You might think that this information is useless to CCC, but it is essential to know these algorithms and data structures to excel in the field of Computer Science.

Big O notation and Time complexity:

Constant time :

constant time is anything that doesn't depend on the size of the input.
For example,

```
def constantRun (num):
        k = num + num
        return k
```

This is a constant time function because no matter what the input is, the asymptotic run time won't change. It will always run at a constant time. We denote this as O(1). It doesn't matter how long the code is. For example,

```
def constantRun2 (num):
        k = num + num
        if(k > someNumber):
                k -= 5
        return k
```

will also have a constant time complexity, and we denote this as O(1) as well.

Let's say you have an array with N elements. To get the sum of all numbers in the array, you'll have to iterate through each element in the array using a forloop like this.

```
sumArray = 0
for i in range(0, len(array)):
        sumArray += array[i]
```

As you can see, depending on the size of the array, the number of times

```
sumArray += array[i]
```

is run increases. If the size of the array is N, this line will run N times. We say that this function as an asymptotic time complexity of O(N).

you may do something like

```
sumArray = sum(array)
```

which will give the same result. This may seem like a O(1), but if you look inside the function "sum" it will actually go through each element in the array.

What's the runtime complexity of

```
for i in range(0, len(array)):
        for j in range(0, len(array)):
                sumArray += array[i]
                sumArray += array[j]
```

Since the inner forloop runs N times, and the outer loop also runs N times, the time complexity is O(N*N) or O(N^2)
Every time the outer loop runs, the inner loop runs N times. And since the outer loop runs N times, it runs N*N times in total.

Exponential runtime:

Let's say that you wanna find out my password. You know that it's made of only numbers. If my password is N digits long, and you program a software to try every possible N digit number to crack my password, there are 10(digits)^N possible numbers to try. This means the software will have an exponential runtime complexity, or O(10^N)

For example, if my password was 4 digits long, the code to try all possible combinations is

```
for i in range(0,10):
        for j in range(0,10):
                for k in range(0,10):
                        for l in range(0,10):
                                if([i,j,k,l] == password):
                                        return True
```

As you can probably guess, exponential times are usually very slow. Much slower than the runtimes we've seen.

Logarithmic runtime:

This runtime complexity will come up very often. Logarithm is the inverse of exponents. It's faster than O(N) (most runtimes)
https://www.google.com/search?q=logarithmic+graph&rlz=1C5CHFA_enCA735CA735&sxsrf=ALeKk038IZeRJyESoKfM8DMs67PITg6REA:1583906581707&source=lnms&tbm=isch&sa=X&ved=2ahUKEwj3xaPp35HoAhWJmHIEHblrD6MQ_AUoAXoECA8QAw&biw=1280&bih=652

Let’s say some program takes X amount of time when you run it with 10 items. If the program has a logarithmic runtime, it will take 2X time to run with 100 items, 3X time to run with 1000 items, etc. This is assuming log base 10.

Most times in algorithms, we refer to log base 2.

Let’s say there is a sorted array with size 14, and we have to find number 12.
[1,2,3,4,5,6,7,8,9,10,11,12,13,14]

We can go through each element in the array, which would give us a runtime of O(N), but there’s a faster way to do this.
We can first look at the element in the middle. array[middle] = 8. We see that the value of this element is smaller than what we’re looking for, so we only have to check the right half of the array. Again, we check the middle of the right half, which is
11.

Usually a good algorithm refers to an algorithm with small big O time. Because processors are very advanced (fast) nowadays, we only really care about the runtime when inputs are large.

Searching and Sorting:

Searching algorithm :
As you’ve seen, there’s linear search, and binary search.
Linear search takes linear time - O(N)
Binary search takes logarithmic time - O(log(N))

However, binary search can only be done on sorted arrays.

There are two types of sorting algorithms.
The O(N^2) sort, and the O(N log(N)) sort.
Typical O(N^2) sorting algorithms are insertion sort, bubble sort and such.
Most common O(N log(N)) sorts are merge sort and quick sort. Quick sort is usually faster than most sorts.
It has an average runtime of O(N log(N)) and worst case runtime of O(N^2).