Searching Algorithms: Linear & Binary Search

DSA TA Section - By **Faiq Ahmed, Omer Shamsi, Areeba Zehra Jafri, and Fatima Khan**

Warm-up: Finding Your Place

Unsorted Attendance List

How would you find your roll number in a jumbled list? You'd likely check each name one by one.

Analogy: Linear Search

Sorted Attendance List

What if the list is sorted alphabetically or numerically? You might open it in the middle, then decide which half to check next.

Analogy: Binary Search

Linear Search - The Idea

Linear search is the most straightforward searching algorithm. It sequentially checks each element of the list until a match is found or the whole list has been searched.

- Starts from the beginning of the list.
- Compares each element with the target value.
- Continues until the target is found or the end of the list is reached.
- Suitable for unsorted data.

Linear Search - Implementation Walkthrough

The Process

01

Initialize

Set a counter or index to the start (e.g., 0).

02

Iterate

Loop through each element of the array.

03

Compare

At each step, compare the current element with the target.

04

Found or End

If a match is found, return the index. If the loop finishes without a match, the element is not present.

Pseudocode Snippet

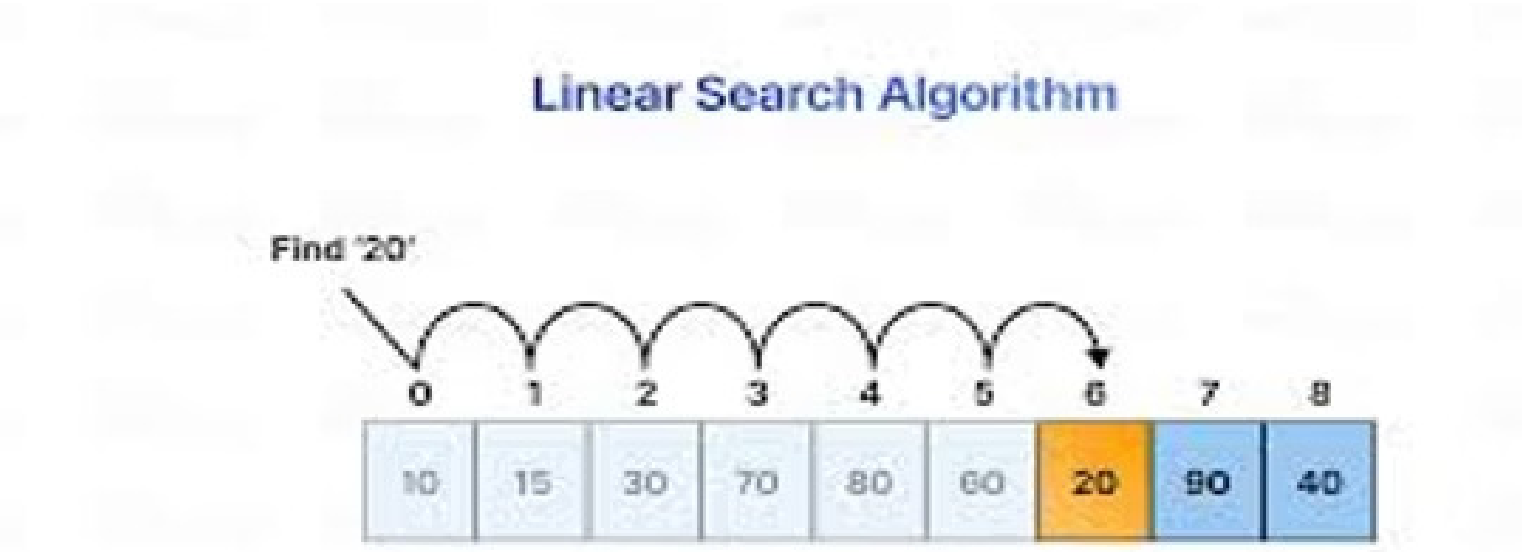
```
function linearSearch(arr, target):
  for i from 0 to arr.length - 1:
    if arr[i] == target:
      return i // Found
  return -1    // Not found
```

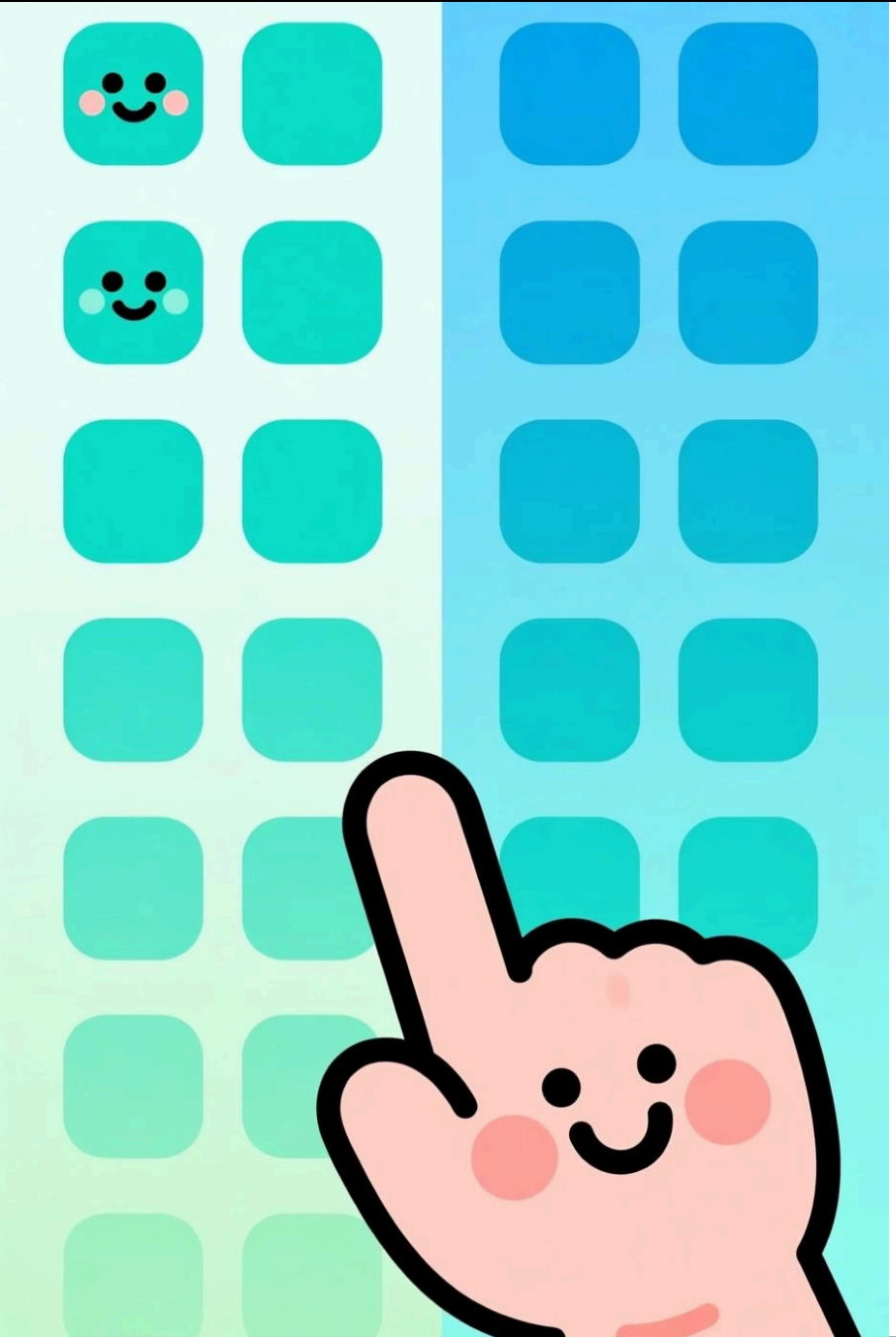
Time Complexity: O(n) – In the worst case, we check every element.

Example:

Array: [12, 5, 9, 34, 7]

Search for 34 → check 12, 5, 9 → found at 4th position





Binary Search - The Idea

Binary search is a highly efficient algorithm for finding an element in a sorted list. It works by repeatedly dividing the search interval in half.

- Start with the middle element.
- If the target matches, you're done!
- If the target is smaller, search the left half.
- If the target is larger, search the right half.
- Repeat the process until the element is found or the interval is empty.

Binary Search - Implementation Walkthrough

The Steps

01

Define Range

Set `low` to the first index and `high` to the last index.

02

Find Mid

Calculate `mid = (low + high) / 2` (integer division).

03

Compare & Adjust

If `arr[mid] == target`, return `mid`. If `arr[mid] < target`, set `low = mid + 1`. If `arr[mid] > target`, set `high = mid - 1`.

04

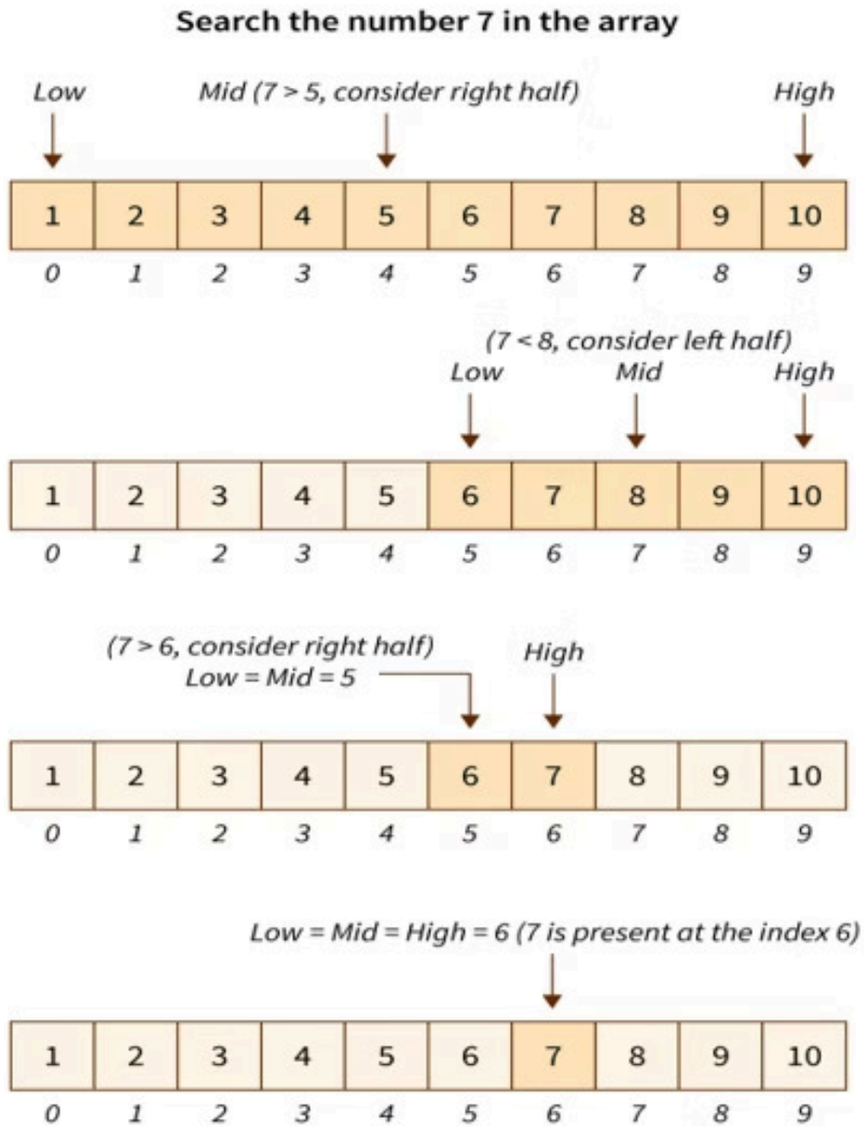
Repeat

Continue until `low > high` (element not found) or target is found.

Pseudocode Snippet

```
function binarySearch(arr, target):
  low = 0
  high = arr.length - 1
  while low <= high:
    mid = (low + high) / 2
    if arr[mid] == target:
      return mid
    else if arr[mid] < target:
      low = mid + 1
    else: // arr[mid] > target
      high = mid - 1
  return -1
```

Time Complexity: O(log n) – Each comparison halves the search space.



Linear vs. Binary Search: A Quick Comparison

Requirement	Unsorted or sorted data	**Sorted data only**
Speed (Time Complexity)	O(n) - Slower for large datasets	**O(log n) - Much faster for large datasets**
Simplicity	Easier to understand and implement	More complex logic
Use Cases	Small lists, unsorted data, linked lists	Large, sorted arrays/lists, databases

Level Up Your Skills: LeetCode Practice

Apply what you've learned to these LeetCode problems. Start with easy and build your way up!

Easy

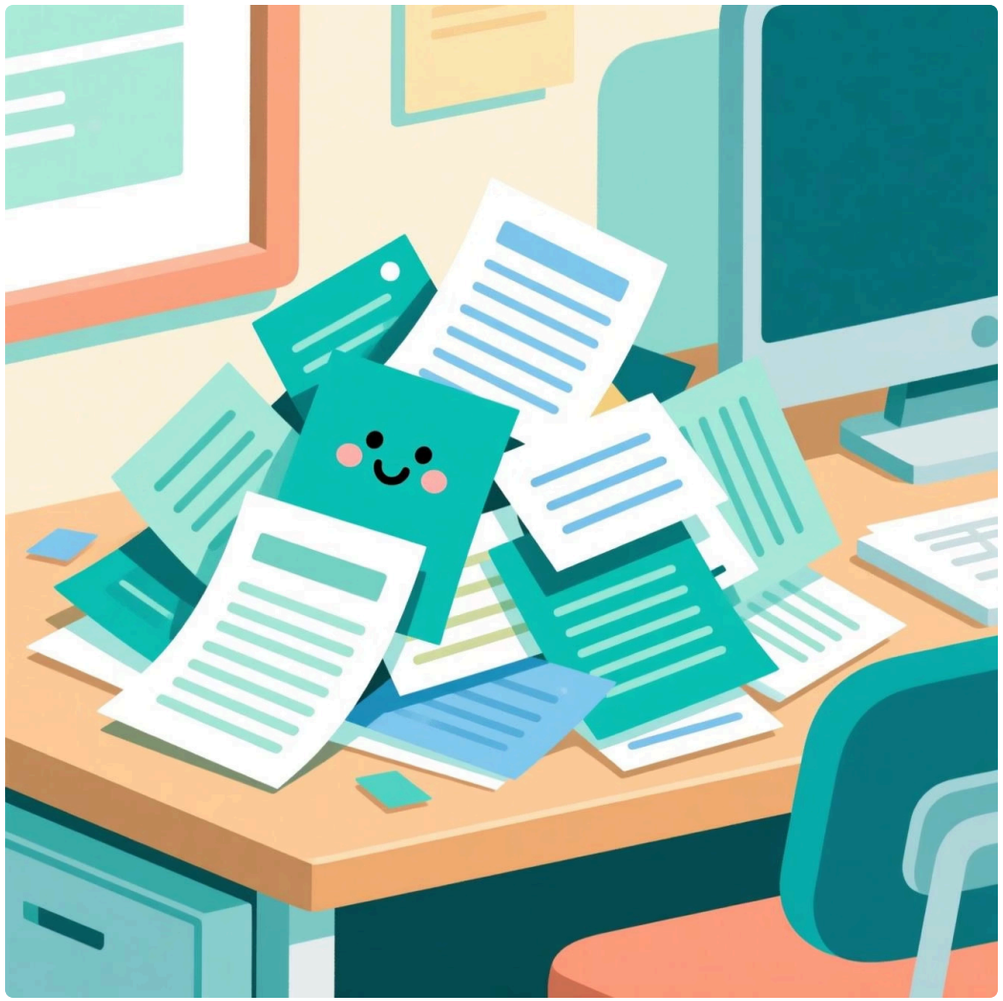
- 704. Binary Search
- 35. Search Insert Position

Medium

- 852. Peak Index in a Mountain Array
- 74. Search a 2D Matrix

Real-Life Scenarios: When to Use Which?

Scenario 1: Jumbled Bundle of Papers



Imagine you have a stack of unorganized papers. Finding a specific document requires you to go through each one until you find it. This is analogous to a Linear Search.

Key: Data is not pre-arranged.

Scenario 2: Sorted Bundle of Roll Numbers

Now, imagine you have a list of student roll numbers, perfectly sorted. To find a specific number, you'd open to the middle, then decide if you need to check the first or second half. This mirrors Binary Search.

Key: Data is ordered, enabling efficient searching.

Wrap-up & Key Takeaways

Linear Search

O(n) complexity, simple, suitable for **unsorted** data. Think of it as checking items one by one.

Binary Search

O(log n) complexity, highly efficient, requires data to be **sorted**. It dramatically reduces search space.

Choose Wisely

The choice of algorithm depends on your dataset's size and whether it's sorted. Always consider the trade-offs!

> **Your Challenge:** Solve at least 5 LeetCode problems on searching before our next session to solidify your understanding!