@miladkoohi

How To Calculate Time Complexity With Big O Notation?

What is **Big O** Notation Explained: Space and Time Complexity

with Python Examples

What do i Learn?

REPOST
if like

Table of content:

Introduction

There are usually multiple ways to solve the problem using a computer program. For instance, there are several ways to sort items in an array - you can use merge sort, bubble sort, insertion sort, and so on. All of these algorithms have their own pros and cons and the developer's job is to weigh them to be able to choose the best algorithm to use in any use case. In other words, the main question is which algorithm to use to solve a specific problem when there exist multiple solutions to the problem.

Introduction and Why is Algorithm Analysis Important?

@reallygreatsite

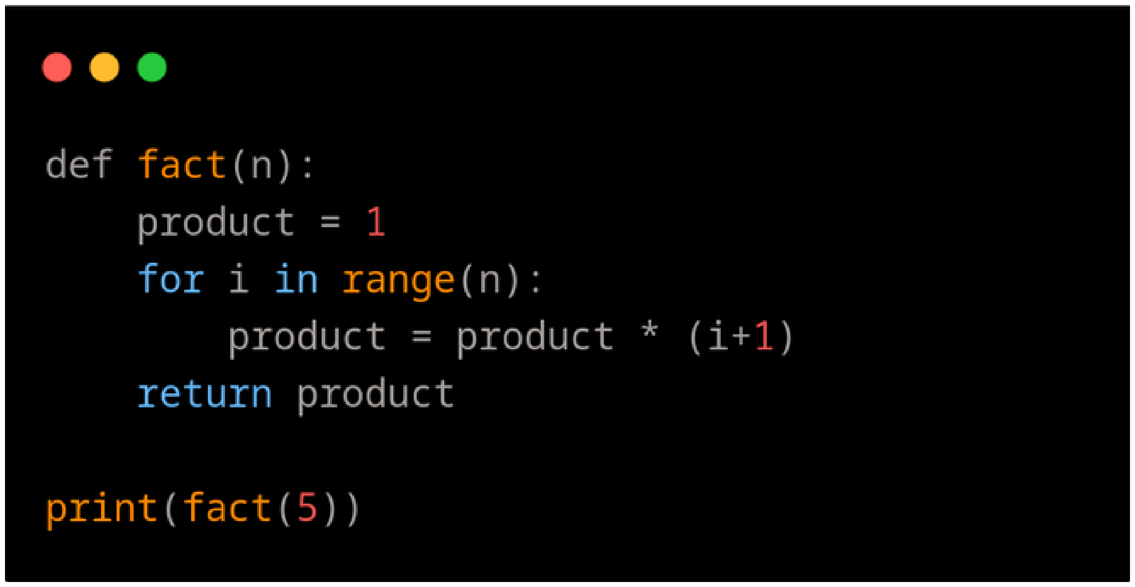
Introduction

Algorithm analysis refers to the analysis of the complexity of different algorithms and finding the most efficient algorithm to solve the problem at hand. Big-O notation is a statistical measure used to describe the complexity of the algorithm.

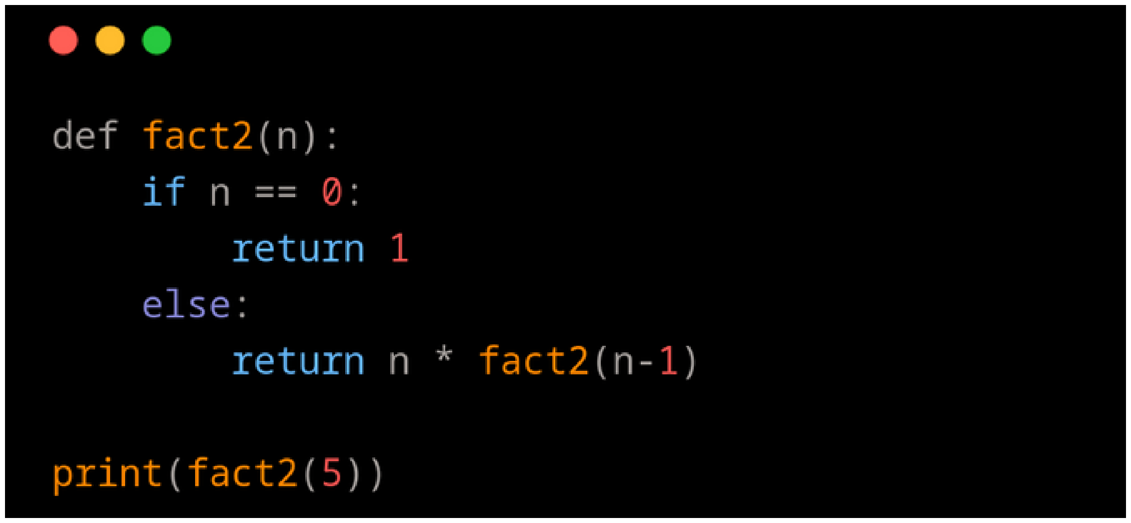
“Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann–Landau notation or asymptotic notation.”

In this guide, we'll first take a brief review of algorithm analysis and then take a deeper look at the Big-O notation. We will see how Big-O notation can be used to find algorithm complexity with the help of different Python functions.

Why is Algorithm Analysis Important?



```python
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```



```python
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

Which algorithm is better?

How can you determine which algorithm code is faster in terms of time **complexity**?

How to calculate time complexity With code?

1

One way to do so is by finding the time required to execute the code on the same input.

- In the Jupyter notebook, you can use the **%timeit** literal followed by the function call to find the time taken by the function to execute:

This will give us:

```
%timeit fact(50)
```

```
9 µs ± 405 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

The output says that the algorithm takes 9 microseconds (plus/minus 45 nanoseconds) per loop.

Similarly, we can calculate how much time the second approach takes to execute:

```
%timeit fact2(50)
```

This will result in:

```
15.7 µs ± 427 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

The second algorithm involving recursion takes 15 microseconds (plus/minus 427 nanoseconds).

The execution time shows that the first algorithm is faster compared to the second algorithm involving recursion. When dealing with large inputs, the performance difference can become more significant.

But

However, execution time is not a good metric to **measure** the complexity of an algorithm since it depends upon the **hardware**. A more objective complexity analysis metric for an algorithm is needed. This is where the Big O notation comes to play.

Algorithm Analysis with Big-O Notation

2

Big-O notation signifies the relationship between the input to the algorithm and the steps required to execute the algorithm. It is denoted by a **big "O"** followed by an opening and closing parenthesis. Inside the parenthesis, the relationship between the input and the steps taken by the algorithm is **presented using "n"**.

The key takeaway is **- Big-O** isn't interested in a particular instance in which you run an algorithm, such as fact(50), but rather, how well does it scale given increasing input. This is a much better metric for evaluating than concrete time for a concrete instance!

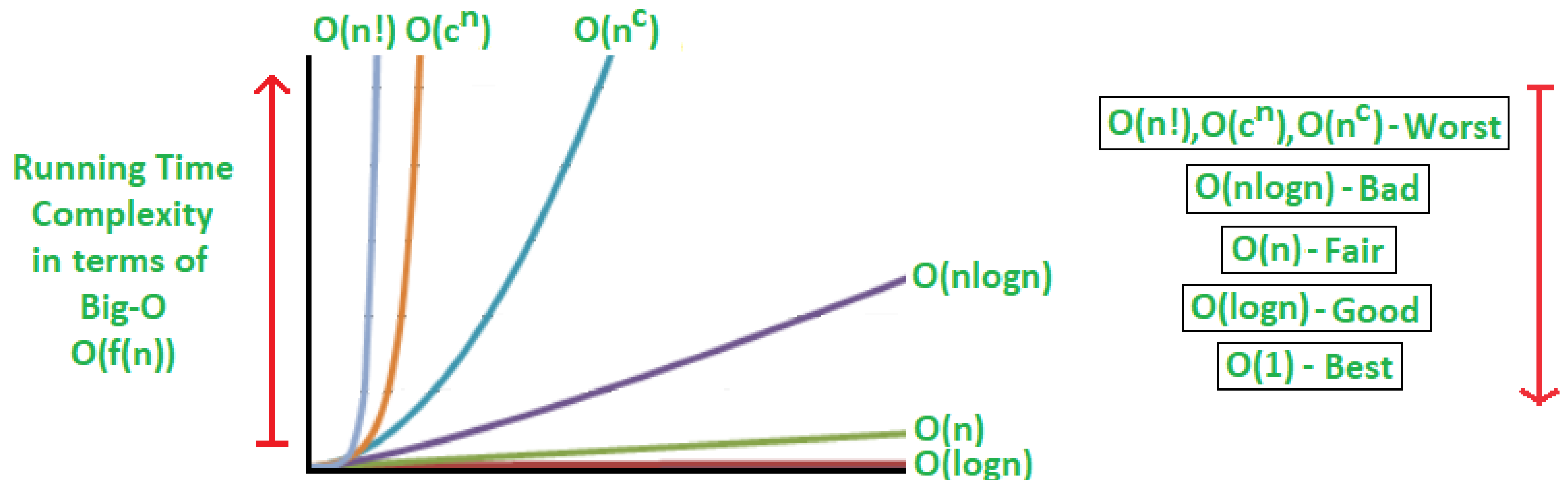
For example, if there is a linear relationship between the input and the step taken by the algorithm to complete its execution, the Big-O notation used will be **O(n)**. Similarly, the Big-O notation for quadratic **functions is $O(n^2)$**.

```
# O(n): at n=1, 1 step is taken. At n=10, 10 steps are taken.
# O(n²): at n=1, 1 step is taken. At n=10, 100 steps are taken.
```

Algorithm Analysis with Big-O Notation

@miladkoohi

visualize these functions and compare them



Generally speaking - anything worse than **linear** is considered a <u>bad complexity</u> (i.e. inefficient) and should be avoided if possible. Linear complexity is okay and usually a necessary evil. Logarithmic is good. Constant is amazing!

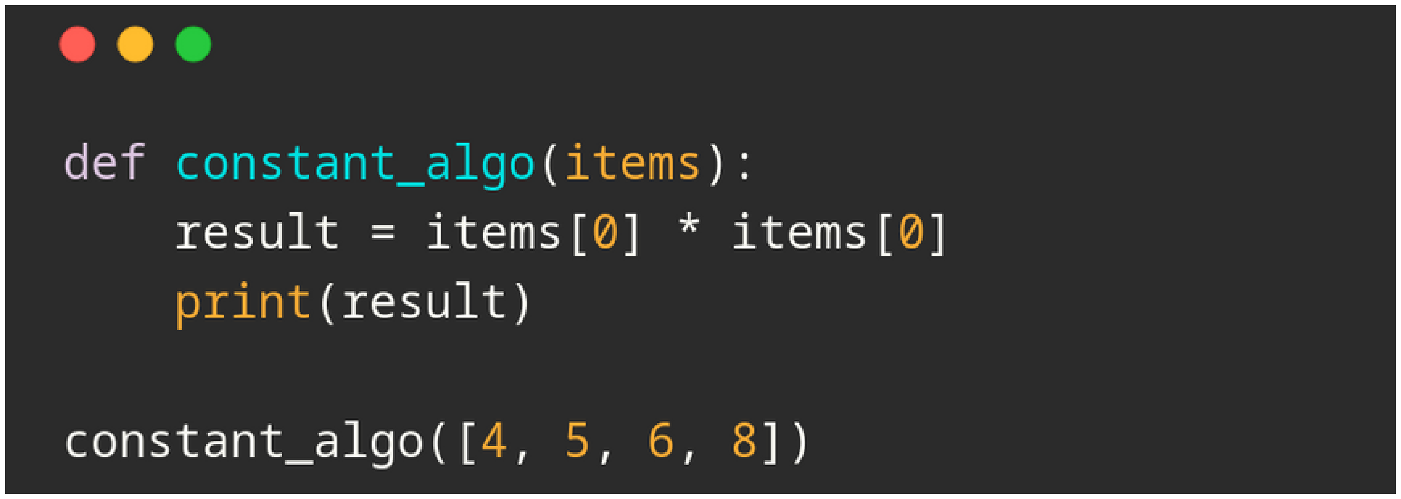
@miladkoohi

some of the most common Big-O functions

Name	Big O
Constant	O(c)
Linear	O(n)
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$
Logarithmic	O(log(n))
Log Linear	O(nlog(n))

1.Constant Complexity - O(C)

The complexity of an algorithm is said to be constant if the steps required to complete the execution of an algorithm remain constant, irrespective of the number of inputs. The constant complexity is denoted by **O(c)** where c can be any constant number.



```
def constant_algo(items):
    result = items[0] * items[0]
    print(result)

constant_algo([4, 5, 6, 8])
```

In the script above, irrespective of the input size, or the number of items in the input list items, the algorithm performs only 2 steps:

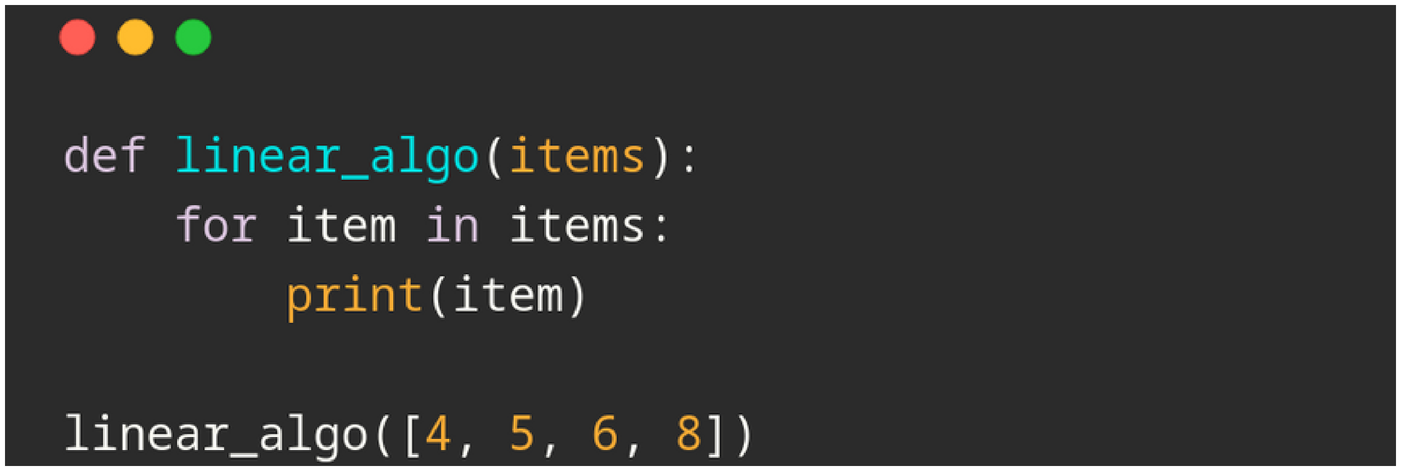
1.Finding the square of the first element
2.Printing the result on the screen.

Hence, the complexity remains constant:O(C).

2.Linear Complexity - O(n)

The complexity of an algorithm is said to be linear if the steps required to complete the execution of an algorithm increase or decrease linearly with the number of inputs. **Linear complexity** is denoted by **O(n)**.

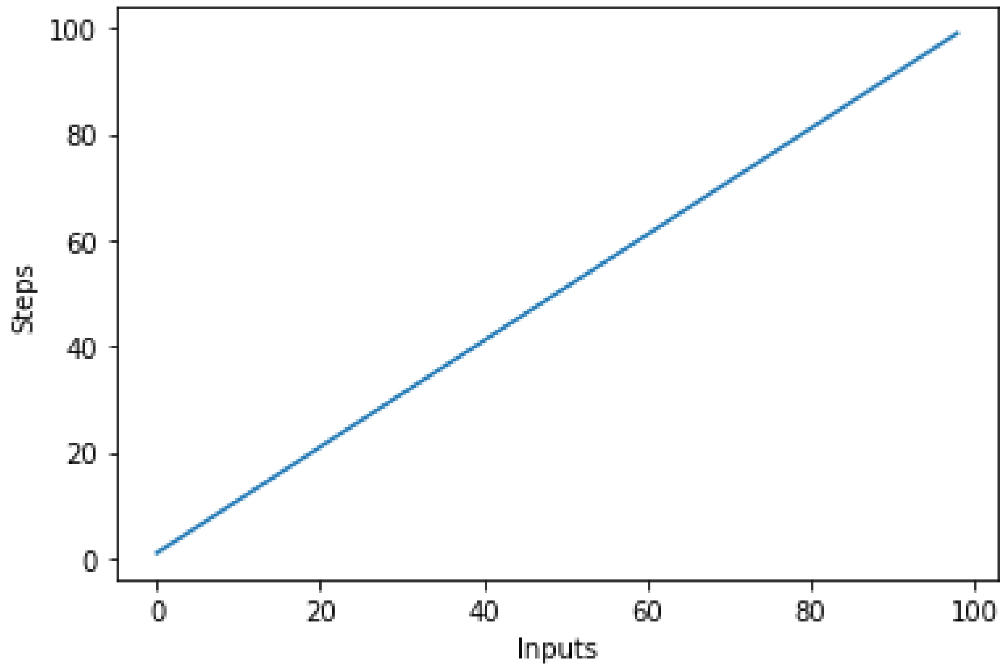
In this example, let's write a simple program that displays all items in the list to the console:



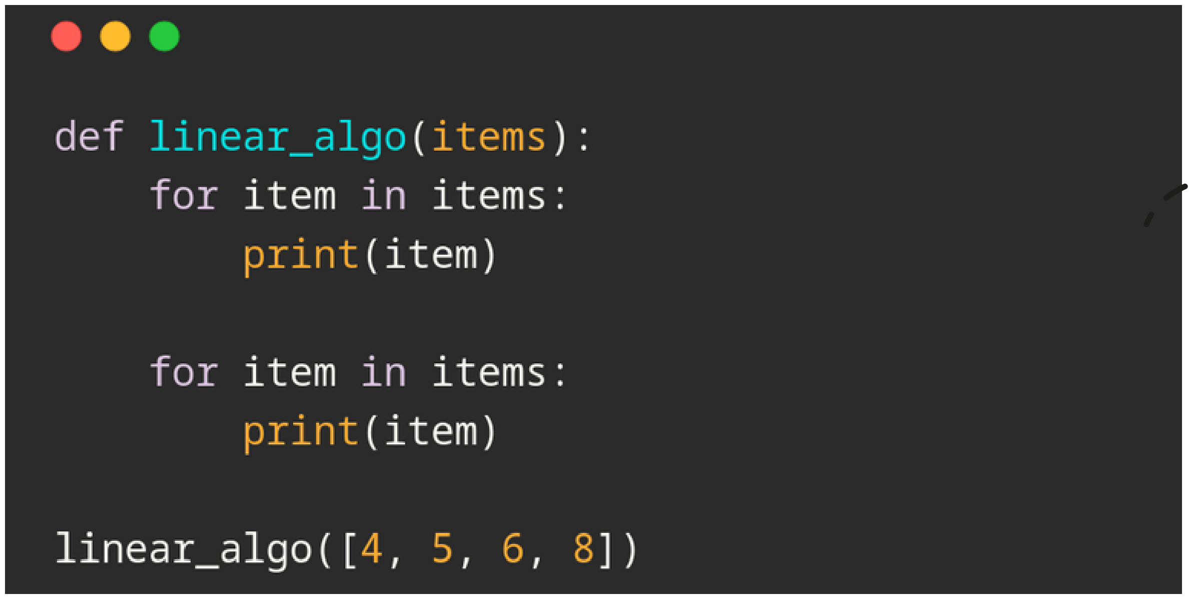
```
def linear_algo(items):
    for item in items:
        print(item)

linear_algo([4, 5, 6, 8])
```

The complexity of the **linear_algo()** function is linear in the above example since the number of iterations of the **for-loop** will be equal to the size of the input items array. For instance, if there are 4 items in the items list, the for-loop will be executed 4 times.

2.Linear Complexity - O(n) -> Note - 1

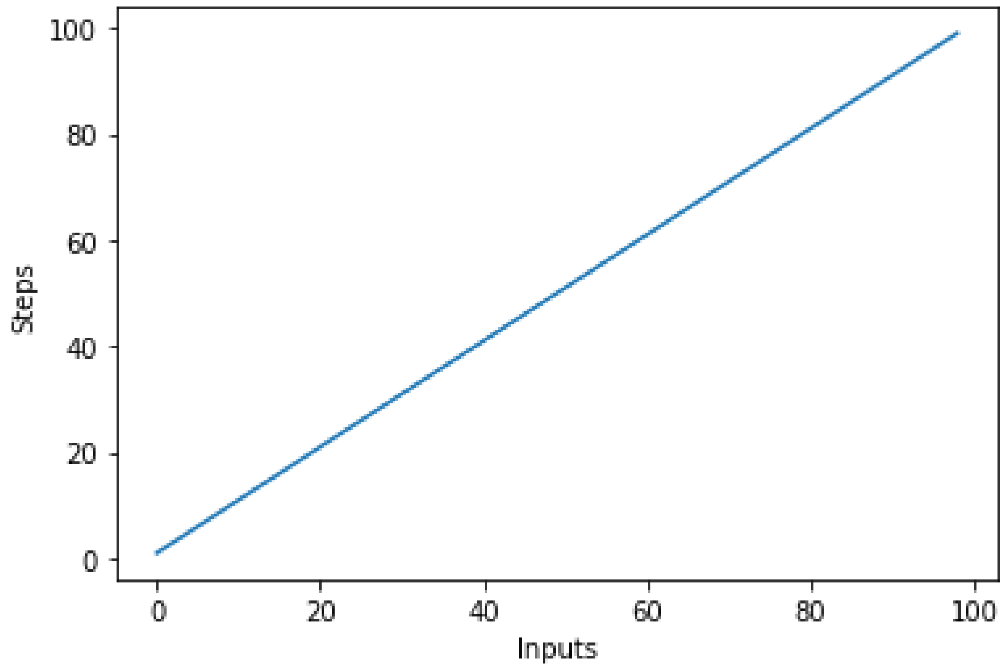


An important thing to note is that with large inputs, constants tend to lose value. This is why we typically remove constants from Big-O notation, and an expression such as O(2n) is usually shortened to O(n). Both O(2n) and O(n) are linear - the linear relationship is what matters, not the concrete value. For example, let's modify the linear_algo():



```
def linear_algo(items):
    for item in items:
        print(item)

    for item in items:
        print(item)

linear_algo([4, 5, 6, 8])
```

2.Linear Complexity - O(n) -> Note-2



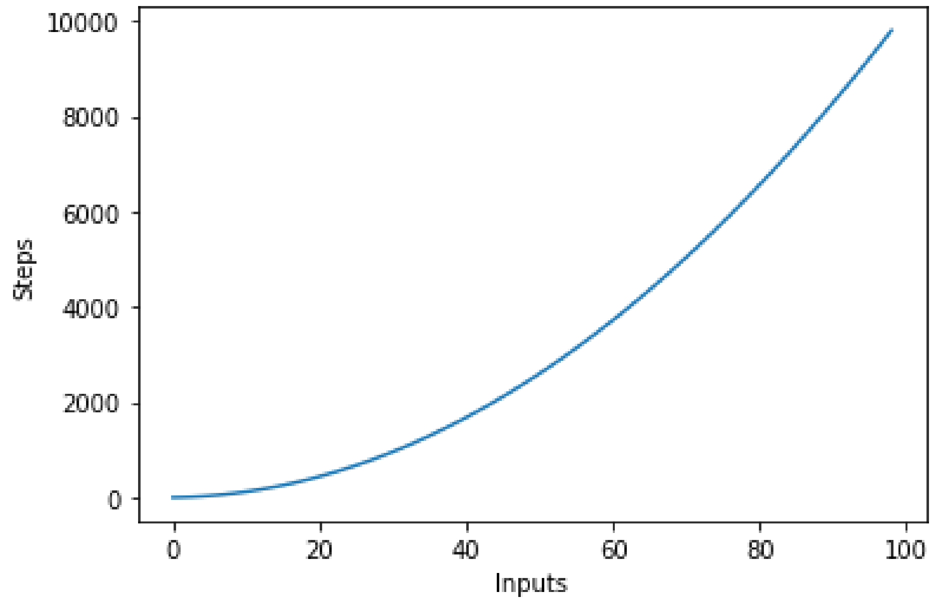
There are <u>two for-loops</u> that iterate over the input items list. Therefore the complexity of the algorithm becomes **O(2n)**, however in the case of infinite items in the input list, the twice of infinity is still equal to infinity. We can ignore the constant **2** (since it is ultimately insignificant) and the complexity of the algorithm remains O(n).

3.Quadratic Complexity - O(n²)

The complexity of an algorithm is said to be **quadratic** when the steps required to execute an algorithm are a quadratic function of the number of items in the input. Quadratic complexity is denoted as **O(n²)**:

```python
def quadratic_algo(items):
    for item in items:
        for item2 in items:
            print(item, ' ' ,item2)

quadratic_algo([4, 5, 6, 8])
```

We have an outer loop that iterates through all the items in the input list and then a nested inner loop, which again iterates through all the items in the input list. The total number of steps performed is **n*n**, where n is the number of items in the input array.



4.Logarithmic Complexity - O(logn)

Some algorithms achieve logarithmic complexity, such as Binary Search. Binary Search searches for an element in an array, by checking the middle of an array, and pruning the half in which the element isn't. It does this again for the remaining half, and continues the same steps until the element is found. In each step, it halves the number of elements in the array.

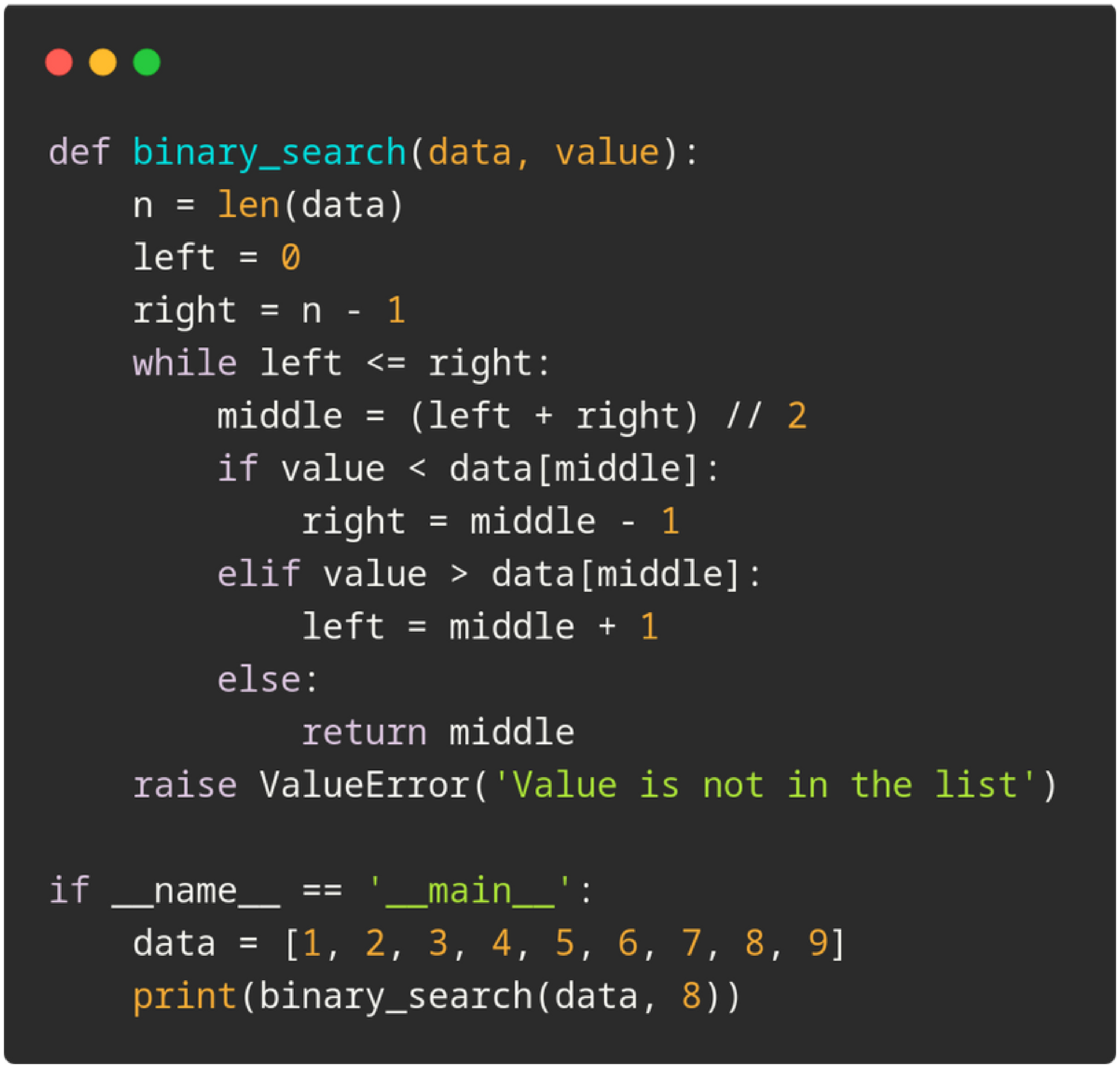
This requires the array to be sorted, and for us to make an assumption about the data (such as that it's sorted).

When you can make assumptions about the incoming data, you can take steps that reduce the complexity of an algorithm. Logarithmic complexity is desirable, as it achieves good performance even with highly scaled input.

4.Logarithmic Complexity - O(logn) -> Example

Algorithms with logarithmic time complexity are commonly found in operations on binary trees or when using binary search. Let's take a look at the example of a binary search, where we need to find the position of an element in a sorted list:

Example:

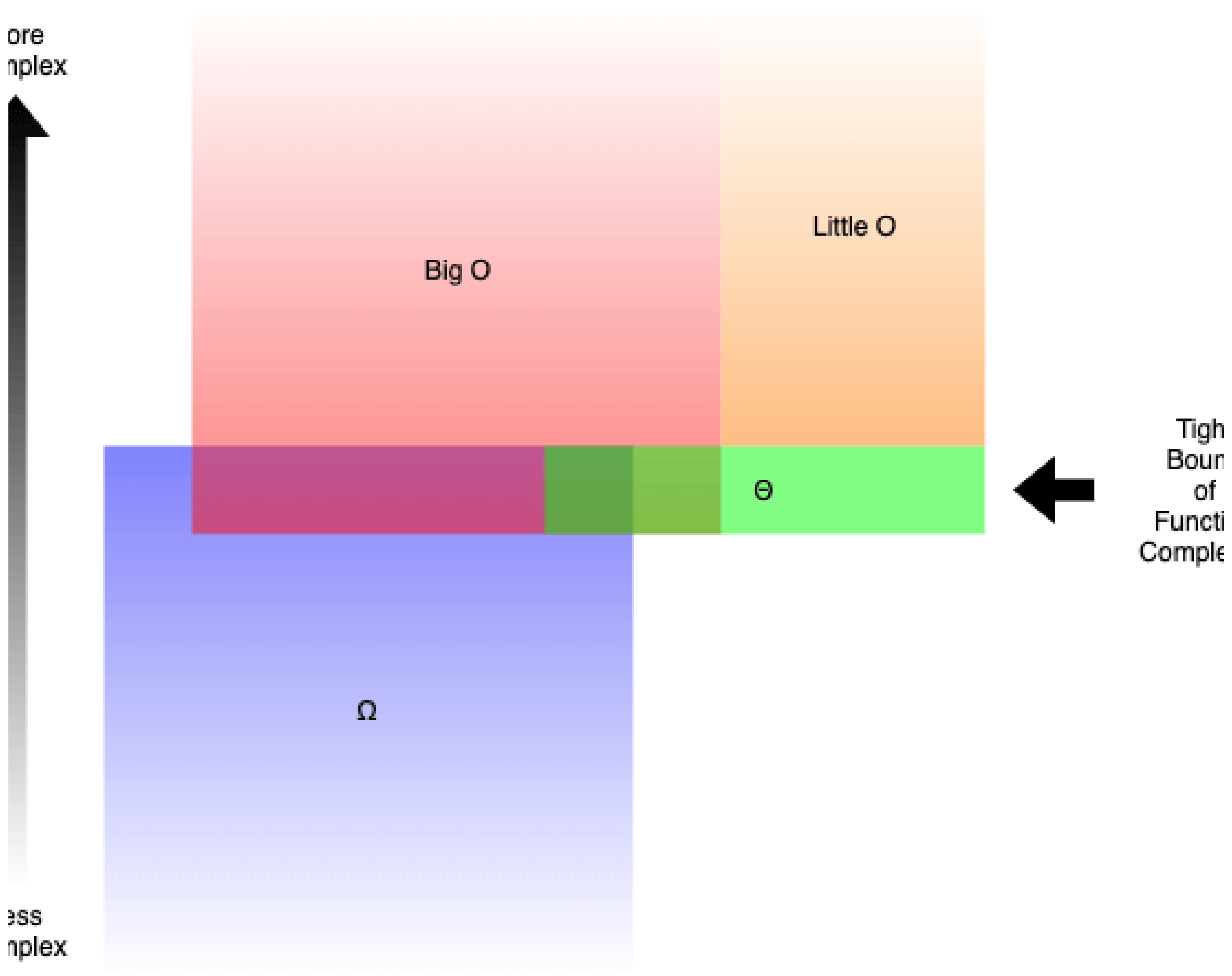


```python
def binary_search(data, value):
    n = len(data)
    left = 0
    right = n - 1
    while left <= right:
        middle = (left + right) // 2
        if value < data[middle]:
            right = middle - 1
        elif value > data[middle]:
            left = middle + 1
        else:
            return middle
    raise ValueError('Value is not in the list')

if __name__ == '__main__':
    data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    print(binary_search(data, 8))
```

Big O, Little O, Omega & Theta->1

@miladkoohi

- **Big O**: “f(n) is O(g(n))” if f for some constants c and N_0, $f(N) \leq cg(N)$ for all $N > N_0$
- **Omega**: “f(n) is Ω(g(n))” if f for some constants c and N_0, $f(N) \geq cg(N)$ for all $N > N_0$
- **Theta**: “f(n) is Θ(g(n))” if f f(n) is O(g(n)) and f(n) is Ω(g(n))
- **Little O**: “f(n) is o(g(n))” if f f(n) is O(g(n)) and f(n) is not Θ(g(n))
- —Formal Definition of Big O, Omega, Theta and Little O



In plain words:

- **Big O (O())** describes the upper bound of the complexity.
- **Omega (Ω())** describes the lower bound of the complexity.
- **Theta (Θ())** describes the exact bound of the complexity.
- **Little O (o())** describes the upper bound excluding the exact bound.

For example, the function $g(n) = n^2 + 3n$ is $O(n^3)$, $o(n^4)$, $\Theta(n^2)$ and $\Omega(n)$. But you would still be right if you say it is $\Omega(n^2)$ or $O(n^2)$.

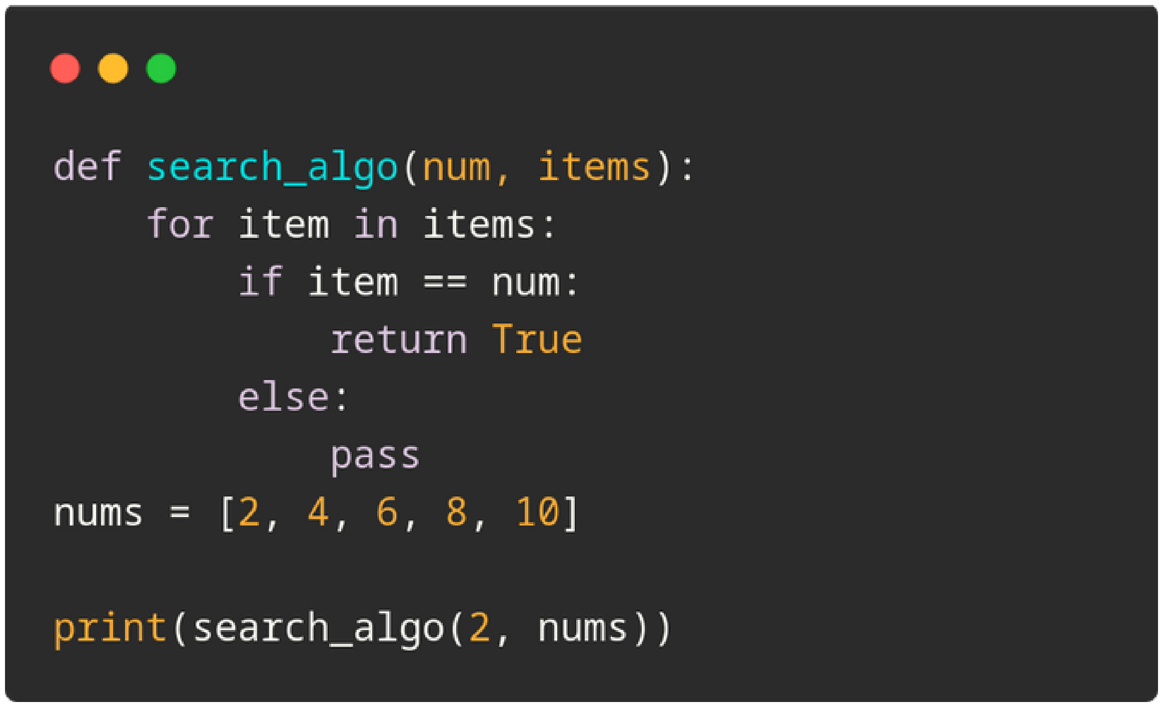
Generally, when we talk about Big O, what we actually meant is Theta. It is kind of meaningless when you give an upper bound that is way larger than the scope of the analysis. This would be similar to solving inequalities by putting ∞ on the larger side, which will almost always make you right.

But how do we determine which functions are more complex than others? In the next section you will be reading, we will learn that in detail.

Worst vs Best Case Complexity->1

Usually, when someone asks you about the complexity of an algorithm - they're interested in the worst-case complexity (Big-O). Sometimes, they might be interested in the **best-case** complexity as well (Big-Omega).

To understand the **relationship** between these, let's take a look at another piece of code:



```python
def search_algo(num, items):
    for item in items:
        if item == num:
            return True
        else:
            pass
nums = [2, 4, 6, 8, 10]

print(search_algo(2, nums))
```

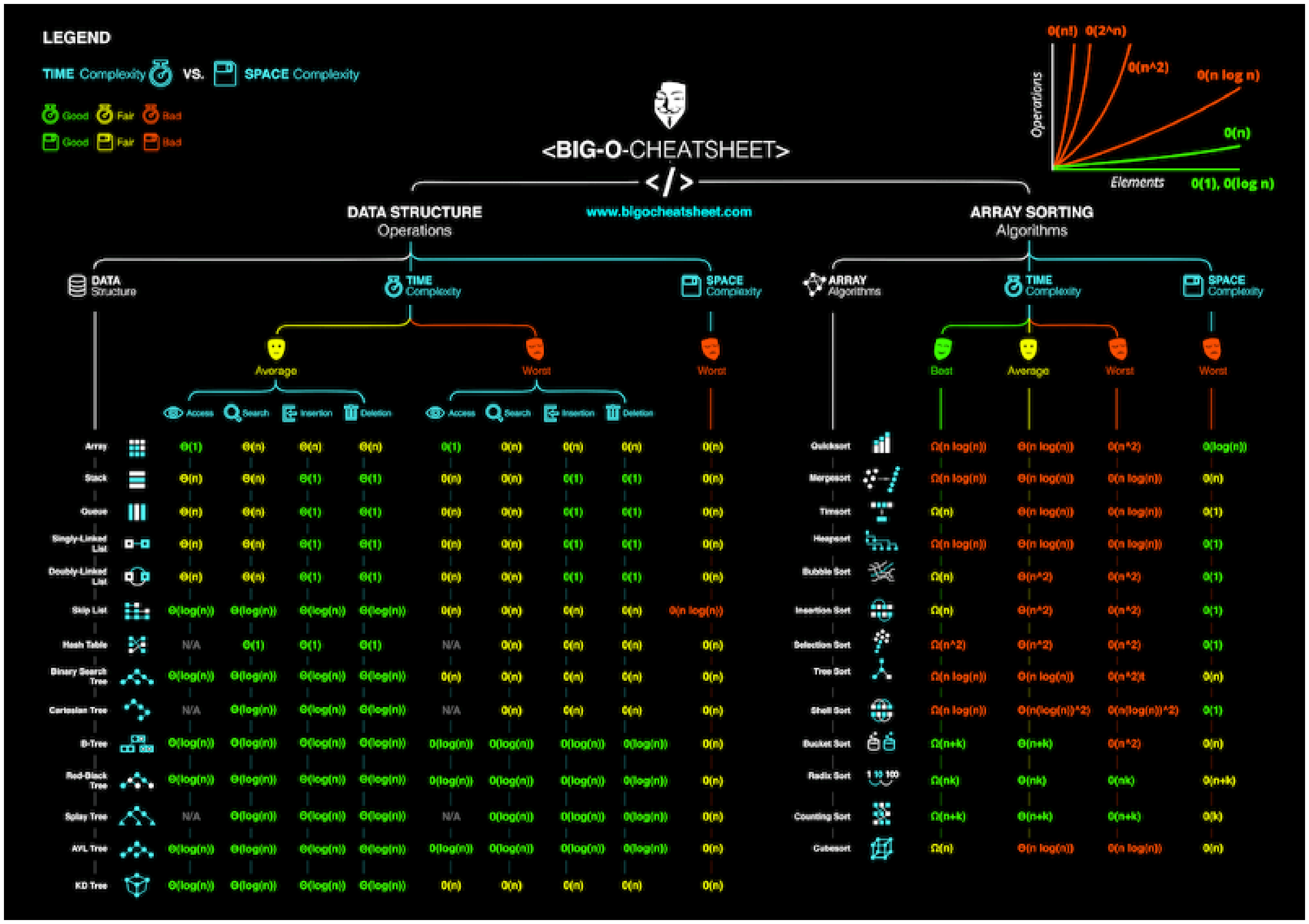
In the script above, we have a function that takes a number and a list of numbers as input. It returns true if the passed number is found in the list of numbers, otherwise, it returns None. If you search for 2 in the list, it will be found in the first comparison. This is the best case complexity of the algorithm in that the searched item is found in the first searched index. The best case complexity, in this case, is O(1). On the other hand, if you search 10, it will be found at the last searched index. The algorithm will have to search through all the items in the list, hence the worst-case complexity becomes O(n).

Worst vs Best Case Complexity->2

@miladkoohi

Note: The worst-case complexity remains the same even if you try to find a non-existent element in a list - it takes n steps to verify that there is no such an element in a list. **Therefore** the worst-case complexity remains O(n).

In addition to best and worst case complexity, you can also calculate the average complexity (Big-Theta) of an algorithm, which tells you "given a random input, what is the expected time complexity of the algorithm"?


LEGEND
TIME Complexity VS. SPACE Complexity
Good Fair Bad
Good Fair Bad
<BIG-O-CHEATSHEET>
</>
www.bigocheatsheet.com
DATA STRUCTURE Operations
ARRAY SORTING Algorithms
O(n!) O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements

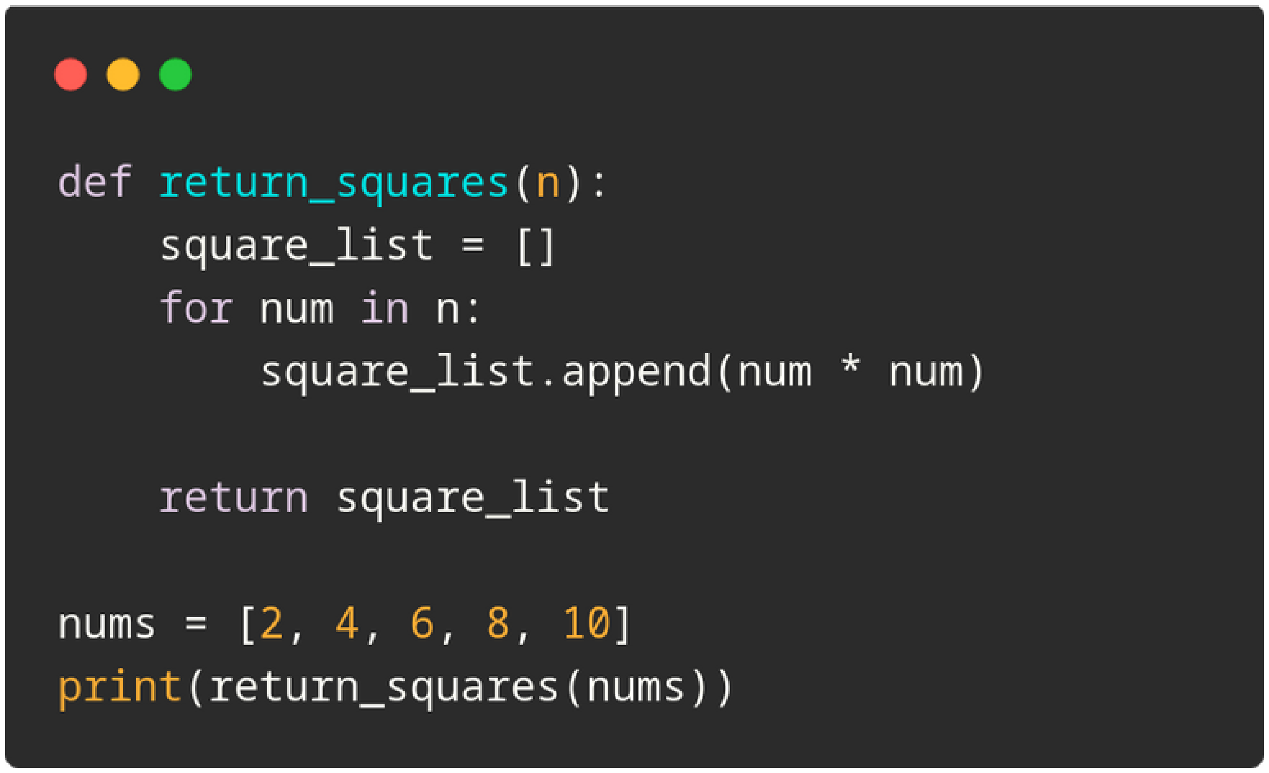
DATA Structure	TIME Complexity								SPACE Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY Algorithms	TIME Complexity			SPACE Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Space Complexity

In addition to the time complexity, where you count the number of steps required to complete the execution of an algorithm, you can also find the space complexity which refers to the amount of space you need to allocate in memory during the **execution** of a program.

Have a look at the following example:



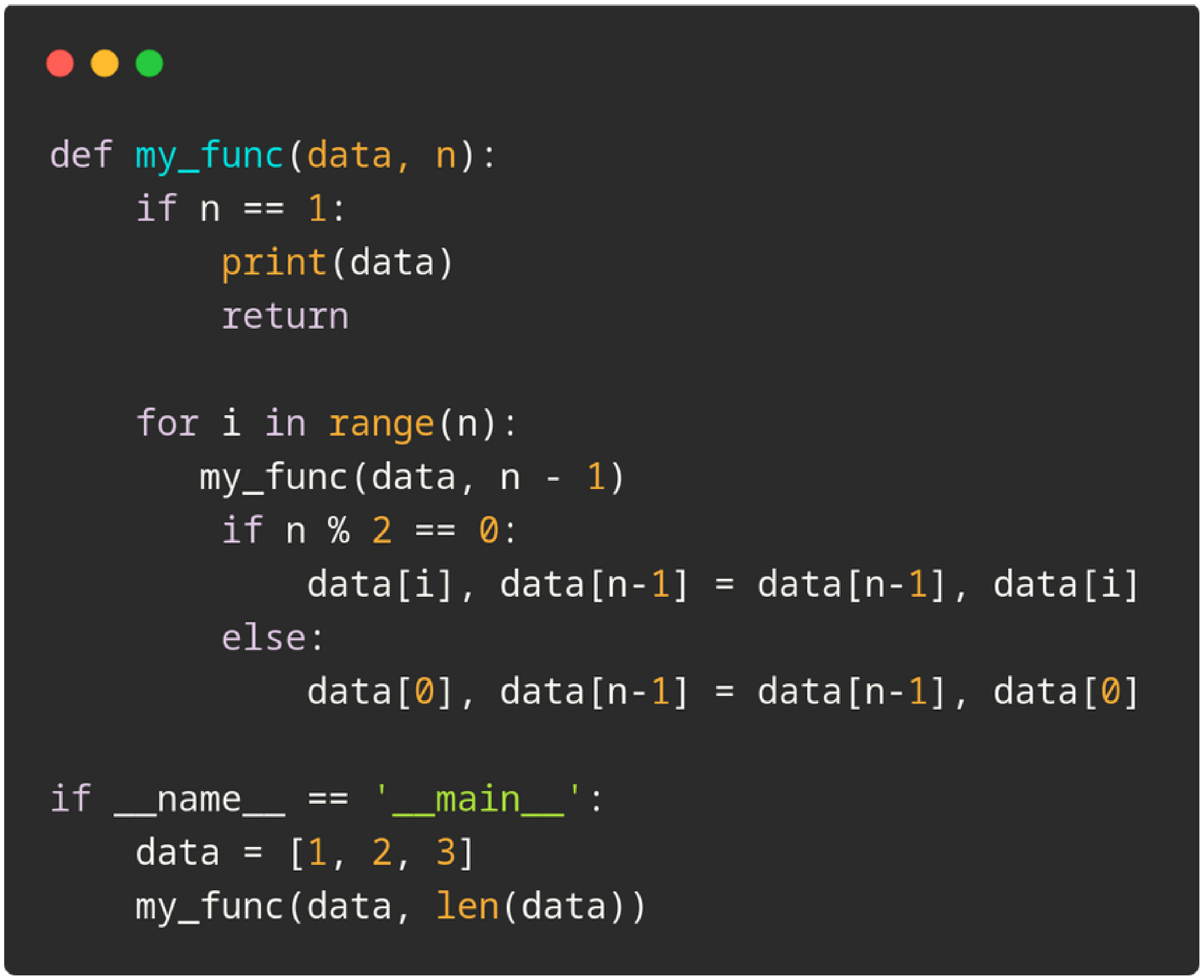
```python
def return_squares(n):
    square_list = []
    for num in n:
        square_list.append(num * num)

    return square_list

nums = [2, 4, 6, 8, 10]
print(return_squares(nums))
```

The *return_squares()* function accepts a list of integers and returns a list with the **corresponding** squares. The algorithm has to allocate memory for the same number of items as in the input list. Therefore, the space complexity of the algorithm becomes O(n).

Challenge!

@miladkoohi

What do you think the time complexity of the following algorithm is?



```python
def my_func(data, n):
    if n == 1:
        print(data)
        return

    for i in range(n):
       my_func(data, n - 1)
        if n % 2 == 0:
            data[i], data[n-1] = data[n-1], data[i]
        else:
            data[0], data[n-1] = data[n-1], data[0]

if __name__ == '__main__':
    data = [1, 2, 3]
    my_func(data, len(data))
```

im milad koohi

write commetns

@miladkoohi

REPOST
if like