702. Search in a Sorted Array of Unknown Size

```
Medium <u>Array</u> <u>Binary Search</u>
                                       Interactive
```

In this interactive problem, you are given access to a sorted array of unique elements, but you do not know its size. To interact with this array, you can use the ArrayReader interface, which has a function get(i) that will return the value at the ith index (0indexed) if it's within the bounds of the array. If i is outside the bounds of the array, get(i) will return 2^31 - 1, which serves

as an indication of being outside the array's boundaries. You are also provided an integer target, and your task is to find the index k in the hidden array where secret[k] == target. If the target is not present in the array, you should return -1. The requirement is to accomplish this with a solution that has a runtime complexity of O(log n), which implies that a binary search algorithm should be used.

Problem Description

Intuition

Since we don't know the array's size, we can't directly apply binary search. However, the problem statement helps by stating that

To find this range, we could intuitively start with an initial guess and then expand our search range exponentially until we find an

the ArrayReader.get(i) function will return a very large number (specifically, 2^31 - 1) if we access an index beyond the array's bounds. This allows us to first locate the possible range where the target could reside.

out-of-bound value. Once we identify an index that returns an out-of-bound value, we know that the array's size is less than this index. We could then perform a binary search between the start of the array and the identified out-of-bound index. The solution code, however, does not initially find the bounds of the array. Instead, it assumes a range of indexes between 0 and

20,000, which should be more than enough to cover typical constraints of the problem since we are told that get(i) will return 2³¹ - 1 for out-of-bound accesses. With this assumption, the solution applies a binary search algorithm right away. During the binary search, if reader.get(mid) is greater than or equal to target, it means the target is located at mid or to the

left of mid (since the array is sorted), so it narrows the search range's upper bound to mid. If reader.get(mid) is less than target, the target can only be to the right of mid, and it narrows the search range's lower bound to mid + 1. This process is repeated until the lower and upper bounds converge to the smallest range possible that might contain the target.

Finally, having honed in on a potential index where target could exist, the solution checks if reader.get(left) is indeed equal to target. If it is, left is returned as the index where target was found. If reader.get(left) does not match target, the function

In the standard binary search, the middle element is typically compared to the target value, and based on that comparison, you can decide if the target would be in the left half or the right half of the current segment. Here's how the implemented solution strategically applies binary search to the given problem: Initial Range Setting: Given that the exact size of the array is unknown, the solution begins by setting a broad search range

The solution uses a variation of <u>binary search</u> to efficiently find the target value within a sorted array of unknown size. Binary

search is a divide-and-conquer algorithm that operates by repeatedly dividing in half the portion of the list that could contain the

number (specifically 2^31 - 1) if i is out of the array bounds, this large upper bound is safe to assume for most inputs as per

Example Walkthrough

to 20,000.

Start Loop:

End Loop.

0

class Solution:

Parameters:

Returns:

left, right = 0, 20000

while left <= right:</pre>

Solution Approach

returns -1, indicating that the target is not in the array.

item, until there are no more segments left or the item is found.

problem constraints. Binary Search Application: While left is less than right, the solution continuously narrows the search range based on the value returned by the ArrayReader.get(mid):

for the binary search, with left set to 0 and right set to 20,000. Since we know ArrayReader.get(i) will return a very large

o If ArrayReader.get(mid) is greater than or equal to the target, it means the target cannot be to the right of mid (since the array is sorted), so the upper bound of the range (right) is set to mid. o If ArrayReader.get(mid) is less than the target, then the target must be to the right of mid, and thus the lower bound of the range (left) is set to mid + 1.

+ right) / 2 but is often faster in many programming languages due to bit shifting.

array, but we need to find the index of the target value 10 using the solution approach described.

Calculate mid as (left + right) >> 1, which is initially (0 + 20,000) >> 1 = 10,000.

Target Verification: We verify the value found at the index 3 to ensure it is indeed the target.

To find the mid index, the average of left and right is calculated using (left + right) >> 1, which is equivalent to (left

Target Verification: After exiting the while loop, the left variable should either be at the location of the target value or at

the smallest index greater than the target. To verify whether the target has been found, ArrayReader.get(left) is

compared to the target: • If they match, the index left is returned, indicating the location of the target. ○ If they do not match, it means the target is not present in the array, so -1 is returned.

complexity, which is a requirement as per the problem. Moreover, in practice, an initial range could be determined dynamically by starting with a small range and exponentially expanding it (doubling it each time) until an out-of-bound access occurs. This step can optimize the solution further if the potential maximum size of the array is significant compared to the actual size of the array.

Let's say we have an ArrayReader that provides access to a sorted array: [3, 5, 7, 10, 15, 20]. We don't know the size of this

Initial Range Setting: We start by establishing an initial large search range for the binary search. We set left to 0 and right

■ Check ArrayReader.get(10,000), it returns 2^31 - 1, which means 10,000 is out of the array bounds. Since the value is very high, we

Continue the loop with left set to 0 and right set now to an index within the array bounds. For this example, let's

It's important to understand that binary search reduces the search space by half with each step, leading to the O(log n) runtime

set the value of right to mid, which is now 10,000. ■ Recalculate mid as (0 + 10,000) >> 1 = 5,000, and so on, until we get to a mid value inside the bound of the array.

Binary Search Application: Now we begin the binary search process.

Here's how the solution would work with this example:

- assume that after several steps (not shown for brevity), we have narrowed down the bounds to left = 2 and right = 5. \blacksquare mid is (2 + 5) >> 1 = 3. ArrayReader.get(3) returns 10, which is the target value. ■ Since ArrayReader.get(mid) == target, the search is over, and we skip to the Target Verification step.
- quickly compared to a linear search, meeting the complexity requirement O(log n).

pass # Placeholder for the get method of ArrayReader.

Search for a target value in a sorted array of unknown length.

We use a binary search approach, defining the initial range based

int: The index of the target value or -1 if the target is not found.

Initialize the left and right pointers for binary search.

Perform binary search within the bounds of left and right.

Please note that the ArrayReader class is not fully implemented above.

public int search(ArrayReader reader, int target) {

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

return reader.get(left) == target ? left : -1;

// by comparing the value at left's index to the target value.

* Interface representing an array reader that supplies a `get` method to access elements at a given index.

* Searches for a target element in a sorted array with an unknown size using the ArrayReader.

* Implements a binary search algorithm assuming a very large array where negative elements

// Return the index if it's the target, or -1 if not.

* @param reader - The array reader providing access to elements.

* @param target - The target value to search for.

// const index: number = search(someReader, targetValue);

pass # Placeholder for the get method of ArrayReader.

def search(self, reader: ArrayReader, target: int) -> int:

def get(self, index: int) -> int:

while left <= right:</pre>

Time and Space Complexity

Calculate the middle index.

class ArrayReader:

class Solution:

return reader.get(left) == target ? left : -1;

if (reader.get(mid) >= target) {

// Initialize the left pointer at the start of the array.

// Binary search algorithm to find the target value.

reader (ArrayReader) - an instance of ArrayReader to access array elements

Retrieve the value at the middle index from the reader interface.

If the middle value is greater than the target, adjust right boundary.

If the loop ends and we haven't returned, the target is not in the array.

If the middle value matches the target, return the index.

def search(self, reader: ArrayReader, target: int) -> int:

on given constraints (here 0 to 20000).

Calculate the middle index.

mid_value = reader.get(mid)

if mid value == target:

elif mid value > target:

return mid

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

target (int) — the target value to search for

ArrayReader.get(3) is equal to 10, which matches the target.

We return the index 3 because this is where the target was found.

Solution Implementation **Python** class ArrayReader: def get(self, index: int) -> int:

In this example, after several iterations of narrowing the search bounds and recomputing mid, we have successfully found the

target value of 10 at index 3 using binary search. This demonstrates the efficiency of binary search in reducing the search range

right = mid - 1# If the middle value is less than the target, adjust left boundary. else: left = mid + 1

return -1

int left = 0;

int right = 20000;

} else {

C++

};

/**

/**

TypeScript

interface ArrayReader {

get(index: number): number;

* denote the sequence has ended.

while (left < right) {</pre>

right = mid;

left = mid + 1;

```
# It only serves a placeholder to illustrate how the get method signature should look.
# Since the original problem was rewritten to match Python 3 syntax and best practices.
# real implementation of get method from ArrayReader class or real input would be required to run the code.
Java
/**
 * This is the implementation of a solution for finding the target element
 * in a sorted array with unknown size using an API provided by ArrayReader.
class Solution {
    /**
     * Searches for the target value in a sorted array with unknown size.
     * @param reader An object with an API to get elements at a specific index.
     * @param target The value to search for in the array.
     * @return The index of the target if found, otherwise returns -1.
```

// Initialize the right pointer with a high enough index to ensure the target is within range.

// The value 20000 is used as an arbitrary high index; in practice, this should cover the array length.

// Calculate the mid index by averaging left and right, shifting right by 1 avoids potential overflow.

// Update the right boundary to mid, as the target can either be at mid or to the left of mid.

// If the value at mid is greater than or equal to the target, narrow the search to the left half.

// After the loop, left should point to the target if it exists. Return the index if the target is found.

// Update the left boundary to mid + 1, excluding mid from the next search range.

```
* This function searches for a target element in a sorted array with unknown size.
* The ArrayReader API is used to access elements by index without knowing the array's length.
* @param reader The array reader that provides access to the elements.
* @param target The target value to search for in the array.
* @return The index of the target if found, otherwise -1.
*/
class Solution {
public:
   int search(const ArrayReader& reader, int target) {
       // Initialize boundaries left and right.
       // The right boundary is set to 20000 as the problem statement indicates that
       // array values will be less than 10000, so 20000 is safely outside the range.
       int left = 0;
       int right = 20000;
       // Run binary search algorithm to find the position of the target.
       while (left < right) {</pre>
           // Calculate the midpoint of the current left and right boundaries.
            int mid = left + (right - left) / 2; // Use a different method to avoid potential overflow.
           // Get the value at the midpoint index from the reader.
            int val = reader.get(mid);
           // If the midpoint value is greater than or equal to the target, move the right boundary to mid,
           // otherwise, move the left boundary to mid + 1.
            if (val >= target) {
                right = mid;
            } else {
                left = mid + 1;
       // Once the search is over, check if the left boundary is at the target's position
```

```
* @returns The index of the target if found, otherwise -1.
function search(reader: ArrayReader, target: number): number {
  let left: number = 0;
  // Artificially setting the right boundary very high since the array size is unknown,
 // but we know the range of possible values based on the problem constraints.
  let right: number = 20000;
  while (left < right) {</pre>
   // Calculate the middle index to partition the array into halves.
   let mid: number = left + Math.floor((right - left) / 2);
   // Retrieve the value at the middle index using the reader.
    let valueAtMid: number = reader.get(mid);
   if (valueAtMid >= target) {
     // If value at mid is greater or equal to the target, we shrink the right boundary.
      right = mid;
   } else {
      // If value at mid is less than the target, we adjust the left boundary.
      left = mid + 1;
  // After exiting the loop, left should be at the smallest index whose value is at least the target.
 // Check if the actual value at the index is equal to the target and return the index if so.
  return reader.get(left) === target ? left : -1;
// Example usage:
// Assume there exists an ArrayReader instance - someReader,
// and a target value - targetValue.
// You can call the search function as follows:
```

```
on given constraints (here 0 to 20000).
Parameters:
reader (ArrayReader) — an instance of ArrayReader to access array elements
target (int) - the target value to search for
Returns:
int: The index of the target value or -1 if the target is not found.
# Initialize the left and right pointers for binary search.
left, right = 0, 20000
```

Perform binary search within the bounds of left and right.

Search for a target value in a sorted array of unknown length.

We use a binary search approach, defining the initial range based

```
mid = left + (right - left) // 2
            # Retrieve the value at the middle index from the reader interface.
            mid_value = reader.get(mid)
            # If the middle value matches the target, return the index.
            if mid value == target:
                return mid
            # If the middle value is greater than the target, adjust right boundary.
            elif mid value > target:
                right = mid - 1
           # If the middle value is less than the target, adjust left boundary.
            else:
                left = mid + 1
        # If the loop ends and we haven't returned, the target is not in the array.
        return -1
# Please note that the ArrayReader class is not fully implemented above.
 It only serves a placeholder to illustrate how the get method signature should look.
# Since the original problem was rewritten to match Python 3 syntax and best practices,
# real implementation of get method from ArrayReader class or real input would be required to run the code.
```

space to be searched is halved at each iteration. The space complexity is 0(1) since the algorithm uses a constant amount of extra space, with variables like left, right, and mid that do not depend on the size of the input.

The time complexity of the provided code is O(log n) where n represents the position of the target value or the position where

the reader's get method returns INT_MAX signaling the end of the array. This is due to the binary search approach where the