Binary Search: An In-Depth Exploration

Notes

November 26, 2024

Contents

1 Introduction **3**

2 Problem Statement **3**

3 Algorithm Explanation **3**

4 Variations of Binary Search **4**
- 4.1 Finding the First or Last Occurrence . . . 4
- 4.2 Finding the Insertion Point . . . 4
- 4.3 Finding the Peak Element . . . 4
- 4.4 Searching in a Rotated Sorted Array . . . 4

5 Code Implementations **4**
- 5.1 Java Implementation . . . 4
  - 5.1.1 Iterative Approach . . . 4
  - 5.1.2 Recursive Approach . . . 5
- 5.2 Python Implementation . . . 6
  - 5.2.1 Iterative Approach . . . 6
  - 5.2.2 Recursive Approach . . . 7

6 Dry Run of Examples **8**
- 6.1 Example 1 . . . 8
- 6.2 Example 2 . . . 9

7 Time and Space Complexity Analysis **9**
- 7.1 Time Complexity . . . 9
- 7.2 Space Complexity . . . 9

8 Applications of Binary Search **10**
- 8.1 Finding Square Roots . . . 10
- 8.2 Searching in a Rotated Sorted Array . . . 10
- 8.3 Finding the First or Last Occurrence . . . 10
- 8.4 Application in Lower and Upper Bounds . . . 10
- 8.5 Optimizing Solutions in Dynamic Programming . . . 10

9 Common Interview Questions **10**
9.1 LeetCode Problem 704: Binary Search 10
9.2 LeetCode Problem 33: Search in Rotated Sorted Array 10
9.3 LeetCode Problem 34: Find First and Last Position of Element in Sorted Array . . . 10
9.4 LeetCode Problem 367: Valid Perfect Square 10
9.5 LeetCode Problem 441: Arranging Coins 11
9.6 LeetCode Problem 852: Peak Index in a Mountain Array 11

10 Conclusion **11**

11 References **11**

1 Introduction

Binary Search is a fundamental algorithm in computer science used to efficiently locate a target value within a sorted array or list. Unlike linear search, which examines each element sequentially, binary search reduces the search space by half with each comparison, achieving logarithmic time complexity. This efficiency makes binary search a critical tool in various applications, including database querying, search engines, and solving algorithmic problems on platforms like LeetCode.

2 Problem Statement

Given a sorted (in ascending order) integer array `nums` of n elements and a target value `target`, write a function to search for `target` in `nums`. If `target` exists, return its index; otherwise, return `-1`.

Example 1:

- **Input**: `nums = [1,3,5,7,9,11]`, `target = 7`
- **Output**: `3`
- **Explanation**: The target value `7` is found at index `3`.

Example 2:

- **Input**: `nums = [2,4,6,8,10]`, `target = 5`
- **Output**: `-1`
- **Explanation**: The target value `5` is not present in the array.

Constraints:

- $1 \leq nums.length \leq 10^4$
- $-10^4 \leq nums[i] \leq 10^4$
- All elements in `nums` are unique.
- `nums` is sorted in ascending order.
- $-10^4 \leq target \leq 10^4$

3 Algorithm Explanation

Binary search operates on the principle of divide and conquer. Given that the array is sorted, the algorithm compares the target value to the middle element of the array:

1. **Initialize Pointers**: Set two pointers, `left` and `right`, to the start and end of the array, respectively.

2. **Find the Middle Element**: Calculate the middle index `mid` as `mid = left + (right - left) / 2`.
3. **Compare with Target**:
   - If `nums[mid]` equals `target`, return `mid`.
   - If `nums[mid]` is less than `target`, adjust the `left` pointer to `mid + 1`.
   - If `nums[mid]` is greater than `target`, adjust the `right` pointer to `mid - 1`.
4. **Repeat**: Continue the process until `left` exceeds `right`.
5. **Target Not Found**: If the target is not found, return `-1`.

4 Variations of Binary Search

Binary search can be adapted to solve various problems beyond simple search operations. Some common variations include:

4.1 Finding the First or Last Occurrence

When the array contains duplicate elements, binary search can be modified to find the first or last occurrence of a target value.

4.2 Finding the Insertion Point

Determine the index at which a target should be inserted to maintain the array's sorted order.

4.3 Finding the Peak Element

Identify a peak element in an array where a peak is defined as an element greater than its neighbors.

4.4 Searching in a Rotated Sorted Array

Locate a target value in an array that has been rotated at an unknown pivot.

5 Code Implementations

Below are implementations of binary search in both Java and Python, covering the basic search functionality.

5.1 Java Implementation

5.1.1 Iterative Approach

Description : The iterative approach uses a loop to repeatedly divide the search interval in half.

```java
public class BinarySearch {
    /**
     * Searches for a target value in a sorted array using binary search.
     * @param nums Sorted array of integers.
     * @param target The integer to search for.
     * @return The index of target if found; otherwise, -1.
     */
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length -1;

        while(left <= right){
            int mid = left + (right - left) / 2;

            if(nums[mid] == target){
                return mid;
            }
            else if(nums[mid] < target){
                left = mid +1;
            }
            else{
                right = mid -1;
            }
        }

        return -1; // Target not found
    }

    public static void main(String[] args) {
        BinarySearch solution = new BinarySearch();
        int[] nums1 = {1,3,5,7,9,11};
        int target1 = 7;
        System.out.println("Index of " + target1 + ": " + solution.search(
            nums1, target1)); // Outputs: 3

        int[] nums2 = {2,4,6,8,10};
        int target2 = 5;
        System.out.println("Index of " + target2 + ": " + solution.search(
            nums2, target2)); // Outputs: -1
    }
}
```

Listing 1: Java: Iterative Binary Search

5.1.2 Recursive Approach

Description : The recursive approach divides the problem into smaller subproblems by recursively searching in the left or right half of the array.

```java
public class BinarySearchRecursive {
    /**
```

```java
     * Searches for a target value in a sorted array using binary search recursively.
     * @param nums Sorted array of integers.
     * @param target The integer to search for.
     * @return The index of target if found; otherwise, -1.
     */
    public int search(int[] nums, int target) {
        return binarySearch(nums, target, 0, nums.length -1);
    }

    private int binarySearch(int[] nums, int target, int left, int right){
        if(left > right){
            return -1; // Target not found
        }

        int mid = left + (right - left) / 2;

        if(nums[mid] == target){
            return mid;
        }
        else if(nums[mid] < target){
            return binarySearch(nums, target, mid +1, right);
        }
        else{
            return binarySearch(nums, target, left, mid -1);
        }
    }

    public static void main(String[] args) {
        BinarySearchRecursive solution = new BinarySearchRecursive();
        int[] nums1 = {1,3,5,7,9,11};
        int target1 = 7;
        System.out.println("Index of " + target1 + ": " + solution.search(nums1, target1)); // Outputs: 3

        int[] nums2 = {2,4,6,8,10};
        int target2 = 5;
        System.out.println("Index of " + target2 + ": " + solution.search(nums2, target2)); // Outputs: -1
    }
}
```

Listing 2: Java: Recursive Binary Search

5.2 Python Implementation

5.2.1 Iterative Approach

Description : The iterative approach in Python follows the same logic as the Java iterative approach.

```python
class BinarySearch:
```

```
    def search(self, nums, target):
        """
        Searches for a target value in a sorted array using binary search.

        :param nums: List[int] - Sorted list of integers.
        :param target: int - The integer to search for.
        :return: int - The index of target if found; otherwise, -1.
        """
        left = 0
        right = len(nums) -1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                left = mid +1
            else:
                right = mid -1

        return -1  # Target not found

if __name__ == "__main__":
    solution = BinarySearch()
    nums1 = [1,3,5,7,9,11]
    target1 = 7
    print(f"Index of {target1}: {solution.search(nums1, target1)}")  # Outputs
        : 3

    nums2 = [2,4,6,8,10]
    target2 = 5
    print(f"Index of {target2}: {solution.search(nums2, target2)}")  # Outputs
        : -1
```

Listing 3: Python: Iterative Binary Search

5.2.2 Recursive Approach

Description : The recursive approach in Python mirrors the Java recursive approach.

```
class BinarySearchRecursive:
    def search(self, nums, target):
        """
        Searches for a target value in a sorted array using binary search
            recursively.

        :param nums: List[int] - Sorted list of integers.
        :param target: int - The integer to search for.
        :return: int - The index of target if found; otherwise, -1.
        """
        return self.binary_search(nums, target, 0, len(nums) -1)
```

```python

    def binary_search(self, nums, target, left, right):
        if left > right:
            return -1  # Target not found

        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            return self.binary_search(nums, target, mid +1, right)
        else:
            return self.binary_search(nums, target, left, mid -1)

if __name__ == "__main__":
    solution = BinarySearchRecursive()
    nums1 = [1,3,5,7,9,11]
    target1 = 7
    print(f"Index of {target1}: {solution.search(nums1, target1)}")  # Outputs
        : 3

    nums2 = [2,4,6,8,10]
    target2 = 5
    print(f"Index of {target2}: {solution.search(nums2, target2)}")  # Outputs
        : -1
```

Listing 4: Python: Recursive Binary Search

6 Dry Run of Examples

6.1 Example 1

Input: `nums = [1,3,5,7,9,11]`, `target = 7`

Execution Steps:

1. **Initialization**: `left = 0`, `right = 5`
2. **First Iteration**:
   - `mid` = $0 + (5 - 0)/2 = 2$
   - `nums[2]` = 5
   - Since $5 < 7$, update `left` to `mid + 1` = 3
3. **Second Iteration**:
   - `mid` = $3 + (5 - 3)/2 = 4$
   - `nums[4]` = 9
   - Since $9 > 7$, update `right` to `mid - 1` = 3
4. **Third Iteration**:

- `mid` = $3 + (3 - 3)/2 = 3$
- `nums[3]` = 7
- Found target at index 3

Output: 3

6.2 Example 2

Input: `nums = [2,4,6,8,10]`, `target = 5`
Execution Steps:

1. **Initialization**: `left = 0`, `right = 4`
2. **First Iteration**:
   - `mid` = $0 + (4 - 0)/2 = 2$
   - `nums[2]` = 6
   - Since $6 > 5$, update `right` to `mid - 1` = 1
3. **Second Iteration**:
   - `mid` = $0 + (1 - 0)/2 = 0$
   - `nums[0]` = 2
   - Since $2 < 5$, update `left` to `mid + 1` = 1
4. **Third Iteration**:
   - `mid` = $1 + (1 - 1)/2 = 1$
   - `nums[1]` = 4
   - Since $4 < 5$, update `left` to `mid + 1` = 2
5. **Termination**: `left = 2` > `right = 1`

Output: `-1`

7 Time and Space Complexity Analysis

7.1 Time Complexity

The time complexity of binary search is **O(log n)**, where `n` is the number of elements in the array. This is because with each comparison, the algorithm halves the search space.

7.2 Space Complexity

- **Iterative Approach**: `O(1)` space complexity, as it uses a constant amount of additional space.
- **Recursive Approach**: `O(log n)` space complexity due to the recursion stack.

8 Applications of Binary Search

Binary search is not limited to searching in arrays. It has diverse applications in various domains:

8.1 Finding Square Roots

Calculating the integer part of the square root of a number can be efficiently done using binary search.

8.2 Searching in a Rotated Sorted Array

Binary search can be adapted to find elements in arrays that have been rotated at an unknown pivot.

8.3 Finding the First or Last Occurrence

In sorted arrays with duplicate elements, binary search can be modified to find the first or last occurrence of a target value.

8.4 Application in Lower and Upper Bounds

Determining the lower and upper bounds of a target in a sorted array can be achieved using binary search.

8.5 Optimizing Solutions in Dynamic Programming

Binary search can be utilized to optimize certain dynamic programming solutions by reducing time complexity.

9 Common Interview Questions

Understanding binary search and its variations is crucial for technical interviews. Below are some commonly asked questions:

9.1 LeetCode Problem 704: Binary Search

Problem: Implement binary search on a sorted array.

9.2 LeetCode Problem 33: Search in Rotated Sorted Array

Problem: Search for a target value in a rotated sorted array.

9.3 LeetCode Problem 34: Find First and Last Position of Element in Sorted Array

Problem: Find the starting and ending position of a given target value in a sorted array.

9.4 LeetCode Problem 367: Valid Perfect Square

Problem: Determine if a number is a perfect square using binary search.

9.5 LeetCode Problem 441: Arranging Coins

Problem: Find the total number of complete rows of coins in a staircase.

9.6 LeetCode Problem 852: Peak Index in a Mountain Array

Problem: Find the peak index in a mountain array using binary search.

10 Conclusion

Binary search is a powerful algorithm that offers efficient search capabilities in sorted datasets. Its logarithmic time complexity makes it highly suitable for large datasets where linear search would be impractical. Understanding binary search not only aids in solving search-related problems but also enhances problem-solving skills applicable to a wide range of algorithmic challenges. Mastery of binary search and its variations is essential for excelling in technical interviews and developing optimized software solutions.

11 References

- Wikipedia: Binary Search Algorithm
- GeeksforGeeks: Binary Search
- LeetCode: Binary Search Problems
- YouTube: Binary Search Explained