

## 1426. Counting Elements

Average Rating: 4.89 (9 votes)

Given an integer array `arr`, count how many elements `x` there are, such that `x + 1` is also in `arr`.

If there're duplicates in `arr`, count them separately.

### Example 1:

```
Input: arr = [1,2,3]
Output: 2
Explanation: 1 and 2 are counted cause 2 and 3 are in arr.
```

### Example 2:

```
Input: arr = [1,1,3,3,5,5,7]
Output: 0
Explanation: No numbers are counted, cause there's no 2, 4, 6, or 8 in arr.
```

### Example 3:

```
Input: arr = [1,3,2,3,5,0]
Output: 3
Explanation: 0, 1 and 2 are counted cause 1, 2 and 3 are in arr.
```

### Example 4:

```
Input: arr = [1,1,2,2]
Output: 2
Explanation: Two 1s are counted cause 2 is in arr.
```

### Example 5:

```
Input: arr = [1,1,2]
Output: 2
Explanation: Both 1s are counted because 2 is in the array.
```

### Constraints:

- $1 \leq \text{arr.length} \leq 1000$
- $0 \leq \text{arr}[i] \leq 1000$

## Solution

### Approach 1: Search with Array

#### Intuition

The simplest way of solving this problem is to loop through each integer, `x`, checking whether or not it should be counted. This requires checking whether or not `x + 1` is in `arr`.

```
define function count_elements(arr):
    count = 0
    for each x in arr:
        if integer_in_array(arr, x + 1):
            count = count + 1
    return count
```

To implement the `integer_in_array` function in the above algorithm, we can use `linear search`. To do a linear search, we need to loop through each integer of `arr`. If we find the integer that we're looking for, then return `true`. If we get to the end of `arr`, then we know the integer is not there, and so should return `false`.

```
define function integer_in_array(arr, target):
    for each x in arr:
        if target is equal to x:
            return true
    return false
```

Many programming languages have a built-in function for checking whether or not an integer is in `arr`, e.g. Python.

#### Algorithm

```
Java Python3
1 def countElements(self, arr: List[int]) -> int:
2     count = 0
3     for x in arr:
4         if x + 1 in arr:
5             count += 1
6     return count
7
8 # note that we could also do this as a one-liner generator comprehension.
9 # return sum(1 for x in arr if x + 1 in arr)
```

#### Complexity Analysis

Let  $N$  be the length of the input array, `arr`.

- Time complexity:  $O(N^2)$ .

We loop through each of the  $N$  integers `x`, checking whether or not `x + 1` is also in `arr`. Checking whether or not `x + 1` is in `arr` is done using linear search, which requires checking through all  $N$  integers in `arr`. Because we're doing  $N$  operations  $N$  times, we get a time complexity of  $O(N^2)$ .

- Space complexity:  $O(1)$ .

We are only using a constant number of single-value variables (e.g. `count`), giving us a space complexity of  $O(1)$ .

### Approach 2: Search with HashSet

#### Intuition

If you're not familiar with the `HashSet` data structure, check out our [Hash Tables Explore Card](#) to get up to speed.

The above algorithm will work fine for the maximum array length we've given here. However, we can do a lot better than  $O(N^2)$ , and an interviewer will no doubt expect you to come up with a better way.

The reason why the algorithm above was so inefficient is because we're performing  $N$  linear searches, each with a cost of  $O(N)$ . When we have an algorithm that is performing many linear searches to check for item existence, we should instead be looking to change the way the data is stored so that the time complexity of doing each search is less.

Recall that looking up items in a `HashSet` has a cost of  $O(1)$ . Creating a `HashSet` from an array of  $N$  items has a cost of  $O(N)$ . We only need to create the `HashSet` once. After that, we can then replace all  $O(N)$  linear searches with  $O(1)$  `HashSet` lookups.

Before we go any further, here is an algorithm that is *incorrect*. Try to spot what the problem is; we'll discuss it just below.

```
define function count_elements(arr):
    hash_set = a new HashSet
    add all integers of arr to hash_set
    count = 0
    for each x in hash_set:
        if hash_set contains x + 1:
            count = count + 1
    return count
```

Did you spot the bug? If there were duplicates in `arr`, then the `count` returned will be too low!

Recall that a `HashSet` removes duplicates. Consider a case like `arr = [1, 1, 2]`. The `HashSet` will be `{1, 2}`. Therefore, the above code will loop over each integer in the `HashSet`, which is only one copy of `1`. Yet `arr` had two copies of `1`.

To fix it, we need to loop over `arr`, but do the existence checks using the `HashSet`.

```
define function count_elements(arr):
    hash_set = a new HashSet
    add all integers of arr to hash_set
    count = 0
    for each x in arr:
        if hash_set contains x + 1:
            count = count + 1
    return count
```

#### Algorithm

```
Java Python3
1 def countElements(self, arr: List[int]) -> int:
2     hash_set = set(arr)
3     count = 0
4     for x in arr:
5         if x + 1 in hash_set:
6             count += 1
7     return count
8
9 # note that we could also do this as a one-liner generator comprehension.
10 # return sum(1 for x in arr if x + 1 in arr)
```

#### Complexity Analysis

Let  $N$  be the length of the input array, `arr`.

- Time complexity:  $O(N)$ .

Creating a `HashSet` from  $N$  integers takes  $O(N)$  time. We then need to loop over each of the  $N$  integers like before, except this time we check for `x + 1` by seeing if it is in the `HashSet`; an  $O(1)$  operation. This gives us a total time complexity of  $O(N) + N \cdot O(1) = O(N) + O(N) = O(N)$ .

- Space complexity:  $O(N)$ .

The `HashSet` needs to store each unique integer from `arr`. In the worst case, all the integers in `arr` will be unique, meaning that the `HashSet` has a space complexity of  $O(N)$ .

It's interesting to note that  $O(N)$  is an *upper bound* on the space complexity. If  $U$  is the number of unique integers in `arr`, then the space complexity could more accurately be represented as  $O(U)$ .

### Approach 3: Search with Sorted Array

#### Intuition

Another way of changing the data storage to allow for more efficient searching is to sort it. Sorting has a time complexity of  $O(N \log N)$ , and searching for integers in a sorted array, using binary search, has a cost of  $O(\log N)$ . This will give us a total time complexity of  $O(N \log N)$ .

```
define function countElements(arr):
    sort arr
    count = 0
    for each x in arr:
        binary search for x + 1 in arr
        if x + 1 is in arr:
            count = count + 1
    return count
```

The main challenge of this approach would be needing to implement your own binary search.

However, we don't actually need to use binary search! If we iterate over the sorted `arr`, then we know that if `x + 1` exists, it will be after all the copies of `x`.

```
[1 | 1 | 1 | 2 | 2 | 4 | 5 | 6 | 8 | 8 | 9 | 12 | 12 | 12 | 12 | 13]
```

Each copy of `x` should be counted if at least one copy of `x + 1` exists. Therefore, we can iterate down the sorted `arr`, keeping track of how many times the current `x` has appeared. When we get to a different integer, we can check if it's `x + 1`, and if it is, then the number of `x` we saw should be added to `count`.

```
define function countElements(arr):
    sort arr
    count = 0
    run_length = 1
    for each i in range 1 to arr.length - 1:
        if arr[i - 1] is not equal to arr[i]:
            if arr[i - 1] + 1 is equal to arr[i]:
                count = count + run_length
            run_length = 0
        run_length = run_length + 1
    return count
```

Here is an animation of this approach.

```
[1 | 1 | 1 | 2 | 2 | 4 | 5 | 6 | 8 | 8 | 9 | 12 | 12 | 12 | 12 | 13]
```

1 / 27

#### Algorithm

```
Java Python3
1 def countElements(self, arr: List[int]) -> int:
2     arr.sort()
3     count = 0
4     run_length = 1
5     for i in range 1 to arr.length:
6         if arr[i - 1] is not equal to arr[i]:
7             if arr[i - 1] + 1 is equal to arr[i]:
8                 count = count + run_length
9             run_length = 0
10            run_length = run_length + 1
11    return count
12
13 # note that we could also do this as a one-liner generator comprehension.
14 # return sum(1 for x in arr if x + 1 in arr)
```

#### Complexity Analysis

- Time complexity:  $O(N \log N)$ .

Sorting using a built-in sorting algorithm has a cost of  $O(N \log N)$ . After that, we do a single pass through `arr`, which has a cost of  $O(N)$ , giving us a total time complexity of  $O(N \log N) + O(N) = O(N \log N)$ .

- Space complexity: varies from  $O(N)$  to  $O(1)$ .

The `Space complexity` of this approach is dependent on the space complexity of the sorting algorithm you're using. The space complexity of sorting algorithms built into programming languages is generally anywhere from  $O(N)$  to  $O(N^2)$ .

Notice that you could implement your own  $O(N \log N)$  time complexity,  $O(1)$  space complexity, sorting algorithm if needed. In practice,  $O(N \log N)$  is not much worse than  $O(N)$ , and so this approach provides an interesting contrast to Approach 2 (which had a space complexity of  $O(N)$ ).

Analysis written by [@hai\\_dee](#)

Rate this article: ★ ★ ★ ★ ★

#### Comments: 4

Type comment here... (Markdown is supported)

© Preview Post

probudho ★ 13 ⏺ April 8, 2020 11:00 PM

The solution above uses two passes. Here is one pass python solution:

```
class Solution(object):
    def countElements(self, arr):
        sort arr
        count = 0
        run_length = 1
        for each i in range 1 to arr.length - 1:
            if arr[i - 1] is not equal to arr[i]:
                if arr[i - 1] + 1 is equal to arr[i]:
                    count = count + run_length
                run_length = 0
            run_length = run_length + 1
        return count
```

Read More

nishadkumar ★ 40 ⏺ April 7, 2020 11:00 PM

Nice idea for the 3rd approach to use sorting! I went with the 2nd approach using  $O(N)$  time and space complexity. However, I am sure it wouldn't make much of a difference between  $O(N \log N)$  and  $O(N)$  as there are only 1000 numbers given as input. What would be ideal? Approach 2 or 3?

1 ⏺ April 8, 2020 11:00 PM

sergitk ★ 10 ⏺ April 8, 2020 4:20 AM

Python collections.Counter

```
class Solution:
    def countElements(self, arr: List[int]) -> int:
        arr = Counter(arr)
        count = 0
        for each x in arr:
            if arr[x] >= 2:
                count += arr[x] - 1
        return count
```

Read More

vanhaxi ★ 157 ⏺ April 8, 2020 2:12 AM

SHOW 1 REPLY

Analysis written by [@hai\\_dee](#)

Rate this article: ★ ★ ★ ★ ★

#### Comments: 4

Type comment here... (Markdown is supported)

© Preview Post

probudho ★ 13 ⏺ April 8, 2020 11:00 PM

The solution above uses two passes. Here is one pass python solution:

```
class Solution(object):
    def countElements(self, arr):
        sort arr
        count = 0
        run_length = 1
        for each i in range 1 to arr.length - 1:
            if arr[i - 1] is not equal to arr[i]:
                if arr[i - 1] + 1 is equal to arr[i]:
                    count = count + run_length
                run_length = 0
            run_length = run_length + 1
        return count
```

Read More

nishadkumar ★ 40 ⏺ April 7, 2020 11:00 PM

Nice idea for the 3rd approach to use sorting! I went with the 2nd approach using  $O(N)$  time and space complexity. However, I am sure it wouldn't make much of a difference between  $O(N \log N)$  and  $O(N)$  as there are only 1000 numbers given as input. What would be ideal? Approach 2 or 3?

1 ⏺ April 8, 2020 11:00 PM

sergitk ★ 10 ⏺ April 8, 2020 4:20 AM

Python collections.Counter

```
class Solution:
    def countElements(self, arr: List[int]) -> int:
        arr = Counter(arr)
        count = 0
        for each x in arr:
            if arr[x] >= 2:
                count += arr[x] - 1
        return count
```

Read More

vanhaxi ★ 157 ⏺ April 8, 2020 2:12 AM

SHOW 1 REPLY