

# Introduction to Algorithm Analysis and Big O

In this lecture we will discuss how to analyze Algorithms and why it is important to do so!

## Why analyze algorithms?

Before we begin, let's clarify what an algorithm is. In this course, an **algorithm** is simply a procedure or formula for solving a problem. Some problems are famous enough that the algorithms have names, as well as some procedures being common enough that the algorithm associated with it also has a name. So now we have a good question we need to answer:

### *How do we analyze algorithms and how can we compare algorithms against each other?*

Imagine if you and a friend both came up with functions to sum the numbers from 0 to N. How would you compare the functions and algorithms within the functions? Let's say you both came up with these two separate functions:

```
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]: sum1(10)
```

```
Out[2]: 55
```

```
In [3]: def sum2(n):
        """
        Take an input of n and return the sum of the numbers from 0 to n
        """
        return (n*(n+1))/2
```

```
In [4]: sum2(10)
```

```
Out[4]: 55.0
```

You'll notice both functions have the same result, but completely different algorithms. You'll note that the first function iteratively adds the numbers, while the second function makes use of:

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}$$

So how can we objectively compare the algorithms? We could compare the amount of space they take in memory or we could also compare how much time it takes each function to run. We can use the built

in **%timeit** magic function in jupyter to compare the time of the functions. The **%timeit** magic in Jupyter Notebooks will repeat the loop iteration a certain number of times and take the best result. Check out the link for the documentation.

Let's go ahead and compare the time it took to run the functions:

```
In [5]: %timeit sum1(100)
```

```
5.17 µs ± 94.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [6]: %timeit sum2(100)
```

```
161 ns ± 9.09 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

We can see that the second function is much more efficient! Running at a much faster rate than the first. However, we can not use "time to run" as an objective measurement, because that will depend on the speed of the computer itself and hardware capabilities. So we will need to use another method, **Big-O!**

In the next lecture we will discuss Big-O notation and why its so important!