Linear and Binary Search

A Comprehensive Guide

Algorithm Study Material

Overview: Searching algorithms are fundamental processes in computer science used to retrieve information stored within some data structure. This document compares the two most common search algorithms: **Linear Search** and **Binary Search**.

1 Linear Search

1.1 Definition

Linear search (also known as sequential search) is the simplest search algorithm. It works by checking every element in an array or list sequentially, one by one, from the beginning to the end, until the desired target element is found or the list ends.

1.2 How It Works

1. Start at the first element (index 0) of the array.
2. Compare the current element with the target value.
3. If it matches, the search is successful, and the index is returned.
4. If it does not match, move to the next element.
5. Repeat steps 2-4 until the end of the array is reached. If the end is reached without a match, the target is not in the array.

1.3 Array Example

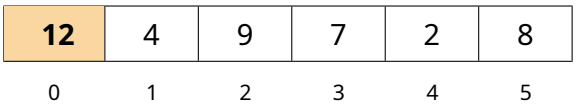
Let's find the target number **7** in the following unsorted array: **Array:** `[ 12, 4, 9, 7, 2, 8 ]`

- **Step 1:** Check index 0. Value is 12. $(12 \neq 7) \rightarrow$ Move to next.
- **Step 2:** Check index 1. Value is 4. $(4 \neq 7) \rightarrow$ Move to next.
- **Step 3:** Check index 2. Value is 9. $(9 \neq 7) \rightarrow$ Move to next.
- **Step 4:** Check index 3. Value is 7. $(7 == 7) \rightarrow$ **Match found! Return index 3.**

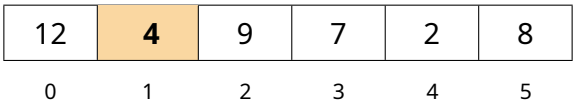
Visualizing the Passes (Target = 7)

Legend: *Checking* *Match Found*

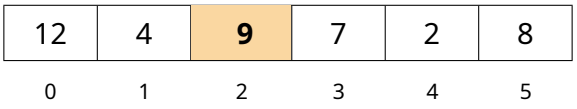
Pass 1: Checking index 0. ($12 \neq 7$).



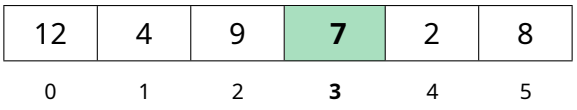
Pass 2: Checking index 1. ($4 \neq 7$).



Pass 3: Checking index 2. ($9 \neq 7$).



Pass 4: Checking index 3. ($7 == 7$). **Match found!**



1.4 Pseudocode

Linear Search Pseudocode

```
FUNCTION LinearSearch(Array, Target):
    FOR index FROM 0 TO length(Array) - 1:
        IF Array[index] == Target:
            RETURN index  // Target found
        END IF
    END FOR

    RETURN -1 // Target not found
END FUNCTION
```

1.5 Complexity & Characteristics

- **Time Complexity:** $O(n)$ in the worst case (target is at the end or missing). $O(1)$ in the best case (target is the first element).
- **Space Complexity:** $O(1)$ because it only requires a single variable (the index) to traverse the array. No extra memory is needed.

Advantages: Extremely simple to implement; works on both sorted and unsorted data; does not require additional memory.
Disadvantages: Very slow for large datasets because it might have to check every single element.

2 Binary Search

2.1 Definition

Binary search is a highly efficient searching algorithm that finds the position of a target value within a **sorted** array. It works by repeatedly dividing the search interval in half.

2.2 How It Works

1. The array **must be sorted** beforehand.
2. Define two pointers: `Low` (start of the array) and `High` (end of the array).
3. Calculate the `Mid` point: Mid $= (\text{Low} + \text{High})/2$.
4. Compare the target with the element at the `Mid` point:
   - If they match, return the `Mid` index.
   - If the target is *less* than the Mid element, the target must be in the left half. Update `High = Mid - 1`.
   - If the target is *greater* than the Mid element, the target must be in the right half. Update `Low = Mid + 1`.
5. Repeat steps 3-4 until `Low` becomes greater than `High` (target not found).

2.3 Array Example

Let's find the target number **25** in the following **sorted** array: **Array:** `[ 3, 8, 14, 25, 39, 42, 55 ]`

- **Initial State:** `Low` = 0 (value 3), `High` = 6 (value 55).
- **Step 1:** Calculate `Mid` = (0 + 6) / 2 = 3.
  - Value at index 3 is **25**.
  - Target (25) == Mid Value (25).
  - **Match found! Return index 3.**

Note: Even though it took 1 step here due to the midpoint, if we were searching for 42, the algorithm would have discarded the entire left half `[3, 8, 14, 25]` in a single step and searched only `[39, 42, 55]`.

Visualizing Multiple Passes (Target = 42)

To better demonstrate the "halving" nature of Binary Search, let's search for **42** in the same array.

Legend: *Mid (Checking)* *Match Found* *Discarded Half*

Pass 1: `Low` = 0, `High` = 6. Calculate `Mid` = (0+6)/2 = 3. Value is **25**.
Since $25 < 42$, the target must be in the right half. Discard index 0 to 3.

3	8	14	**25**	39	42	55
0	1	2	**3 (Mid)**	4	5	6

Pass 2: `Low` = 4, `High` = 6. Calculate `Mid` = (4+6)/2 = 5. Value is **42**.
Since $42 == 42$, **Match found!**

3	8	14	25	39	**42**	55
0	1	2	3	4	**5 (Mid)**	6

2.4 Pseudocode

Binary Search Pseudocode

```
FUNCTION BinarySearch(Array, Target):
    Low = 0
    High = length(Array) - 1

    WHILE Low <= High:
        Mid = (Low + High) / 2

        IF Array[Mid] == Target:
            RETURN Mid            // Target found
        ELSE IF Array[Mid] < Target:
            Low = Mid + 1         // Search right half
        ELSE:
            High = Mid - 1        // Search left half
        END IF
    END WHILE

    RETURN -1 // Target not found
END FUNCTION
```

2.5 Complexity & Characteristics

- **Time Complexity:** $O(\log n)$ in the worst/average case. Every step cuts the search area in half. Best case is $O(1)$ (target is exactly in the middle).
- **Space Complexity:** $O(1)$ for the iterative approach shown above. (Recursive approach would take $O(\log n)$ call stack space).

Advantages: Incredibly fast for large datasets. It drastically minimizes the number of comparisons needed.
Disadvantages: The array *must* be sorted first. Sorting takes time. If the array frequently changes (elements added/removed), maintaining the sorted order can be costly.

3 Differences Between Linear and Binary Search

Below is a side-by-side difference table of the two algorithms, highlighting 6 simple one-liner points.

Linear Search	Binary Search
Does not require the dataset to be sorted.	Requires the dataset to be sorted beforehand.
Follows a sequential element-by-element access approach.	Follows a divide and conquer mathematical approach.
Worst-case time complexity is significantly higher at $O(n)$.	Worst-case time complexity is significantly lower at $O(\log n)$.
Highly efficient and preferred for small datasets.	Highly efficient and exclusively preferred for large datasets.
Can be easily used on both arrays and linked lists.	Best suited for arrays with direct index access.
Easy to implement and utilize on multidimensional arrays.	Complex to implement and utilize on multidimensional arrays.

"Linear search reads the whole book to find a word.
Binary search opens the dictionary exactly in the middle."