Working with time complexity

Introduction

In this reading, you will explore a worked example of a piece of code written in Python, along with how you would evaluate it using Big-O notation.

Evaluating an application's performance ensures that the code written is good and fit for purpose. The question is how do we evaluate efficiency? When we measure electricity, we use kilowatt-hours, which means how many kilowatts an appliance will use if it runs for an hour. The appliance will not always run for an hour, and it may have different requirements depending on the setting used, it is more of a general rule-of-thumb for evaluating cost.

When evaluating coding solutions, Big-O notation is used. So, Big-O notation is the kilowatt hour of code evaluation. It can be applied to measuring how much time a piece of code will take or how much space it will use in memory. Not all processors will run at the same speed, so instead of timing an application, you count the number of instructions an application initiates.

Which measurement reflects the quickest possible execution of some code?

Let's explore which measurement reflects the quickest possible execution of some code.

`O(1)`

You use a constant time algorithm that takes `O(1)` (O-of-one) time to compute. This determines that it will only take one computation to complete a task. An example of this is to print an item from an array.

```python
# An array with 5 numbers
array = [0,1,2,3,4]

# retrieve the number found at index location 3
print(array[3])
```

Run

Reset

In this instance, no matter how many values exist in the array, the approach has a Big-O of one. This means that running this code is considered `O(1)`.

O(n)

Next, let's explore an example of `O(n)`. Taking the same array, an `if` statement is written that looks for the number 5. To establish that 5 is not there, it has to check every item in the array.

```python
# An array with 5 numbers
array = [0,1,2,3,4]

if 5 in array:
    print("five is alive")
```

Run

Reset

In the above example, there is no 5, so there is no printout. To establish this, five checks were made on this array. As the input `n = 5`, this code is said to have a Big – O of `O(n)`. To better understand this, let's extend the array to 10, leaving out the 5.

```python
# an array with 10 numbers
array = [0,1,2,3,4,6,7,8,9,10]
```

```
if 5 in array:
    print("five is still alive")
```

By extending the array 10 integers, the number of computations has now become 10. This is still called `O(n)` because the input size is 10, which is how many checks must be made before the program ends.

O(log n)

This search is less intensive than `O(n)` but more work than `O(1)`. `O(log n)` is a logarithmic search and it will increase as new inputs are added but these inputs only offer marginal increases. An excellent example of this in action is a binary search. Binary search is covered in more detail later in the course.

Now, imagine playing a guessing game with the following prompts: too high, too low, or correct. You are given a range of 100 to 1. You may decide to approach the problem systematically. First, you guess 50 – too high. So, you guess 25 – which is too high. You may choose then to go 12 or 13. What is happening here is that you are halving the search space with each guess.

So, while the input to this function was 100 using a binary search approach, you should come upon the answer in under 5 or 6 guesses. This solution would have a time complexity of `O(log n)`. Even if n (the range of numbers entered) is ten times bigger. It will not take ten times as many guesses.

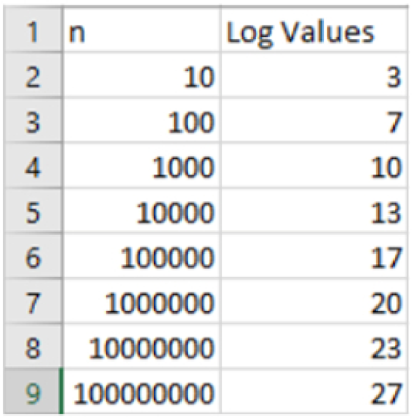
Here is a breakdown of those steps on the array.

```
array = [0,1,2,3,4,6,7,8,9,10]

print("##Step One")
print("Array")
print(array)
midpoint = int(len(array)/2)
print("the midpoint at step one is: " , array[midpoint])

print()

print("##Step Two")
array = array[:midpoint] # 6 is the midpoint of the array
print("Array")
print(array)
# running this shows the numbers left to check
# is 5 < 3
# no
# so discard the left hand side

# so the array is halved again
midpoint=int(len(array)/2)
print("the midpoint is: ",  array[midpoint])

print()
print("##Step Three")
array = array[midpoint:] # so the array is halved at the midpoint
print(array)
# check for the midpoint
midpoint=int(len(array)/2)
print("the midpoint is: " , array[midpoint])
# is 4 < 5
# yes look to the right

print()
print("##Step Four")
print(array[midpoint:])
# check for the midpoint
array = array[midpoint:] # so the array is halved at the midpoint
midpoint=int(len(array)/2)

```

You will notice that to determine if 5 is present, it took 5 steps. That is a big-O score of `O(5)`. You can see that this is bigger than `O(1)` but smaller than `O(n)`. Now, what happens when the array is extended to 100? When looking for a number in an array of 10, it took 5 guesses. Looking at an array of 100 will not take 50 guesses; it will take no more than 10. Equally, if the list is extended to 1000, the guesses will only go up to 15-20.

From this, we can see that it is not `O(1)` because the answer is not immediate. It is not `big-O(n)` because the number of guesses does not go up with the size n of the array. So here, one says that the complexity is `O(log(n))`.

To gain greater insight into how the log values are only a gradual rise, look at a log table up to 100,000,000. This lens shows that `O(log n)` incurs only a minimal processing cost. Running a binary search on an array with any `n` values will, in a worst-case scenario, always make the number of computations found in the log values column.



1	n	Log Values
2	10	3
3	100	7
4	1000	10
5	10000	13
6	100000	17
7	1000000	20
8	10000000	23
9	100000000	27

`O(n^2)` is heavy on computation. This is quadratic complexity, meaning that the work is doubled for every element in the array. An excellent way to visualize this is to consider that you have a variety of arrays. In keeping with the earlier example, let's explore the following code:

```python
new_array=[] # an array to hold all of the results
# array with five numbers
array = [0,1,2,3,4]
for i in range(len(array)): # the array has five values, so this is n=5
    for j in range(len(array)): # still the same array so n = 5
        new_array.append(i*j) # every computation made is stored here

print(len(new_array)) #how big is this new array ?
```

Run

Reset

The first loop will equal the number of elements input, `n`. The second loop will also look at the number of input elements, `n`. So, the overall complexity of running this approach can be said to be `n*n` which is `n^2` (n-squared). To find out how many computations were made, you have to print out the number of times `n` was used in the loop as below.

```python
n = 5 #size of array
print(n*n) # how big is this new array ?
```

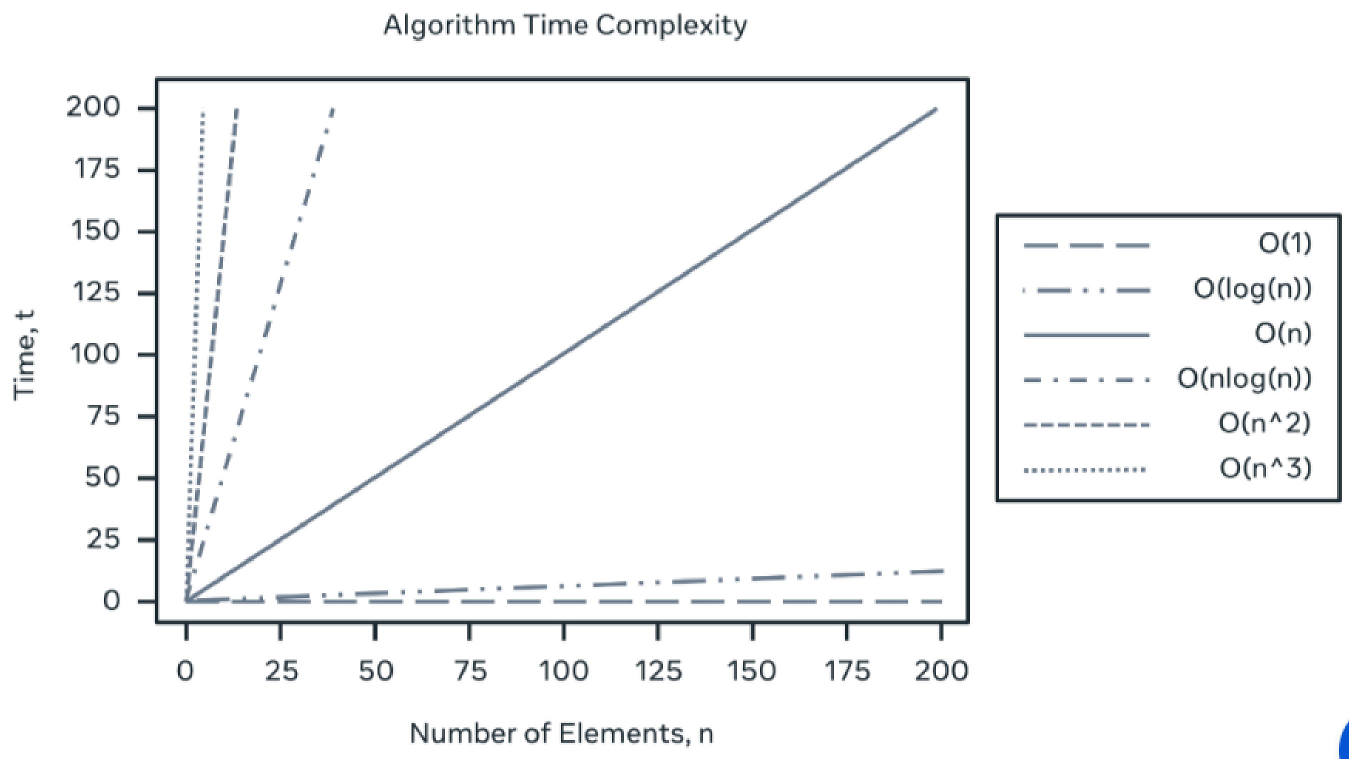
Run

Reset

If you know that the array has 25 elements, then you understand the principles of calculating Big-O notation. To further test your knowledge, how many computations would be required if `n = 6`? Meaning the array had 6 values? The answer is 6 x 6 so 36.

Visual representation of the problem

Below is a graphical representation of how n relates to the number of computations taken.



As you can see, the best time to aim for is `O(1)`; `O(log n)` is still excellent. `O(n)` is ok and `O(n^2)` is not great.

Worst case, best case and average case

Of course, it is not always possible to tell how long an approach will take. When looking at the initial loop example, there was a search performed for an element that was absent. You can say that to search a loop takes `O(n)` times but this might not always be the case.

Consider that the item being searched for is the first in the array. Then the return will be pretty good in `O(1)` time! In the example provided every item must be searched before determining that it was absent: `O(n)` time. The middle case would be that it is found around the middle of the loop `O(n/2)`. When evaluating an approach, three definitions are used: best case, worst case and average case.

Conclusion

This reading introduced the notion of time in relation to complexity and you explored a worked example of a piece of code written in Python. You also investigated how you would evaluate it using Big-O notation.

A good question to ask yourself before you start is, "how many computations does my solution employ and is there a better way?" Now that you know how to use a metric to evaluate the solution to a given problem, you can start thinking of its efficacy concerning time complexity.