Searching Algorithms Week 2.1

Mayank Pratap Singh

July 16, 2024

1 Introduction

Linear search is a simple and intuitive search algorithm that checks each element in the array sequentially until the target element is found or the end of the array is reached. It is used for unsorted arrays or when a simple search is required.

1.1 Algorithm Explanation

Given an array `arr` and a target value, the linear search algorithm iterates through each element of the array to find the target.

1.2 Algorithm Steps

1. Start from the first element and iterate through the array.
2. Compare each element with the target value.
3. If the element matches the target, return the index.
4. If the end of the array is reached and no match is found, return -1.

1.3 Time and Space Complexity Analysis

- **Time Complexity:** $O(n)$ because in the worst case, the algorithm may need to check every element in the array.
- **Space Complexity:** $O(1)$ because it requires a constant amount of additional memory space, We are not using extra space.

1.4 Python Implementation

Here is the implementation of the linear search algorithm in Python:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

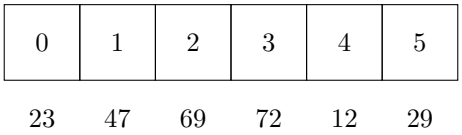

Figure 1: Array Representation

Linear search is an efficient method for searching unsorted arrays, offering a straightforward approach to finding target elements. Its simplicity and ease of implementation make it a fundamental concept in computer science.

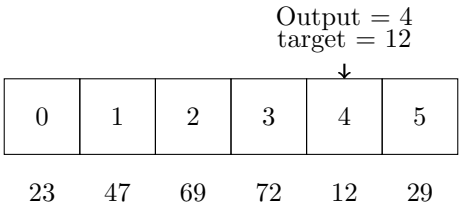

Figure 2: Finding Target 12 in the Array

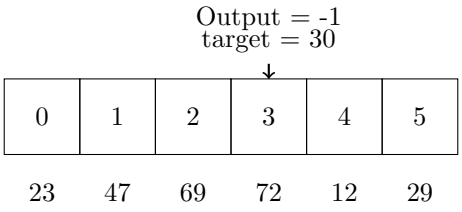

Figure 3: Searching for a Non-existing Target 30

2 Avoid Build-In Functions

In data structure and algorithm (DSA) interviews, it is crucial to demonstrate your understanding of fundamental concepts and your ability to implement algorithms manually. While Python provides built-in functions like `index()` and `sort()`, relying on these during an interview can indicate a lack of depth in your knowledge,and why it's important to write algorithms manually and how to optimize search algorithms effectively.

Built-in functions such as `index()` for searching or `sort()` for sorting are highly optimized for performance and convenience. However, in a DSA interview, the primary goal is to showcase your problem-solving skills and understanding of the underlying principles.

2.1 The `index()` Function

The `index()` function performs a linear search to find the first occurrence of a specified value in a list. Here's an example:

Listing 1: Using the index() function

```
arr = [23, 47, 69, 72, 12, 29]
index = arr.index(12)  # Returns 4
```

Why Avoid It? - It hides the complexity of the search algorithm. - It does not demonstrate your ability to implement the search logic manually.

2.2 The `sort()` Function

The `sort()` function sorts the list in place using Timsort, a highly efficient sorting algorithm.

Listing 2: Using the sort() function

```
arr.sort()  # Sorts the list in-place
# arr becomes [12, 23, 29, 47, 69, 72]
```

Why Avoid It? - It abstracts away the details of sorting algorithms. - It prevents you from showing your understanding of different sorting techniques.

3 Binary Search

Binary Search is a highly efficient algorithm used for finding an element in a sorted array. The array must be sorted in either ascending or descending order for the binary search to work correctly. The algorithm repeatedly divides the search interval in half, making it faster than linear search.

3.1 Algorithm Explanation

Given a sorted array and a target value, the binary search algorithm finds the position of the target value.

3.2 Constraints

- The array must be sorted. - The array can be in ascending or descending order.

3.3 Example

Consider the sorted array:

0	1	2	3	4	5	6	7
2	4	8	12	27	45	49	52

We are looking for the target value 45.

3.4 Steps of the Algorithm

1. Initialize the start and end indices: `start = 0` and `end = 7`.
2. Calculate the middle index:

$$\texttt{mid} = \left\lfloor \frac{\texttt{start} + \texttt{end}}{2} \right\rfloor$$

   For the first iteration: `mid` = 3.

3. Compare the middle value with the target:
   - If `arr[mid] == target`, return `mid`.
   - If `arr[mid] < target`, search in the right half: `start = mid + 1`.
   - If `arr[mid] > target`, search in the left half: `end = mid - 1`.
4. Repeat the process until `start  end`.

3.5 Binary Search Implementation

Listing 3: Binary Search Implementation

```
def binary_search(arr, target):
    start, end = 0, len(arr) - 1
    while start <= end:
        mid = (start + end) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            start = mid + 1
        else:
            end = mid - 1
    return -1

arr = [2, 4, 8, 12, 27, 45, 49, 52]
index = binary_search(arr, 45)  # Returns 5
```

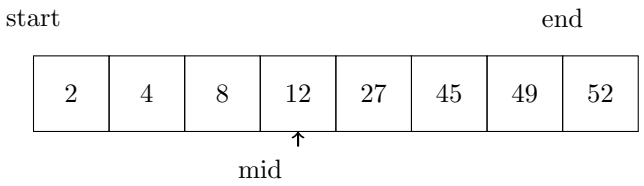

Figure 4: Initial State: start = 0, end = 7, mid = 3

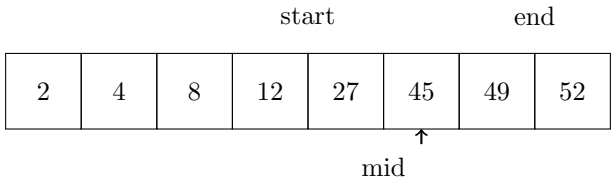

Figure 5: Updated State: start = 4, end = 7, mid = 5

.

3.6 Time and Space Complexity

- **Time Complexity:** $O(\log n)$, because we reduce the search space by half in every iteration.
- **Space Complexity:** $O(1)$, because we use a constant amount of additional space.

3.7 Overflow Consideration

To avoid overflow when calculating the middle index, use the following formula:

$$\texttt{mid} = \texttt{start} + \frac{\texttt{end} - \texttt{start}}{2}$$

This approach prevents the potential overflow that could occur with:

$$\texttt{mid} = \frac{\texttt{start} + \texttt{end}}{2}$$

3.8 Ternary Search

Ternary Search is another search algorithm that divides the array into three parts. Although it has a time complexity of $O(\log_3 n)$, it is computationally more expensive and not commonly used in practice.

3.9 Comparison with Binary Search

While ternary search divides the search space into more parts, the constant factors in its time complexity make it less efficient compared to binary search. Hence, binary search is generally preferred.

3.10 Conclusion

Binary Search is a crucial algorithm for efficiently finding elements in a sorted array. Understanding and implementing this algorithm manually is essential for demonstrating problem-solving skills during DSA interviews. While built-in functions offer convenience, writing the algorithm from scratch shows a deep understanding of the underlying principles. Additionally, the formula to calculate the middle index, `start + (end - start) // 2`, should be used to avoid overflow issues.

4 Modified Binary Search for Finding the First Infinite Value

Binary Search is typically used for searching in sorted arrays. However, we can modify the binary search algorithm to solve problems even in unsorted arrays under certain conditions. Here, we tackle a hypothetical question: finding the first occurrence of an infinite value in a highly unsorted array.

4.1 Hypothetical Problem

Given a highly unsorted array where the initial values are numeric but eventually become infinite, we need to find the index of the first infinite value.

4.2 Example Array

Consider the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
7	9	1	12	14	17	23	20	∞	∞	∞	∞	∞	∞

We need to find the index of the first infinite value.

5 Approach

5.1 Linear Search

A simple approach is to use a linear search:

Listing 4: Linear Search Implementation

```
def linear_search(arr):
    for i in range(len(arr)):
        if arr[i] == float('inf'):
            return i
    return -1
```

Time Complexity: $O(n)$
Drawback: This brute force approach is not efficient for large arrays.

5.2 Modified Binary Search

We can implement a modified version of binary search to solve this problem more efficiently, even though the array is not sorted.

1. Initialize `start = 0` and `end = n-1`.
2. Calculate `mid` as:

$$\texttt{mid} = \left\lfloor \frac{\texttt{start} + \texttt{end}}{2} \right\rfloor$$

3. If `arr[mid]` is not infinite, move to the right half: `start = mid + 1`.
4. If `arr[mid]` is infinite but `arr[mid - 1]` is also infinite, move to the left half: `end = mid - 1`.
5. If `arr[mid]` is infinite and `arr[mid - 1]` is not infinite, `mid` is the index of the first infinite value.
6. Repeat until `start  end`.

5.3 Modified Binary Search Implementation

Listing 5: Modified Binary Search Implementation

```
def modified_binary_search(arr):
    start, end = 0, len(arr) - 1
    while start <= end:
        mid = start + (end - start) // 2
        if arr[mid] != float('inf'):
            start = mid + 1
        else:
            if mid == 0 or arr[mid - 1] != float('inf'):
                return mid
            end = mid - 1
    return -1

arr = [7, 9, 1, 12, 14, 17, 23, 20, float('inf'), float('inf'), float('inf'), fl
index = modified_binary_search(arr)  # Returns 8
```

5.4 Explanation with Diagrams

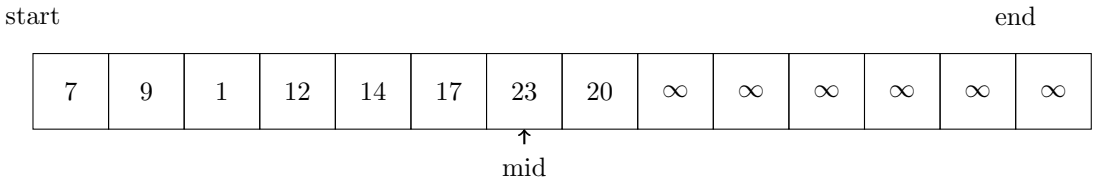

Figure 6: Initial State: start = 0, end = 13, mid = 6

5.5 Conclusion

The modified binary search algorithm provides an efficient solution to find the first occurrence of an infinite value in a highly unsorted array. By adjusting the binary search approach, we can achieve better performance than the brute force

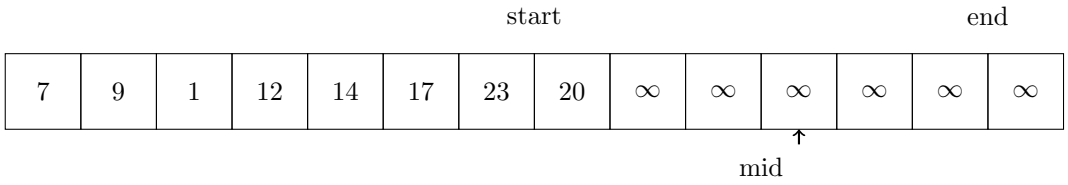

Figure 7: Updated State: start = 7, end = 13, mid = 10

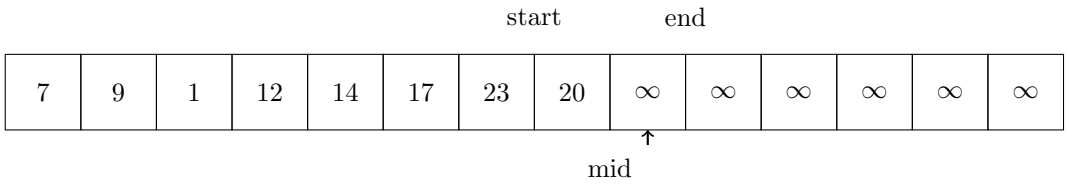

Figure 8: Final State: start = 7, end = 9, mid = 8

linear search method. Understanding and implementing these modifications is essential for demonstrating problem-solving skills during DSA interviews.

6 Bit Manipulation and Truth Tables

In terms of interviews, two important topics are Binary Search and Bit Manipulation. Bit manipulation involves operations on bits, which are fundamental units of information in computing, represented as 0 or 1. A byte consists of 8 bits.

Bit manipulation is crucial for competitive programming and interviews because code that operates on bits can be much faster than code that operates on higher-level representations such as decimal numbers.

7 Truth Tables

A truth table is a mathematical table used to determine the output of a particular logical operation based on all possible combinations of its inputs. Common logical operations include AND, OR, NOT, and XOR.

7.1 AND Operation

The AND operation returns true only if both inputs are true.

A	B	Output
1	0	0
0	0	0
0	1	0
1	1	1

Table 1: Truth Table for AND Operation

7.2 OR Operation

The OR operation returns true if at least one of the inputs is true.

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table 2: Truth Table for OR Operation

7.3 NOT Operation

The NOT operation inverts the input value.

A	Output
1	0
0	1

Table 3: Truth Table for NOT Operation

7.4 XOR Operation

The XOR (exclusive OR) operation returns true if the inputs are different.

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 4: Truth Table for XOR Operation

In terms of variables, XOR has some useful properties:

- $x \oplus x = 0$
- $0 \oplus x = x$

7.5 Explanation with Diagrams

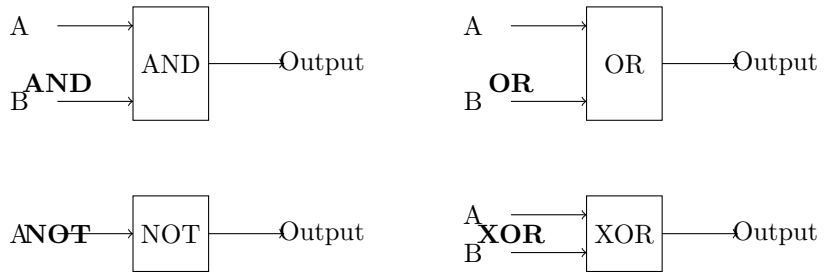

Figure 9: Logical Operation Diagrams

7.6 Conclusion

Understanding bit manipulation and truth tables is essential for solving complex problems efficiently in competitive programming and technical interviews. The logical operations AND, OR, NOT, and XOR form the foundation of these manipulations, and their properties can be leveraged to write optimized and faster code.

8 Left Shift and Right Shift Operations

Bitwise shift operations are crucial for efficient computation in many programming tasks, especially in competitive programming and technical interviews. Understanding how left shift and right shift work can help optimize code performance significantly.

9 Left Shift Operation

The left shift operation (<<) shifts the bits of a number to the left by a specified number of positions, filling the vacated positions with zeros.

9.1 Decimal to Binary Conversion

To understand left shift, let's first convert a decimal number to binary.

$$5_{10} \rightarrow 101_2$$

2	5
2	2 remainder 1
	1 remainder 0

The binary representation of 5 is 101_2.

9.2 Binary to Decimal Conversion

Now, let's convert a binary number back to decimal.

$$101_2 \rightarrow 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 0 + 1 = 5_{10}$$

Another example:

$$1010_2 \rightarrow 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 0 + 2 + 0 = 10_{10}$$

9.3 Left Shift Example

Consider the binary number 101_2 (which is 5 in decimal):

$$5 \ll 1 \rightarrow 101_2 \ll 1 = 1010_2 = 10_{10}$$

Performing another left shift:

$$10 \ll 1 \rightarrow 1010_2 \ll 1 = 10100_2 = 20_{10}$$

Note: Left shift operation effectively multiplies the number by 2.

- $5 \ll 1 = 10$
- $10 \ll 1 = 20$
- $20 \ll 1 = 40$

10 Right Shift Operation

The right shift operation (>>) shifts the bits of a number to the right by a specified number of positions, filling the vacated positions with the sign bit (for signed numbers) or zeros (for unsigned numbers).

10.1 Right Shift Example

Consider the binary number 101_2 (which is 5 in decimal):

$$5 \gg 1 \rightarrow 101_2 \gg 1 = 010_2 = 2_{10}$$

Performing another right shift:

$$2 \gg 1 \rightarrow 010_2 \gg 1 = 001_2 = 1_{10}$$

Note: Right shift operation effectively divides the number by 2.

- $5 \gg 1 = 2$
- $2 \gg 1 = 1$

$$5\ (101) \longrightarrow \ll 1 \longrightarrow \cancel{1}0\ (1010)$$

$$10\ (1010) \longrightarrow \ll 1 \longrightarrow \cancel{20}\ (10100)$$

$$5\ (101) \longrightarrow \gg 1 \longrightarrow \cancel{2}\ (010)$$

$$2\ (010) \longrightarrow \gg 1 \longrightarrow \cancel{1}\ (001)$$

Figure 10: Left Shift and Right Shift Diagrams

10.2 Conclusion

Bitwise shift operations are powerful tools for optimizing computations. Left shifts effectively multiply the number by 2, while right shifts divide the number by 2. Understanding these operations can lead to more efficient code, especially in performance-critical applications.

11 Finding Repeated and Missing Elements Using XO

Given an array where one element is repeated and one element is missing, we aim to find both elements using bit manipulation techniques, specifically the XOR operation. This approach is highly efficient and optimized.

11.1 Problem Statement

Consider the array:

$$\text{Input} = [3, 1, 2, 5, 3]$$

We need to find the repeated element (3) and the missing element (4).

11.2 XOR Operation

The XOR operation has useful properties for solving this problem:

- $x \oplus x = 0$
- $x \oplus 0 = x$

11.3 Approach

1. Compute the XOR of all array elements and the numbers from 1 to n. 2. The result will be the XOR of the repeated and missing elements. 3. Find the rightmost set bit in the XOR result to partition the array into two groups. 4. XOR each group separately to find the repeated and missing elements.

11.4 Step-by-Step Solution

1. Compute XOR of all elements in the array and all numbers from 1 to n:

$$\text{XOR1} = 3 \oplus 1 \oplus 2 \oplus 5 \oplus 3$$

$$\text{XOR2} = 1 \oplus 2 \oplus 3 \oplus 4 \oplus 5$$

$$\text{XOR} = \text{XOR1} \oplus \text{XOR2} = (3 \oplus 3 \oplus 1 \oplus 1 \oplus 2 \oplus 2 \oplus 5 \oplus 5) \oplus 4 = 4 \oplus 3$$

2. Compute the XOR result:

$$4 \oplus 3 = 7 \quad \text{(in binary: 0111)}$$

3. Find the rightmost set bit (bit position where the bit is 1):

$$\text{Rightmost set bit} = \text{XOR\&} - \text{XOR}$$

For 7:

$$0111\&1001 = 0001 \quad \text{(bit position 1)}$$

4. Partition the array elements and numbers from 1 to n based on this bit:

$$\text{Group X} = \{3, 3, 1\} \quad \text{Group Y} = \{1, 2, 5, 4\}$$

5. Compute the XOR for each group separately:

$$\text{XOR}_X = 3 \oplus 3 \oplus 1 = 1$$

$$\text{XOR}_Y = 1 \oplus 2 \oplus 5 \oplus 4 = 4$$

6. Determine which is the repeated and which is the missing element:

   If x is in the array, x is the repeated element and y is the missing element.

$$\text{Repeated element} = 3, \quad \text{Missing element} = 4$$

11.5 Diagrammatic Representation

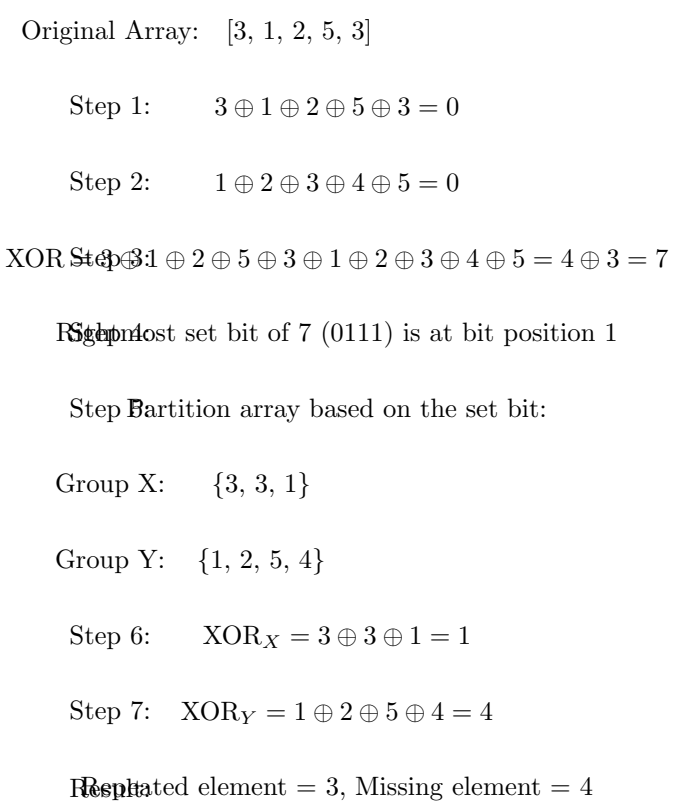

Figure 11: Finding Repeated and Missing Elements Using XOR

11.6 Conclusion

The XOR operation provides an efficient method to find the repeated and missing elements in an array. By leveraging the properties of XOR and bit manipulation, we can achieve an optimized solution with constant space complexity.