VerifyThis 2026 - Challenge 1

Ada and her papers

Based on a contribution from Jean-Christophe Filliâtre and Mário Pereira

1 Context

Ada is a young researcher who is actively keeping track of her citation counts. She keeps them is an array, which is sorted in reverse order. These days, it looks as follows:

12	5	3	3	3	3	2	1	0	0

In other words, her most cited paper is cited 12 times, the second most cited paper is cited 5 times, and so on, for a total of 10 papers. Ada notices that she has three papers that are cited at most three times each, but that she is not yet famous enough to have four papers cited at most four times each. She defines her score as the greatest number h such that at least h elements in the array are greater or equal to h. Ada just invented the h-index — but luckily for her, her administration has not yet come with the same idea.

Ada is a good programmer, so she quickly writes a C function to compute the h-index (function `compute` in Fig. 1). A moment later, she realizes that one of the most fundamental algorithm can be used to compute it more efficiently, and she writes a second C function to compute the h-index (function `compute_opt` in Fig. 1).

Whenever Ada discovers a new citation to one of her papers, she updates her array. She locates the position in the array corresponding to the paper, increments the value at that position, and then moves it to the left until the array is sorted again. For instance, if her paper at position 5 (counting from 0) gets a new citation, the array ends up in the following state:

12	5	4	3	3	3	2	1	0	0

The h-index is still 3, though. If later, the paper at position 4 gets a new citation, then the array is updated as follows

12	5	4	4	3	3	2	1	0	0

and this times the h-index becomes 4.

Ada figures out that updating both the array and the h-index value can be conveniently done at the same time. This is function `update` in Fig. 1, where parameter `h` is the current h-index value and `i` is the index of the count to be incremented. The function updates the array and returns the new h-index value.

```c
// given an array `a` of size `n`, compute its h-index
int compute(int a[], int n) {
  int h = 0;
  while (h < n && h < a[h])
    h++;
  return h;
}

// the same, more efficiently
int compute_opt(int a[], int n) {
  int lo = 0, hi = n;
  while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] <= mid) hi = mid;
    else lo = mid + 1;
  }
  return lo;
}

// increments the value at index `i`,
// updates the array and returns the new h-index
int update(int a[], int h, int i) {
  int x = a[i];
  int lo = 0, hi = i;
  while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == x) hi = mid;
    else lo = mid + 1;
  }
  a[lo]++;
  if (lo == h && a[lo] == h+1) return h+1;
  return h;
}
```

Figure 1: Ada's C code.

2 Tasks

Specification.

1.1 Provide a predicate definition for "the array a is sorted in reverse order"

1.2 Provide a predicate definition for "the array a has h-index value h" (your definition should work for any array, not just those sorted in reverse order).

Function `compute`. Input: an array `a` containing non-negative integers, sorted in reverse order.

2.1 Prove the safety, *i.e.*, the function terminates and does not access array `a` out of bounds.

2.2 Prove that the function returns the h-index.

Function `compute_opt`. Input: an array `a` containing non-negative integers, sorted in reverse order.

3.1 Prove the safety, *i.e.*, the function terminates and does not access array `a` out of bounds.

3.2 Prove that the function returns the h-index.

Function `update`. Input: an array `a` containing non-negative integers, sorted in reverse order; its h-index value `h`; a valid position `i` in array `a`.

4.1 Prove the safety, *i.e.*, the function terminates and does not access array `a` out of bounds.

4.2 Prove that array `a` is sorted in reverse order at the end of the function.

4.3 Prove that the contents of array `a` at the end of the function is consistent with the increment of the value that was at index `i`.

4.4 Prove that the returned value is the h-index of the final array.

Bonus tasks

B.1 Prove that the complexity of the function `compute` is $O(n)$, and that the complexity of the functions `compute_opt` and `update` is $O(\log n)$.

B.2 Implement a reverse-sorting function `rsorted`, and prove that `compute_opt(rsorted(a, n), n)` returns the h-index of the array `a`.

B.3 Implement and prove correct a function `update_k(int a[], int h, int i, int k)` that adds `k` citations to the i-th paper, and returns the updated h-index. The function should maintain the sorted order of the array, and should work for any $k \geq 1$.