EXPERIMENT NO.01

Roll No.: 24141001
Batch:I1

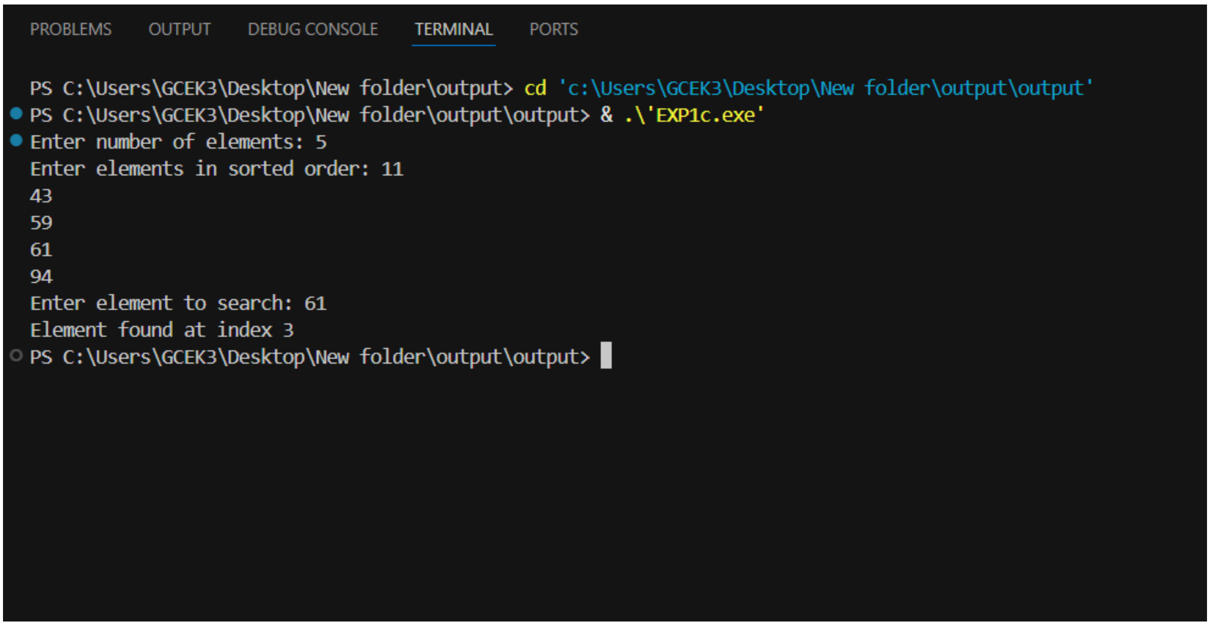
Title: Binary search techniques using array and recursion. Analyse time and space complexity.

1) **Binary search using Array.**

```
 #include <stdio.h>
int binarySearch(int arr[], int n, int key) {
   int low = 0, high = n - 1;

   while (low <= high) {
      int mid = (low + high) / 2;

      if (arr[mid] == key)
         return mid;

      else if (arr[mid] < key)
         low = mid + 1;

      else
         high = mid - 1;
   }

   return -1;
}
int main() {
   int n, key, result;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter elements in sorted order: ");
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("Enter element to search: ");
   scanf("%d", &key);
   result = binarySearch(arr, n, key);
   if (result == -1)
      printf("Element not found\n");
   else
      printf("Element found at index %d\n", result);
```

```
    return 0;
}
```

OUTPUT



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\GCEK3\Desktop\New folder\output> cd 'c:\Users\GCEK3\Desktop\New folder\output\output'
PS C:\Users\GCEK3\Desktop\New folder\output\output> & .\'EXP1c.exe'
Enter number of elements: 5
Enter elements in sorted order: 11
43
59
61
94
Enter element to search: 61
Element found at index 3
PS C:\Users\GCEK3\Desktop\New folder\output\output>
```

Time complexity & space complexity:

Time Complexity

Best Case: O(1)
When the middle element is the key (found in the first comparison).

Worst Case / Average Case: O(log n)
Because each step halves the search space. For n elements, maximum comparisons are about $\log_2(n)$.

Space Complexity

O(1)
Only a few extra variables (low, high, mid, key, etc.) are used, no additional data structures.

Applications & Limitations:

Applications of Binary Search

1. Searching in sorted arrays/lists
   Quickly find elements in a sorted dataset (e.g., roll numbers, IDs).

2. Databases and Libraries
   Used to index and quickly retrieve records.

3. Gaming / Leaderboards
   To locate player scores or ranks efficiently.

4. Dictionary / Spell Checkers
   To check if a word exists in a sorted word list.

5. Range searching / Lower and Upper Bound
   Used in competitive programming and algorithms like lower_bound in C++ STL.

Limitations of Binary Search

1. Array must be sorted
   Cannot work on unsorted data; sorting adds extra overhead (O(n log n)).

2. Random Access Required

   Works efficiently only on data structures that allow direct access (like arrays, not linked lists).

3. Static Data
   Not suitable if elements are frequently inserted or deleted (re-sorting is required).

4. Not suitable for very small datasets
   Linear search can sometimes be faster for tiny arrays due to less overhead.

2) Binary search using recursion:

```
#include <stdio.h>
int binarySearch(int arr[], int low, int high, int key) {
if (low > high)
```

```
        return -1; // Element not found

    int mid = (low + high) / 2;

    if (arr[mid] == key)
        return mid; // Element found
    else if (arr[mid] < key)
        return binarySearch(arr, mid + 1, high, key);
 else
        return binarySearch(arr, low, mid - 1, key);  }

int main() {
    int n, key, result;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter elements in sorted order: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter element to search: ");
    scanf("%d", &key);

    result = binarySearch(arr, 0, n - 1, key);

    if (result == -1)
        printf("Element not found\n");
    else
        printf("Element found at index %d\n", result);

    return 0;
}
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

Enter number of elements: 5
Enter elements in sorted order: 12
34
56
87
99
Enter element to search: 56
Element found at index 2
PS C:\Users\GCEK3\Desktop\New folder\output\output>
```

Time complexity & space complexity:

Time Complexity

Best Case: O(1)
→ When the middle element is the key (found in first call).

Worst Case / Average Case: O(log n)
→ Each recursive call halves the search range, so total calls are about $\log_2(n)$.

Space Complexity

O(log n)
→ Each recursive call adds a new function frame to the call stack.
→ For n elements, maximum recursive depth = $\log_2(n)$.

Applications & Limitations:

Applications

1. Searching in sorted arrays – Fast search in roll numbers, IDs, etc.

2. Databases/Indexing – Used in sorted datasets for quick retrieval.

3. Gaming/Leaderboards – Searching scores or rankings efficiently.

4. Dictionary/Spell Checkers – Check word existence in sorted lists.

5. Range Queries (lower/upper bound) – Common in competitive programming.

Limitations

1. Extra Space Usage

Each recursive call uses stack memory, so space complexity is O(log n).

2. Risk of Stack Overflow

For very large arrays, deep recursion may cause stack overflow.

3. Same Sorting Requirement

Array must be sorted; otherwise, it won't work.

4. Slightly Slower than Iterative

Function calls add overhead compared to iterative binary search.

Conclusion:

From this experiment I understand the time and space complexity of binary search using recursion and array also there applications and limitations.