Big O Notation

1.1 Definitions

- What is Big O Notation?

  > Big O Notation measures an algorithm's time and space complexity. In other words, it describes the computational complexity of an algorithm.

- What are the three types of time complexities?

  > Big Theta, Big Omega, and Big O.
  >
  > > **Big Omega** is the best case scenario.
  > >
  > > **Big Theta** is between worst and best case scenario.
  > >
  > > **Big O** is the worst case scenario.

1.2 Runtime Complexities

There are many time complexities results:

- O(1): Constant time
- O(N): Linear time
- O(log N): Logarithmic time
- $O(n^2)$: Quadratic time
- $O(2^n)$: Exponential time

1.3 Big O Operations

1.3.1 Dropping constants and dominant terms

- As N tends to infinite, constant factors aren't a big deal, and are always ignored.

  > O(2N) == O(99999999N) == O(N)

- Some complexities run faster than others, so keep only the worst running ones.

  > $O(N^2 + N) == O(N^2)$
  >
  > O(N + log N) == O(log N)

1.3.2 Adding / Multiplying completixies

- When the loops are separated, then you add the complexity terms:

```
for a in arrayA:      # O(N)
  print(a)          # O(1)
for b in arrayB:      # O(N)
  print(b)          # O(1)
                    # O(N) + O(N) + O(1) + O(1) => O(2N) + O(2) ==>
O(2N) => O(N)
```

> The first and second loops run **N** times depending on the input size. This can be represented as **O(N) + O(N) => O(2N) ==> O(N)**. *Remember to remove non dominant terms and constant values*.

- When the loops are nested, and the array variable is the same, then you multiply complexity terms:

```
for a in arrayA:      # O(N)
  print(a)          # O(1)
  for i in arrayA:  # O(N)
      print(i)      # O(N)
                    # O(N) * O(N) + O(2) ==> O(N^2)
```

> The first loops run N times depending on the input size, for each iterarion, the second loop is executed **N** times, therefore N*N is the complexity of the above code ==> **O(N^2)**

1.3.3 Some Extra Rules to find the time complexity of an algorithm

```
1. If statements or declarations/asignments                    =>
O(1)
2. Loops                                                       =>
O(N)
3. Nested Loops (same array)                                   =>
O(N<sup>2</sup>
4. Nested Loops (two arrays)                                   =>
O(M + N)
5. Loops where controlling param is divided by x at each step  =>
O(logN)
```

1.4 Measuring Time Complexity of Recusive Algorithms

1.4.1 Regular N-1 Calls

```
def findMaxNumRec(Array A, N):
=> M(N)
    if n ==1:
=> O(1)
        return ArrayA[0]
=> O(1)
    else:
```

```
        return MAX(findMaxNumRec(ArrayA[n-
1],findMaxNumRec(ArrayA,n-1)   => M(N-1)
```

> For recursive functions we assume **M(N)**
>
> Based on the above code and values, M(N) = M(N-1) + O(2) => **M(N) = M(N-1) + O(1)** ` M(N) = M(N-1) + O(1) => We need to continue until N = 1 M(1) = O(1)

```
M(N-1) = M((N-1)-1) + O(1) = M(N-2) + O(1)
M(N-2) = M((N-2)-1) + O(1) = M(N-3) + O(1)

* Substitute values into M(N) equation:
M(N) = O(1) + [M(N-2) + O(1)] => M(N-2) + O(2)
    => O(2) + [M(N-3) + O(1)  => M(N-3) + O(3)
    => ...
    => M(N-a) + a                          => This follows a
pattern where A = N, where A tends to infinity

* Subtitute a to N-1 in the above equation:
=> M(N-(N-1)) + N-1
=> M(N-N+1) + N-1
=> M(1) + N -1                                    => drop constant
terms
=> N
```

`

> The result time complexity is **O(N)**.

1.4.2 N/x Recursive Calls

```
def func(n):
=> M(n)
    if n <= 0:
=> O(1)
        return 1
=> O(1)
    else:
        return 1 + func(n/5)
=> M(n/5)
```

> For recursive functions we assume **M(n)**
>
> Based on the above code and values, M(n) = M(n/5) + O(2) => **M(n) = M(n/5) + O(1)** ` M(n) = M(n/5) + O(1) => We need to continue until n = 1 M(1) = M(1/5) + O(1) = O(1)

```
M(n/5) = M((n/5)/5) + O(1) = M(n/25) + O(1)
M(n/25) = M((n/25)/5) + O(1) = M(n/125) + O(1)
```

```
* Substitute values into M(n) equation:
M(n) = O(1) + [M(n/25) + O(1)] => M(n/25) + O(2)
    => O(2) + [M(n/125) + O(1)  => M(n/125) + O(3)
    => ...
    => M(n/a) + a                                   => follows a
pattern where A = n; A tends to infinity

* Subtitute a to n/5 in the above equation:
=> M(n/(n/5)) + n/5
=> M(5) + n/5                                       => drop constant
terms
=> n/5
```

`

> Since n gets divided in each recursive call, the time complexity is **O(log N)**.

1.4.3 Tree Recursion

```
def f(n):
=> M(n)
    if n <= 1:
=> M(1)
        return 1
=> M(1)
    else:
        return f(n-1) + f(n-1)
=>

       *Let's call f(4):
                      f(4)                              calls
              f(3)              f(3)                      calls
         f(2)   f(2)       f(2)   f(2)                   calls
       f(1)f(1)f(1)f(1)  f(1)f(1)f(1)f(1)                returns
```

> There are 2^0 nodes in f(4), 2^1 nodes in f(3), 2^2 nodes in f(2), 2^3 nodes in f(1)...
>
> *The time complexity is* $\mathbf{2^N}$.