Big O Examples

In the first part of the Big-O example section we will go through various iterations of the various Big-O functions. Make sure to complete the reading assignment!

Let's begin with some simple examples and explore what their Big-O is.

O(1) Constant

In [2]:
```
def func_constant(values):
    '''
    Prints first item in a list of values.
    '''
    print values[0]

func_constant([1,2,3])
```

```
1
```

Note how this function is constant because regardless of the list size, the function will only ever take a constant step size, in this case 1, printing the first value from a list. so we can see here that an input list of 100 values will print just 1 item, a list of 10,000 values will print just 1 item, and a list of **n** values will print just 1 item!

O(n) Linear

In [4]:
```
def func_lin(lst):
    '''
    Takes in list and prints out all values
    '''
    for val in lst:
        print val

func_lin([1,2,3])
```

```
1
2
3
```

This function runs in O(n) (linear time). This means that the number of operations taking place scales linearly with n, so we can see here that an input list of 100 values will print 100 times, a list of 10,000 values will print 10,000 times, and a list of **n** values will print **n** times.

O(n^2) Quadratic

In [8]:
```
def func_quad(lst):
    '''
    Prints pairs for every item in list.
    '''
    for item_1 in lst:
        for item_2 in lst:
            print item_1,item_2

lst = [0, 1, 2, 3]

func_quad(lst)
```

```
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
3 0
3 1
3 2
3 3
```

Note how we now have two loops, one nested inside another. This means that for a list of n items, we will have to perform n operations for *every item in the list!* This means in total, we will perform n times n assignments, or n^2. So a list of 10 items will have 10^2, or 100 operations. You can see how dangerous this can get for very large inputs! This is why Big-O is so important to be aware of!

Calculating Scale of Big-O

In this section we will discuss how insignificant terms drop out of Big-O notation.

When it comes to Big O notation we only care about the most significant terms, remember as the input grows larger only the fastest growing terms will matter. If you've taken a calculus class before, this will remind you of taking limits towards infinity. Let's see an example of how to drop constants:

```
In [10]: def print_once(lst):
             '''
             Prints all items once
             '''
             for val in lst:
                 print val
```

```
In [11]: print_once(lst)
```

```
0
1
2
3
```

The print_once() function is O(n) since it will scale linearly with the input. What about the next example?

```
In [12]: def print_3(lst):
             '''
             Prints all items three times
             '''
             for val in lst:
                 print val

             for val in lst:
                 print val

             for val in lst:
                 print val
```

```
In [13]: print_3(lst)
```

```
0
1
2
3
0
1
2
3
0
1
2
3
```

We can see that the first function will print O(n) items and the second will print O(3n) items. However for n going to inifinity the constant can be dropped, since it will not have a large effect, so both functions are O(n).

Let's see a more complex example of this:

In [14]:
```python
def comp(lst):
    '''
    This function prints the first item O(1)
    Then it prints the first 1/2 of the list O(n/2)
    Then prints a string 10 times O(10)
    '''
    print lst[0]

    midpoint = len(lst)/2

    for val in lst[:midpoint]:
        print val

    for x in range(10):
        print 'number'
```

In [16]:
```python
lst = [1,2,3,4,5,6,7,8,9,10]

comp(lst)
```

```
1
1
2
3
4
5
number
number
number
number
number
number
number
number
number
number
```

So let's break down the operations here. We can combine each operation to get the total Big-O of the function:

$$O(1 + n/2 + 10)$$

We can see that as n grows larger the 1 and 10 terms become insignificant and the 1/2 term multiplied against n will also not have much of an effect as n goes towards infinity. This means the function is simply O(n)!

Worst Case vs Best Case

Many times we are only concerned with the worst possible case of an algorithm, but in an interview setting its important to keep in mind that worst case and best case scenarios may be completely different Big-O times. For example, consider the following function:

In [20]:
```
def matcher(lst,match):
    '''
    Given a list lst, return a boolean indicating if match item is in the list
    '''
    for item in lst:
        if item == match:
            return True
    return False
```

In [21]:
```
lst
```

Out[21]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [22]:
```
matcher(lst,1)
```

Out[22]:
```
True
```

In [24]:
```
matcher(lst,11)
```

Out[24]:
```
False
```

Note that in the first scenario, the best case was actually O(1), since the match was found at the first element. In the case where there is no match, every element must be checked, this results in a worst case time of O(n). Later on we will also discuss average case time.

Finally let's introduce the concept of space complexity.

Space Complexity

Many times we are also concerned with how much memory/space an algorithm uses. The notation of space complexity is the same, but instead of checking the time of operations, we check the size of the allocation of memory.

Let's see a few examples:

In [25]:
```
def printer(n=10):
    '''
    Prints "hello world!" n times
    '''
    for x in range(n):
        print 'Hello World!'
```

In [26]:
```
printer()
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Note how we only assign the 'hello world!' variable once, not every time we print. So the algorithm has O(1) **space** complexity and an O(n) **time** complexity.

Let's see an example of O(n) **space** complexity:

In [27]:
```
def create_list(n):
    new_list = []

    for num in range(n):
        new_list.append('new')

    return new_list
```

In [29]:
```
print create_list(5)
```

```
['new', 'new', 'new', 'new', 'new']
```

Note how the size of the new_list object scales with the input **n**, this shows that it is an O(n) algorithm with regards to **space** complexity.

Thats it for this lecture, before continuing on, make sure to complete the homework assignment below:

Homework Assignment

Your homework assignment after this lecture is to read the fantastic explanations of Big-O at these two sources:

- Big-O Notation Explained (http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o/487278#487278)
- Big-O Examples Explained (http://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly)