Orders of Growth

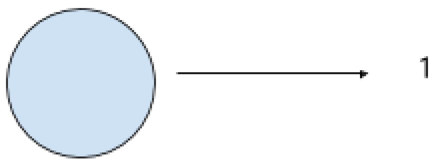
Orders of growth tell us the runtime of a function as n, or our input increases. It tells us how efficient our functions are. We represent orders of growth with Big O notation -- don't worry too much about the different types of notation: that's what 61B goes more in depth into!

There are different ways to approach orders of growth questions, but I wanted to give an overview of how I typically do these type of questions!

1. Figure out what exactly the function is doing.
2. Draw a picture that shows how much work is being done at each "call" or "level" and sum up the work done at each level. This might sound a little vague, but I'll go more in depth into this in the problems!

Constant:

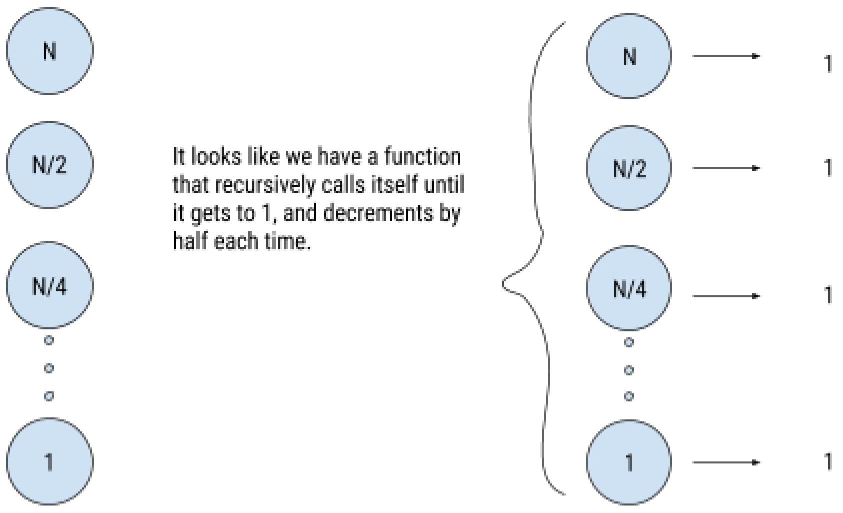
```
def f(x):
      return x * x
```



It looks like we only have one level that does a constant amount of work. Regardless of how big N gets, the total amount work is always constant. So this function is O(1).

Logarithmic:

```
def f(x):
      if x == 1:
            return 1
      else :
            return f(x//2)
```



It looks like we have a function that recursively calls itself until it gets to 1, and decrements by half each time.

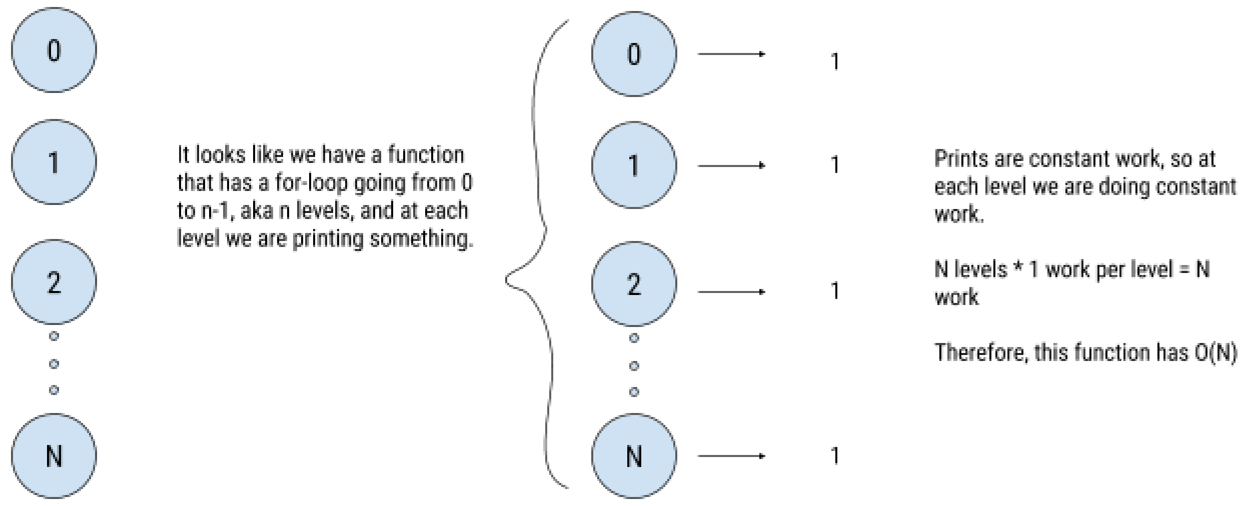
At each level we are doing constant work. Since we are dividing by 2, we have logN levels.

logN levels * 1 work per level = logN work

Therefore, this function has O(logN)

Linear:

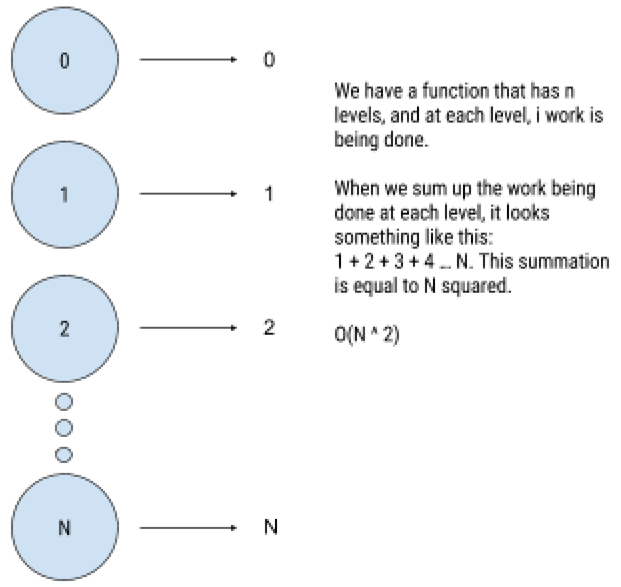
```
def f(x):
      for i in range(x):
            print(i)
```



0
1
2
N
It looks like we have a function that has a for-loop going from 0 to n-1, aka n levels, and at each level we are printing something.
0
1
2
N
1
1
1
1
Prints are constant work, so at each level we are doing constant work.
N levels * 1 work per level = N work
Therefore, this function has O(N)

Quadratic:

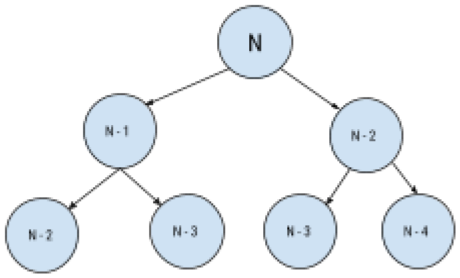
```
def f(x):
      for i in range(x):
            for h in range(i):
                  print(h)
```



0
1
2
N
0
1
2
N
We have a function that has n levels, and at each level, i work is being done.
When we sum up the work being done at each level, it looks something like this:
1 + 2 + 3 + 4 ... N. This summation is equal to N squared.
O(N ^ 2)

Exponential:

```
def f(x):
      if x == 1:
            return 1
      else:
            return f(x - 1) + f(x -2)
```



For this function, we are making two calls every level and are decrementing by 1 and 2. This is what the picture looks like -- very tree like! The number of nodes in this tree are 2 ^ N, because we have N levels and are splitting by 2 each time.

Each node is doing constant work, so the total work being done is O(2^N).