Binary Search

What is Searching?

- Searching is a method to **find some relevant information in a data set**.
- In computer programming, searching is usually referred to as finding a particular element in a data structure.
- Two things very important to be properly defined in a searching problem :
    - **Target :** What are you trying to find ?
    - **Search Space :** Where exactly do you need to search ?

Types of Searching Algorithms

- *If you're searching for a product on Amazon, and it takes minutes for the search result to pop up, you probably won't be a very satisfied customer. That is why efficient searching is one of the most important components of user experience.*

- Hence, it is important to understand different searching algorithms and why it is important to use one over other. Below are some of the most commonly used ones.
    - Linear Search
    - Binary Search

Linear Search

- In Linear search, we traverse each element of the array, one by one, and check whether it is equal to the element to be searched. It is also called sequential search because it checks all the elements sequentially.

Time Complexity of Linear Search Algorithm

- In the best case, we might find the element in the first iteration only, i.e. it is the first element of the array.
- In the worst case, we might find the element in the last iteration or not find K in the array at all.
- On average, it might be present somewhere around the middle of the array.

- So, we can probably conclude that the average case time complexity of the linear search is O(n/2) ~ **O(n)**, where n is the number of elements in the array. Hence, the time complexity of the linear search is in fact, linear.

Space Complexity

- As it is evident, we are not using any significant extra memory in linear search. So the **space complexity is constant, i.e. O(1).**

Applications of Linear Search Algorithm

- Linear search is the most straightforward and easiest algorithm to implement and understand. It is particularly useful if the data set is small, randomized and there are memory constraints, since it always uses constant memory.
- If multiple elements need to be searched in a data set, linear search is not the most efficient approach as it will take linear time in each case on average.

Binary search

- Binary search is an efficient algorithm for searching for a particular data in a given data set. It works by repeatedly dividing in half the portion of the search space that could contain the item, until you've narrowed down the possible locations to just one.

Idea of Binary Search

- Let's try to understand binary search using a basic example of a dictionary. Let us say we want to search the word "dog" in the dictionary. A dictionary always contains words in alphabetical order.
- Let's say we randomly open the dictionary and we find the word "goat". Now think, will the "dog" be present before "goat" or after it. Definitely before it. Can we reject the complete right part of the dictionary without even looking into it ? Yes.

- A B C D E F Goat H I J K L M N O P Q R S T U V W X Y Z

- Now our search space has reduced. We'll again randomly open any page and let's say that we land on "cat" , now can we say that definitely "dog" won't be present on the left side so we reject the complete left and search on the right part.

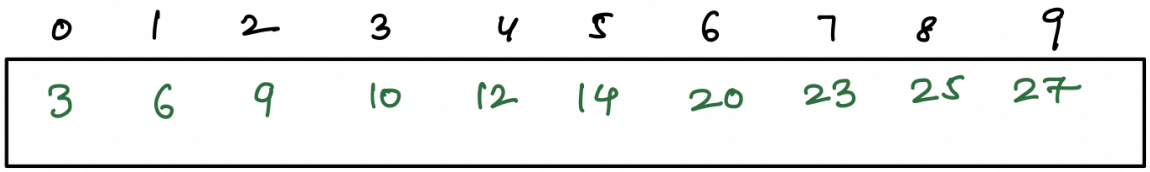
- A B Cat D E F

- Now our search space has again reduced. We'll again randomly open any page and let's say that we land on "dog". Our search is complete.

- *The idea is to keep on rejecting one half of the search space every time based on some conditions and find the answer in a much faster time.*

Problem

Given a sorted array of size N. Find the index of element k. If it doesn't exist, return -1.

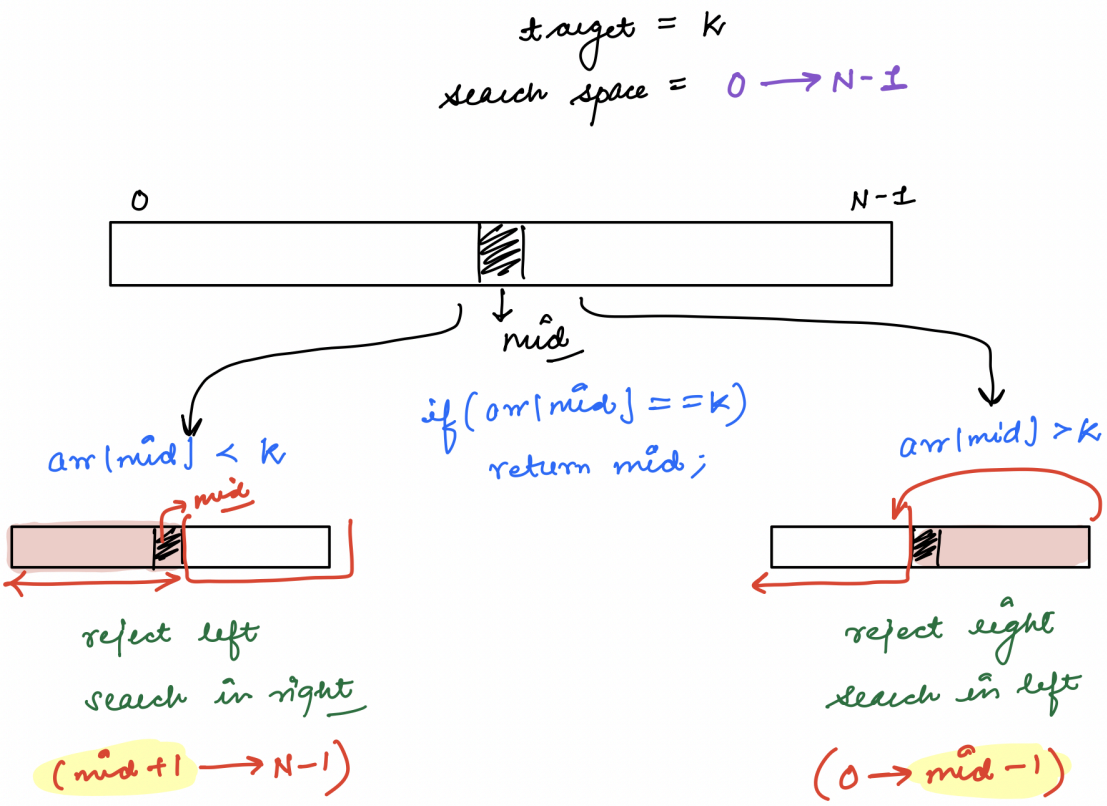


In the above example data set -

- What is the index if the value of K is equal to 12 ?
    - Index = 4
- What is the index if the value of K is equal to 5 ?
    - Index = -1

Solution :

1. **Liner Search** : We can traverse the array and look for the index of given element k.
    a. TC: O(N) , SC: O(1)
2. **Binary Search** : Can we use a similar idea like we used in the dictionary ? Instead of randomly going to any element, we'll always go to the middle of the existing search space.
    a. Let's define the search space first. Our search space is the complete array from 0th index to N - 1 index. Let's take it as l = 0 and r = N - 1
    b. We go to middle of the search space , mid = (l + r) / 2
    c. Following are the possible conditions when you are checking the middle element:
        i. Arr[mid] == k : In this case search is complete
        ii. Arr[mid] < k : the target that we are searching for is greater than the current element, since the array is sorted, the target won't be present on left side , so reject left and search in right
        iii. Arr[mid] > k : the target that we are searching for is smaller than the current element, since the array is sorted, the target won't be present on the right side , so reject right and search on left.



d. We can keep on executing this on repeat until either you find the target or the search space is exhausted (l > r)
e. Time complexity of Binary search : O(log2(search space size))

Q. AGGRESSIVE COWS (Binary Search on Answer)

Given an array representing the positions of the stalls and N cows. Cows are aggressive towards each other so the farmer wants the cows to be maximize the minimum distance between two cows, find maximum possible minimum distance if the farmar has N cows.

Example

Input: arr[] = **{1, 2, 5, 8, 10}**
N = **3**

Output:
4

Explanation:

- We place the first cow at position 1.
- The second cow at position 5
- The thirst cow at position 10
- So distance between first and second cow is 4, and the second and third cow is 5. So the maximum possible minimum distance is **4**.

Solution

Brute Force Solution:

A brute force approach would involve trying all possible combinations of stall positions for placing the cows and calculating the minimum distance for each combination. This would involve iterating through all possible subsets of stalls and calculating the minimum distance. However, the time complexity of this approach would be exponential and impractical for large inputs.

Binary Search Solution:
The binary search solution for the Aggressive Cows problem aims to find the maximum minimum distance that allows a given number of cows to be placed in sorted stalls.

We start with a range of possible distances, from 0 (minimum possible distance) to the maximum distance between the first and last stall. Using binary search, we repeatedly check if it is possible to place the cows with a certain minimum distance.

For each iteration, we calculate the middle value within the range and check if it is feasible to place the cows with that minimum distance. If it is possible, we update the result and adjust the range to look for larger minimum distances. If it is not possible, we adjust the range to look for smaller minimum distances.

```
def canPlaceCows(stalls, cows, min_distance):
      count = 1
      prev_pos = stalls[0]

      for i in range(1, len(stalls)):
            if stalls[i] - prev_pos >= min_distance:
                  count += 1
                  prev_pos = stalls[i]
            if count == cows:
                  return True
      return False
```

```
def maxMinDistance(stalls, cows):
      stalls.sort() # Sort the stalls in ascending order
      start = 0
      end = stalls[-1] - stalls[0]
      result = 0

      while start <= end:
            mid = start + (end - start) // 2
            if canPlaceCows(stalls, cows, mid):
                  result = mid start = mid + 1
            else:
                  end = mid - 1
      return result
```

Revision Videos -

https://drive.google.com/drive/folders/15M4y3UGOg1HkBIZ0NRQ9Nai8We7ltKoA?usp=share_link

Two Pointers -
https://drive.google.com/file/d/1K79LGCQLWLSj6mxBO-wwYyRbSli8Mmso/view?usp=share_link