Task 1

1- Write a code of binary search

```python
def binary_search(lst, target):
    left, right = 0, len(lst) - 1
    while left <= right:
        mid = (left + right) // 2
        if lst[mid] == target:
            return mid
        elif lst[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return "Not Found"

# Test the function
target1 = int(input("enter your target:"))
print(binary_search([1, 2, 3, 4, 5], target1))
target2 = int(input("enter your target:"))
print(binary_search([1, 2, 3, 4, 5], target2))
```

```
enter your target:3
2
enter your target:7
Not Found
```

2- What is the Complexity of a Code and How to Calculate It?

The complexity of a code refers to measures that describe how efficient and easy to maintain a program is. The main types of complexity are time complexity and space complexity.

1-Time Complexity:

Definition: Time complexity measures how long an algorithm takes to complete as the input size grows.

How to Calculate:

1. **Identify the main operations:** Look for the parts of the code that take the most time.
2. **Count how often they run:** See how the number of operations changes with the input size.
3. **Use Big O notation:** Simplify the expression to show the largest factor that affects the time.

Examples:

- **O(1):** Constant time - the time doesn't change with input size.
- **O(n):** Linear time - the time increases directly with input size.
- **O(n^2):** Quadratic time - the time increases with the square of the input size.

1-Space Complexity:

Definition: Space complexity measures how much memory an algorithm uses as the input size grows.

How to Calculate:

1. **Identify memory use:** Look at the variables and data structures.
2. **See how it grows with input size:** Check how memory usage changes with larger inputs.
3. **Use Big O notation:** Simplify to show the largest factor affecting memory use.

Examples:

- **O(1):** Constant space - memory use doesn't change with input size.
- **O(n):** Linear space - memory use increases directly with input size.

3- choose 2 types of searching and explain them

Breadth-First Search (BFS)

Definition: BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or an arbitrary node) and explores all its neighbors at the current depth level before moving on to nodes at the next depth level.

How It Works:

1. Start with a queue and enqueue the starting node.
2. Dequeue a node, mark it as visited, and enqueue all its unvisited neighbors.
3. Repeat until the queue is empty.

Key Points:

- **Time Complexity:** O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity:** O(V), as it needs to store all the vertices in the queue.
- **Use Cases:** BFS is useful for finding the shortest path in unweighted graphs and level-order traversal of trees.

Depth-First Search (DFS)

Definition: DFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or an arbitrary node) and explores as far as possible along each branch before backtracking.

How It Works:

1. Start with a stack (or use recursion) and push the starting node.
2. Pop a node, mark it as visited, and push all its unvisited neighbors.
3. Repeat until the stack is empty.

Key Points:

- **Time Complexity:** O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity:** O(V), due to the stack used for recursion or the iterative process.
- **Use Cases:** DFS is useful for topological sorting, detecting cycles in a graph, and solving puzzles like mazes.