# 274. H-Index

Given an array of integers `citations` where{" "} `citations[i]` is the number of citations a researcher received for their{" "} $i^{th}$ {" "} paper, return compute the researcher's `h` **-index**.

According to the{" "} [definition of h-index on Wikipedia](): A scientist has an index `h` if `h` of their{" "} `n` papers have at least `h` citations each, and the other `n − h` papers have no more than `h` citations each.

If there are several possible values for `h`, the maximum one is taken as the `h` **-index**.

**Example 1:**

```
Input: citations = [3,0,6,1,5]{"\n"}
Output: 3{"\n"}
Explanation: [3,0,6,1,5] means the researcher has 5 papers
in total and each of them had received 3, 0, 6, 1, 5 citations respectively.
{"\n"}Since the researcher has 3 papers with at least 3 citations each and
the remaining two with no more than 3 citations each, their h-index is 3.
{"\n"}
```

**Example 2:**

```
Input: citations = [1,3,1]{"\n"}
Output: 1{"\n"}
```

**Constraints:**

- `n == citations.length`
- `1 <= n <= 5000`
- `0 <= citations[i] <= 1000`

---

## Solution

We can try all possible values of `h` from `0`to `n`. For each `h`, loop through `citations` to see if `h` is a possible h-index, using the condition we are given:

> A scientist has an index `h` if `h` of their `n` papers have at least h citations each, and the other `n−h` papers have no more than `h` citations each.

The answer is the highest `h` for which this is true.

This takes $\mathcal{O}(n^2)$ time because for each of the `n+1` possible `h` values, we have to loop through `n` citations.

Create a function `hasAtLeastHPapersWithHCitations` with a parameter `h` to check if there are at least `h` papers with `>= h` citations. When `hasAtLeastHPapersWithHCitations(x)` is true, `hasAtLeastHPapersWithHCitations(x−1)` is also true. This means that `hasAtLeastHPapersWithHCitations` is a monotonic function, so we can binary search for the highest `h` for which it return `true`. This `h` is our h-index.

### Time Complexity

Each call to `hasAtLeastHPapersWithHCitations` checks all $n$ papers, taking $\mathcal{O}(n)$.

Binary searching the range $[0, n]$ takes $\mathcal{O}(\log n)$.

Multiplying these together, we take $\mathcal{O}(n \log n)$.

### Space Complexity

`citations` is passed by reference, so we aren't allocating any memory for it. We allocate a constant amount of memory for a couple of variables, so the space complexity is $\mathcal{O}(1)$.

```cpp
class Solution {
public:
    bool hasAtLeastHPapersWithHCitations(int h, vector<int>& citations) {
        int count = 0;
        for (int cite_count : citations) {
            if (cite_count >= h)
                count++;
        }
        return count >= h;
    }
    int hIndex(vector<int>& citations) {
        int low = 0, high = citations.size();
        while (low <= high) {
            int mid = (low + high) / 2;
            if (hasAtLeastHPapersWithHCitations(mid, citations))
                low = mid + 1;
            else
                high = mid − 1;
        }
        return high;
    }
};
```

```java
class Solution {
    static boolean hasAtLeastHPapersWithHCitations(int h, int[] citations) {
        int count = 0;
        for (int cite_count : citations) {
            if (cite_count >= h)
                count++;
        }
        return count >= h;
    }
    public int hIndex(int[] citations) {
        int low = 0, high = citations.length;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (hasAtLeastHPapersWithHCitations(mid, citations))
                low = mid + 1;
            else
                high = mid − 1;
        }
        return high;
    }
}
```

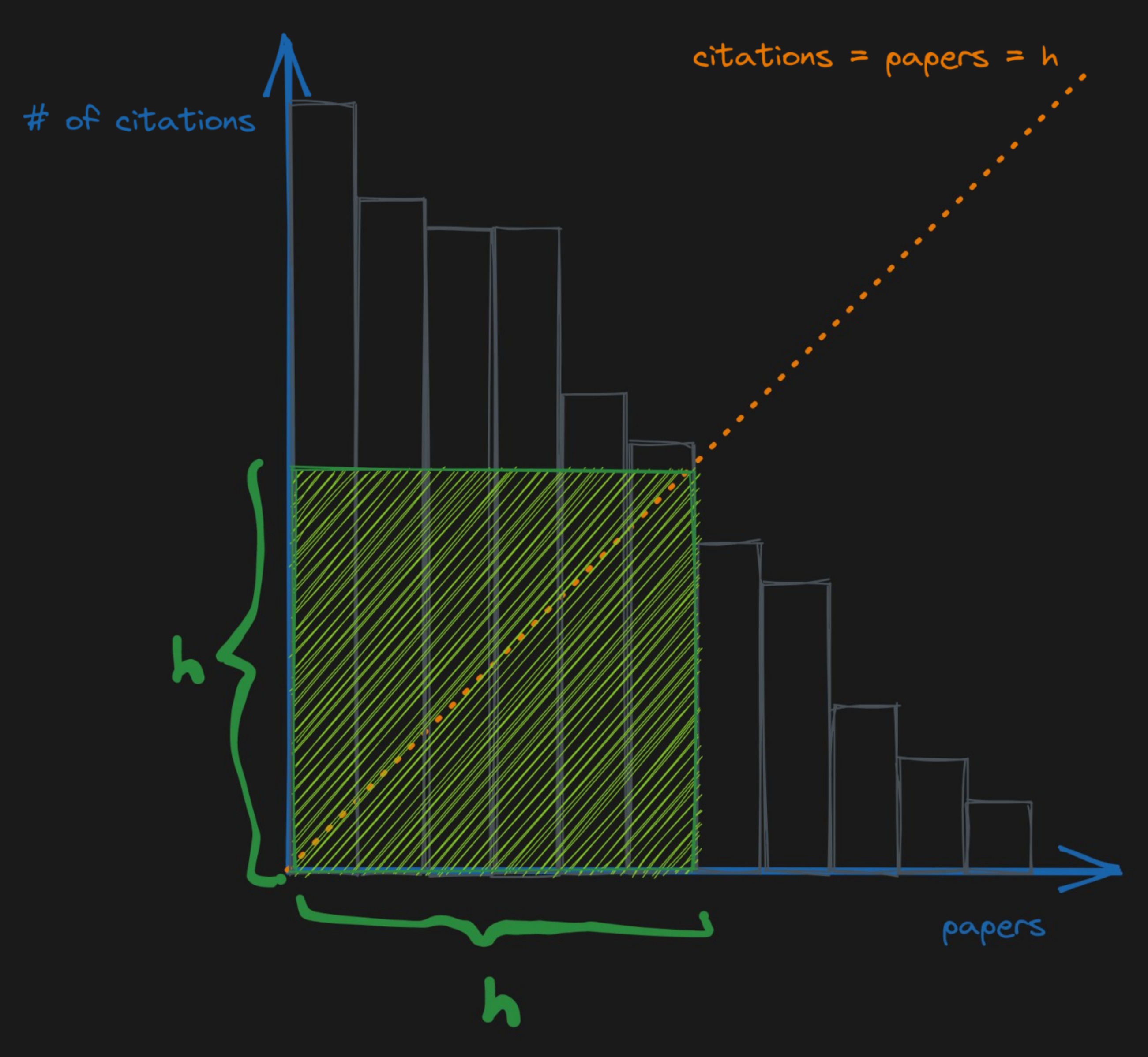
```python
class Solution:
    def hIndex(self, citations: List[int]) -> int:

        def hasAtLeastHPapersWithHCitations(h, citations):
            return sum(cite_count >= h for cite_count in citations) >= h

        low = 0
        high = len(citations)
        while low <= high:
            mid = (low + high) // 2;
            if hasAtLeastHPapersWithHCitations(mid, citations):
                low = mid + 1;
            else:
                high = mid − 1;
        return high
```

First we sort the papers by decreasing # of citations. Imagine a histogram where each bar represents a paper and its height is the # of citations it has.

<img src={require('@site/static/img/274/1.png').default} style={{ height: 450 }}/>




If the h-index were `h`, we'd need exactly `h` bars with height as least `h`. That is to say, we'd need the green square covered by bars. To find the `h` index, first set `h = 0`. Then keep increasing `h` by `1` as long as the next tallest bar is `>= h+1`. When we can no longer increase `h`, we have our answer.

In the diagram above, if we continued to increase `h`, the next added bar would not be tall enough for the new `h`.

### Time Complexity

Sorting is $\mathcal{O}(n \log n)$. Looping `h` is $\mathcal{O}(n)$. So the time complexity is $\mathcal{O}(n \log n)$.

### Space Complexity

The only memory we allocate is the integer `h`, so the space complexity is $\mathcal{O}(1)$.

```cpp
class Solution {
public:
    int hIndex(vector<int>& citations) {
        sort(citations.rbegin(), citations.rend());
        int h = 0;
        while (h < citations.size() and citations[h] >= h+1) {
            h++;
        }
        return h;
    }
};
```

```java
class Solution {
    public int hIndex(int[] citations) {
        // Sorting an int[] in reverse in Java is annoying
        // We first sort normally then reverse the array
        Arrays.sort(citations);
        for (int i = 0; i < citations.length/2; i++) {
            int tmp = citations[i];
            citations[i] = citations[citations.length−1−i];
            citations[citations.length−1−i] = tmp;
        }

        int h = 0;
        while (h < citations.length && citations[h] >= h+1) {
            h++;
        }
        return h;
    }
}
```

```python
class Solution:
    def hIndex(self, citations: List[int]) -> int:
        citations.sort(reverse=True)
        h = 0
        while h < len(citations) and citations[h] >= h+1:
            h += 1
        return h
```