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?

2 Algorithm Analysis

There are two aspects to consider when analyzing algorithms : **space complexity** and **time complexity**. Space complexity is the amount of memory space 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 a constant amount of time to execute.

This approach is independent of the machine speed and the language specifics. However, it is also 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

Therefore, we are interested in how the worst-case run-time of an algorithm grows as the input-size grows to very large values. 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, those 5 operations will run n times. Hence the total number of operations are calculated as : $5n + 2$. After computing the number of basic operations, it is time to get 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 would grow 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 : $\mathrm{O}(n)$

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 :

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

- $n^2 + 2^n \Rightarrow \mathrm{O}(2^n)$

In order to get the order of growth for each of the above, we have simply recognized the highest order term in the expression and ignored the constants and lower order terms. It is because, 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-time costs.

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 $\mathrm{O}(n)$. 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 a **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 : $\mathrm{O}(n)$ as we focus on the dominant term and ignore the multiplicative constant.

3.2 Nested Loops

How about **nested** loops? :

```
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 $\mathrm{O}(n^2)$ 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 works by repeatedly dividing a sorted sequence in half, in order to find the required element :

```
# Recursive Binary Search on a 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 only 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 it in half to reach the last remaining element, 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 the time complexity for divide and conquer approaches 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, 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 at each step. For every step, You will need to compute two sub-steps, as there would be an $n-1$ and an $n-2$ for every n. 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 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 a summary :

- O($log(n)$) : Logarithmic complexity. Characteristic of divide and conquer approaches which reduce the search space by half at 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.

4 Search Algorithms

Searching algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various real-world applications such as information retrieval, data analysis, web-search engines and more. Let us therefore look at two commonly used search algorithms :

4.1 Linear Search

The linear search algorithm simply iterates through each element of a list to find the target element, as shown below :

```
# Linear Search via Iteration

def linear_search(arr, target):
    for i in range(len(arr)):
        if target == arr[i]:
            return f"Element found at index : {i}"
    return "Element not found"
```

In the worst-case, we will have to loop through all elements of the list to decide if the target element is there or not. Hence, time complexity is linear : O(n) where n is the length of the list. The full code can been seen here : linear_search.py

4.2 Binary Search

Binary search algorithm follows the divide and conquer approach, reducing the search space in half at every step. First, we find the middle index and check if it is equal to the target element. If not, we eliminate half the list, depending on whether the middle index element is greater or lesser than the target element. Further, we use pointer variables to keep track of the first and last index of the list, as shown below :

```
# Binary Search on Sorted List via Iteration

def binary_search(arr, target):
    start = 0
    end = len(arr) - 1

    while start <= end:
        mid = (start + end) // 2

        if arr[mid] == target:
            return f"Element found at index : {mid}"
        elif arr[mid] < target:
            start = mid + 1
        else:
            end = mid - 1

    return "Element not found"
```

As seen in the divide and conquer section above, in the worst case we will be dividing the search space in half until we reach the very last element of the list. This leads to logarithmic time complexity : O($log(n)$) where n is the length of the list. The full code can been seen here : binary_search.py

5 Sorting Algorithms

In the binary search algorithm above, we saw that the divide and conquer approach really helped us speed up the searching process. We were able to go from O(n) in linear search to O($log(n)$) time complexity. That is a significant speed-up. However, the catch is that binary search was performed on an already sorted list. Naturally, the sorting operation will add its own time complexity.

So the question now is, does it make sense to sort a list before searching, even after adding the time complexity for sorting? Well, it turns out the answer is : Yes. That is because, we only need to sort once and then we can search multiple times. For a large number of search operations, the sort time becomes irrelevant, if we can do the sorting part in a reasonable amount of time. Let us look at the following two sorting algorithms :

5.1 Selection Sort

The selection sort algorithm works by finding the smallest element and placing it at the beginning of the list with each iteration. So after n iterations, the first n elements are already sorted. This is implemented with the help of nested loops, as shown below :

```
def selection_sort(arr):
i = 0
while i < len(arr):
    for j in range(i+1, len(arr)):
        if arr[j] < arr[i]:
            arr[i], arr[j] = arr[j], arr[i]
    i += 1
```

The outer while loop goes through each element of the list with i being used to keep track of the current index. The inner for loop looks for the smallest element in the remaining unsorted portion of the list, from $i + 1$ till the end. The swap operation places the smaller element at the current index position i as soon as it is found. We then move on to the next index position and continue sorting until the entire list is sorted.

As this algorithm uses the nested loop design, the time complexity is naturally quadratic : O(n^2) where n is the list length. The full code can been seen here : selection_sort.py

5.2 Merge Sort

Merge sort works on the divide and conquer approach. The list is divided recursively until we reach the base case, which is just a single element. We then merge them back together, keeping the correct order, thus returning a fully sorted list as shown below :

```
# Merge Sort Algorithm

def merge_sort(arr):
    if len(arr) <= 1:
```

```
        return arr
    else:
        mid = len(arr) // 2
        left_half = merge_sort(arr[:mid])
        right_half = merge_sort(arr[mid:])
        return merge(left_half, right_half)


# Merge Function to sort and combine the two halves
def merge(left, right):
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and
    right_index < len(right):
        if left[left_index] < right[right_index]:
            merged_list.append(left[left_index])
            left_index += 1
        else:
            merged_list.append(right[right_index])
            right_index += 1

    while left_index < len(left):
        merged_list.append(left[left_index])
        left_index += 1

    while right_index < len(right):
        merged_list.append(right[right_index])
        right_index += 1

    return merged_list
```

The **merge_sort** function first checks for the base case. If true : the list is naturally sorted and is returned as it is. Otherwise, the list is recursively split into two halves until we reach singular elements. The **merge** function is then called, which compares the smallest elements from both halves one by one and then appends them in the correct order into a single fully sorted **merged_list**.

Dividing the list into half at each step leads to logarithmic time complexity. Merging them back up into one sorted list takes linear time as we have to compare all n elements. Hence, leading to a log-linear time complexity : O($n\ log(n)$) where n is the length of the list. The full code can been seen here : merge_sort.py

The End.