Common Complexity Scenarios

This lesson summarizes our discussion of complexity measures and includes some commonly used examples and handy formulas to help you with your interview.

We'll cover the following

List of Important Complexities#

In this lesson, we are going to study some common examples and handy formulas for solving time complexity problems.

The following list shows some common loop statements and how much time they take to execute.

Simple for-loop#

```
for x in range(n):
    # statement(s) that take constant time
```

Running Time Complexity = n = $O(n)$

Explanation: Python's `range(n)` function returns an array that contains integers from 0 till n-1 ([0, 1, 2, ..., n-1]). The `in` means that `x` is set equal to the numbers in this array at each iteration of the loop sequentially. So n is first 0, then 1, then 2, ..., then n-1. This means the loop runs a total of n times, hence the running time complexity is n.

For-loop with Increments#

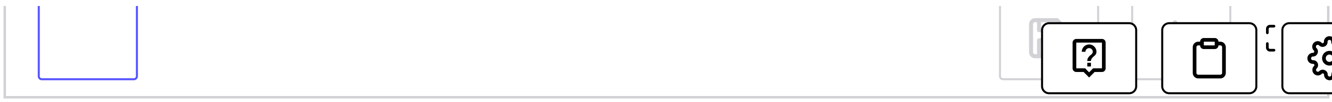
```
for x in range(1, n, k):
    # statement(s) that take constant time
```

Runing Time Complexity = $floor(\frac{n}{k})$ = $O(n)$

Explanation: With this Python statement, `x` is initially set equal to 1 and then gets set to values incremented by `k` until it reaches a number greater than or equal to `n`. In other words, `x` will be set to [$1, 1+k, 1+2k, 1+3k, \cdots, (1+mk) < n$]. It takes $floor(\frac{n}{k})$ time since there are that many numbers that `x` is set to. Also, note that in Python, changing the value of `x` within the loop would not effect the time complexity since `x` is *initialized* at every iteration unlike in other languages such as C++ where `x` is simply incremented/decremented/multiplied/divided. Try it for yourself!

```
k = 2
n = 10

for x in range(1, n, k):
    print(x)
    x = 100  # x is set equal to 100
    print(x)
```



Changing the value of x in Python for-loops

Simple Nested For-loop#

```
for i in range(n):
    for x in range(m):
        # Statement(s) that take(s) constant time
```

Running Time Complexity = $n \times m = O(nm)$

Explanation: The inner loop is a simple for loop that takes m time and the outer loop *runs* it n times. In other words, the outer loop runs n times and the inner loop runs m times at each iteration of the outer loop. So that makes it so that it takes $n \times m$ time in total.

Nested For-loop with Dependant Variables#

```
for i in range(n):
    for x in range(i):
        # Statement(s) that take(s) constant time
```

Running Time Complexity = $\frac{(n-1)((n-1)+1)}{2}$ = $O(n^2)$

Explanation: The outer loop runs n times and for each time the outer loop runs, the inner loop runs `i` times. So, the statements in the inner loop do not run at the first iteration of the outer loop since `i` is 0 then; they run *once* at the second iteration of the outer loop since `i` is equal to 1 at that point, then they run *twice*, then *thrice*, until `i` is $n-1$. So, they run a to

$1+2+3+\cdots+n-1$ times = $\left(\sum_{i=1}^{n-1} i\right) = \frac{(n-1)((n-1)+1)}{2}$ = $O(n^2)$

Nested For-loop with Index Modification#

```
for i in range(n):
    i *= 2
    for x in range(i):
        # Statement(s) that take(s) constant time
```

Running Time Complexity = $n(n-1) = n^2 - n = O(n^2)$

Explanation:

Outer Loop	Inner Loop
`i` = 0	`i` = 0*2 = **0**
`i` = 1	`i` = 1*2 = **2**
`i` = 2	`i` = 2*2 = **4**
$\ldots$	$\ldots$
`i` = n-1	`i` = $(n-1) \times 2$ = **2n - 2**

So this means that the constant time statements in the inner loop are run a total of $0+2+4+\cdots+(2n-2)$ times. So, plugging this into the summation formula, we get

$$\left(\sum_{i=1}^{n-1} f(n)\right) = \frac{n-1}{2}(f(1) + f(n-1))$$

$$\Rightarrow \left(\sum_{i=1}^{n-1} 2n\right) \frac{n-1}{2}(2 \times 1 + 2(n-1)) =$$

$\frac{n-1}{2}(2 + 2n - 2))$ =

$\frac{n-1}{2}(2n)$ =

$\frac{2n^2-2n}{2}$ =

$n^2 - n$ =

$O(n^2)$

Loops with log(n) time complexity#

```
i = #constant
n = #constant
k = #constant
while i < n:
    i*=k
    # Statement(s) that take(s) constant time
```

Running Time Complexity = $\log_k(n)$ = $O(\log_k(n))$

Explanation: A loop statement that multiplies/divides the loop variable by a constant such as the above takes $\log_k(n)$ time because the loop runs that many times. Let's consider an example where i = 1, n = 16, and k = 2:

i	Count
1	1

i	Count
2	2
4	3
8	4
16	-

$$\log_k(n) = \log_2(16) = 4$$

Now that you have all the tools necessary to solve complexity problems, let's look at some exercises in the next few lessons.

Back

Useful Formulas

Next

Challenge 1: Big O of Nested Loop wit...

Completed

Report an Issue