Revision Note - Binary Search

Binary search is an efficient algorithm for finding a specific element in a sorted array or list by repeatedly dividing the search space in half. It follows a divide-and-conquer approach to quickly locate the target element.

- Efficiency: Binary search has a time complexity of O(log n), making it much faster than linear search, especially for large datasets.
- Suitable for sorted data: Binary search requires the input data to be sorted, but it provides a significant advantage in terms of search time for sorted arrays or lists.
- Midpoint comparison: It compares the target element with the midpoint of the search space and eliminates the half of the search space that does not contain the target.

Pseudo Code

```
binarySearch(arr, target):
    low = 0
    high = length(arr) - 1

    while low <= high:
        mid = (low + high) / 2

        if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1  // Element not found
```

Time and Space Complexity:

Worst Case Time Complexity: O(log n) when the target element is not present or is located at the extreme ends of the array.
Average Case: O(log n) when the elements are uniformly distributed.

Binary search has a **space complexity** of **O(1)** as it only requires a few extra variables for storing indices and temporary values. It does not require additional memory proportional to the input size.

Question 1

Given a sorted array of distinct integers, find the index of a given targer.

Example Input:
Arr = **[2, 4, 5, 7, 11, 14, 16, 20]**
Target = **11**

Example Output:
4

Explanation: The target **11** is present at index **4**.

Solution

Brute Force: Linearly iterate through the array and when Arr[i] equals the target value, return the index.
TC: O(n)

Optimal Solution: Given a the fact the array is sorted, we can apply binary search.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2	4	5	7	11	14	16	20

Iteration Count	**Left Limit Index**	**Right Limit Index**	**Mid Index**	**Arr[Mid Index] == target**	**Next iteration half**
1	0	7	3 (Arr[Mid] = 3)	No	Arr[mid]<target, right half
2	4	7	5 (Arr[Mid] = 14)	No	Arr[mid]>target, left
3	4	4	**4 Arr[mid] = 11**	Yes	**Value Found!**

Pseudo Code:

```
function binarySearch(arr, target):
    low = 0  // Set the lower bound of the search range
    high = length(arr) - 1  // Set the upper bound of the search range

    while low <= high:
        mid = (low + high) / 2  // Calculate the middle index of the searc

        if arr[mid] == target:  // If the middle element is the target, re
            return mid
        else if arr[mid] < target:  // If the middle element is smaller th
            low = mid + 1
        else:  // If the middle element is greater than the target, update
            high = mid - 1

    return -1  // Element not found
```

Concept of Binary Search on Answer

The concept behind binary search on answers is to define a feasible range for the answer and iteratively narrow down this range until we find the desired answer. This technique is particularly useful when we can determine if a solution with a certain value is feasible or not.

Question 2

Problem Statement

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.

Pseudo Code:

```python
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
```

Problem 3

Problem Statment

Q. Find index of the given element in rotated sorted array of distinct elements. (No info on count of rotations)

Example

Input:
arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3}
key = 3

Output:
Index of key 3 in the array: 8

Explanation: The position of the value 3 in the rotated sorted array is at 8. Used 0-based indexing.

Solution

Brute Force Approach:

- Iterate through the array and check each element one by one.
- If the element matches the key, return its index.
- If the end of the array is reached and the key is not found, return -1 to indicate that the key is not present in the array.

Binary Search Approach (Intuition):

- Find the pivot index (the index where the rotation occurred) using a modified binary search.
- Divide the array into two parts at the pivot index.
- Determine in which part the key might be located based on the comparison of the key with the first element of each part.
- Perform a binary search on the selected part of the array to find the index of the key.

Pseudo Code:

```
function pivotedBinarySearch(arr, n, key):
    // Find the pivot index
    pivot = findPivot(arr, 0, n - 1)

    // If no pivot found, perform regular binary search
    if pivot == -1:
        return binarySearch(arr, 0, n - 1, key)

    // Check if the pivot element is the key
    if arr[pivot] == key:
        return pivot

    // Determine which side of the pivot the key lies and perform binary s
    if arr[0] <= key:
        return binarySearch(arr, 0, pivot - 1, key)
    return binarySearch(arr, pivot + 1, n - 1, key)

function findPivot(arr, low, high):
    // Base cases for recursive function
    if high < low:
        return -1
    if high == low:
        return low

    // Calculate the middle index
    mid = (low + high) / 2

    // Check if the mid element is the pivot
    if mid < high and arr[mid] > arr[mid + 1]:
        return mid
    if mid > low and arr[mid] < arr[mid - 1]:
        return (mid - 1)

    // Recursively search for the pivot on the appropriate side
    if arr[low] >= arr[mid]:
        return findPivot(arr, low, mid - 1)
    return findPivot(arr, mid + 1, high)

function binarySearch(arr, low, high, key):
    // Base case for recursive function
    if high < low:
        return -1

    // Calculate the middle index
    mid = (low + high) / 2

    // Check if the key is found at the middle index
    if key == arr[mid]:
        return mid

    // Recursively search for the key on the appropriate side
    if key > arr[mid]:
        return binarySearch(arr, mid + 1, high, key)
```

```
    return binarySearch(arr, low, mid - 1, key)
```

- **Time Complexity:** O(log n) Binary Search requires log n comparisons to find the element.
- **Space Complexity:** O(1).

Summary of Binary Search:

- Binary search is an efficient search algorithm for sorted arrays.
- It follows a divide-and-conquer approach to narrow down the search space.
- It compares the target element with the middle element to determine the next search space.
- The search space is divided in half at each step until the target element is found or the search space is exhausted.
- Binary search on answer is a technique where the binary search is applied to search for an optimal or desired solution by checking the validity of a solution based on a condition or property.