# Lesson:


# Searching in an array- Linear & Binary Search

# Pre Requisites:

- Basic java syntax
- Loops in JAVA

# List of concepts involved :

- Linear search
- Binary search

# Linear search in 1D array :

To search for an element in the array we need to traverse over the array. In linear search we continuously keep on traversing the array linearly i.e one by one every element until the required condition is satisfied.

Here follows an example of linear search to find a particular element X in a given array.

**Q1. Given an array and an integer 'x'. Return true if 'x' is present in the array otherwise false.**
Input 1: A = [3 , 4 , 2 , 1 , 8 , 9] , x = 4
Output 1: true
Input 2: A = [3 , 4 , 2 , 1 , 8 , 9] , x = 5
Output 2: false

LP_CODE1_LS.java

Output:

```
enter the number of elements you want : 5
enter the elements :
2 3 6 55 1
enter the target: 6
True
```

**Approach:**

- We linearly traverse over the array using a for loop and as soon as any element that we are currently on, is equal to the target variable 'target' we return a true.
- If even after complete traversal we are unable to find the element, we will return false.
- We have taken a "flag" variable just to distinguish that if we have already printed a true we need not to print false again.

**Time complexity :** Since in the worst case we are traversing the whole array therefore the time complexity will be O(size of the array).

**Space complexity :** For linear search we are not using any extra space therefore we can say that space complexity will be constant or O(1).

**Advantages of Linear Search:**

- Linear search is simple to implement and easy to understand algorithms.
- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.

- It is a well suited algorithm for small datasets.

**Disadvantages of Linear Search:**

- Linear search has a time complexity of O(size of the dataset or array), which in turn makes it slow for large datasets.

**Binary search algorithm**:

- This algorithm is designed to search an element in a sorted array with better efficiency than linear search. Secondly, it is not only confined to sorted arrays but can be applied to any monotonic range of data where during search, in every iteration some portion of the dataset can always be neglected.

**Advantages of Binary Search**

It is more efficient for large size data but when we have a dataset arranged in monotonic order.

**Disadvantages of Binary Search**

It works just on records that are arranged and kept arranged. That isn't generally feasible, particularly assuming components are continually being added to the rundown.

**Steps involved in binary search algorithm:**

- The main concept of this algorithm is to contract the search region by half in every iteration.
- This is done by comparing the middle most value of the array by the given target. If the elements match we can simply return a true.
- Other than this case there can be 2 more possibilities. One is the target element is greater than the middle element. Now since the array is sorted, all elements before the middle element are less than the middle element so none of them can ever match the target.
- But we still have a chance in the right region from the middle element to the last element where we can search for the target.
- This way we will strictly ignore the elements from the starting index to the middle one while we continue our search from the middle element to the last element.
- The second case can be just opposite if the target element is found to be smaller than the middle element.
- So we will ignore all the elements from the middle element to the last element because they all will be definitely greater than the middle element and thus greater than the target element. Now the search space will be from the first element till the middle element.
- This way in each iteration we keep on reducing our space of search. If at any point we find a match with the target we will return true otherwise at last when no element is left in our search space we will return false.

**Example - 1:** Search if the given value is present in the array or not.

Array = [1, 5 , 6 , 9, 15, 18, 19, 23], Value = 15

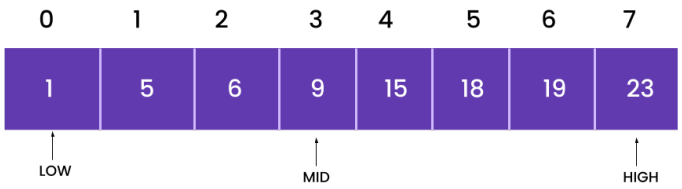
In the above example, we need to search for the value 15 if it is present. A simpler way could be to use linear search directly giving us a time complexity of O(n). But since we are given that the array is sorted, we can use binary search here.

Array-

| 1 | 5 | 6 | 9 | 15 | 18 | 19 | 23 |
|---|---|---|---|---|---|---|---|

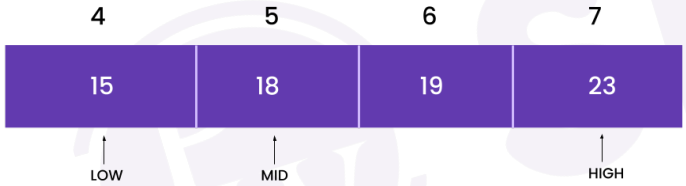
**Step 1:** For the given array, the middle element is 9. The method to calculate the middle element is (low+high)/2 but incase we know (low+high) is going to overflow, then we can use low+ (high-low)/2 if, where low is a pointer that points to the lowest index of the search space and high is a pointer that points to the highest index of the search space.

Here low = 0 and high = 7
Therefore mid = (0 + 7)/2 = 3




As we can see 9 is not equal to 15. So we will continue our search. 9 < 15


=>our answer lies in the latter half {right portion} of the array.
With the change in search space will change our low and mid pointers.
High pointer will not change.
**Step 2:** In the second step, we will consider only the remaining half of the array.




mid = (low + high)/2 = (4 + 7)/2 = 5
For this half the middle element is 18.
As we can see 18 is not equal to 15. So we will continue our search.
18 > 15
=> our answer lies in the first half{left half} of the array.
This time our high and mid pointers will be changed whereas low pointer will remain intact.


**Step 3:** We will again consider only the remaining part of the array.


In this case our middle , low and high pointers will be on the same index and on comparing the middle element with the target we found our match.
15 = 15.
=> we have found the value that we were looking for.

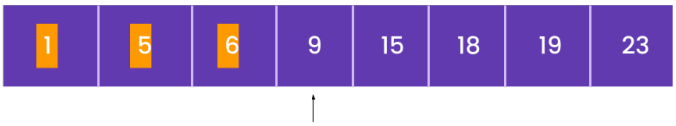
**Example-2:** In this let's consider the same array as before but this time we will search for the value 7 in the array.

**Example - 2:** In this let's consider the same array as before but this time we will search for the value 7 in the array.

We can see that 7 doesn't exist in our array. Let's see how our algorithm comes to the same conclusion.

Array-


**Step 1:** For the given array, the middle element is 9.




As we can see, 9 is not equal to 7. So we will continue our search.
9 > 7
=>our answer lies in the first half of the array.
**Step 2:** In the second step, we will consider only the remaining half of the array.


For this half the middle element is 5.
As we can see, 5 is not equal to 7. So we will continue our search.
5 < 7
=>our answer lies in the second half of the array.
**Step 3:** We will again consider only the remaining part of the array.


In this case we our middle element will be 6 i.e. the only element in the array.
As we can see, 6 is not equal to 7. So we will continue our search.
6 < 7
=>our answer lies in the second half of the array.
=>but here we can see that there is no second half of the array left to explore.
=>7 doesn't exist in the array.
We've now understood how the algorithm works. So let's see its code too.

**LP_CODE2_BS.java**

Output:

```
enter the number of elements you want : 6
enter the elements :
2 3 4 1 6 7
enter the target: 6
True
```

# Time complexity analysis of binary search algorithm:

At every step we reduce the interval ranging from low to high to half its size. Initial size of the search space is N where N represents the length of the given array and every step, that search space is reduced by half.

**N -> N/2 -> N/4 -> ... -> N/(2^k)**
Here k represents the total number of iterations.
N/2^k = 1 [because we will stop when we have just one element in our search space.]
N = 2^k
Taking logarithm both sides with base 2:
$\log_2 N = k$
Hence we can say that total iterations required will be $\log_2 N$ in the worst case.
Thus, giving us a time complexity of $O(\log_2 N)$.
$O(\log_2 N)$ is a better time complexity than O(n) i.e. the time complexity for linear search. Hence we can see why it is preferred over linear search.

# Space Complexity Analysis

In the iterative version of the binary search, we are generating only 3 variables- low, high and mid. Therefore the space complexity of binary search is O(1).

# Interview Problem: Lower Bound/First occurrence in an Array

Lower bound is also known as the first occurrence of an element in an array. When the element is present single or in duplicates, the index of its first occurrence is known as Lower Bound.

**Q2. Given a sorted array containing duplicates and an integer 'target'. Find the first occurrence of the target in the array. If target does not exist return -1.**
Input 1: arr = [1 1 1 2 2 3 4 5 5 5] , target = 5
Output 1: 7
Input 2: arr = [1 2 2 5 5 5 6 8 8 8 8] , target = 7
Output 2: -1

LP_CODE3_BS.java

Output:

```
enter the number of elements you want : 6
enter the elements :
1 2 3 3 4 4
enter the target: 3
The first occurrence of given target is at 2
```

**Approach:**

- Given that the array is sorted, to search the given element we can use Binary Search Algorithm.
- If we are able to find any occurrence of the target in the array, we reduce our search space to the first half of the array because we want the minimum index that has the value equal to target.
- But there is an equal chance that there may not be any occurrence of the target in the first half, that's why we recorded the current index.
- Because this will be serving as an answer if nothing better works out.

# Interview Problem 2: Square root of a number

**Q1. Given a positive number, return the square root of it. If the number is not a perfect square, return the floor of its square root.**
**Floor of a number: floor(4.15) = 4, floor(5.98) = 5**

For example,
Input: x = 12
Output: 3
Input: x = 16
Output: 4

LP_CODE4_BS.java

Output :

```
enter the number whose square root you want : 19
The square root of the given number is : 4
```

**Approach:**

- We know that the square of the number will always be less than it, so we chose the range from 0 to the given number.
- Here you can observe one thing that we do not have any array over which we are planning to apply binary search rather we have a monotonic search region.
- So it is not always mandatory to apply binary search on sorted arrays but any monotonic region of search will work.
- Now we have found the 'mid' element, this can be a probable candidate for square root to we checked if 'mid*mid' equals the given number. If yes, then we found the square root.
- Second possibility could be mid*mid < x, if this is the case that means we need to increase the mid value, so we will move to the right half / latter half of the array. But since there is no guarantee of finding a perfect square root, we need to store the currently obtained mid value as our answer until we get something more precise and accurate than this value.
- The last case would be if mid*mid > x, in that case we need to reduce the value of mid. So we lower the 'high' pointer.
- We will terminate once the value of low will be equal to or greater than high. That indicates we have traversed over all the possibilities.

# Next Class Teaser

- Sorting in array
- Bubble Sort
- Insertion Sort
- Selection Sort