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 time complexity. It is about understanding how choices in the design of an algorithm come with certain costs and consequences. Therefore, we now start identifying common algorithm design choices and their corresponding run-times.

3.1 Loops

What is the typical cost of using a loop to iterate through some sequence in terms of its run-time? Let us look at it, through the following example :

```
for i in range(n):
    print(1)
```

In the worst case, you will have to loop through all n elements. Therefore, the order of growth will be O(n). It is important to notice the pattern here that, using a single loop generally leads to **linear** time complexity.

How about two loop statements in **sequence**? 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. For example :

```
for i in range(n):
    print(1)

for j in range(n):
    print(2)
```

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** as : $n + n = 2n$. The order of growth however remains the same : O(n) as we focus on the dominant term and ignore the multiplicative constant.

3.2 Nested Loops

How about **nested** loops? Well, as seen before :

```
for i in range(n):
    for j in range(n):
        print(1)
```

Here, the outer-loop runs n times and the inner-loop in turn runs n times for every outer-loop iteration. Hence, their time complexities **multiply** and we get $n * n$ or O(n^2) in terms of order of growth. Nested loops generally lead to **polynomial** or more specifically **quadratic** time complexity.

3.3 Divide & Conquer

Let us look at how the divide and conquer approach works in context of binary search on a sorted list. The binary search algorithm is a searching algorithm that works by repeatedly dividing a sorted sequence in half in order to find the required element :

```
# Recursive Binary Search Algorithm on Sorted List

def binary_search(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L) // 2
        if L[half] > e:
            return binary_search(L[:half], e)
        else:
            return binary_search(L[half:], e)
```

Essentially, we pick an index that divides the list in half. Check if L[half] is greater, smaller or equal to e. Accordingly, search the left or right half of the list. Repeat until the element is found or the list is exhausted.

As evident in the above algorithm, at each step we eliminate half the list. In the worst case, we keep dividing the list in half until we are left with just one element. So, if the list length is n and k is the number of times we divide the list in half, then :

$$n/2^k = 1$$

Therefore in the worst-case we would require $k = log_2(n)$ number of steps or O($log(n)$) in terms of order of growth. Hence, it can be recognized that time complexity for the divide and conquer approach is usually **logarithmic** in the length of the list.

3.4 Recursion & Iteration

Let us compare the iterative and recursive versions of the Fibonacci-number algorithm in terms of their order of growth.

```
# Iterative Fibonacci

 def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_1 = 0
        fib_2 = 1
        for i in range(n-1):
            temp = fib_1
            fib_1 = fib_2
            fib_2 = temp + fib_2
        return fib_2
```

In the above algorithm, the point to we have a bunch of basic operations taking constant time and then we have a loop with basic operations running inside it, hence the overall order of growth is linear : O(n).

What about the recursive version?

```
# Recursive Fibonacci

def fib_recur(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

The above code is much cleaner to look at. However, notice the two recursive calls. With each step, you will need to compute two sub-problems, as there would be an $n-1$ and an $n-2$ for every n that you try to compute. This naturally leads to an **exponential** order of growth : O(2^n). The pattern to recognize here is that anything that has multiple recursive calls at each step or a single recursive call that grows exponentially will likely lead to exponential order of growth.

3.5 Complexity Classes

In the above sections, we have learned to reason about the cost of an algorithm by recognizing certain common patterns in its design. Here is summary of different time complexities :

- O($log(n)$) : Logarithmic complexity. Characteristic of the divide and conquer approach which reduces the problem size in half with each step.

- O(n) : Linear complexity. Characteristic of simple iterative or recursive programs.
- O(n^c) : Polynomial or commonly quadratic complexity. Characteristic of nested loops or recursive calls. Here c is a constant.
- O(c^n) : Exponential complexity. Characteristic of multiple recursive calls at every level or a single recursive call that grows exponentially.