Why Asymptotic Notation

Learn why asymptotic notation is an essential tool for becoming an efficient programmer.

When writing programs, it’s important to make smart programming choices so that code runs most efficiently. Computers seem to take no time evaluating programs, but when scaling programs to deal with massive amounts of data, writing efficient code becomes the difference between success and failure. In computer science, we define how efficient a program is by its runtime.

We can’t just time the program, however, because different computers run at different speeds. My dusty old PC does not run as fast as your brand new laptop. Programming is also done in many different languages, how do we account for that in the runtime We need a general way to define a program’s runtime across these variable factors. We do this with Asymptotic Notation.

With asymptotic notation, we calculate a program’s runtime by looking at how many instructions the computer has to perform based on the size of the program’s input. For example, if I were calculating the maximum element in a collection, I would need to examine each element in the collection. That examining step is the same regardless of the language used, or the CPU that’s performing the calculation. In asymptotic notation, we define the size of the input as N. I may be looking through a collection of 10 elements, or 100 elements, but we only need to know how many steps are performed relative to the input so N is used in place of a specific number. If there is a second input, we may define the size of that input as M.

There are varieties of asymptotic notation that focus on different concerns. Some will communicate the best case scenario for a program. For example, if we were searching for a value within a collection, the best case would be if we found that element in the first place we looked. Another type will focus on the worst case scenario, such as if we searched for a value, looked in the entire dataset and did not find it. Typically programmers will focus on the worst case scenario so there is an upper bound of runtime to communicate. It’s a way of saying “things may get this bad, or slow, but they won’t get worse!”

In this next module, we will learn more about asymptotic notation, how to properly analyze the runtime of a program through asymptotic notation, and how to take into consideration the runtime of different data structures and algorithms when creating programs. Learning these skills will change the way you think when you design programs and it will prepare you for the software engineering world where creating efficient programs is an essential skill.

Let’s dive into the world of asymptotic notation!

Review

Let’s review what we learned:

We use asymptotic notation to describe the runtime of a program. The three types of asymptotic notation are big Theta, big Omega, and big O.

We use big Theta (Θ) to describe the runtime if the runtime of the program is the same in every case.

The different common runtimes from fastest to slowest are: Θ(1), Θ(log N), Θ(N), Θ(N log N), Θ(N2), Θ(2N), Θ(N!).

We use big Omega (Ω) to describe the best-case running time of a program.

We use big O (O) to describe the worst-case running time of a program.

We typically describe a program’s running time in terms of big O.

When finding the runtime of a program with multiple steps, you can divide the program into different sections and add the runtimes of the various sections. You can then take the slowest runtime and use that runtime to describe the entire program.

When analyzing the runtime of a program, we care about which part of the program is the slowest.

Now that we conceptually understand what asymptotic notation is, let’s practice analyzing and improving runtimes with real programming language examples!

Space complexity

这两个function的space complexity def double\_array(input\_array): # Returns an array that is the double of the input array length = len(input\_array) doubled\_array = [0] \* length for i in range(length): doubled\_array[i] = input\_array[i] \* 2 return doubled\_array def find\_min(input\_array): # Returns the smallest element in the array minimum = input\_array[0] for i in input\_array: if i < minimum: minimum = i return minimum

ChatGPT

让我们分析一下这两个函数的空间复杂度。

1. **double\_array 函数**

这个函数创建了一个新的列表 **doubled\_array**，它的大小与输入的 **input\_array** 相同。

* 输入：**input\_array** 占用的空间是 O(n)，其中 n 是数组中的元素数量。但是，我们通常不把输入计算在内。
* 额外空间：**doubled\_array** 是一个新列表，它需要额外的 O(n) 空间。函数中的其他变量（如 **length** 和循环变量 **i**）都是常量空间，即 O(1)。
* 总体来看，**double\_array** 函数的空间复杂度是 O(n) 因为你需要与输入数组相同大小的空间来存储结果。

1. **find\_min 函数**

这个函数不会创建与输入大小相关的任何新数据结构。它只是遍历数组，比较元素，找到最小值。

* 输入：**input\_array** 占用的空间是 O(n)，其中 n 是数组中的元素数量。但是，这不计算在函数的空间复杂度内。
* 额外空间：**minimum** 是一个单独的变量，用于存储当前找到的最小值。它需要常量空间 O(1)。循环变量 **i** 也是 O(1)。
* 总体来看，**find\_min** 函数的空间复杂度是 O(1)，因为无论输入数组的大小如何，它都只使用固定（常量）数量的额外空间。

这两个例子展示了空间复杂度如何反映一个函数在执行过程中对额外内存的需求。即使处理相同的输入（在这种情况下是 **input\_array**），不同的算法和操作可能会导致不同的空间需求。