Chapter 8 : Algorithms

Aviral Janveja

1 Introduction

We come back to where we started from, in chapter 1 :

> “**Programming** is the art of instructing computers to perform specific tasks. At the core of a computer program, an **algorithm** serves as a systematic set of steps or instructions designed to solve a computational problem. So, while an algorithm provides the strategy, a program brings that strategy to life in a formal language that computers can comprehend.”

Till now we have focused on the programming toolbox, syntax and specific features of the Python language. However, there could be many ways to solve a particular problem in the very same programming language. Hence, we now take that algorithmic or strategic part into consideration. The question here is :

> “How can we relate our choices in the design of an algorithm, to the amount of time it will take to solve a particular problem. For example, how will the program efficiency differ if you choose to use recursion versus iteration.”

Why care about program efficiency when the computers are getting faster every year? **Because**, the size of data-sets and problems are increasing as well. For example, being able to search through the billions of web-pages on the internet.

2 Complexity

In terms of complexity, there are two aspects to consider : **space complexity** and **time complexity**. The amount of storage space required versus the amount of time required to solve a particular problem. There might be a trade-off between the two at times. However, in this chapter we are going to focus on the **time complexity** of our algorithms.

2.1 Order of Growth

Independent of the machine or a specific program implementation.

Just evaluate the algorithm by counting the number of basic operations. Focus on what happens as the input size grows.

The number of steps will depend on the input. for example, searching for an element through the list. If it is the very first element, best case scenario. If the element is not in the list at all, you search through the entire list before halting, hence worst case scenario.

In our case, we will consider the worst case scenario in order to get an upper-bound on the amount of time required.

Expressing the growth of the program run-time as the input size grows. We are not looking for the exact run-time. We are trying to capture the relationship between the run-time and input size. For example, if doubling the input size leads to doubling of the run-time, then it is a linear relation.

Order of Growth Definition : We are looking for an upper-bound on the growth of run-time as a function of the size of the input in the worst case scenario.

2.2 Big O Notation

This is the notation that we are going to use to convey an algorithms order of growth.

Let us see how we can go from counting operations to getting the order of growth.

let us look at the factorial example, calculated the iterative way.

```
answer = 1  # 1 basic operation

while n > 1:  # 1 basic operation
    answer = answer * n  # 2 basic operations
    n = n - 1  # 2 basic operations

return answer  # 1 basic operation
```

There are 7 basic operations in total, with 5 operations being part of the while loop. In the worst case scenario, those 5 operations will run n-times. Hence the total number of operations are be calculated as : $5n + 2$.

After computing the number of steps : The point to remember is, that we want to look at the algorithm behavior when n gets really large. And when n gets really large, the small factors in the equation won't really matter.

Hence, in $5n + 2$, we ignore the additive constants and the multiplicative constants. We focus the dominant term in the equation that grows most rapidly and drop the constants.

In the above example, that means ignoring the 5 and the plus 2, which leaves us with n. And hence, it is a linear relation between input-size and run-time for the above iterative factorial algorithm. In the Big O notation, it is written as : O(n)

Let us look at some more example equations like above and focus on their dominant terms to get the order of growth.

- $n^2 + 2n + 2$: O(n^2)
- $n^2 + 1000n + 3^{1000}$: O(n^2)
- $log(n) + n + 4$: O(n)
- $nlog(n) + 300n$: O($nlog(n)$)
- $n^2 + 2^n$: O(2^n)

3 Complexity Classes

Analyzing algorithms and their complexity. Analyze statements inside functions, focus on the dominant term.

3.1 Sequential Statements : Add Complexity

For example :

```
for i in range(n):
    print('a')

for j in range(n*n):
    print('b')
```

In the above example, the first loop runs n times, whereas the second one runs n^2 times in the worst case. Both of them added together is equal to $n+n^2$. Therefore, order of growth can will be O$(n + n^2)$ = O(n^2) focusing on the dominant, fastest growing term.

3.2 Nested Statements : Multiply complexity

For example :

```
for in range(n):
    for j in range(n):
        print('a')
```

The outer loop runs n times and the inner loop in turn runs n times for every outer loop iteration. Hence, the order of growth for the nested loop is O$(n * n)$ = O(n^2)