275. H-Index II

Medium Array Binary Search

Problem Description

This problem involves finding the h-index of a researcher based on the array of their paper citations, which is sorted in ascending order. The h-index represents a metric that aims to measure both the productivity and citation impact of the publications of a scientist or scholar. It's defined as the maximum number 'h' such that the researcher has at least 'h' papers cited 'h' times each. Thus, if a researcher has an h-index of 5, they have published at least 5 papers with 5 or more citations each. The challenge in this problem includes implementing an efficient algorithm that runs in logarithmic time - a clear indicator that a binary search approach is likely the most appropriate solution given the sorted nature of the array.

Intuition

The intuition behind using a binary search for this problem arises from the sorted nature of the citations array and the specific property we're trying to find - the h-index. By definition, we know that an h-index of value 'h' means that there must be 'h' papers with at least 'h' citations each. This implies a monotonic relationship between the number of papers and the number of citations in a sorted list.

For a sorted array, if we find a point where the citation number is greater than or equal to the number of papers counted from the end of the array backwards, it's possible that our current position is part of the h-index count. If the citation count at a given array position is equal to or more than the papers count from that point to the array end, then any smaller number of papers will also have an equal or greater number of citations.

Binary search fits well here since this algorithm efficiently narrows down the search space by half with each step. We perform a binary search to pinpoint the exact point in the array where the number of citations crosses from being less than the h-index count to satisfying the h-index condition. Iteratively, we search for the largest 'mid' such that `citations[n - mid] >= mid`, which means there are at least 'mid' papers with 'mid' or more citations. This process of splitting the search space in half and validating the h-index condition at each midpoint quickly locates the correct h-index value in logarithmic time.

The implemented solution starts with binary search boundaries from 0 to the number of citations (left and right, respectively). We iteratively adjust these boundaries based on whether the mid-value satisfies the h-index condition until left and right converge, which will happen when we find the maximum 'h' that satisfies the condition.

Solution Approach

The given solution approach employs a binary search algorithm that takes advantage of the sorted nature of the input array 'citations'. Given that binary search is a divide-and-conquer algorithm that halves the search space with each iteration, it is a perfect match for the requirement of the problem to run in logarithmic time.

The binary search process in this problem is somewhat unique. The goal is to find the maximum `h` index possible. To facilitate this, we set two variables, `left` and `right`, which represent the current range we consider for our binary search. The variable `left` starts at 0 (the lowest possible h-index), and `right` starts at `n` (the length of the citations array which is the highest initial guess for h-index).

Within the `while` loop, the algorithm calculates the middle point `mid` by taking the average of `left` and `right` and incrementing it by one to handle the integer division correctly. The condition `(left < right)` ensures that we continue the search as long as there is a range to check.

At each iteration of the binary search, we check if there are at least `mid` papers with `mid` or more citations, which translates to the condition `citations[n - mid] >= mid`. High values of `mid` mean higher possible h-index values, and conversely, lower values mean lower possible h-index values.

- If the condition is true, it means we have at least `mid` papers with at least `mid` citations, and we should look to see if there's a higher h-index possible, so we move the `left` pointer to `mid`.
- Conversely, if the condition is false, it means we have fewer than `mid` papers with at least `mid` citations, so we must lower our expectations and thus move the `right` pointer to `mid - 1`.

The loop continues adjusting `left` and `right` until they converge, at which point `left` will hold the maximum h-index that satisfies the h-index condition (`citations[n - left] >= left`). This `left` value is what's returned as the solution to the problem.

The entire process requires no additional data structures as it operates directly on the input array and uses a handful of variables for the pointers and the middle index. The binary search pattern itself serves as the primary algorithmic strategy, exploiting the ordered nature of the input and requiring only `O(log n)` time to complete.

Example Walkthrough

Let's consider a researcher whose papers have the following citation counts, sorted in ascending order: `[0, 1, 3, 5, 6]`. The goal is to find the h-index using the binary search approach explained above.

There are 5 papers, so we set `left` to 0 and `right` to 5, representing possibility for the h-index from 0 to 5.

Step 1: `while (left < right)`

- `left` = 0, `right` = 5
- Calculate `mid` = (0 + 5)/2 = 2 (since we're using integer division, the fractional part is discarded)
- Check the condition `citations[n - mid] >= mid`, with `n` being the number of papers.
- `citations[5 - 2]` is `citations[3]` which is 5. Since `5 >= 2`, the condition is true.
- We might have a higher h-index, so we set `left` to `mid`. Now, `left` = 2 and `right` = 5.

Step 2: `while (left < right)`

- `left` = 2, `right` = 5
- Calculate `mid` = (2 + 5)/2 = 3
- `citations[5 - 3]` is `citations[2]` which is 3. Since `3 >= 3`, the condition is true.
- We could still have a higher h-index, so again set `left` to `mid`. Now, `left` = 3 and `right` = 5.

Step 3: `while (left < right)`

- `left` = 3, `right` = 5
- Calculate `mid` = (3 + 5)/2 = 4
- `citations[5 - 4]` is `citations[1]` which is 1. Since `1 < 4`, the condition is false.
- We've overshot the possible h-index, so we need to lower it, set `right` to `mid - 1`. Now, `left` = 3 and `right` = 4 -1 = 3.

In the next step, `left` will equal `right` and the loop will stop. `left`, which also equals `right`, is our h-index which is 3. This means the researcher has 3 papers with at least 3 citations each. This matches the definition of the h-index, and since the search ended, this is the maximum `h` possible for the given citations array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def hIndex(self, citations: List[int]) -> int:
        # Initialize the length of the citations list.
        number_of_citations = len(citations)
        # Set initial search range from 0 to number_of_citations.
        left, right = 0, number_of_citations
        # Perform binary search to find the h-index.
        while left < right:
            # Mid-point of the current search range calculated using bitwise right shift, equivalent to dividing by 2.
            # Adding 1 to include the mid element in the right part of the search during the next iteration.
            mid = (left + right + 1) // 2  # Using '//' for integer division in Python3
            # Check if mid citations have at least 'mid' each.
            if citations[number_of_citations - mid] >= mid:
                # If condition satisfied, search in the right half.
                left = mid
            else:
                # Else search in the left half, exclude 'mid' by subtracting one.
                right = mid - 1
        # The left pointer indicates the maximum number of citations that at least 'left' papers have.
        return left
```

Java

```java
class Solution {
    public int hIndex(int[] citations) {
        int numPapers = citations.length; // Number of papers
        int low = 0;                      // Lower bound of binary search
        int high = numPapers;             // Upper bound of binary search

        // Perform binary search
        while (low < high) {
            // Calculate the middle index. "+1" ensures we go to upper half if not found.
            int mid = (low + high + 1) >>> 1; // Using unsigned right shift for safe mid calculation

            // Check if mid papers have at least 'mid' citations each
            if (citations[numPapers - mid] >= mid) {
                // If yes, move the lower bound up
                low = mid;
            } else {
                // Otherwise, bring the upper bound down
                high = mid - 1;
            }
        }

        // After exiting the loop, low is the h-index
        return low;
    }
}
```

C++

```cpp
class Solution {
public:
    int hIndex(vector<int>& citations) {
        // The size of the citations vector
        int numPapers = citations.size();

        // Initialize binary search bounds
        int left = 0, right = numPapers;

        while (left < right) {
            // Calculate the mid point, avoid overflow by using left + (right - left) / 2
            // The "+1" is to make sure we find the upper bound in case of even number of elements
            int mid = left + (right - left + 1) / 2;

            // Check if mid number of papers have at least mid citations each
            // Since 'citations' is sorted in non-increasing order,
            // we check the value at index 'numPapers - mid'
            if (citations[numPapers - mid] >= mid)
                // If mid papers have at least mid citations, this could be our h-index,
                // but there may be a larger one. Move the search to the right (upper half).
                left = mid;
            else
                // If mid papers do not have at least mid citations, look for lower h-index
                // in the left (lower half).
                right = mid - 1;
        }
        // After the loop, 'left' points to the largest number such that there are 'left' papers
        // that have at least 'left' citations.
        return left;
    }
};
```

TypeScript

```typescript
function hIndex(citations: number[]): number {
    // Initialize the size of the citations array.
    const length = citations.length;
    // Define binary search bounds.
    let left = 0,
        right = length;

    // Perform a binary search to find the h-index.
    while (left < right) {
        // Calculate the mid point. The bitwise operation is used to perform integer division by 2.
        const mid = (left + right + 1) >> 1;
        // Check if the mid element fulfills the h-index property.
        if (citations[length - mid] >= mid) {
            // If the condition is met, move the lower bound up, as we are looking for the maximum h.
            left = mid;
        } else {
            // Otherwise, move the upper bound down.
            right = mid - 1;
        }
    }

    // The left variable will have settled on the correct h-index value by the end of the loop.
    return left;
}
```

```python
from typing import List

class Solution:
    def hIndex(self, citations: List[int]) -> int:
        # Initialize the length of the citations list.
        number_of_citations = len(citations)
        # Set initial search range from 0 to number_of_citations.
        left, right = 0, number_of_citations
        # Perform binary search to find the h-index.
        while left < right:
            # Mid-point of the current search range calculated using bitwise right shift, equivalent to dividing by 2.
            # Adding 1 to include the mid element in the right part of the search during the next iteration.
            mid = (left + right + 1) // 2  # Using '//' for integer division in Python3
            # Check if mid citations have at least 'mid' each.
            if citations[number_of_citations - mid] >= mid:
                # If condition satisfied, search in the right half.
                left = mid
            else:
                # Else search in the left half, exclude 'mid' by subtracting one.
                right = mid - 1
        # The left pointer indicates the maximum number of citations that at least 'left' papers have.
        return left
```

Time and Space Complexity

The time complexity of the code is `O(log n)`, where `n` is the length of the `citations` list. This is because the algorithm uses a binary search technique, repeatedly dividing the search interval in half, which results in a logarithmic number of iterations in relation to the size of the input list.

The space complexity of the code is `O(1)`. There are no extra data structures used that grow with the input size. The additional space used for variables `left`, `right`, and `mid` is constant, and thus does not depend on the size of the input list.