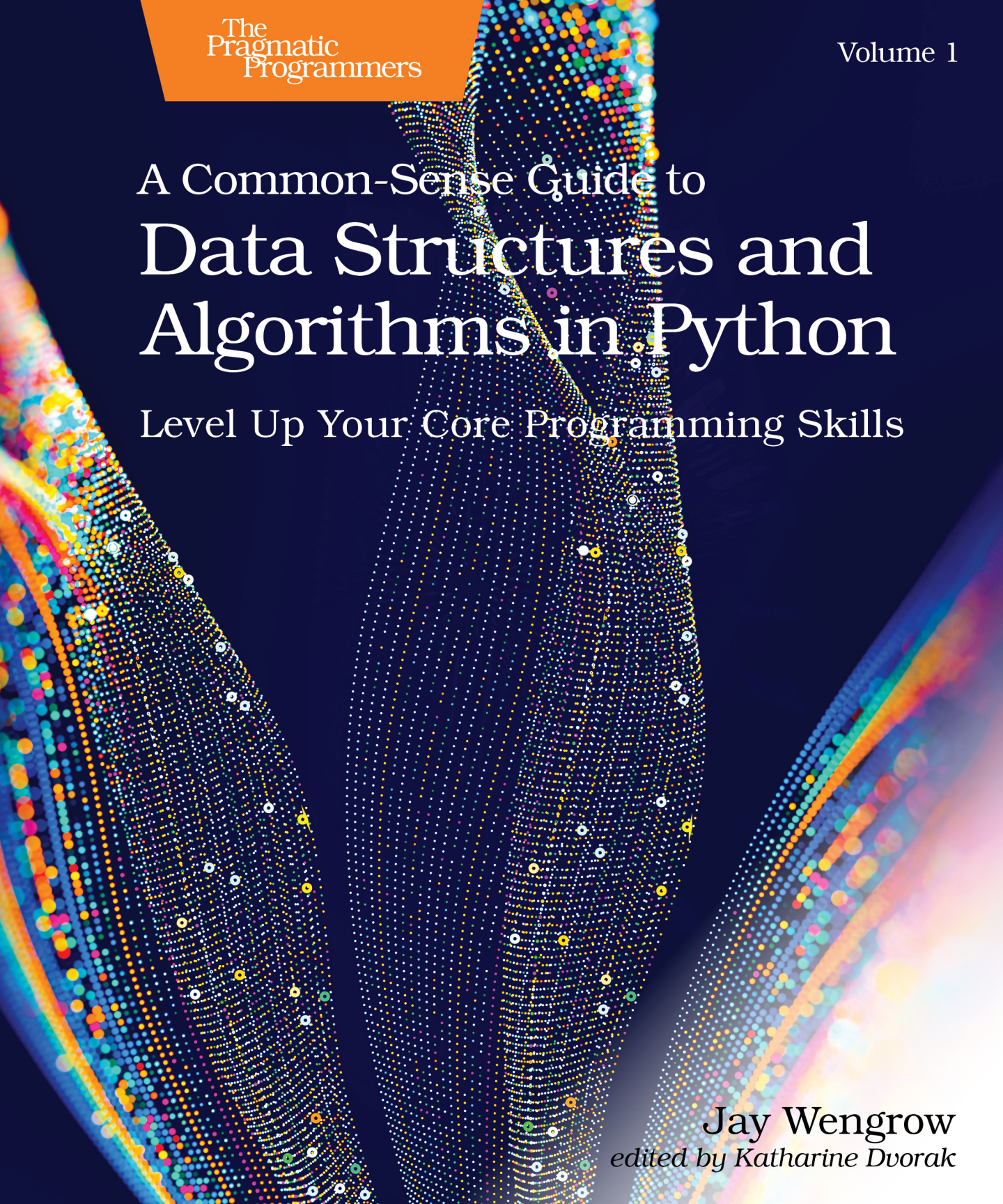
Extracted from:

A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1

Level Up Your Core Programming Skills

The Pragmatic Bookshelf

Dallas, Texas


The Pragmatic Programmers
Volume 1
A Common-Sense Guide to
Data Structures and Algorithms in Python
Level Up Your Core Programming Skills
Jay Wengrow
edited by Katharine Dvorak

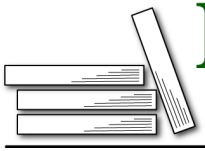
A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1

Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf

Dallas, Texas



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

Publisher: Dave Thomas
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
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: 979-8-88865-035-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2023

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, quite a few details 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 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?

```
def average_of_even_numbers(array):
    sum = 0
    count_of_even_numbers = 0

    for number in array:
        if number % 2 == 0:
            sum += number
            count_of_even_numbers += 1

    if count_of_even_numbers == 0:
        return None

    return sum // count_of_even_numbers
```

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 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 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 two 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 an implementation of this algorithm. Let's see if we can figure out its Big O efficiency:

```
def word_builder(array):
    collection = []

    for index_i, i in enumerate(array):
        for index_j, j in enumerate(array):
            if index_i != index_j:
                collection.append(i + 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 of i. For each index_i, we run an inner loop that iterates again over each character in the same array using the index index_j. Within this inner loop, we concatenate the characters at index_i and index_j, with the exception of when index_i and index_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? For our example array of ["a", "b", "c", "d"], our function would return the following 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?

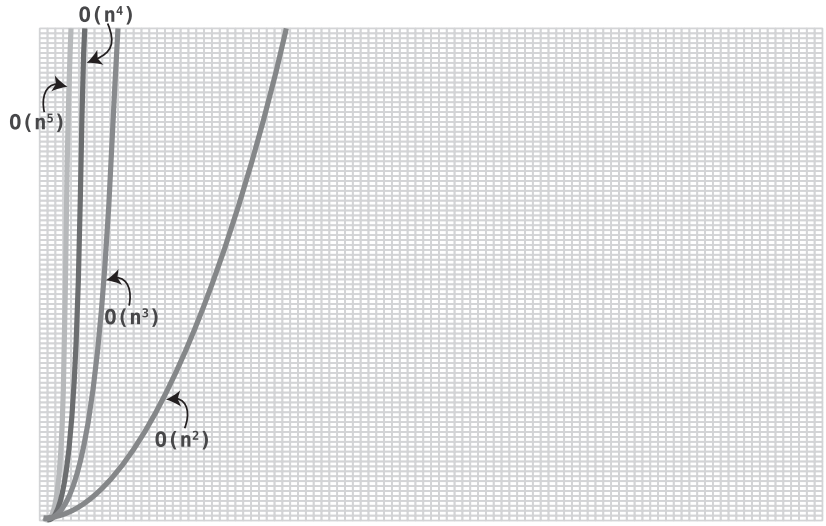
```
def word_builder(array):
    collection = []

    for index_i, i in enumerate(array):
        for index_j, j in enumerate(array):
            for index_k, k in enumerate(array):
                if (index_i != index_j and
                        index_j != index_k and index_i != index_k):
                    collection.append(i + j + 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:



Optimizing any code from a speed of $O(N^3)$ to $O(N^2)$ would be a big win since the code becomes exponentially faster. However, the algorithm above remains stuck at $O(N^3)$.

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's an implementation of this function. See if you can identify its efficiency in Big O:

```
def sample(array):
    if not array:
        return None

    first = array[0]
    middle = array[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).