**Search Algorithms**

**Task = Finding index of 9, if not found return -1**

**1. Linear Search**

Linear search is where an array is parsed sequentially. Imagine a for loop, where we go over the array, performing some checks on every element, to find a particular element of interest. The search typically starts from 0th element and continues until the element of interest is found or else end of list is reached.

Hence, if you are lucky, the element to be found is at the start of the list and the search ends in fewer iterations. But if element is towards the end of the list, or not at all in the list, more iterations will be needed. Consider, the following scenarios.

Input = [9, 2, 1, 3, 6, 5, 4, 8, 7], iterations needed = 1

Input = [1, 2, 7, 3, 6, 5, 9, 8, 4], iterations needed = 7

Input = [1, 2, 7, 3, 6, 5, 4, 8, 9], iterations needed = 9

Input = [1, 2, 7, 3, 6, 5, 4, 8, 10], iterations needed = 9

Best execution time 1, worst execution time N, where N is the total number of items in the array.

**That makes the time complexity is** O(N)

Simplest pythonic solution

for idx, src in enumerate(nums):

if src == target:

return idx

return -1

Tube light moment for greenhorns:

Using fancy looking methods like using “in” keyword in python, may make your code look smarter but it does not affect time complexity. As in the background python will still use a for loop in the background and parse every element.

        if target in nums:

            return nums.index(target)

        else:

            return -1

This method may be the viable when very simple logic is to be implemented and when using additional memory (creating additional variables) is not allowed.

This method will become inefficient and time consuming when going through larger array. Hence when elements in the array are sorted, or can be sorted first, it is good idea to use Binary Search instead.

**Task = Finding index of 7, if not found return -1**

**2. Binary Search**

Binary search is a **Divide and Conquer** algorithm. Like any divide-and-conquer algorithm, binary search first divides a large **sorted** array into two smaller subarrays. Then recursively (or iteratively) it discards one subarray and continues on the second subarray. This decision of discarding one subarray is made in just one comparison.

So binary search reduces the search space to half at each step. Search space is the possible subarray where our element of interest may exist. Therefore, the time complexity of the binary search algorithm is O(log2n), which is very efficient. The auxiliary space required by the program is O(1) for iterative implementation and O(log2n) for recursive implementation due to call stack.

Let’s track the search space by using two indexes – start and end. Initially, start = 0 and end = n-1 (as initially, the whole array is our search space). At each step, find the mid-value in the search space, so now we have mid-value, left hand search space, right hand search space.

After which there are three possible cases:

1. If target = nums[mid], return mid.
2. If target < nums[mid], discard all elements in the right search space, including the middle element, i.e., . Now our new high would be mid-1.
3. If target > nums[mid], discard all elements in the left search space, including the middle element, i.e., . Now our new low would be mid+1.

Input = [2, 3, 5, 7, 8, 10, 12, 15, 18, 20]

start = 0, end = 9, mid = (start + end) // 2 = 4, input[mid] = 8

target < input[mid] – condition 2 satisfied

start = 0, end = 3 (mid-1), mid = (start + end) // 2 = 1, input[mid] = 3

target > input[mid] – condition 3 satisfied

start = 2 (mid+1), end = 3, mid = (start + end) // 2 = 2, input[mid] = 5

target > input[mid] – condition 3 satisfied

start = 3 (mid+1), end = 3, mid = (start + end) // 2 = 3, input[mid] = 7

target == input[mid] – condition 1 satisfied