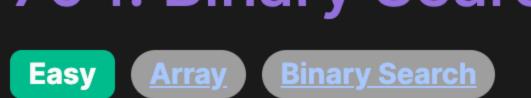
704. Binary Search



Problem Description

This problem asks us to create a function that takes two inputs: an array of integers nums sorted in ascending order, and an integer target. The goal is to search for the target value within the nums array. If the target is found, the function should return the index of the target in the array. If target is not in the array, the function should return -1.

The critical constraint is that the algorithm used to search for the target value must have a runtime complexity of 0(log n). This is a strong hint that a <u>binary search</u> algorithm should be implemented because binary search has a logarithmic time complexity

is a strong hint that a <u>binary search</u> algorithm should be implemented because binary search has a logarithmic time complexit and typically operates on sorted data.

To satisfy the <code>0(log n)</code> runtime complexity requirement, we use a <u>binary search</u> algorithm, which effectively splits the array in

Intuition

half during each iteration to narrow down the possible location of the target value.

Here are the key steps of the intuition behind the <u>binary search</u> solution approach:

1. Initialize two pointers, left and right, which represent the start and end of the range within the array we're currently considering. Initially, left is 0 and right is len(nums) - 1.

- 2. Enter a while loop which runs as long as left is less than right. This loop will continue until we either find the target or the search space is empty.
- 3. Determine the mid point index by averaging left and right. The >> 1 operation is a bitwise shift that divides the sum by 2, efficiently finding the midpoint.
- 4. Compare the middle element nums[mid] with the target. If nums[mid] is greater than or equal to the target, we move the right pointer to mid, as the target, if it exists, must be to the left of mid.
- as the target, if it exists, must be to the left of mid.

 5. If nums [mid] is less than the target, the target, if it exists, must be to the right of mid, so we update left to mid + 1.
- 6. The loop continues, halving the search space each time, until left becomes equal to right.

 7. After exiting the loop, we check if nums[left] is equal to target. If it is, we return left, which is the index of the target. If not, we return -1 to
- This solution approaches the problem using classic <u>binary search</u>, ensuring the runtime complexity meets the required <code>0(log n)</code> by continuously halving the search space, which is characteristic of logarithmic runtime algorithms.

Solution Approach

The implementation of the solution uses a <u>binary search</u> algorithm. Binary search is a classic algorithm in computer science for

efficiently searching for an element in a sorted list. The algorithm repeatedly divides the search interval in half, and because the

Initialize two pointers: left is set to 0, which is the first index of nums, and right is set to len(nums) - 1, which is the last

search space has been fully narrowed down. We're searching for the exact placement of the target or concluding that it's

equivalent to mid = (left + right) // 2 but is more efficient. The mid variable represents the index of the element in the

list is sorted, it can quickly determine if the element can only be in either the left or the right half of the list. Here's a step-by-step walkthrough of the implementation based on the provided code:

indicate that the target is not present in the array.

index of nums.

2. Enter a while loop that continues as long as left is less than right. This ensures that we don't stop searching until the

not in the array.

3. Calculate the mid-point index mid by adding left and right, then shifting the result to the right by one bit (>>> 1). This is

middle of the current search space.

4. The if statement if nums[mid] >= target: evaluates whether the middle element is greater than or equal to the target. If so, that means the target, if it exists, must be at mid or to the left of mid within the search space. In response, right is

updated to mid, narrowing the search to the left half of the current search space (inclusive of the mid point).

mid + 1, now searching to the right of the original mid point.

6. The loop continues until left is equal to right, which means we can no longer divide the search space, and we must check if we have found the target.

If the middle element is less than the target, the target can only be to the right of mid. Therefore, we update left to be

Once the loop exits, we have one final check to perform. We compare the element at the index left with the target. If

they're equal, then left is the index where target is found in the array nums. Otherwise, if they're not equal, this means the

- target is not present in the array, and we return -1.

 No other data structures are utilized in this implementation as <u>binary search</u> operates directly on the sorted input list, and no
- the expected outcome within the required 0(log n) time complexity.

 Example Walkthrough

This solution leverages the efficiency of binary search, taking advantage of the sorted nature of the input array nums to deliver

nums = [1, 3, 5, 6, 9] target = 5

Calculate mid index: mid = (0 + 4) >> 1 = 4 >> 1 = 2 (bitwise shift of 4 to the right by 1 is the same as integer division by 2).

1. Initialize two pointers: left = 0 (beginning of the array) and right = 4 (index of the last element in the array).

The element at index 2 in nums is 5.

1. Check if nums[mid] >= target. In this case, nums[2] is 5, which is equal to target (5). Therefore, we found our target, and right is updated to mid.

Here's how the binary search algorithm would be applied step by step:

Our while loop starts because left (0) is less than right (4).

additional memory is required other than a few variables for indices.

Let's consider an example to illustrate the solution approach described above.

Suppose we have the following array of integers nums and a target integer target:

6. The loop concludes, and we no longer need to narrow down our search.

to the target value. We return left, which is 2, the index where target is found.

By following this method, we efficiently find our target using a binary search approach, ensuring the search process conforms to the logarithmic runtime, $0(\log n)$.

The array is not further divided since we already located the target. Therefore, the while loop does not continue further.

Finally, outside of the loop, we compare nums[left] with target. Here, nums[left] is nums[2] which is 5, and this is equal

left, right = 0, len(nums) - 1
Use a binary search algorithm
while left < right:</pre>

Calculate the middle index using bit shifting (equivalent to floor division by 2)

If the middle element is greater than or equal to the target

Narrow the search to the left half including the middle element

Narrow the search to the right half excluding the middle element

After exiting the loop, left should be the smallest index of the target value # Check if the element at the left index is the target return left if nums[left] == target else -1

print(result) # Output: 2

else:

Solution Implementation

def search(self, nums: List[int], target: int) -> int:

Initialize the left and right pointers

mid = (left + right) >> 1

if nums[mid] >= target:

left = mid + 1

If the element is not found, return -1

right = mid

result = sol.search([1, 2, 3, 4, 5], 3)

// Perform binary search

// Calculate the midpoint to avoid potential overflow

// we need to move the right boundary to the middle

// After the loop ends, left should point to the smallest

// otherwise return -1 as the target is not found

* @return {number} The index of the target if found; otherwise, -1.

return (nums[left] == target) ? left : -1;

* @param {number} target - The value to search for.

// number not smaller than target. Check if it's the target,

* Searches for a target value in a sorted array of numbers using binary search.

// Continue to search while the left pointer has not surpassed the right.

* @param {number[]} nums - Sorted array of numbers where we search for the target.

// If the middle element is greater or equal to the target,

// The target is greater than the middle element,

// move left boundary one step to the right of mid

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

if (nums[mid] >= target) {

right = mid;

left = mid + 1;

while (left < right) {</pre>

} else {

while (left < right) {</pre>

from typing import List

class Solution:

Example usage

Java

sol = Solution()

class Solution {

Python

```
public int search(int[] nums, int target) {
        // Initialize the starting index of the search range.
        int left = 0;
        // Initialize the ending index of the search range.
        int right = nums.length - 1;
        // Continue searching while the range has more than one element.
        while (left < right) {</pre>
            // Calculate the middle index of the current range.
            int mid = left + (right - left) / 2;
            // If the middle element is greater than or equal to the target,
            // narrow the search range to the left half (including the middle element).
            if (nums[mid] >= target) {
                right = mid;
            } else {
                // If the middle element is less than the target,
                // narrow the search range to the right half (excluding the middle element).
                left = mid + 1;
       // At this point, left is the index where the target may be if it exists.
        // Check if the element at the 'left' index is the target.
        // If it is, return the index. Otherwise, return -1 indicating the target is not found.
        return nums[left] == target ? left : -1;
C++
#include <vector> // Include the vector header for using the std::vector container
class Solution {
public:
    int search(std::vector<int>& nums, int target) {
        int left = 0; // Initialize left boundary of the search
        int right = nums.size() - 1; // Initialize right boundary of the search
```

*/ const search = (nums: number[]. target: number): number => { // Initialize left and right pointers for the binary search. let left: number = 0; let right: number = nums.length - 1;

};

TypeScript

```
// Calculate the middle index using bitwise right shift
       // equivalent to Math.floor((left + right) / 2)
       const mid: number = (left + right) >> 1;
       // If the middle element is greater or equal to target, move the right pointer to the middle.
       if (nums[mid] >= target) {
            right = mid;
        } else {
           // If the middle element is less than the target, move the left pointer past the middle.
            left = mid + 1;
   // Once the search space is narrowed down to a single element,
   // check if it's equal to the target and return the appropriate index or -1.
   return nums[left] === target ? left : -1;
};
// Example usage:
const nums: number[] = [1, 3, 5, 7, 9];
const target: number = 7;
const result: number = search(nums, target); // result should be 3 as nums[3] = 7
from typing import List
class Solution:
   def search(self, nums: List[int], target: int) -> int:
       # Initialize the left and right pointers
        left, right = 0, len(nums) - 1
       # Use a binary search algorithm
       while left < right:</pre>
           # Calculate the middle index using bit shifting (equivalent to floor division by 2)
           mid = (left + right) >> 1
           # If the middle element is greater than or equal to the target
           if nums[mid] >= target:
                # Narrow the search to the left half including the middle element
                right = mid
           else:
                # Narrow the search to the right half excluding the middle element
                left = mid + 1
       # After exiting the loop, left should be the smallest index of the target value
       # Check if the element at the left index is the target
       return left if nums[left] == target else -1
```

result = sol.search([1, 2, 3, 4, 5], 3) # print(result) # Output: 2

Time Complexity:

Time and Space Complexity

Example usage

sol = Solution()

If the element is not found, return -1

The given Python code performs a binary search on the array nums by repeatedly dividing the search interval in half. The search ends when left and right indices converge, at which point it either finds the target or concludes that the target is not in the array.

The while loop will run until left is equal to right. On each iteration, the interval size is halved, which means the time complexity is logarithmic with regard to the size of the array nums. Hence, the time complexity is $O(\log n)$, where n is the length of nums.

Space Complexity: The space complexity of the code is 0(1) since it uses a constant amount of extra space. The variables left, right, and mid

only take up a fixed amount of space regardless of the input list size.