Best, Worst and Average Case Analysis:

A program finds it best when it is effortless for it to function and worse otherwise.

By considering a search algorithm used to perform a sorted array search, we will analyze this feature.

Analysis of a Search Algorithm:

Consider an array that is sorted in increasing order

1 7 18 28 50 180

We have to search a given number in this array and report whether it's present in the array or not. In this case, we have two algorithms, and we will be interested in analyzing their performance separately.

Algorithm 1 – Start from the first element until an element greater than or equal to the number to be searched is found.

Algorithm 2 – Check whether the first or the last element is equal to the number. If not, find the number between these two elements (center of the array); if the center element is greater than the number to be searched, repeat the process for the first half else, repeat for the second half until the number is found. And this way, keep dividing your search space, making it faster to search.

Analyzing Algorithm 1: (Linear Search)

We might get lucky enough to find our element to be the first element of the array. Therefore, we only made one comparison which is obviously constant for any size of the array.

- Best case complexity = O(1)

If we are not that fortunate, the element we are searching for might be the last one. Therefore, our program made 'n' comparisons.

- Worst-case complexity = O(n)

For calculating the average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases. Here, we found it to be just O(n). (Sometimes, calculation of average -case time gets very complicated.)

Analyzing Algorithm 2: (Binary Search)

If we get really lucky, the first element will be the only element that gets compared. Hence, a constant time.

- Best case complexity = O(1)

If we get unlucky, we will have to keep dividing the array into halves until we get a single element. (that's the array gets finished)

Hence the time taken: n + n/2 +n/4 + ... +1 = log n with base 2

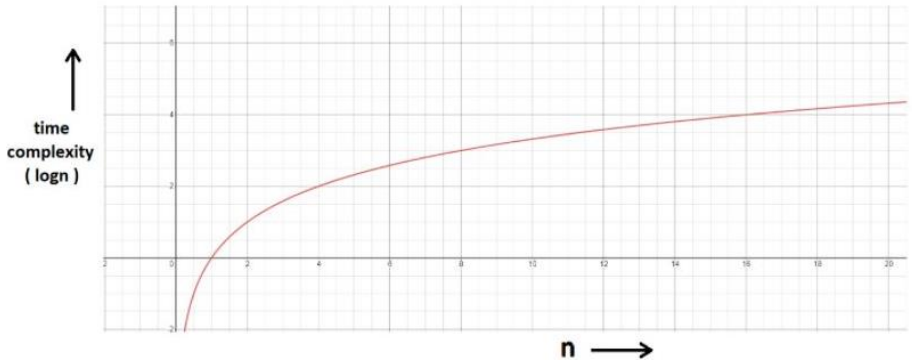
- Worst-case complexity = O(log n)

N.B: log n refers to how many times needed to divide n units until they can no longer be divided (into halves).

log 8 = 3 ⇒ 8/2 + 4/2 + 2/2 → Can't break anymore.

log 4 = 2 ⇒ 4/2 + 2/2 → Can't break anymore.

We can see in the graph, how slowly the time complexity (Y- axis) increases when we increase the input n (X - axis).

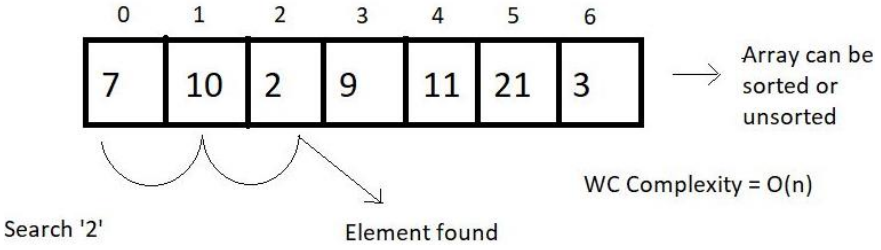


Why we can't calculate complexity in seconds when dealing with time complexities:

a. Not everyone's computer is equally powerful. So, we avoid handling absolute time taken.
b. We just measure how time (runtime) grows with input in the asymptotic analysis.

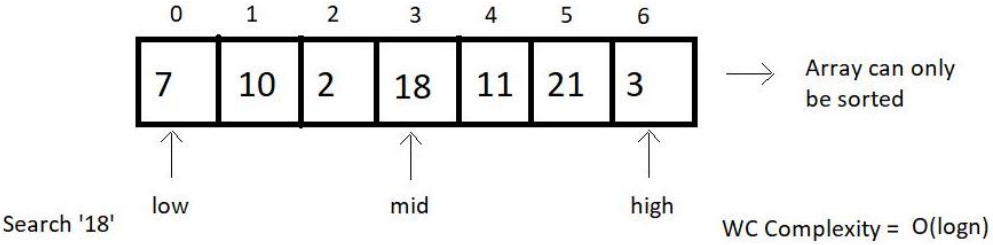
Linear Search and Binary Search:

Linear Search method searches for an element by visiting all the elements sequentially until the element is found or the array finishes. It follows the array traversal method.



Binary Search method searches for an element by breaking the search space into half each time it finds the wrong element.

This method is limited to a sorted array. The search continues towards either side of the mid, based on whether the element to be searched is lesser or greater than the mid element of the current search space.



Notes Made by

SOIKOT SHAHRIAR

[t.me/soikot_shahriaar]

[github.com/soikot-shahriaar]