Name: Bhakti Anil Baraf
Reg No: 241071907
BTech SY CE (Batch D)

Assignment 2

Aim:

Understanding Linear Search and Binary Search algorithms. Design the algorithm and develop the program for the same.

Searching:

Searching is the process of finding an element in a list. There are several search algorithms available, but two common types are:

1. **Linear Search**:
   - This is the simplest form of search. It sequentially checks each element of the list until a match is found or the whole list has been searched.
   - In linear search, we go through each element in the array one by one. Starting from the first element, we compare it with the target value. If the element matches the target, we return the index where it was found. If it doesn't match, we move to the next element and repeat the process.
   - Time Complexity: O(n), where n is the number of elements in the array.
   - Example:

   Array: [5, 8, 12, 20, 30]
   Target: 20

   Explanation:
   In this example, we compare 5, 8, and 12 with 20, and none of them match. When we reach the fourth element, 20, it matches the target, so we stop and return the index 3 (since arrays are zero-indexed).

   Output:
   Target found at index 3

2. **Binary Search**:
    - Binary Search works on sorted arrays. It divides the array into halves, comparing the target value to the middle element, and repeatedly narrows the search interval based on the comparison.
    - Binary search works by dividing the sorted array in half repeatedly to reduce the search space. We start by looking at the middle element. If the middle element is equal to the target, we stop. If it's smaller, we continue searching in the right half of the array; if it's larger, we search in the left half.
    - Time Complexity: O(log n), where n is the number of elements in the array.
    - Example:

Sorted Array: [2, 5, 8, 12, 20, 25]
Target: 12

Explanation:
In this example, we first compare the middle element 8 with 12. Since 12 is larger, we ignore the left half and focus on the right half of the array. We then take the middle of the remaining elements, which is 12. It matches the target, so we return the index 3.

Output:
Target found at index 3

Steps for Linear Search and Binary Search:

1. **Linear Search:**

1. Start from the first element of the array.
2. Compare the target element with the current array element.
3. If the target matches, return the index.
4. If the target does not match, move to the next element.
5. Repeat steps 2–4 until the target is found or the array is fully traversed.
6. If the target is not found, return -1.

2. Binary Search:

1. Start with the entire sorted array.
2. Set two pointers: low at the beginning and high at the end of the array.

3. Find the middle element mid.
4. Compare the target element with the middle element:
    - If the target equals the middle element, return the index.
    - If the target is smaller than the middle element, search the left half by setting high = mid - 1.
    - If the target is larger than the middle element, search the right half by setting low = mid + 1.
5. Repeat until the target is found or the low pointer exceeds the high pointer.
6. If the target is not found, return -1.

Algorithms for Linear Search and Binary Search

1. Linear Search Algorithm:

Input:

- A list/array A of size n.
- A target value target to be searched.

Output:

- The index of the target if found, or -1 if not found.

Steps:

1. **Start**
2. Set i = 0 (starting index).
3. While i < n, repeat the following:
    a. If A[i] == target, return i.
    b. Otherwise, increment i by 1.
4. If the target is not found after the loop ends, return -1.
5. **End**

2. Binary Search Algorithm:

Input:

- A sorted list/array A of size n.
- A target value target to be searched.

Output:

- The index of the target if found, or -1 if not found.

Steps:

1. **Start**
2. Initialize two pointers:
   - low = 0 (starting index).
   - high = n - 1 (last index).
3. While low <= high, repeat the following:
   - a. Set mid = (low + high) // 2 (middle index).
   - b. If A[mid] == target, return mid.
   - c. If A[mid] > target, set high = mid - 1 (search in the left half).
   - d. If A[mid] < target, set low = mid + 1 (search in the right half).
4. If the target is not found after the loop ends, return -1.
5. **End**

Pseudocode:

1. **Linear Search Pseudocode:**

```
Procedure LinearSearch(A, target)

   for i = 0 to length(A) - 1 do

      if A[i] == target then

         return i   // Target found at index i

   return -1           // Target not found

End Procedure
```

2. **Binary Search Pseudocode:**

```
Procedure BinarySearch(A, target, low, high)

   while low <= high do

      mid = (low + high) // 2

      if A[mid] == target then

         return mid       // Target found at index mid
```

```
        else if A[mid] > target then
            high = mid - 1   // Search in the left half
        else
            low = mid + 1    // Search in the right half
    return -1                // Target not found
End Procedure
```

·) Linear Search Algorithm:

Input:
- A list of size n
- A target value to be searched.

Output:
- The index of the target if target found otherwise return -1.

Pseudocode:

```
                          List
def   LinearSearch (A, target )

      for i = 0 to length (A) - 1 :
          if A[i] == target:
               return i

      return -1

end def
```

2) Bineary Search Algorithm:

Input:

- A sorted list of size n
- A target value to be searched.

Output:

- The index of the target/value will be returned oth if target found otherwise return -1.

Pseudocode:

```
def binarysearch (list, target, low, high)
    while low <= high
        mid = (low + high)/2

        if list[mid] == target :
            return mid
        else if list[mid] > target:
            high = mid - 1
        else
            low = mid + 1
    return -1

end def.
```

Example for Linear Search and Binary Search

1. Linear Search Example:

Input:
Array A = [15, 3, 9, 8, 20, 11, 13]
Target target = 20

Steps:

1. Start at index i = 0. Compare A[0] = 15 with target = 20.
   - Not a match, so move to the next index.
2. At index i = 1. Compare A[1] = 3 with target = 20.
   - Not a match, so move to the next index.
3. At index i = 2. Compare A[2] = 9 with target = 20.
   - Not a match, so move to the next index.
4. At index i = 3. Compare A[3] = 8 with target = 20.
   - Not a match, so move to the next index.
5. At index i = 4. Compare A[4] = 20 with target = 20.
   - **Match found** at index 4. Return 4.

Output:
Target found at index 4

2. Binary Search Example:

Input:
Sorted array A = [2, 6, 12, 18, 24, 32, 47, 59]
Target target = 24

Steps:

1. Set low = 0 and high = 7 (since the length of the array is 8).
2. Calculate mid = (0 + 7) // 2 = 3. Compare A[3] = 18 with target = 24.
   - Since 24 > 18, search in the right half by setting low = mid + 1 = 4.
3. Set low = 4 and high = 7. Calculate mid = (4 + 7) // 2 = 5. Compare A[5] = 32 with target = 24.
   - Since 24 < 32, search in the left half by setting high = mid - 1 = 4.
4. Now, low = 4 and high = 4. Calculate mid = (4 + 4) // 2 = 4. Compare A[4] = 24 with target = 24.
   - **Match found** at index 4. Return 4.

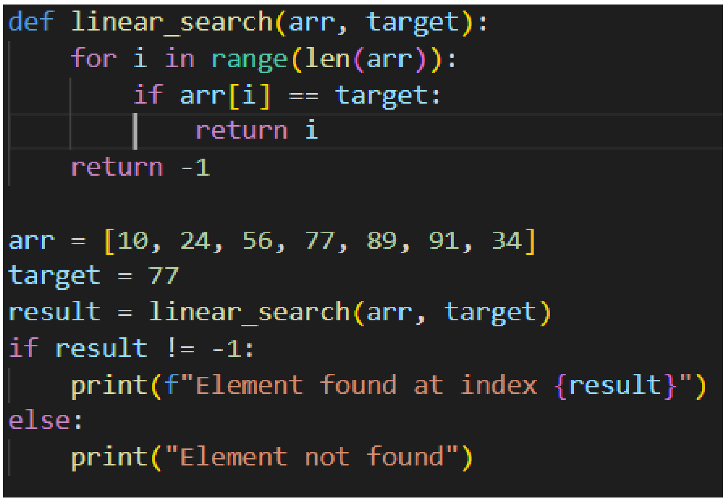
Output:
Target found at index 4

Python Program:

Linear Search Program:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

arr = [10, 24, 56, 77, 89, 91, 34]
target = 77
result = linear_search(arr, target)
if result != -1:
    print(f"Element found at index
{result}")
else:
    print("Element not found")
```



```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

arr = [10, 24, 56, 77, 89, 91, 34]
target = 77
result = linear_search(arr, target)
if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

Output:
Element found at index 3

Binary Search Program:

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

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

        if arr[mid] == target:
            return mid

        elif arr[mid] > target:
            high = mid - 1

        else:
            low = mid + 1

    return -1

arr = [10, 24, 56, 77, 89, 91, 100]
target = 77
result = binary_search(arr, target)
if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

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

        # Check if target is present at mid
        if arr[mid] == target:
            return mid  # Target found at index mid

        # If target is smaller than mid, search the left half
        elif arr[mid] > target:
            high = mid - 1

        # If target is larger than mid, search the right half
        else:
            low = mid + 1

    return -1  # Target not found

# Test the function
arr = [10, 24, 56, 77, 89, 91, 100]
target = 77
result = binary_search(arr, target)
if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

Output:
Element found at index 3

Time Complexity:

Comparison

- **Linear Search**: Simple to implement and works on both sorted and unsorted lists. It has a time complexity of O(n), where n is the number of elements in the list.
- **Binary Search**: More efficient for large, sorted lists, with a time complexity of O(log n). However, it requires that the list is sorted.

Time complexity:

1) Linear search:

Input size = n

∵ The Linear search will search element from start to end.
Then, ~~T~~
Total number of comparisions = n = C(n)

$$C(n) = \sum_{i=1}^{n} 1 = n-1+1 = n \in O(n)$$

∴ The time complexity is O(n)

2) Binary search:

Input size = n

Operation:
1. ~~In elem~~ each step the search is halved.
2. If elem is found at mid then stop.
3. If elem not found at mid then set ~~k~~upper and lower limit.

• After every comparision the input size is reduced to $\frac{n}{2}$

∵ The value of ~~ea~~ n is about halved on each repetation of loop, then answer ~~i~~should be $\log_2 n$

∴ The number of times we can divide n by 2 until we reach 1:

$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \cdots \rightarrow 1$

i.e $n = \frac{n}{2} \rightarrow \frac{n}{2^2} \rightarrow \frac{n}{2^3} \rightarrow \cdots \rightarrow \frac{n}{2^k}$

Termination occurs at

$\frac{n}{2^k} < 1$ $\quad k = lo$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2 n$

∴ $O(\log_2 n)$

∴ The time complexity is $O(\log_2 n)$

Test Cases for Linear Search and Binary Search:

1. Positive TestCases:

Testcases:

1) Linear Search & Binary Search:

• Positive test cases:

i) Target = 77
List elem = 10, 24, 56, 77, 89, 91.

Linear search: element found at index 3
Binary search: element found at index 3

ii) Target = 5
List elem = 10, 2, 5, 4, 3, 2, 1

Linear Search = element found at index 2
Binary Search = element found at index 3
Target = 4

iii) Target = 100
List elem = 100, 200, 300, 112, 150

Linear search = element found at index 0
Binary search = element found at index 0.

iv) Target = 50
List elem = 10, 20, 5, 4, 50, 21

Linear search = element found at index 4
Binary Search = element found at index 5

List elem = 4, 5, 10, 20, 21, 50

```
PS C:\Users\BHAKTI\OneDrive\Documents\Python> & c:/Users/BHAKTI/OneDrive/Doc
on/vprac2.py
Element found at index 3
PS C:\Users\BHAKTI\OneDrive\Documents\Python> & c:/Users/BHAKTI/OneDrive/Doc
on/vprac2.py
Element found at index 3
Element found at index 3
```

2. **Negative Test Cases**:

• Negative test cases:

i) ~~En~~ Target = 0
List elem = 10 20 30 40

Linear search = element not found
Binary search = element not found

ii) Target = 1
List elem = 2 5 4 0 3

Linear search = element not found
Binary search = element not found.

iii) Target = 20
List elem = 16, 2 3 6 8

Linear search = element not found
Binary search = element not found.

```
PS C:\Users\BHAKTI\OneDrive\Documents\Python> & c:/Users/BHAKTI/
on/vprac2.py
Element not found
PS C:\Users\BHAKTI\OneDrive\Documents\Python>
```

Conclusion: In this Experiment we learned about Searching Algorithms such as Linear Search and Binary Search.