Big-O-FIrst

January 29, 2021

0.1 Practical Session on Understanding the Big O Notation

0.1.1 Same task to Two person

Create a function to calculate the factorial

[1]:
```
def fact1(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product
```

[2]:
```
print(fact1(5))
```

```
120
```

The algorithm simply takes an integer as an argument. Inside the fact function a variable named product is initialized to 1. A loop executes from 1 to N and during each iteration, the value in the product is multiplied by the number being iterated by the loop and the result is stored in the product variable again. After the loop executes, the product variable will contain the factorial

Second Persons Algo

[3]:
```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)
```

[4]:
```
print (fact2(5))
```

```
120
```

decide which algorithm to use.

[5]:
```
%timeit fact1(50)
```

```
3.5 µs ± 20.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[6]:
```
%timeit fact2(50)
```

```
6.24 µs ± 28.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

first algorithm is faster compared to the second algorithm involving recursion.

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 metrics for the algorithms is needed. This is where Big O notation comes to play.

0.2 Algorithm Analysis with Big-O Notation

0.2.1 Constant Complexity O(1)

[7]:
```
def constant_algo(items):
    result = items[0] * items[0] # constant time -- even if the number of items␣
↪in arg  in very large O(1)
    print (result)  # constant time operation - print out the result... O(1)

    #O(1)+O(1)+ O(1)+ O(1)=  O(4x1 ) === O(1)
```

[9]:
```
constant_algo([2,3,4,5, 5, 6, 8])
```

```
4
```

irrespective of the input size, or the number of items in the input list items, the algorithm performs only 2 steps: Finding the square of the first element and printing the result on the screen. Hence, the complexity remains constant.

0.2.2 Linear Complexity (O(n))

if the steps required to complete the execution of an algorithm increase or decrease linearly with the number of inputs.

[11]:
```
def linear_algo(items): # n elements in items--list
    for item in items: # O(1)-- initialising for loop  * n times
        print(item)    # O(1)-- printing the element..

# O(n*1) + O(1)  --> O(n*1)--->O(n)
```

[13]:
```
linear_algo([4, 5, 6, 7,7,8,9,908])
```

```
4
5
6
7
7
8
9
908
```

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, and so on.

Another point to note here is that in case of a huge number of inputs the constants become insignificant.

[14]:
```
def linear_algo(items):
    for item in items:  #  O(n)
        print(item)

    for item in items:  # O(n)
        print(item)

        # O(n) +O(n)--> O(2n)-->O(n)

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

```
4
5
6
8
4
5
6
8
```

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

0.2.3 Quadratic Complexity (O(n^2))

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

[15]:
```
def quadratic_algo(items):  # n inputs in items
    for item in items:     #O(n*1)
        for item2 in items:  #O(n*1)
            print(item, ' ' ,item) # O(1)
#O(n*n*1)== O(n^2)
```

[16]:
```
quadratic_algo([4, 5, 6, 8])
```

```
4   4
4   4
4   4
4   4
5   5
5   5
5   5
5   5
6   6
```

```
6   6
6   6
6   6
8   8
8   8
8   8
8   8
```

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.

0.2.4 Finding the Complexity of Complex Functions

[26]:
```
for i in range(1,10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

[28]:
```
def complex_algo(items): # n input elements 1000,10000,0000

    for i in range(5): # O(5)
        print ("This is random print")

    for item in items:# directly depending on  number of elements in items
        print(item)  # O(n)

    for item in items: # O(n)
        print(item)

    print("Big Oh") # O(1)
    print("Big Oh") # O(1)
    print("Big Oh") # O(1)
```

[29]:
```
complex_algo([4, 5, 6, 8])
```

```
This is random print
This is random print
This is random print
This is random print
This is random print
```

```
4
5
6
8
4
5
6
8
Big Oh
Big Oh
Big Oh
```

several tasks are being performed, first, a string is printed 5 times on the console using the print statement. Next, we print the input list twice on the screen and finally, another string is being printed three times on the console. To find the complexity of such an algorithm, we need to break down the algorithm code into parts and try to find the complexity of the individual pieces.

O(5)+O(n)+O(n)+O(3).

O(8)+O(2n).

O(n)

0.2.5 Worst vs Best Case Complexity

[31]:
```python
def search_algo(num, items):
    for item in items:  # O(n*1){worst case}  : best case -->
                        #for loop needs to be initialized only once.. O(1)
        if item == num: # O(1)
            return True # O(1)
        else:
            return False # O(1)
nums = [2, 4, 6, 8, 10]
```

[32]:
```python
print(search_algo(2, nums))
```

```
True
```

If you search 2 in the list, it will be found in the first comparison. This is the best case complexity of the algorithm 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).

[ ]:
```python

```