Big O notation ignores constants.

## Count the Ones

Here's another algorithm where the Big O is different from what it seems at first glance. This function accepts an *array of arrays*, where the inner arrays contain `1`'s and `0`'s. The function then returns how many `1`'s there are.

So, for this example input:

```
[
  [0, 1, 1, 1, 0],
  [0, 1, 0, 1, 0, 1],
  [1, 0]
]
```

our function will return `7`, since there are seven `1`'s. Here's the function in Python:

```
def count_ones(outer_array):
  count = 0

  for inner_array in outer_array:
    for number in inner_array:
      if number == 1:
        count += 1

  return count
```

What's the Big O of this algorithm?

Again, it's easy to notice the nested loops and jump to the conclusion that it's $O(N^2)$. However, the two loops are iterating over two completely different things.

The outer loop is iterating over the inner arrays, and the inner loop is iterating over the actual numbers. At the end of the day, our inner loop only runs for as many numbers as there are *in total.*

Because of this, we can say that N represents how many numbers there are. And since our algorithm simply processes each number, the function's time complexity is O(N).

## Palindrome Checker

A *palindrome* is a word or phrase that reads the same both forward and backward. Some examples include "racecar," "kayak," and "deified."

Here's a JavaScript function that determines whether a string is a palindrome:

```
function isPalindrome(string) {

  // Start the leftIndex at index 0:
  let leftIndex = 0;
  // Start rightIndex at last index of array:
  let rightIndex = string.length - 1;

  // Iterate until leftIndex reaches the middle of the array:
  while (leftIndex < string.length / 2) {

    // If the character on the left doesn't equal the character
    // on the right, the string is not a palindrome:
    if (string[leftIndex] !== string[rightIndex]) {
      return false;
    }

    // Move leftIndex one to the right:
    leftIndex++;
    // Move rightIndex one to the left:
    rightIndex--;
  }

  // If we got through the entire loop without finding any
  // mismatches, the string must be a palindrome:
  return true;
}
```

Let's determine the Big O of this algorithm.

In this case, N is the size of the `string` passed to this function.

The guts of the algorithm takes place within the `while` loop. Now, this loop is somewhat interesting because it only runs until it reaches the midpoint of the string. That would mean that the loop runs N / 2 steps.

However, Big O ignores constants. Because of this, we drop the division by 2, and our algorithm is O(N).

## Get All the Products

Our next example is an algorithm that accepts an array of numbers and returns the product of every combination of two numbers.

For example, if we passed in the array, `[1, 2, 3, 4, 5]`, the function returns:

```
[2, 3, 4, 5, 6, 8, 10, 12, 15, 20]
```

This is because we first multiply the 1 by the 2, 3, 4, and 5. Then we multiply the 2 by the 3, 4, and 5. Next, we multiply the 3 by the 4 and the 5. And finally, we multiply the 4 by the 5.

Note something interesting: when we multiply, say, the 2 by the other numbers, we only have to multiply it by the numbers that are to the right of it. We don't have to go back and multiply 2 by the 1, because that was already covered back when we multiplied by the 1 by the 2. So, each number only needs to be multiplied by the remaining numbers to the right of it.

Here's a JavaScript implementation of this algorithm:

```
function twoNumberProducts(array) {
  let products = [];

  // Outer array:
  for(let i = 0; i < array.length - 1; i++) {

    // Inner array, in which j always begins one index
    // to the right of i:
    for(let j = i + 1; j < array.length; j++) {
      products.push(array[i] * array[j]);
    }
  }

  return products;
}
```

Let's break this down. N is the number of items in the array passed to this function.

We run the outer loop N times. (We actually run it N - 1 times, but we'll drop that constant.) The inner loop, though, is different. Since j always begins one index to the right of i, the inner loop's number of steps decrease each time that it's launched by the outer loop.

Let's see how many times the inner loop runs for our example array, which contains five elements:

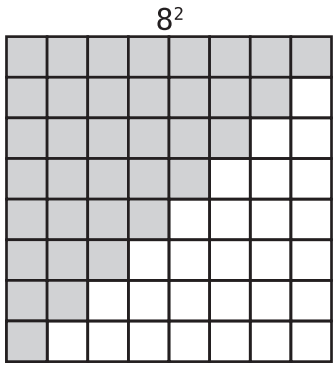
When i is 0, the inner loop runs while j is 1, 2, 3, and 4. When i is 1, the inner loop runs while j is 2, 3, and 4. When i is 2, the inner loop runs while j is 3, and 4. When i is 3, the inner loop runs while j is 4. When all is said and done, the inner loop runs:

4 + 3 + 2 + 1 times.

To put this in terms of N, we can say that the inner loop runs approximately:

N + (N - 1) + (N - 2) + (N - 3) ... + 1 times.

This formula always turns out to compute to about $N^2$ / 2. We can visualize this in the diagram on page 15. For the purposes of the diagram, we'll say that N is 8, so there are $8^2$, or 64, squares.




If you work your way from the top row to the bottom, you'll see that the top row has all N squares shaded gray. The next row has N - 1 squares shaded gray, and the one after that has N - 2 gray squares. This pattern continues until the bottom row, which has just one shaded square.

You can also see at a glance that half of the squares are shaded. This demonstrates that the pattern of N + (N - 1) + (N - 2) + (N - 3)... + 1 is equivalent to $N^2 / 2$.

We've figured out, then, that the inner loop runs for $N^2 / 2$ steps. But because Big O ignores constants, we express this as $O(N^2)$.

## Dealing with Multiple Datasets

Now, what happens if instead of computing the product of every two numbers from a single array, we instead compute the product of every number from one array by every number of a *second* array?

For example, if we had the array, `[1, 2, 3]` and the array, `[10, 100, 1000]`, we'd compute the products as:

```
[10, 100, 1000, 20, 200, 2000, 30, 300, 3000]
```

Our code would be similar to the previous snippet, with some slight modifications:

```
function twoNumberProducts(array1, array2) {
  let products = [];

  for(let i = 0; i < array1.length; i++) {
    for(let j = 0; j < array2.length; j++) {
      products.push(array1[i] * array2[j]);
    }
  }

  return products;
}
```

Let's analyze the time complexity of this function.

First, what is N? This is the first hurdle, as we now have *two* datasets, namely, the two arrays.

It's tempting to lump everything together and say that N is the total number of items of both arrays combined. However, this is problematic for the following reason:

Here's a tale of two scenarios. In Scenario 1, there are two arrays of size 5. In Scenario 2, there's one array of size 9, and another of size 1.

In both scenarios, we'd end up saying that N is 10, since 5 + 5 = 10 and 9 + 1 = 10. However, the efficiency of both scenarios is *very* different.

In Scenario 1, our code takes 25 (5 * 5) steps. Because N is 10, this is equivalent to $(N / 2)^2$ steps.

In Scenario 2, though, our code takes 9 (9 * 1) steps, which is close to about N steps. This is dramatically faster than Scenario 1!

So, we don't want to consider N to be the total number of integers from both arrays, since we'd never be able to pin down the efficiency in terms of Big O Notation, as it varies based on the different scenarios.

We're in a bit of a bind here. We have no choice but to express the time complexity as O(N * M), where N is the size of one array, and M is the size of the other.

This is a new concept: whenever we have two distinct datasets that have to interact with each other through multiplication, we have to identify both sources separately when we describe the efficiency in terms of Big O.

While this is the correct way of expressing this algorithm in terms of Big O Notation, it's a little less useful than other expressions of Big O. Comparing an O(N * M) algorithm to algorithms that only have an N (and not an M) is a little like comparing apples to oranges.

However, we do know that there's a specific range in which O(N * M) lies. That is, if N and M are the same, it's equivalent to $O(N^2)$. And if they're not the same, and we arbitrarily assign the smaller number to be M, even if M is as low as 1, we end up with O(N). In a sense then, O(N * M) can be construed as a range between O(N) and $O(N^2)$.

Is this great? No, but it's the best we can do.