Time vs Space Complexity

what is the time complexity for Linear Search algorithm?

The **time complexity** of the **Linear Search** algorithm is:

- **Best Case: O(1)** (if the target element is the first element in the array)
- **Worst Case: O(n)** (if the target element is at the end or not present in the array)
- **Average Case: O(n)** (on average, the target is found in the middle of the array)

Explanation:

Linear Search scans each element of the array **one by one** until it finds the target or reaches the end.

Linear Search Algorithm:

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Found, return index
    return -1  # Not found

arr = [10, 20, 30, 40, 50]
target = 30
print(linear_search(arr, target))  # Output: 2 (index of 30)
```

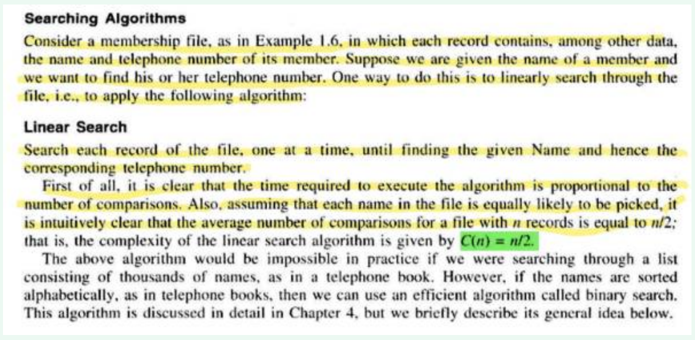
Why is Linear Search O(n) in the worst case?

- In the worst case, we check **every element** before concluding the element is absent.
- For an array of size **n**, we perform **n comparisons** → **O(n)**.

Can we do better?

- **If the array is sorted**, we can use **Binary Search (O(log n))** instead of **Linear Search (O(n))**.
- **For searching in large datasets,** using **hashing (O(1) on average)** is even faster.

Would you like a comparison with **Binary Search** for better understanding? 😊



Searching Algorithms

Consider a membership file, as in Example 1.6, in which each record contains, among other data, the name and telephone number of its member. Suppose we are given the name of a member and we want to find his or her telephone number. One way to do this is to linearly search through the file, i.e., to apply the following algorithm:

Linear Search

Search each record of the file, one at a time, until finding the given Name and hence the corresponding telephone number.

First of all, it is clear that the time required to execute the algorithm is proportional to the number of comparisons. Also, assuming that each name in the file is equally likely to be picked, it is intuitively clear that the average number of comparisons for a file with n records is equal to $n/2$; that is, the complexity of the linear search algorithm is given by $C(n) = n/2$.

The above algorithm would be impossible in practice if we were searching through a list consisting of thousands of names, as in a telephone book. However, if the names are sorted alphabetically, as in telephone books, then we can use an efficient algorithm called binary search. This algorithm is discussed in detail in Chapter 4, but we briefly describe its general idea below.

why does the book say here that the time complexity is n/2?

The book states that the average number of comparisons in a **Linear Search** is **n/2** because, on average, the algorithm finds the target halfway through the list.

Breaking it Down:

1. **Best Case (O(1))**: If the target element is **the first element**, we find it in **1 comparison**.
2. **Worst Case (O(n))**: If the target element is **the last element (or not present)**, we scan through **all n elements**.
3. **Average Case (O(n))**:
   - If the target element is equally likely to be at any position in the array, then, on average, we will find it **at the middle position**.
   - This means we check approximately **n/2 elements** before finding the target.
   - Mathematically, if the target is randomly distributed, the **expected number of comparisons** is:

$$C(n) = \frac{1 + 2 + 3 + ... + n}{n} = \frac{n+1}{2} \approx \frac{n}{2}$$

   - This gives an **average time complexity of O(n), ignoring constant factors**.

Why is it Still O(n)?

The Big-O notation focuses on **growth rate** and ignores constant factors (like **1/2** in **n/2**). So, even though the average case is **n/2 comparisons**, in asymptotic notation, we still write it as **O(n)**.

Would you like a visual example or code demonstration for better clarity? 😊