**Data Structures and Algorithms**

**Big O notation:** It is a type of notation used to measure efficiency of an algorithm, it actually measures how much time a functions takes to run.

There are two parts of measuring the efficiency Time complexity and Space Complexity.

Big O notation actually uses a score criteria to measure efficiency of a data structure on 4 commonly used metrics

* Accessing
* Deleting
* Searching
* Inserting

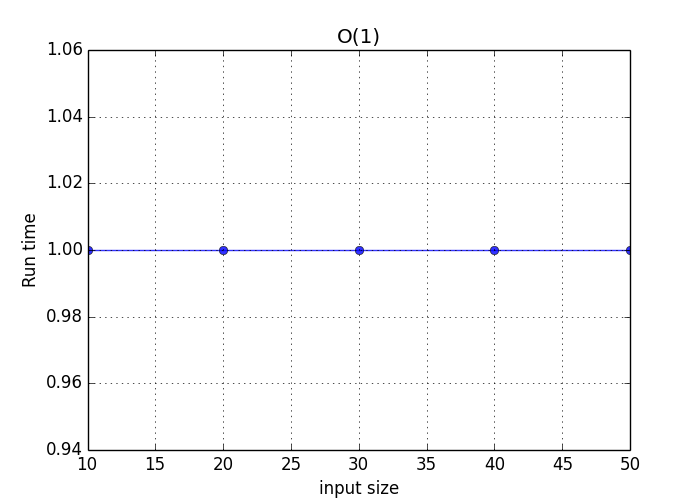
We basically use Big O notation to create a report card. We can measure the score of a certain data structure by using Big O notation Time Complexity Equation.

a BigO Notation Time Complexity equation - or just Time Complexity equations from now on - works by inserting the size of the data-set as an integer n, and returning the number of operations that need to be conducted by the computer before the function can finish. The integer n is simply the size or amount of elements contained within the data set. N represents the number of integers inside a data structure.

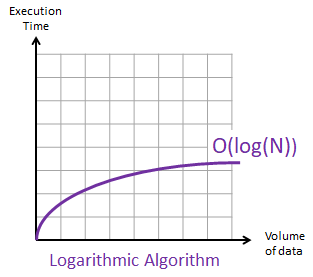
**Types of Big O notation**

O(1)

The absolute best a data structure can “score” on each criteria is a Time Complexity equation of  O(1). This essentially means that NO MATTER what the size of your data set is, the task will be completed in a single step. If your data set has 1 element, the computer will finish the task in one step, if your data has 100 elements, 1 step, 1 Million elements? 1 step. O(1) is the gold standard, absolute best, top of its class time complexity equation. It is basically the Michael Jordan of Big O Notation when it comes to Time Complexity equations.

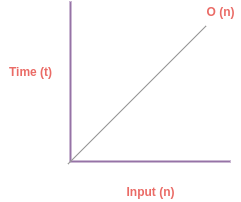


O(log n)

The next fastest type of time complexity equation is O(log n). While not as fast as instantaneous time, a function having a Time Complexity of O(log n) will still provide you with very fast completion. Now if you don’t fully understand logarithms entirely, just know that this efficiency is slower than instantaneous time, O(1), and faster than the next level of efficiency known as O(n). Additionally, because the volume of data versus time graph follows a logarithmic curve, the larger the data set you use, the more bang for your buck you’re going to get. Basically, as the amount of data you try to perform one of our 4 criteria on increases, the rate of change of the amount of operations it takes to complete that certain task decreases.

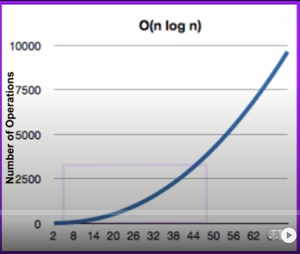
O(n)

O(n) is the next common time complexity equation type that’s going to come up frequently during this lecture. The graph of volume of data versus instructions needed for this function is linear, meaning for every element you add to the data set, the amount of instructions needed to complete the function will increase by the same amount. So to perform a function with a time complexity of O(n) on a data set with 10 elements, it will take 10 instructions. 50 elements will take 50 instructions, 1,000 elements 1,000 instructions and so on. O(n) is really the last “good” Time Complexity equation that exists. Anything above this is considered inefficient and not very practical when it comes to data structures in Computer Science.

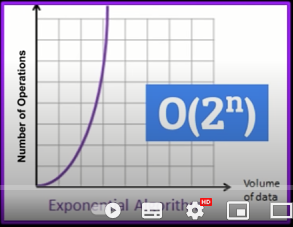


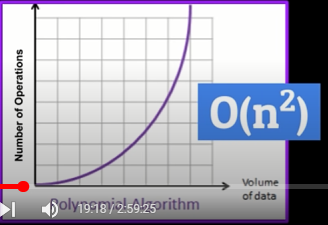
O(n log n)

The next type of equation that will come up is O(n log n). This equation is the first that is relatively bad in terms of efficiency. The graph of volume versus operations shows a somewhat linear but increasing graph, meaning unlike O(log n) it won’t be better in terms of efficiency as the size of the data-set increases. Instead, the slope actually increases as the volume of data does.



The last 2 types of equations are O(n^2) and O(2^n). These are incredibly inefficient equations which should be avoided if at all possible. Because they are both exponential in structure, which can be seen from their graphs of volume versus operations, the larger the data-set you use, the more inefficient it will become.





**Random Access Data structure:**

* Random access data structure as we can randomly access any element
* Provide Accessing of O(1)
* Elements are independent

**Arrays:**

Random access data structure as we can randomly access any element

**Time Complexity Equations:**

Accessing: O(1)

Searching: O(n)

Inserting: O(n)

Deleting: O(n)

**Array list:**

Array list consists of mainly 6 methods

Add Method

Remove Method

Set Method

Clear Method

Get Method

toArray Method

**Time Complexity Equations:**

Accessing: O(1)

Searching: O(n)

Inserting: O(n)

Deleting: O(n)

**Sequential Access Data structure**

* Each element is dependent on other
* Does not provide O(1) accessing
* Access elements through a sequence

**Stack:**

* A sequential data structure in which we can add elements and remove elements according to LIFO principle
* Last In First Out principle states that the element added to the stack at the last will be removed from the list first
* Elements are added from top
* Elements are removed from top

Four methods of stack are:

* Push
* Pop
* Peek/Top
* Contains/Size

**Time Complexity Equations:**

* Accessing: O(n)
* Searching: O(n)
* Inserting: O(1)
* Deleting: O(1)

**Queue:**

* A queue is a sequential access data structure that follows FIFO principle
* In First in First out principle element added first to the list will be removed first
* Elements are added from left(tail) to right(head) in a queue
* Elements are removed from right to left

Four methods of stack are:

* Enqueue
* Dequeue
* Peek/Top
* Contains/Size

**Linked List:**

Linked list is a sequential linear access data structure in which every element is a separate object called a node.

A node consists of 2 parts:

* Data
* Pointer that references to next node in the list

**Time Complexity Equations:**

* Accessing: O(n)
* Searching: O(n)
* Inserting: O(1) or O(n) because it is not specified either adding is from start, end or middle
* Deleting: O(1) or O(n) because it is not specified either deleting is from start, end or middle

**Doubly Linked List:**

Doubly linked list is a sequential access data structure in which every element is a separate object called node.

In doubly linked list node consist of three parts

* Data
* Pointer that references to previous value
* Pointer that references to next value

**Time Complexity Equations:**

* Accessing: O(n)
* Searching: O(n)
* Inserting: O(1) or O(n)
* Deleting: O(1) or O(n)