**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Experiment No. 2**

**Title:**

Assignment Based on Divide and Conquer Strategy. (Implement Recursive and Non-Recursive Binary Search Algorithm using C++ or java. Determine Time and space complexity).

|  |  |  |  |
| --- | --- | --- | --- |
| Title of Experiment | CO Mapping | CO Statements | PO Mapping |
| Implement Recursive and Non-Recursive Binary Search Algorithm using C++ or java. Determine Time and space complexity | CO1, CO2, CO3 | Co1: To formulate computational problems in abstract and mathematically precise manner.  CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm.  CO3:To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO1, PO2, PO3, PO4, PO5, PO11 |

**Theory:**

**1. Introduction**

In this lab exercise, you will learn how to implement program to manage a integer numbers. The program will store this information in sorted order, and it will allow you to search number using binary search (both recursive and non-recursive methods).

**2. Theory**

**a. Binary Search**

Binary search is an efficient algorithm used to search for an element in a sorted list or array. It works by repeatedly dividing the search interval in half.

**Algorithm:**

1. Compare the target value with the middle element.
2. If the target matches the middle element, the search is successful.
3. If the target is less than the middle element, continue the search on the left half of the list.
4. If the target is greater than the middle element, continue the search on the right half of the list.
5. Repeat the process until the element is found or the search interval becomes empty.

**Time Complexity:**

**Best-case time complexity:**

* The best-case scenario occurs when the target element is the middle element of the array.
* In this case, the algorithm will find the element on the first iteration itself.
* Thus, the time complexity for the best case is O(1).

**Worst-case and Average-case time complexity**:

* In the worst-case scenario, the algorithm will continue splitting the array in half until the subarray is reduced to a single element.
* The number of iterations is proportional to the logarithm of the number of elements in the array because the array is halved with each iteration.
* Therefore, the time complexity for the worst and average cases is O(log n), where n is the number of elements in the array.

**Space Complexity of Binary Search**

The space complexity depends on the approach we use:

**Iterative Binary Search:**

* The iterative version of binary search does not use additional memory for recursion calls.
* The space complexity is O(1) because only a few variables (low, high, mid) are used to store the indices.

**Recursive Binary Search:**

* In the recursive version, each recursive call adds a new frame to the call stack.
* Since there are log n recursive calls (in the worst case), the space complexity is O(log n) due to the recursion stack.

**3. Lab Exercise**

**a. Program Requirements**

Your program should fulfill the following requirements:

1. Create an array of integers.
2. Implement a function to insert numbers into array.
3. Implement a function to search for number using binary search (both recursive and non-recursive methods).

**b. Step-by-Step Implementation**

Follow these steps to implement the program:

1. Create an empty array.
2. Implement a function to insert number into the array. Ensure that the array remains sorted.
3. Implement a recursive binary search function to search for a friend's mobile number.
4. Implement a non-recursive binary search function to achieve the same result.
5. Test the program with various scenarios.

**c. Testing the Program**

Test your program with various test cases to ensure it works correctly. Make sure to test:

* Inserting new element.
* Searching for existing element.
* Searching for non-existing element.

**Algorithm:**

**Data Structures:**

array to store numbers .

Functions:

**1. add\_Element(number):**

1. Create a new entry in the array.

2. Ensure the array remains sorted.

**2**. **recursive\_binary\_search(number):**

1. Initialize low = 0 and high = length of array - 1.

2. While low <= high:

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

b. If number == array[mid], return array[mid].

c. If name < array[mid], set high = mid - 1.

d. Otherwise, set low = mid + 1.

3. If the loop terminates without finding the name, return "Not found."

**3**. **non\_recursive\_binary\_search(name):**

1. Initialize low = 0 and high = length of array - 1.

2. While low <= high:

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

b. If array[mid] == number, return array[mid].

c. If name < array[mid], set high = mid - 1.

d. Otherwise, set low = mid + 1.

3. If the loop terminates without finding the name, return "Not found."

**4**. **main():**

1. Initialize an empty array.

2. Display a menu with the following options:

a. Insert number.

b. Search for a number (recursive).

c. Search for a number (non-recursive).

d. Exit.

3. Repeat the following until the user chooses to exit:

a. Prompt the user for their choice.

b. If the choice is 'a':

i. Prompt the user for a number.

c. If the choice is 'b':

i. Prompt the user for a number.

ii. Call recursive\_binary\_search(number) and display the result.

d. If the choice is 'c':

i. Prompt the user for number.

ii. Call non\_recursive\_binary\_search(number) and display the result.

e. If the choice is 'd', exit the program.

f. If the choice is invalid, display an error message.

4. End the program.

**4. Conclusion**

In this lab exercise, we learned how to create a program to manage and search number using binary search.

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

import java.util.Scanner;

public class BinarySearchExample {

    public static int recursiveBinarySearch(int[] arr, int low, int high, int x) {

        if (high >= low) {

            int mid = low + (high - low) / 2;

            if (arr[mid] == x)

                return mid;

            if (arr[mid] > x)

                return recursiveBinarySearch(arr, low, mid - 1, x);

            return recursiveBinarySearch(arr, mid + 1, high, x);

        }

        return -1;

    }

    public static int iterativeBinarySearch(int[] arr, int x) {

        int low = 0, high = arr.length - 1;

        while (low <= high) {

            int mid = low + (high - low) / 2;

            if (arr[mid] == x)

                return mid;

            if (arr[mid] < x)

                low = mid + 1;

            else

                high = mid - 1;

        }

        return -1;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();

        int[] arr = new int[n];

        for (int i = 0; i < n; i++)

            arr[i] = sc.nextInt();

        int key = sc.nextInt();

        int resRec = recursiveBinarySearch(arr, 0, n - 1, key);

        System.out.println(resRec);

        int resIter = iterativeBinarySearch(arr, key);

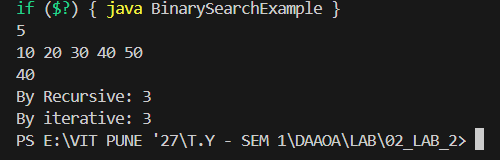
        System.out.println(resIter);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

**Recursive Binary Search:**

* Time Complexity:
  + Best Case: **O(1)** because if the middle element is the target, it takes just one comparison.
  + Average Case: **O(log n)** since the search space halves with each recursive call, reducing the problem size logarithmically.
  + Worst Case: **O(log n)** arises when the element is at one of the ends, requiring maximum recursive steps.
* Space Complexity: **O(log n)** due to the recursion call stack that holds at most one call per level down to log n levels.

**Iterative (Non-Recursive) Binary Search:**

* Time Complexity:
  + Best Case: **O(1)** when the middle element matches the target immediately.
  + Average Case: **O(log n)** because the halving of the search space continues iteratively until found.
  + Worst Case: **O(log n)** when the search narrows down to the edges of the array.
* Space Complexity: **O(1)** since only a few variables are used regardless of input size, with no recursion stack overhead.

FUNCTION recursiveBinarySearch(A, low, high, x) // Space: O(log n) for stack frames (recursion)

IF high >= low // Time: +1

mid ← low + (high - low) / 2 // Time: +1, Space: +1

IF A[mid] == x // Time: +1

RETURN mid // Time: +1

IF A[mid] > x // Time: +1

RETURN recursiveBinarySearch(A, low, mid-1, x) // Each call divides input; Time: \*log n

ELSE

RETURN recursiveBinarySearch(A, mid+1, high, x)// Each call divides input; Time: \*log n

RETURN -1 // Time: +1

ENDFUNCTION

// Recursive Binary Search: Time O(log n), Space O(log n) [web:26][web:27][web:28]

FUNCTION iterativeBinarySearch(A, x) // Space: O(1) extra variables

low ← 0 // Time: +1, Space: +1

high ← length(A) - 1 // Time: +1, Space: +1

WHILE low <= high // Time: \*log n (range halves every iteration)

mid ← low + (high - low) / 2 // Time: +1, Space: +1

IF A[mid] == x // Time: +1

RETURN mid // Time: +1

IF A[mid] < x // Time: +1

low ← mid + 1 // Time: +1

ELSE

high ← mid - 1 // Time: +1

RETURN -1 // Time: +1

ENDFUNCTION

// Iterative Binary Search: Time O(log n), Space O(1) [web:26][web:27][web:28]

FUNCTION main

DECLARE n // Space: +1

INPUT n // Time: +1

DECLARE array A of size n // Space: +n

FOR i = 0 TO n-1 // Time: +n

INPUT A[i] // Time: +1 per iteration

ENDFOR

INPUT key // Time: +1

resRec ← recursiveBinarySearch(A, 0, n-1, key) // Time: O(log n), Space: O(log n)

PRINT resRec // Time: +1

resIter ← iterativeBinarySearch(A, key) // Time: O(log n), Space: O(1)

PRINT resIter // Time: +1

ENDFUNCTION

**Complexity Analysis:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Time Complexity** | **Space Complexity** |
| **Recursive** | O(log n) | O(log n) |
| **Iterative** | O(log n) | O(1) |