Marwadi University Marwadi Chandarana Group NAAC A+	Marwadi University Faculty of Engineering and Technology Department of Information and Communication Technology	
Subject: DAA (01CT0512)	Aim: **Implementing the Searching Algorithms and understanding the time and space complexities**	
Lab-2		**Enrolment No**: 92301733046

Programming Language: Python

1) Linear search:

- **Theory:**
  - **How It Works:**
    - Start from the first element.
    - Compare each element with the target.
    - Stop when you find it or reach the end.

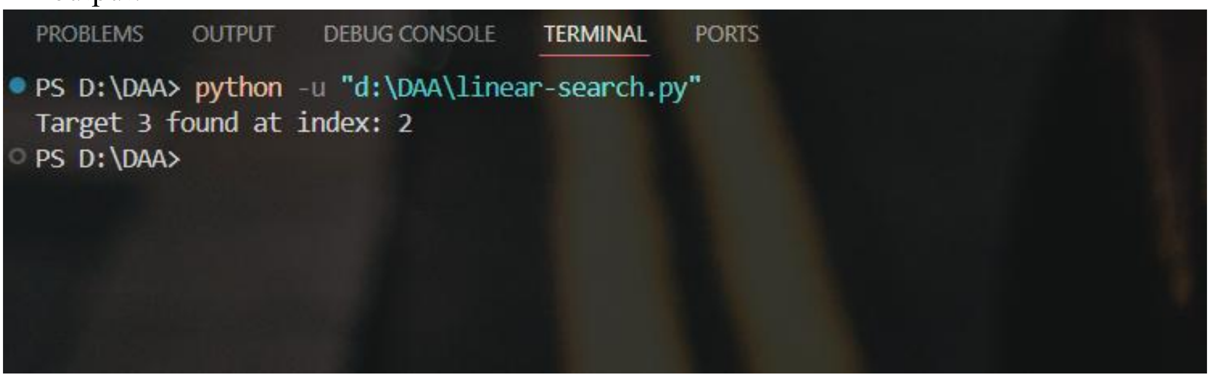
  - **Code:**

```
#LInear Search Algorithm

def linear_search(arr, target):
    if(target in arr):
        return arr.index(target)
    else:
        return -1

arr=[1, 2, 3, 4, 5]
target=3
result=linear_search(arr, target)
if result!=-1:
    print(f"Target {target} found at index: {result}")
else:
    print(f"Target {target} not found in the array")
```

  - output:



- Time Complexity:
  I. Worst time complexity:O(n)

- If the target is not present or is the last element, we must check every element once.n = length of the array.

II. Best case time complexity: O(1)

- If the target is at the **first index**, we find it immediately.

➢ Space Complexity: O(1)
Justification:

- No **extra data structures** or **recursive calls** are used.

2) <u>Binary Search:</u>

➢ Theory:

- Binary search is a fast and intelligent search technique used to find an item in a sorted list.
- It works by repeatedly dividing the search range in half and eliminating the part that doesn't contain the target.
- Rather than going through every item one by one, binary search tries to zero in on the answer quickly cutting the search space in half with every step.

➢ Code:

```
def binary_search(arr, target):
    if len(arr) == 0:
        return 0

    mid = len(arr)// 2
    if mid==1:
        return arr[mid]
    if arr[mid]==target:
        return mid
    elif arr[mid]>target:
        return binary_search(arr[:mid], target)
    elif arr[mid]<target:
        return binary_search(arr[mid+1:], target)
    else:
        return -1

arr=[1,2,3,4,5,6,7]
target = 3
print(binary_search(arr, target))
```

- output:

- Time Complexity: O(log n)

  Justification:
  - Each recursive call cuts the array in half.
  - Suppose the array has n elements:
    - First call → size n
    - Next → size n/2
    - Then → n/4, n/8, ... until size becomes 1.
    - So the number of steps is roughly $\log_2(n)$.
- Worst case time complexity: O(log n)
  - At each recursive step, the array is divided into half → just like cutting a book in half to find a page faster.
  - Even in the worst case (when the element is at the very end or not in the list), the function performs at most $\log_2(n)$ steps before concluding.
  - So, for n = 8 → max steps ≈ 3 (since $2^3 = 8$)
- Space Complexity: O(log n)

  Justification:
  - You're using slicing like arr[:mid] and arr[mid+1:] in every recursive call.
  - This creates a new array copy each time, which takes space.
  - Also, recursion uses call stack memory.
  - Both slicing and recursion depth go up to log n → hence, space = O(log n).