Chapter 8 : Algorithms

Aviral Janveja

1 Introduction

In this final chapter, we circle 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 more than one way 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 Algorithm Analysis

In order to analyse different algorithms, there are two aspects to consider : **space complexity** and **time complexity**. Space complexity is the amount of memory required and time complexity is 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 Time Complexity

When it comes to analysing the time complexity of our algorithms, we are looking for a measure that is Independent of the machine speed and the specific program

implementation. We are not interested in minor tweaks here and there, but rather the time required by a particular algorithmic strategy in general. One way to achieve that is by **counting** the number of **basic operations** in our algorithm, such as assignments, comparisons, mathematical operations and assuming each of these basic operations take constant time to execute.

This approach is independent of machine speed and language specifics. However, it is important to consider that the number of basic operations required will depend on the input. For example, when searching for an element through a list. If it is the very first element, then best-case scenario. If the element is not in the list at all and you search through the entire list before halting, then worst-case scenario. Here, we will be interested in the **worst-case scenario** in order to get an **upper-bound** on the run-time. Further, our focus is not on calculating the exact run-time down to the milliseconds. Instead, we are focusing on how the run-time grows corresponding to the input-size.

2.2 Order Of Growth

So, we are interested in how the run-time grows in the worst-case as the input-size grows very large. This, in lesser words is called the : **order of growth** of an algorithm. Now, let us see how we go from counting basic operations to getting the order of growth. Let us look at the following algorithm for the same :

```
# Computing factorial using iteration

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 calculated as : $5n + 2$. After counting the number of basic operations, it is important to remember that we are interested in the order of growth.

So, when n starts to get really large, constant terms and small factors in the expression won't really matter. Hence, we ignore the additive constants and the multiplicative constants and focus on the dominant term in the expression that grows most rapidly. In the above example, that means ignoring the 5 and the plus 2, which leaves us with n. Hence, the order of growth for the above algorithm is said to be **linear** and the same is expressed using the **Big O** notation as : O(n)

2.3 Examples

Let us look at a couple more run-time expressions and focus on the dominant term to reveal their 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)

In order to get the order of growth for each of the above, we have simply recognized the dominant term in the expression and ignored the constants and lower order terms. As this term will decide the behavior of the algorithm as n tends to get really large, which is exactly what we are interested in.

3 Algorithm Design

The core idea for this chapter is not just about calculating the time complexity. It is about understanding how choices in algorithm design lead to certain costs and consequences. Hence, now we start identifying common algorithm design choices and their corresponding time complexities.

3.1 Loops

What is the typical cost of using a loop to iterate through some list in terms of its time complexity? Let us look at it through the following example of a linear search algorithm on an unsorted list :

```
def linear_search(L, e)
    found = false
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

In the worst case, you will have to loop through all n elements of the list. Hence, the number of steps can be calculated as $4n + 2$ and the order of growth will be O(n) or linear growth. Notice the pattern here that, using a single loop leads to **linear time complexity**.

Sequential Statements : Adds Complexity

In order to analyze the time complexity of multiple blocks of code, simply analyze the individual blocks first, focus on the dominant term and then combine them accordingly. For example :

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

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

In the above example, the first loop runs n times in the worst case, same for the second one. Both of them are placed sequentially one after the other. Therefore, their time complexity adds up : $n + n = 2n$. The order of growth will thus be O(n) focusing on the dominant term and ignoring the multiplicative constant.

3.2 Nested Loops

Nested Statements : Multiplies complexity Taking another example :

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

Here, the outer loop runs n times and the inner loop in turn runs n times for every outer loop iteration. Hence, their time complexity gets multiplied and we get $n * n$ or O(n^2) in terms of order of growth.

How about nested loops? Well, as seen before :

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

The outer loop iterates n times and the inner loop further runs n times for every outer loop iteration. Hence, we get $n * n$ steps and thus O(n^2) order of growth, which means nested loops generally lead to **quadratic time complexity**.

3.3 Divide & Conquer

bisection search on a sorted list example. logarithmic cost log(n)

3.4 Recursion

Fibonacci example. Iterative vs recursive. each recursion opens up two more recursive calls, leading to 2 raised to n exponential order of growth.

complexity classes : time stamp 42:50