1064. Fixed Point

Easy Array Binary Search

Problem Description

In this problem, we are given a sorted array of distinct integers, `arr`. The array is sorted in ascending order. The task is to find the smallest index `i` such that the element at that index is equal to its index value, i.e., `arr[i] == i`. If no such index exists, the function should return `-1`. This problem is essentially asking us to find the point, if any, where the value of the array matches its index.

Intuition

The intuitive approach to this problem relies on the properties of the sorted array. To find the smallest index where the value equals the index, we don't have to look at each element of the array; instead, we can use binary search because of the array's sorted nature. Binary search works by repeatedly dividing the search interval in half, eliminating half of the remaining elements at each step.

If the value at the middle index is less than the middle index itself, then we know that the fixed point, if it exists, must be on the right side of the middle index. This is because the values are distinct and increasing, and if `arr[mid] < mid`, then all values left of `mid` must also be less than their indices, so we can ignore the left side.

Conversely, if the value at the middle index is greater than or equal to the middle index, the fixed point might be at this position or to the left. In this case, we set the new right bound to be the current middle index.

After iteratively applying binary search and narrowing down the search space, if we find an index where `arr[i] == i`, we return it. If we exhaust the search space without finding such an index, we return `-1` because no element satisfies the condition.

Solution Approach

The implementation of the solution uses a binary search algorithm. Here's how it is applied to the given problem:

1. We initialize two pointers, `left` and `right`, which represent the bounds of the search range. Initially, `left` is set to `0` (the first index of the array) and `right` is set to `len(arr) - 1` (the last index of the array).
2. We enter a while loop, which runs as long as `left < right`. Inside the loop, we find the midpoint `mid` by averaging the current `left` and `right` indices (using bit shifting `>>` for efficiency, which is the same as dividing by 2).
3. At each step, we compare the value at the midpoint `arr[mid]` with the `mid` itself to determine where to continue our search:
   - If `arr[mid] >= mid`, then we move the `right` pointer to `mid` because the fixed point could be at this position or to the left of it.
   - If `arr[mid] < mid`, then we move the `left` pointer to `mid + 1` because a fixed point cannot exist to the left of `mid`, given the properties of the sorted array.
4. The loop ends when `left` and `right` converge, meaning the search space has been narrowed down to a single element. At this point, we have potentially found the smallest index `i` where `arr[i] == i`.
5. Finally, we perform a check by comparing the element at the `left` index with `left` itself. If they are equal, we have found the fixed point and return `left`. If they are not, it means there are no elements where `arr[i] == i`, so we return `-1`.

The code does not require any additional data structures, as it only uses the input array `arr` and manipulates the `left` and `right` pointers to perform the binary search.

Example Walkthrough

Let's go through a small example to illustrate the solution approach.

Suppose the input array `arr` is `[-10, -5, 0, 3, 7]`. We need to find the smallest index `i` such that `arr[i] == i`.

1. We initialize two pointers: `left` is `0` and `right` is `4` (since the length of `arr` is `5`, `right` is `len(arr) - 1`).
2. Our first while loop iteration starts:
   - Calculate `mid` `(left + right) / 2`, which is `(0 + 4) / 2`. So, `mid` is `2`.
   - The value at index `2` is `0`. Since `arr[mid] == mid` (`0 == 2` is false), we check if `arr[mid] < mid`.
   - Since `0` (the value at `mid`) is less than `2` (`mid`), we set `left` to `mid + 1`, which is `3`.
3. Our next iteration begins with `left` as `3` and `right` as `4`:
   - Calculate the new `mid`: `(left + right) / 2`, which is `(3 + 4) / 2`. So, `mid` is `3`.
   - The value at index `3` is `3`. Check if `arr[mid] == mid`. Since `3 == 3` is true, we have found our fixed point.
4. Since we have found the index where `arr[i] == i`, we return the value of `left`, which is `3`.

To summarize, in our example array `[-10, -5, 0, 3, 7]`, the smallest index at which the value of the array matches its index is `3`, because `arr[3] == 3`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def fixedPoint(self, arr: List[int]) -> int:
        # Initialize the left and right pointers
        left, right = 0, len(arr) - 1

        # Use a binary search approach to find the fixed point
        while left < right:
            # Find the middle index using bitwise right shift
            # which is equivalent to dividing by 2.
            mid = (left + right) // 2

            # If the middle element is greater than or equal to its index,
            # the fixed point must be to the left, including mid.
            if arr[mid] >= mid:
                right = mid
            else:
                # If the middle element is less than its index,
                # the fixed point must be to the right, so we
                # exclude the middle element by incrementing left.
                left = mid + 1

        # After the loop, if the element at the left index is equal
        # to the index itself, then we've found the fixed point.
        # Otherwise, return -1 as the fixed point does not exist.
        return left if arr[left] == left else -1
```

Java

```java
class Solution {

    /**
     * Finds a fixed point in the array where arr[i] == i.
     * If no such point exists, returns -1.
     *
     * @param arr The input array to be searched for a fixed point.
     * @return The index of the fixed point or -1 if none exists.
     */
    public int fixedPoint(int[] arr) {
        int left = 0;
        int right = arr.length - 1;

        // Use binary search to find the fixed point
        while (left < right) {
            // Calculate mid index
            int mid = left + (right - left) / 2; // Avoids potential overflow compared to (left + right) >> 1

            // Check if the mid index is a fixed point
            if (arr[mid] >= mid) {
                right = mid; // The fixed point is at mid or to the left of mid
            } else {
                left = mid + 1; // The fixed point can only be to the right of mid
            }
        }

        // After the loop, left should point to the smallest candidate for fixed point
        // Check if the final left index is indeed a fixed point
        return arr[left] == left ? left : -1;
    }
}
```

C++

```cpp
class Solution {
public:
    int fixedPoint(vector<int>& arr) {
        // Initialize the search space boundaries.
        int left = 0;
        int right = arr.size() - 1;

        // Use binary search within the loop.
        while (left < right) {
            // Compute the middle index avoiding potential overflow.
            int mid = left + (right - left) / 2;

            // If the value at the mid index is greater than or equal to mid,
            // we need to search on the left side including mid, as the fixed point
            // could be at mid or to its left.
            if (arr[mid] >= mid) {
                right = mid;
            } else {
                // The value at mid is less than mid, so the fixed point
                // must be on the right side, excluding mid.
                left = mid + 1;
            }
        }

        // At the end of the loop, left is the smallest index where arr[left] >= left.
        // If arr[left] == left, we have found the fixed point; otherwise, return -1.
        return arr[left] == left ? left : -1;
    }
};
```

TypeScript

```typescript
function fixedPoint(numbers: number[]): number {
    // Initialize pointers for the start and end of the array
    let start = 0;
    let end = numbers.length - 1;

    // Use binary search to find the fixed point
    while (start < end) {
        // Calculate the middle index
        const middle = (start + end) >> 1; // same as Math.floor((start + end) / 2)

        // If the element at the middle index is greater than or equal to the index,
        // the fixed point must be to the left, including the middle
        if (numbers[middle] >= middle) {
            end = middle;
        } else {
            // If the element at the middle index is less than the index,
            // the fixed point must be to the right
            start = middle + 1;
        }
    }

    // After the loop, start is the potential fixed point.
    // Check if the value at this index equals the index itself
    return numbers[start] === start ? start : -1;
}
```

```python
from typing import List

class Solution:
    def fixedPoint(self, arr: List[int]) -> int:
        # Initialize the left and right pointers
        left, right = 0, len(arr) - 1

        # Use a binary search approach to find the fixed point
        while left < right:
            # Find the middle index using bitwise right shift
            # which is equivalent to dividing by 2.
            mid = (left + right) // 2

            # If the middle element is greater than or equal to its index,
            # the fixed point must be to the left, including mid.
            if arr[mid] >= mid:
                right = mid
            else:
                # If the middle element is less than its index,
                # the fixed point must be to the right, so we
                # exclude the middle element by incrementing left.
                left = mid + 1

        # After the loop, if the element at the left index is equal
        # to the index itself, then we've found the fixed point.
        # Otherwise, return -1 as the fixed point does not exist.
        return left if arr[left] == left else -1
```

Time and Space Complexity

Time Complexity

The provided code uses a binary search algorithm to find a fixed point in the array. Binary search repeatedly divides the array into half to look for the fixed point, resulting in a logarithmic time complexity. Since it narrows down the search space by half at each step and performs a constant number of operations at each level, the time complexity is $O(\log n)$, where n is the length of the array.

Space Complexity

The space complexity of the algorithm is $O(1)$. The solution is an in-place algorithm, meaning it requires a constant amount of additional space regardless of the input size. The variables `left`, `right`, `mid`, and the space for the return value do not scale with the size of the input array.