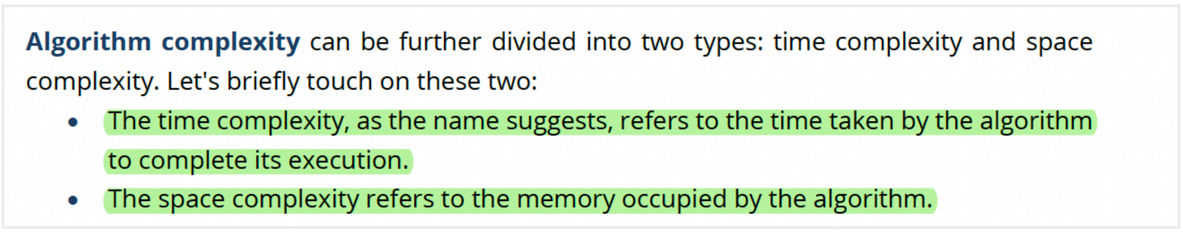
00 Big O notation

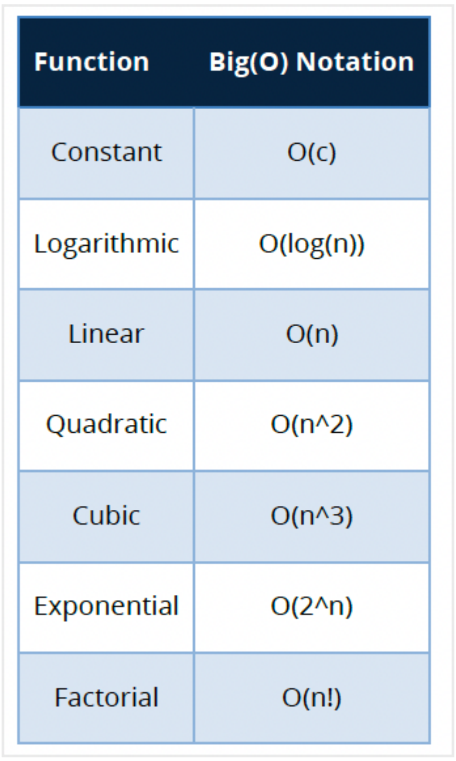


Algorithm complexity can be further divided into two types: time complexity and space complexity. Let's briefly touch on these two:

- The time complexity, as the name suggests, refers to the time taken by the algorithm to complete its execution.
- The space complexity refers to the memory occupied by the algorithm.

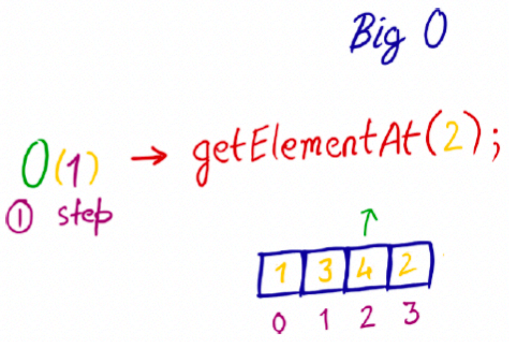
Big(o) notation is an algorithm complexity metric. It defines the relationship between the number of inputs and the steps taken by the algorithm to process those inputs.

** It is about measuring the amount of work a program has to do as an input scales.



Function	Big(O) Notation
Constant	O(c)
Logarithmic	O(log(n))
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Exponential	O(2^n)
Factorial	O(n!)

Constant Complexity

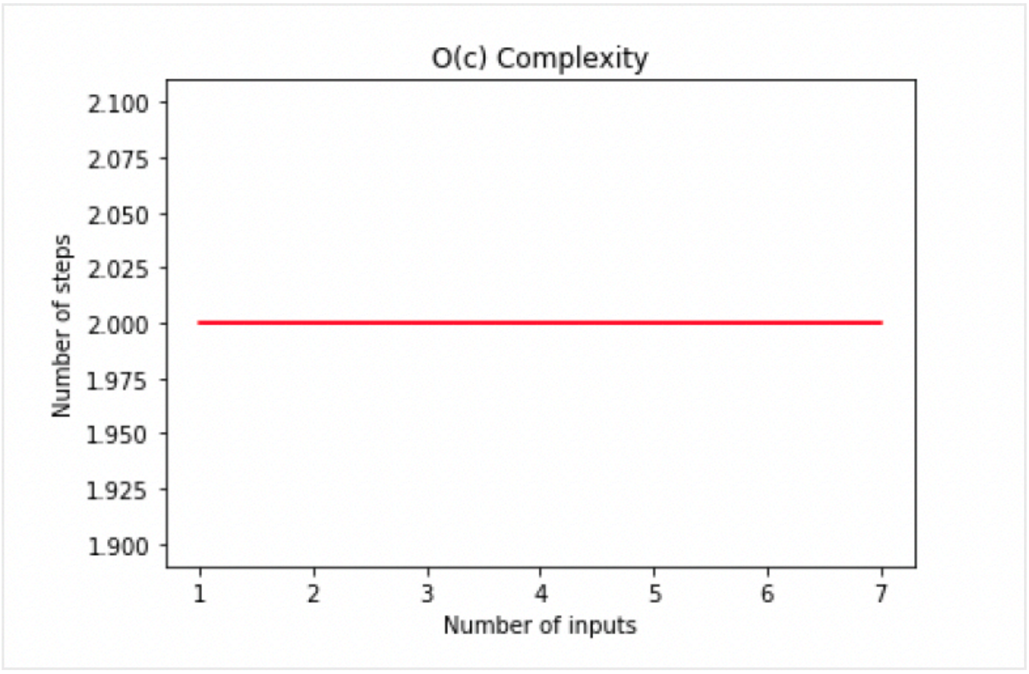


In the constant complexity, the steps taken to complete the execution of a program remains the same irrespective of the input size. Look at the following example:

PYTHON
```python
import numpy as np

def display_first_cube(items):
    result = pow(items[0],3)
    print (result)

inputs = np.array([2,3,4,5,6,7])
display_first_cube(inputs)
```

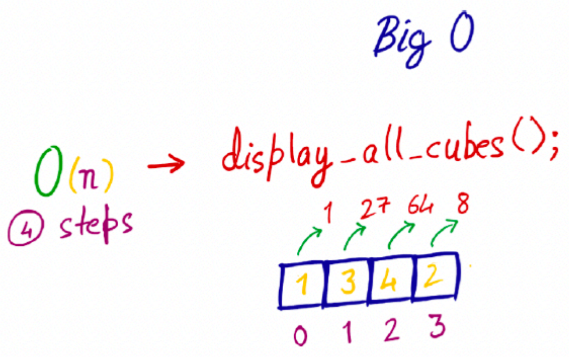
In the above example, the display_first_cube element calculates the cube of a number. That number happens to be the first item of the list that passed to it as a parameter. No matter how many elements there are in the list, the display_first_cube function always performs two steps. First, calculate the cube of the first element. Second, print the result on the console. Hence the algorithm complexity remains constant (it does not scale with input).

Graph- Constant Complexity



Linear Complexity

Linear Complexity*



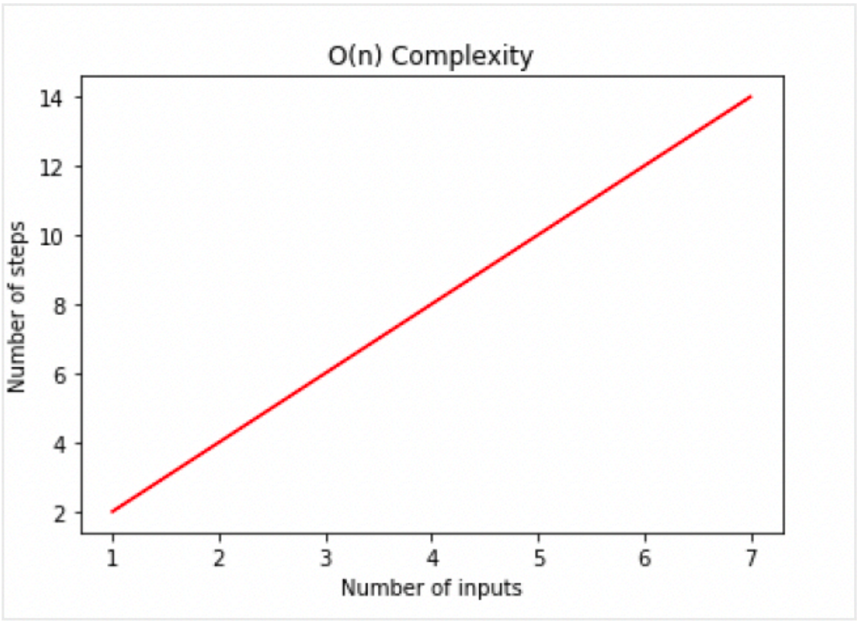
In functions or algorithms with linear complexity, a single unit increase in the input causes a unit increase in the steps required to complete the program execution.

A function that calculates the cubes of all elements in a list has a linear complexity. This is because as the input (the list) grows, it will need to do one unit more work per item. Look at the following script.

PYTHON
```python
import numpy as np

def display_all_cubes(items):
    for item in items:
        result = pow(item,3)
        print (result)

inputs = np.array([2,3,4,5,6,7])
display_all_cubes(inputs)
```

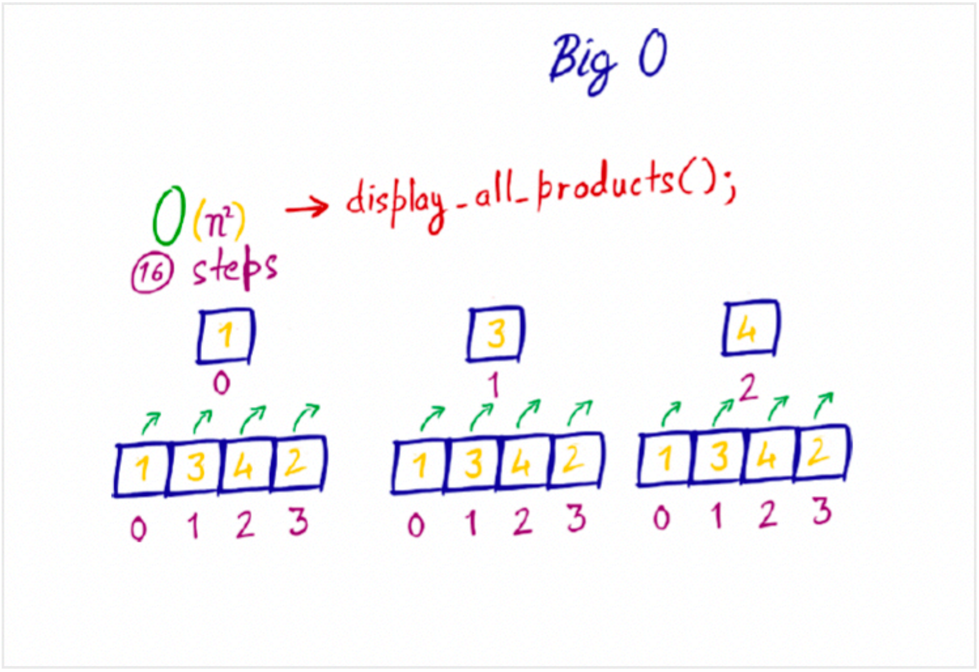
For each item in the `items` list passed as a parameter to `display_all_cubes` function, the function finds the cube of the item and then displays it on the screen. If you double the elements in the input list, the steps needed to execute the `display_all_cubes` function will also be doubled. For the functions, with linear complexity, you should see a straight line increasing in positive direction as shown below:

Graph - Linear Complexity

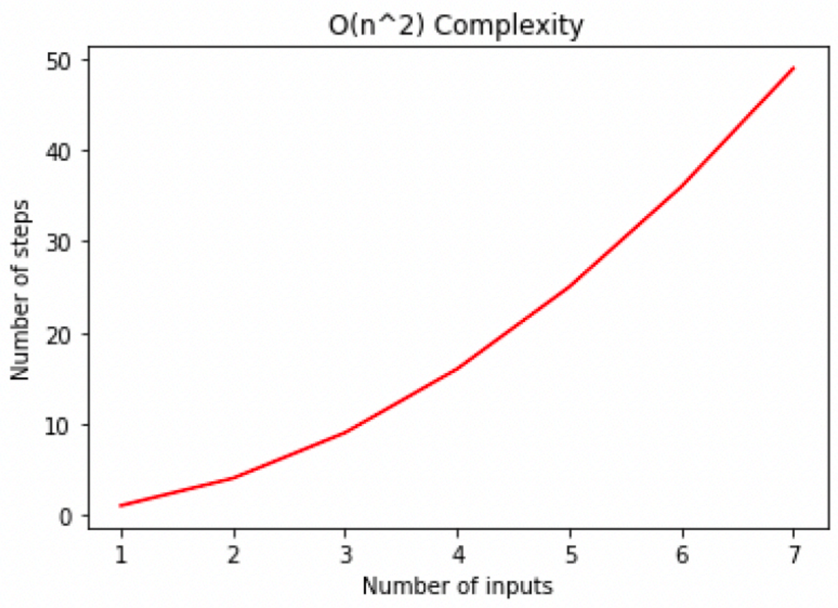


Quadratic Complexity

In contrast to linear complexity, in the case quadratic complexity the output steps **increase quadratically** with the increase in the inputs

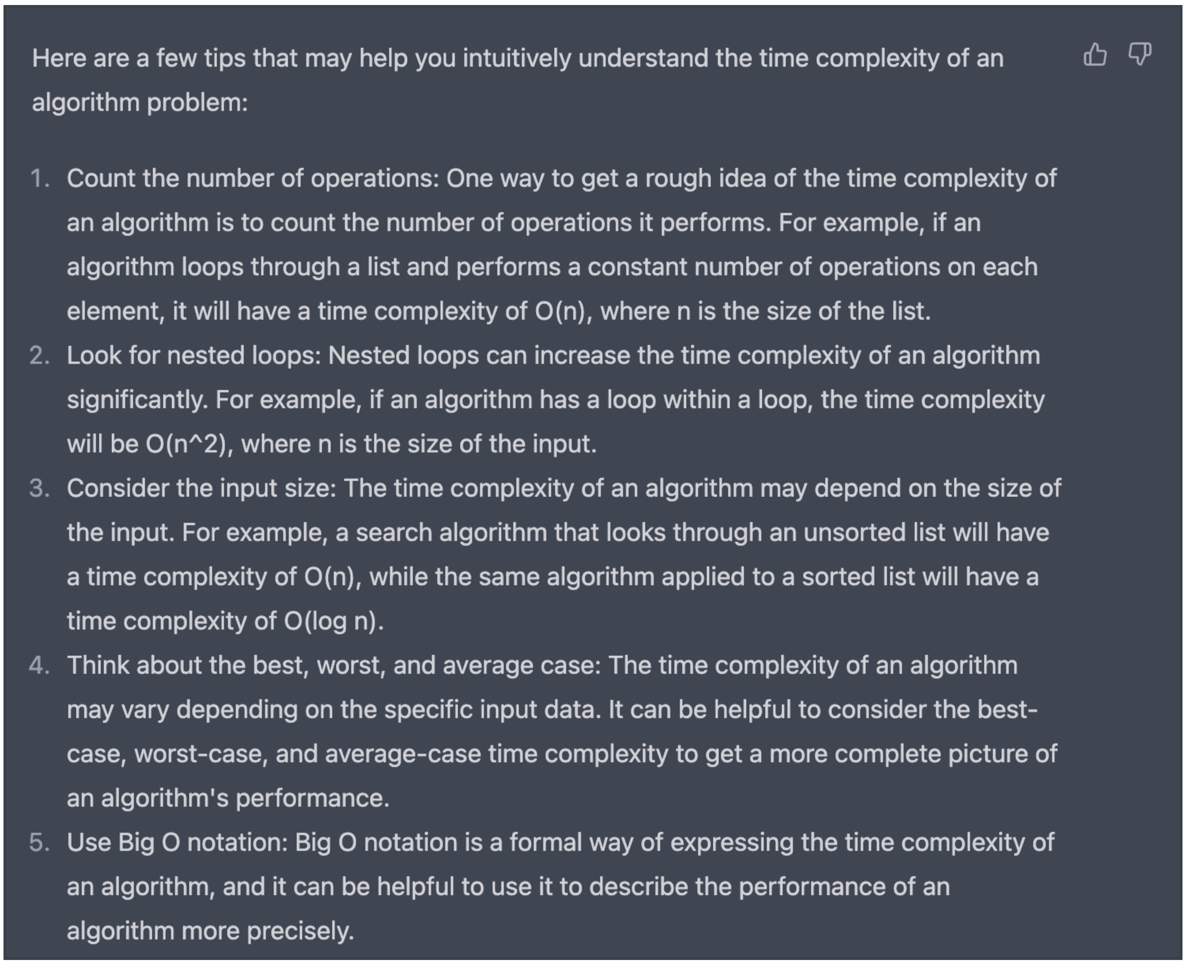


PYTHON
```python
import numpy as np

def display_all_products(items):
    for item in items:
        for inner_item in items:
            print(item * inner_item)

inputs = np.array([2,3,4,5,6])
display_all_products(inputs)
```



The best case complexity refers to the complexity of an algorithm in the ideal situation.

TIPS


Here are a few tips that may help you intuitively understand the time complexity of an algorithm problem:

1. Count the number of operations: One way to get a rough idea of the time complexity of an algorithm is to count the number of operations it performs. For example, if an algorithm loops through a list and performs a constant number of operations on each element, it will have a time complexity of O(n), where n is the size of the list.
2. Look for nested loops: Nested loops can increase the time complexity of an algorithm significantly. For example, if an algorithm has a loop within a loop, the time complexity will be O(n^2), where n is the size of the input.
3. Consider the input size: The time complexity of an algorithm may depend on the size of the input. For example, a search algorithm that looks through an unsorted list will have a time complexity of O(n), while the same algorithm applied to a sorted list will have a time complexity of O(log n).
4. Think about the best, worst, and average case: The time complexity of an algorithm may vary depending on the specific input data. It can be helpful to consider the best-case, worst-case, and average-case time complexity to get a more complete picture of an algorithm's performance.
5. Use Big O notation: Big O notation is a formal way of expressing the time complexity of an algorithm, and it can be helpful to use it to describe the performance of an algorithm more precisely.