Big O Notation

In this lecture we will go over how the syntax of Big-O Notation works and how we can describe algorithms using Big-O Notation!

We previously discussed the functions below:

In [1]:
```
# First function (Note the use of xrange since this is in Python 2)
def sum1(n):
    '''
    Take an input of n and return the sum of the numbers from 0 to n
    '''
    final_sum = 0
    for x in range(n+1):
        final_sum += x

    return final_sum
```

In [2]:
```
def sum2(n):
    """
    Take an input of n and return the sum of the numbers from 0 to n
    """
    return (n*(n+1))/2
```

Now we want to develop a notation to objectively compare the efficiency of these two algorithms. A good place to start would be to compare the number of assignments each algorithm makes.

The original **sum1** function will create an assignment **n+1** times, we can see this from the range based function. This means it will assign the final_sum variable n+1 times. We can then say that for a problem of n size (in this case just a number n) this function will take 1+n steps.

This **n** notation allows us to compare solutions and algorithms relative to the size of the problem, since sum1(10) and sum1(100000) would take very different times to run but be using the same algorithm. We can also note that as n grows very large, the **+1** won't have much effect. So let's begin discussing how to build a syntax for this notation.

Now we will discuss how we can formalize this notation and idea.

Big-O notation describes *how quickly runtime will grow relative to the input as the input get arbitrarily large*.

Let's examine some of these points more closely:

- Remember, we want to compare how quickly runtime will grows, not compare exact runtimes, since those can vary depending on hardware.
- Since we want to compare for a variety of input sizes, we are only concerned with runtime grow *relative* to the input. This is why we use **n** for notation.

- As n gets arbitrarily large we only worry about terms that will grow the fastest as n gets large, to this point, Big-O analysis is also known as **asymptotic analysis**

As for syntax sum1() can be said to be **O(n)** since its runtime grows linearly with the input size. In the next lecture we will go over more specific examples of various O() types and examples. To conclude this lecture we will show the potential for vast difference in runtimes of Big-O functions.

Runtimes of Common Big-O Functions

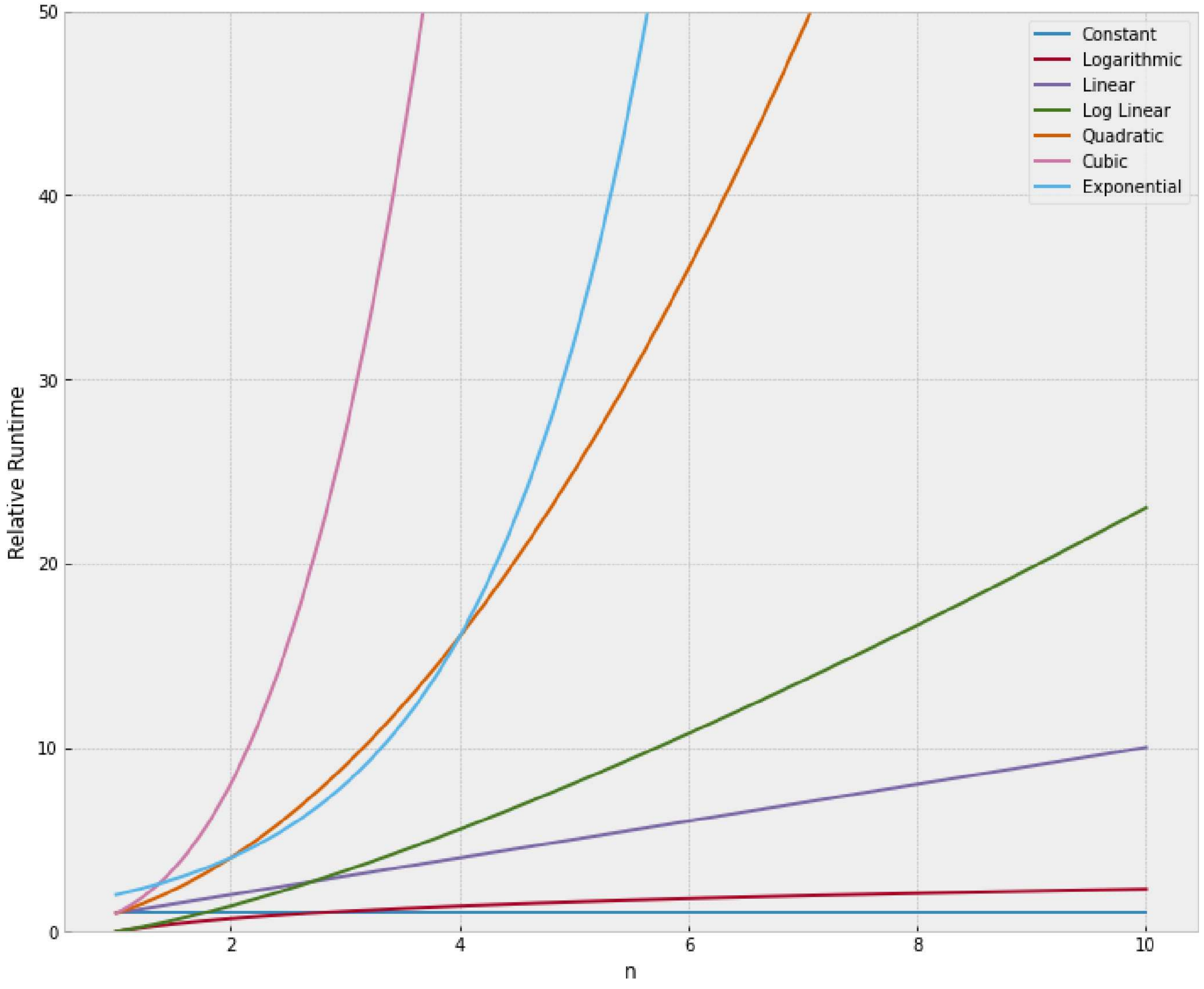
Here is a table of common Big-O functions:

Big-O	Name
1	Constant
log(n)	Logarithmic
n	Linear
nlog(n)	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Now let's plot the runtime versus the Big-O to compare the runtimes. We'll use a simple matplotlib for the plot below. (Don't be concerned with how to use matplotlib, that is irrelevant for this part).

In [3]:
```
from math import log
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')

# Set up runtime comparisons
n = np.linspace(1,10,1000)
labels = ['Constant','Logarithmic','Linear','Log Linear','Quadratic','Cubic','Exponential']
big_o = [np.ones(n.shape),np.log(n),n,n*np.log(n),n**2,n**3,2**n]

# Plot setup
plt.figure(figsize=(12,10))
plt.ylim(0,50)

for i in range(len(big_o)):
    plt.plot(n,big_o[i],label = labels[i])


plt.legend(loc=0)
plt.ylabel('Relative Runtime')
plt.xlabel('n')
```

Out[3]:
```
Text(0.5, 0, 'n')
```



Note how much of a difference a Big-O efficiency can make for the same n value against the projected runtime! Clearly we want to choose algorithms that stay away from any exponential, quadratic, or cubic behavior!

In the next lecture we will learn how to properly denote Big-O and look at examples of various problems and calculate the Big-O of them!