Data Structures:

1. It organizes and store data

Example: Array, Tree

1. Each has its own strengths and weakness

Example: Array is great when we know the index to access, when we don’t know the index we cant access the array

An algorithm defines the steps needed to perform task

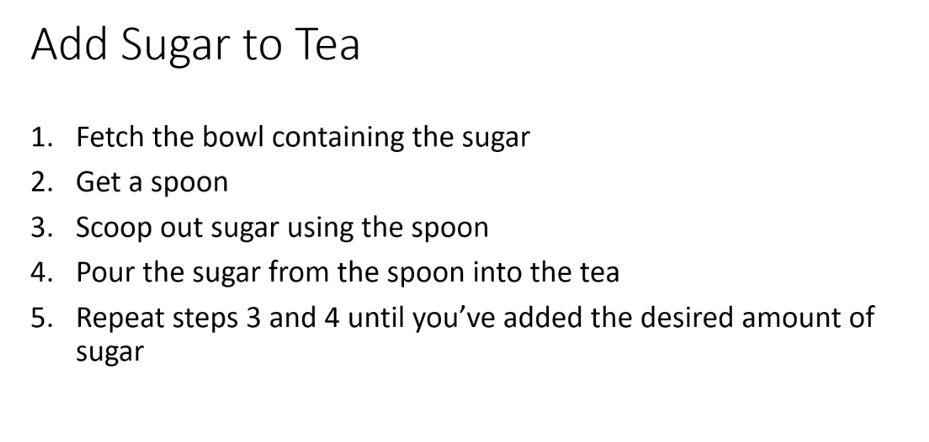
An algorithm is not the implementation, it just describes the steps

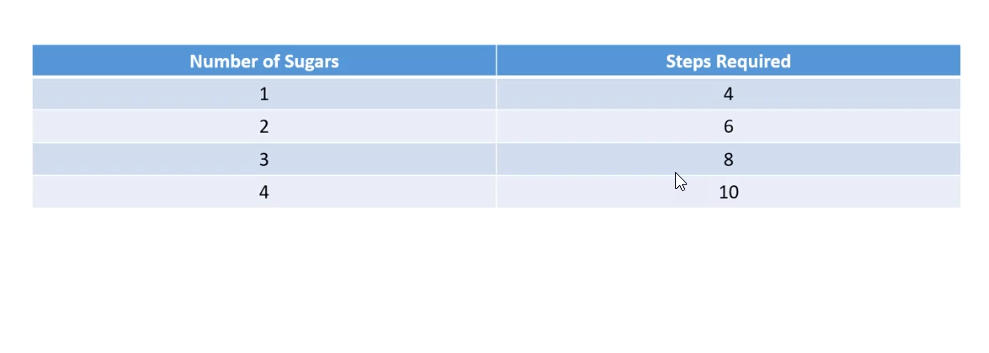
Big O Notation : Way of expressing the complexity of algorithm, Way of measuring, how an algorithm performs

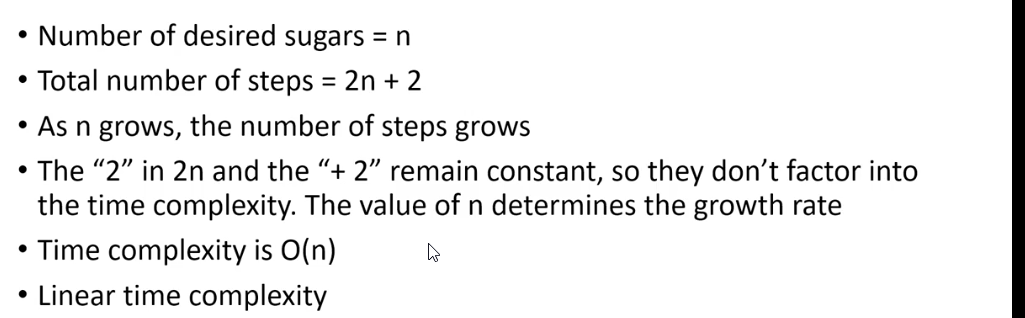
Time Complexity: Number of steps involved to run an alg

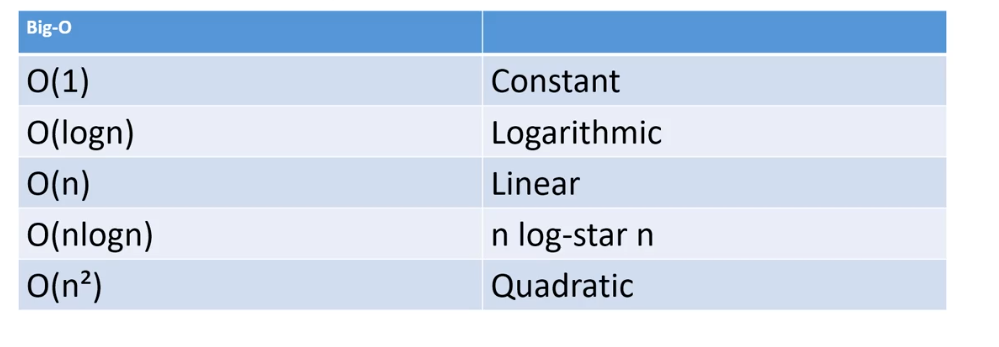
Memory Complexity: Amount of memory requires to run an alg

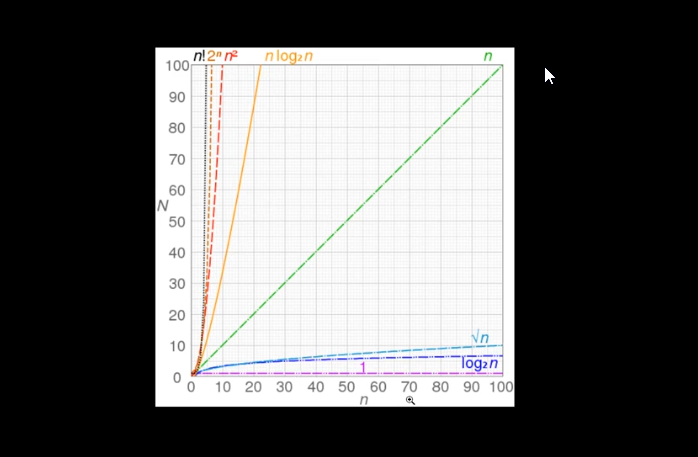
Example:









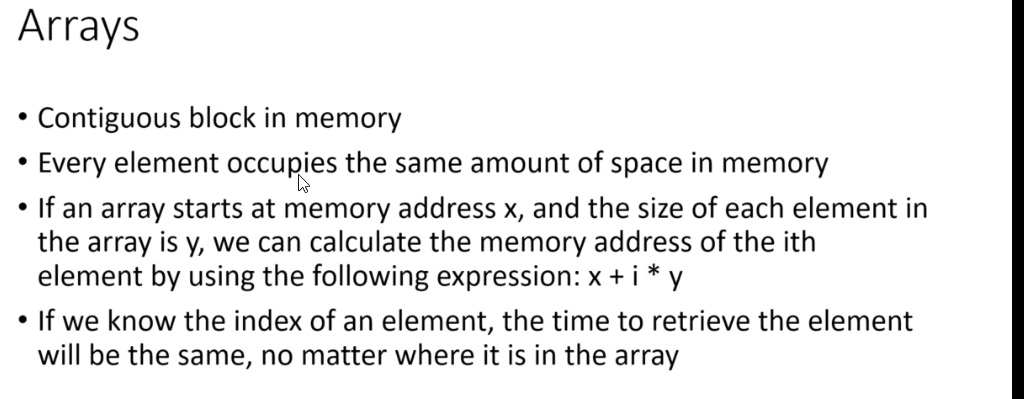


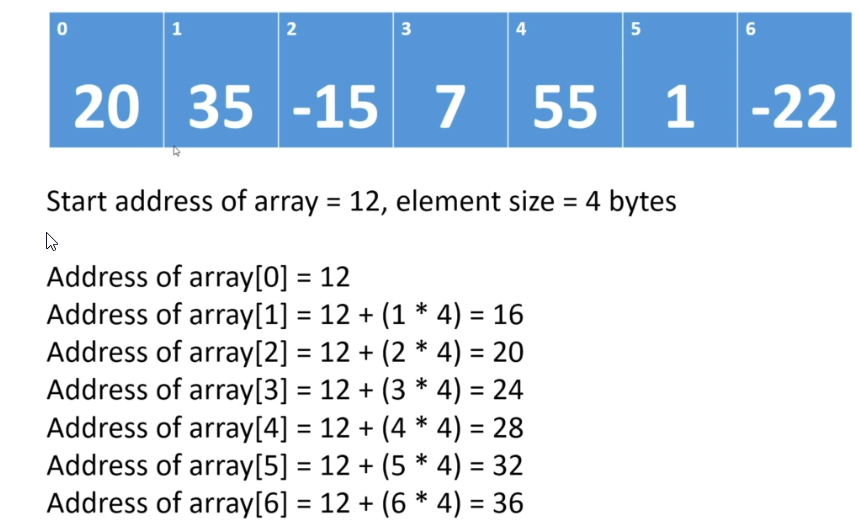
<https://en.wikipedia.org/wiki/Big_O_notation#/media/File:Comparison_computational_complexity.svg>

Arrays:

Contiguous block in memory – It is one huge block – means size needs to be defined (<https://www.geeksforgeeks.org/difference-between-contiguous-and-noncontiguous-memory-allocation/>)

Every element occupies the same amount of space in memory





Here 12 is the Bogus memory address space

Why Array index starts at 0

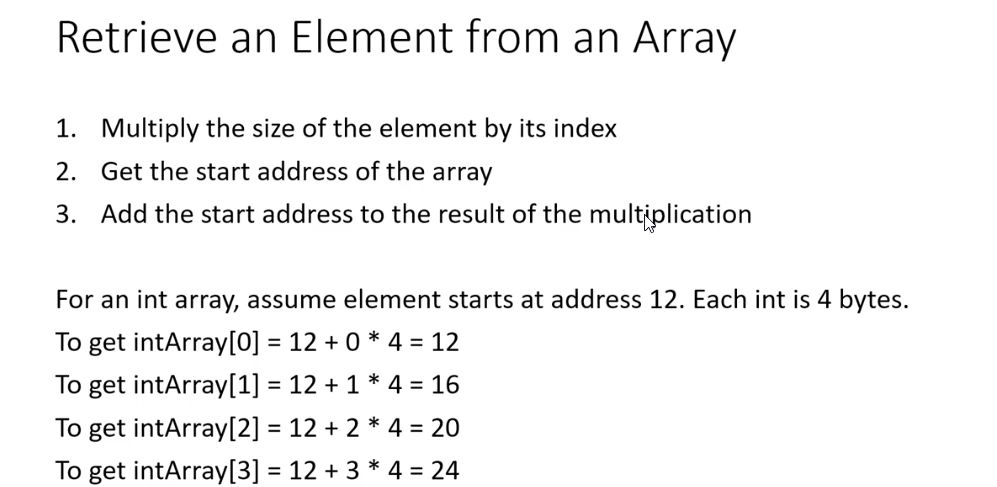
If we look in above image, the formula Is x + (I \* y)

X = some memory address

I = index element

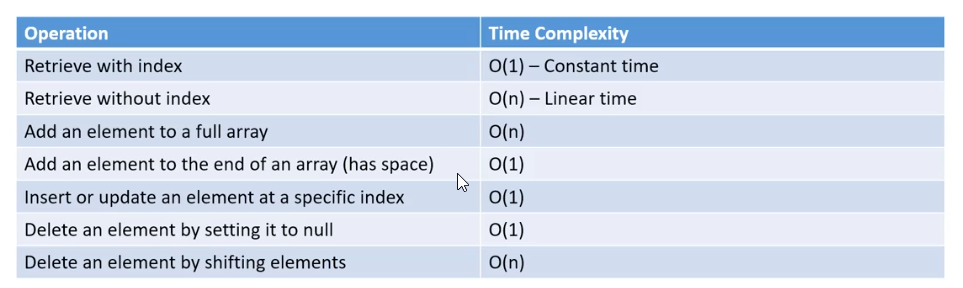
Y = size of each element

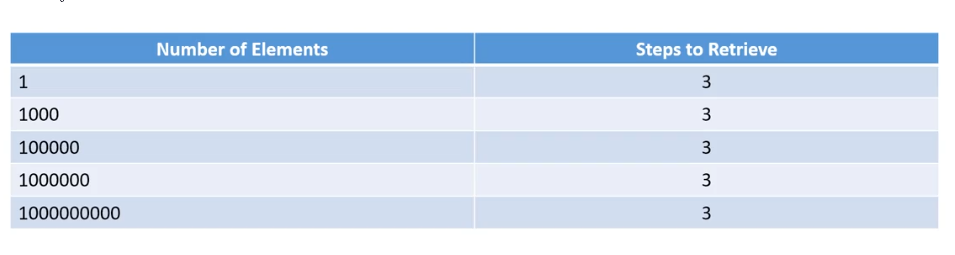
Arrays are efficient when we know the index value from the array



Retrieval in array when you know the index value can be done in O(1) times

Retrieval in array when we not know the index value is 0(n) times – Linear Complexity

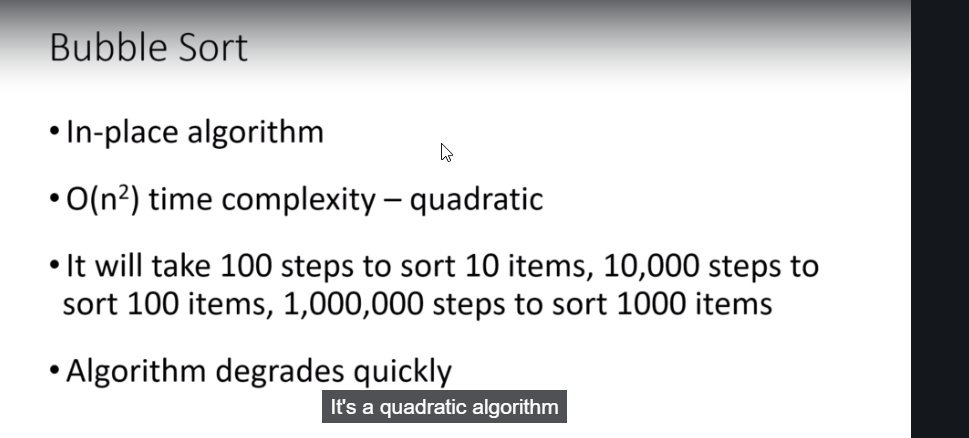




**Sort Algorithms:**

**Bubble Sort**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.  
**Example:**   
**First Pass:**   
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.   
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4   
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2   
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.  
**Second Pass:**   
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )   
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2   
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )   
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )   
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.  
**Third Pass:**   
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )   
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )   
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )   
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** ) 



An in-place algorithm may require a small amount of extra memory for its operation. However, the amount of memory required must not be dependent on the input size and should be constant.

Several sorting algorithms rearrange the input into sorted order in-place, such as insertion sort, selection sort, quick sort, bubble sort, heap sort, etc. All these algorithms require a constant amount of extra space for rearranging the elements in the input array.