1.9 BINARY SEARCH

Question:

Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.

AIM:

To design an algorithm that checks if a given number x exists in a sorted array using binary search and analyze its performance.

ALGORITHM:

1. Start with two pointers:

   low = 0, high = len(arr)-1
2. Find the middle index:

   mid = (low + high) // 2
3. If arr[mid] == key, element is found → return index.
4. If arr[mid] < key, search in the right half (low = mid+1).
5. If arr[mid] > key, search in the left half (high = mid-1).
6. Repeat until low > high.
7. If not found, return -1.

PROGRAM:

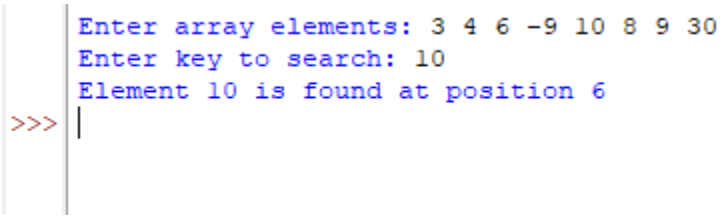
```
def binary_search():
    arr = sorted(list(map(int, input("Enter array elements: ").split())))
    key = int(input("Enter key to search: "))
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == key:
            print(f"Element {key} is found at position {mid}")
            return
        elif arr[mid] < key:
            left = mid + 1
        else:
            right = mid - 1
    print(f"Element {key} is not found")

binary_search()
```

Input:

Array: [3, 4, 6, -9, 10, 8, 9, 30]

Key: 10

Output:



RESULT:

Thus the program is successfully executed, and the output is verified.

PERFORMANCE ANALYSIS:

Time Complexity

1. Best Case: Key found at the middle on first try → O(1)

2. Worst Case: Array repeatedly divided in half until one element left →

Number of comparisons = $\log_2(n)$ → O(log n)

3. Average Case:

Same as worst case since search space halves each step → O(log n)

Space Complexity:

- Iterative version uses O(1) extra space.
- Recursive version would use O(log n) stack space.