Data structures and Alghoritms 1 – BIG O notation

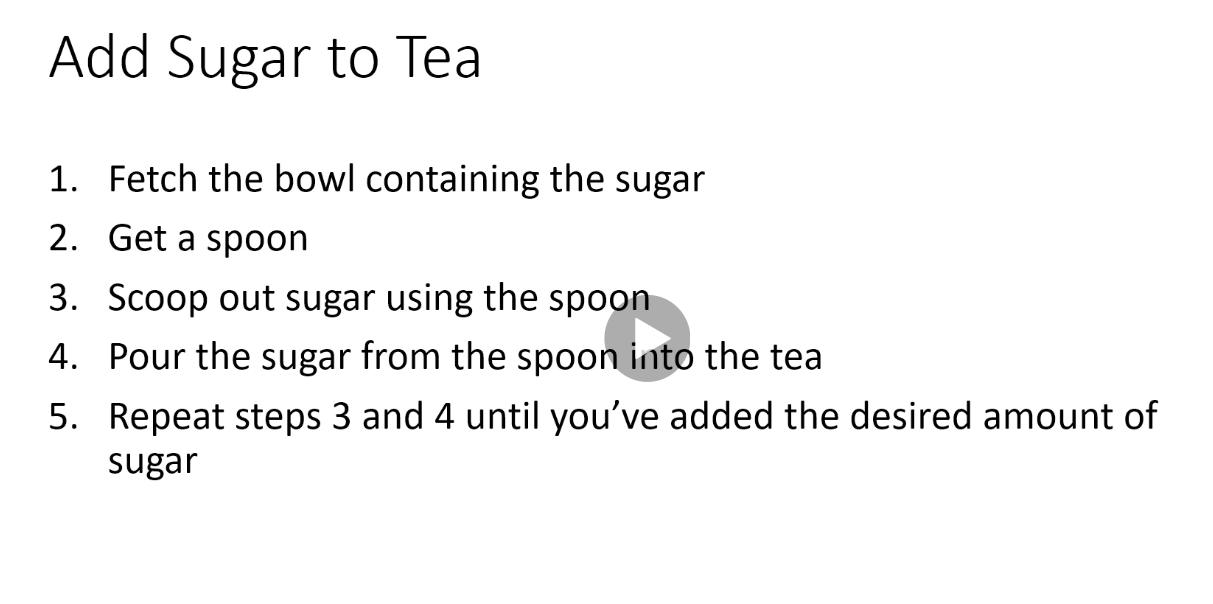
allows us to compare alghoritms in an objective manner, without the differences of hardware implementations.

Time complexity – Number of steps involved in running an alghoritm. How will the alghoritm perform when the number of steps increases ? How well will alghoritm scale. O()

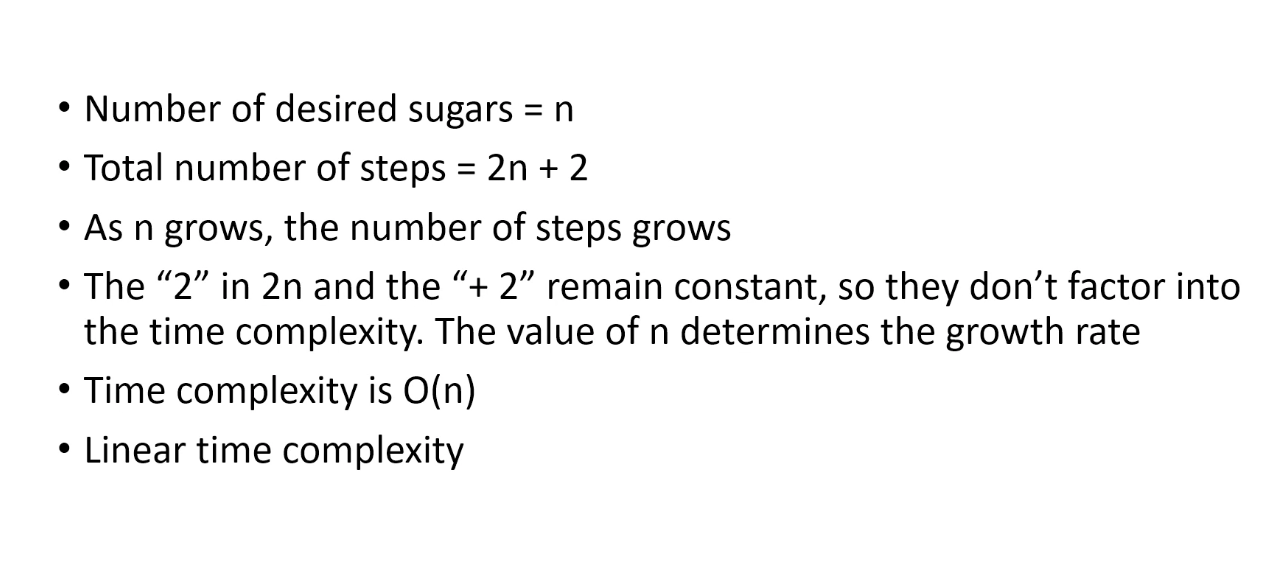
Memory complexity – Amount of memority it takes in running an alghoritm.

We want to look at the worst case scenario. Average case can be also usefull but the worst case is the most usefull. We compare worst case scenarios when comparing two alghoritms

Example:





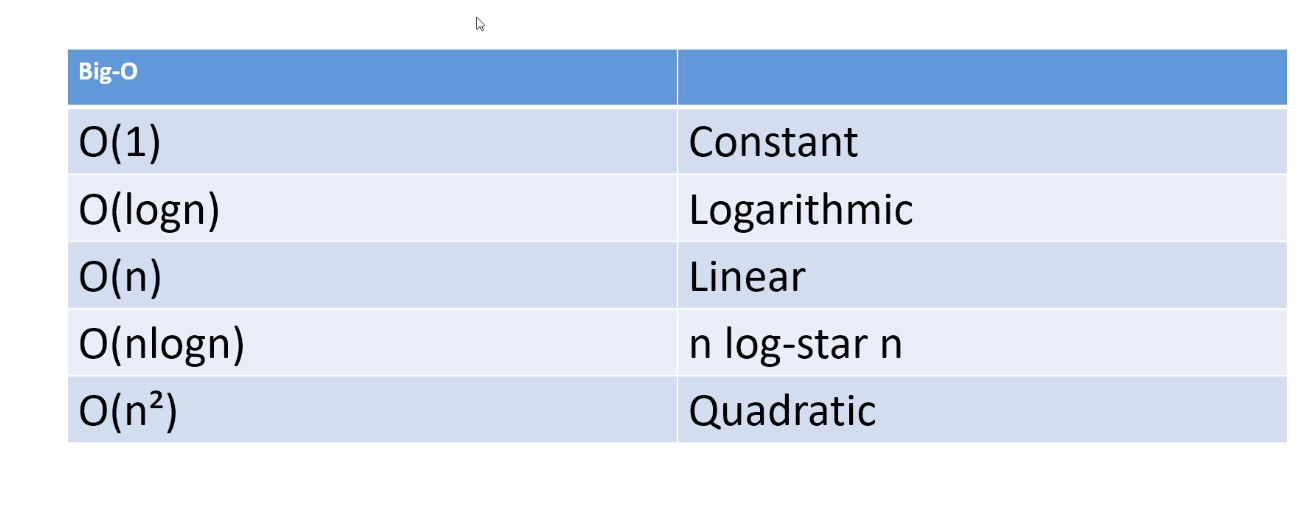


n = number of items

2\*(1)+ 2 = 4

2\*(2)+2 = 6

…



O(log(n)) – log 2 natural logaritm, not log 10 !



X – number of items

Y – number of steps

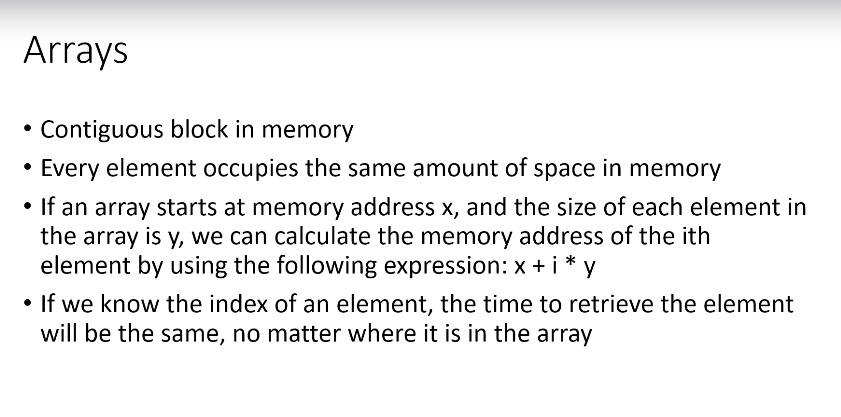
Logn is a very good complexity.

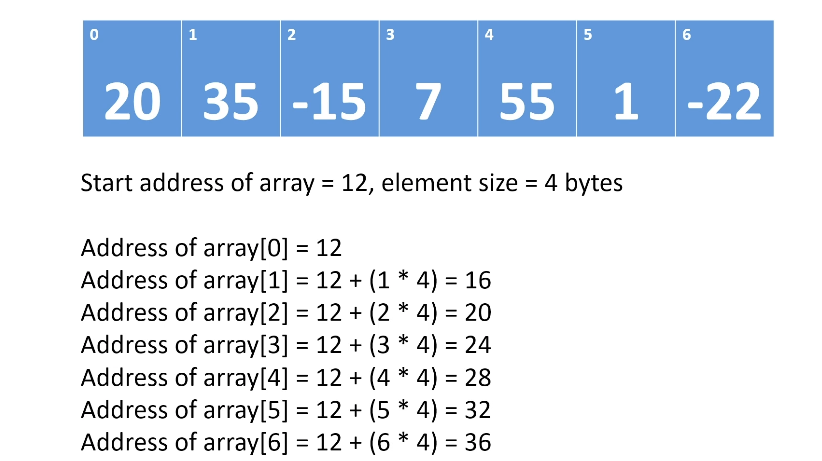
Linear time n is still „ pretty good“

# Arrays

Contiguos block in memory, all items in array are stored in one contiguos block in memory. For example: array starts at memory block 100, it has 200kb, so we have to say how big arrays are because the memory is pre alocated by JVM. Its a static structure alocated in one block of memory.

Every element in array occupies the same amount of space in memory. Int is for example 4 bytes, so each element occupies 4 bytes. When we are working with objects, in array is stored only reference to the objects, not objects themself! Each reference has the same size !!!!

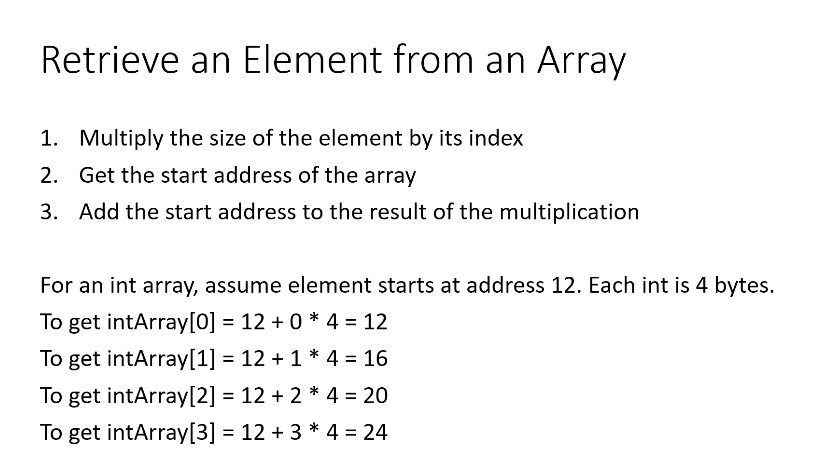




We can use this because array is one continuos block in memory and we know the size of each element in array.

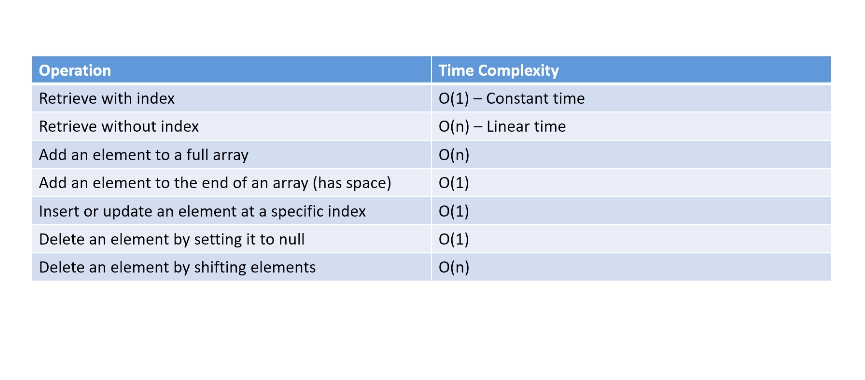
Why are arrays zero based, we would have to always subtract one from the calculations.

If we know the index of element we want we can get to it very quickly (O(1))





We have constant time complexity, O(1). It doesn’t matter how many elements we have in the array.



If an array is full, we have to create brand new array and then copy the integers over. It would take linear time (n). Creating new array is O(1), adding one element ad the end is O(1), copying the entire array takes O(n) because we have to go through all the points in previous araray.

If we delete and set it to null, its O(1) we can jump right to the element if we know index and change it to null. If we were not to know the index we would have O(n) because we would have to find the element first if we were to use brute force method in an unnordered array.

If we want to delete element and we need to shift all the other elements down, it would be O(n). Worst case we want to delete the first element and push each and every of the others elements down. So its O(n).