Previous Next

540. Single Element in a Sorted Array

Oct. 30, 2019 | 22.6K views

Average Rating: 4.94 (36 votes)

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.

Follow up: Your solution should run in O(log n) time and O(1) space.

Example 1:

```
Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2
```

Example 2:

```
Input: nums = [3,3,7,7,10,11,11]
Output: 10
```

Constraints:

- `1 <= nums.length <= 10^5`
- `0 <= nums[i] <= 10^5`

Solution

Approach 1: Brute Force

Intuition

We can use a linear search to check every element in the array until we find the single element.

Algorithm

Starting with the first element, we iterate over every 2nd element, checking whether or not the next element is the same as the current. If it's not, then we know this must be the single element.

If we get as far as the last element, we know that it must be the single element. We need to treat it as a special case after the loop, because otherwise we'll be going over the end of the array.

C++ Java Python Copy

```python
def singleNonDuplicate(self, nums: List[int]) -> int:
    for i in range(0, len(nums) - 2, 2):
        if nums[i] != nums[i + 1]:
            return nums[i]
    return nums[-1]
```

Complexity Analysis

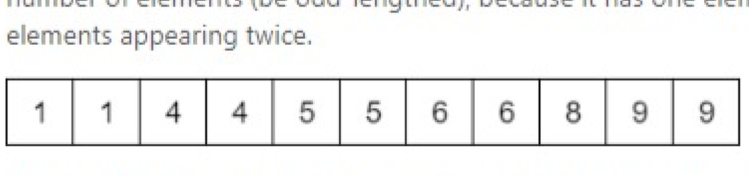
- Time complexity : $O(n)$. For linear search, we are looking at every element in the array once.
- Space complexity : $O(1)$. We are only using constant extra space.

While this approach will work, the question tells us we need a $O(\log n)$ solution. Therefore, this solution isn't good enough.

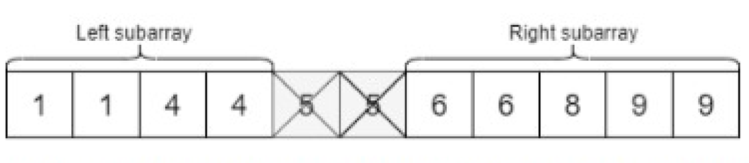
Approach 2: Binary Search

Intuition

It makes sense to try and convert the linear search into a binary search. In order to use binary search, we need to be able to look at the middle item and then determine whether the solution is the middle item, or to the left, or to the right. The key observation to make is that the starting array must always have an odd number of elements (be odd-lengthed), because it has one element appearing once, and all the other elements appearing twice.



Here is what happens when we remove a *pair* from the center. We are left with a left subarray and a right subarray.



Like the original array, the subarray containing the single element must be odd-lengthed. The subarray *not* containing it must be even-lengthed. So by taking a pair out of the middle and then calculating which side is now odd-lengthed, we have the information needed for binary search.

Algorithm

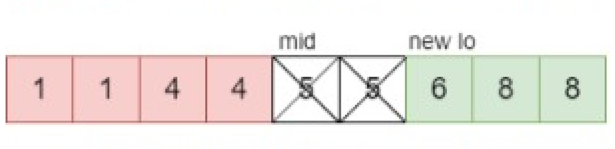
We start by setting `lo` and `hi` to be the lowest and highest index (inclusive) of the array, and then iteratively halve the array until we find the single element or until there is only one element left. We know that if there is only one element in the search space, it must be the single element, so should terminate the search.

On each loop iteration, we find `mid`, and determine the odd/ evenness of the sides and save it in a variable called `halvesAreEven`. By then looking at which half the middle element's *partner* is in (either last element in the left subarray or first element in the right subarray), we can decide which side is now (or remained) odd-lengthed and set `lo` and `hi` to cover the part of the array we now know the single element must be in.

The trickiest part is ensuring we update `lo` and `hi` correctly based on the values of `mid` and `halvesAreEven`. These diagrams should help you understand the cases. When solving problems like this, it's often good to draw a diagram and think really carefully about it to avoid off-by-one errors. Avoid using a guess and check approach.

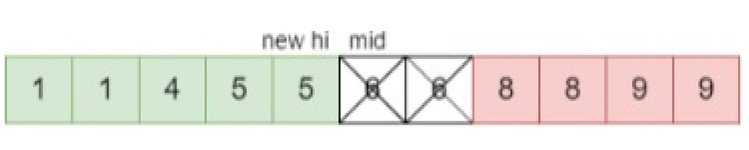
Case 1: Mid's partner is to the right, and the halves were originally even.

The right side *becomes* odd-lengthed because we removed `mid`'s partner from it. We need to set `lo` to `mid + 2` so that the remaining array is the part above `mid`'s partner.



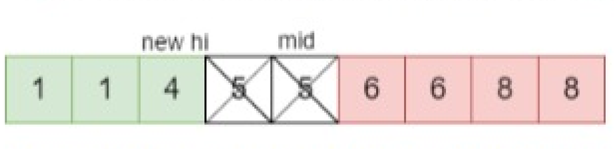
Case 2: Mid's partner is to the right, and the halves were originally odd.

The left side *remains* odd-lengthed. We need to set `hi` to `mid - 1` so that the remaining array is the part below `mid`.



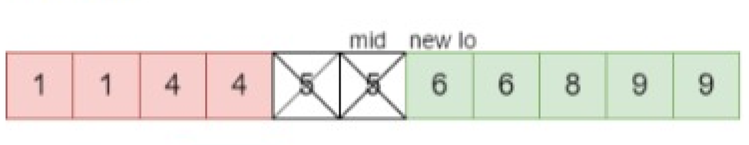
Case 3: Mid's partner is to the left, and the halves were originally even.

The left side *becomes* odd-lengthed because we removed `mid`'s partner from it. We need to set `hi` to `mid - 2` so that the remaining array is the part below `mid`'s partner.



Case 4: Mid's partner is to the left, and the halves were originally odd.

The right side *remains* odd-lengthed. We need to set `lo` to `mid + 1` so that the remaining array is the part above `mid`.



C++ Java Python Copy

```python
def singleNonDuplicate(self, nums: List[int]) -> int:
    lo = 0
    hi = len(nums) - 1
    while lo < hi:
        mid = lo + (hi - lo) // 2
        halves_are_even = (hi - mid) % 2 == 0
        if nums[mid + 1] == nums[mid]:
            if halves_are_even:
                lo = mid + 2
            else:
                hi = mid - 1
        elif nums[mid - 1] == nums[mid]:
            if halves_are_even:
                hi = mid - 2
            else:
                lo = mid + 1
        else:
            return nums[mid]
    return nums[lo]
```

Another interesting observation you might have made is that this algorithm will still work even if the array isn't fully sorted. As long as pairs are always grouped together in the array (for example, `[10, 10, 4, 4, 7, 11, 11, 12, 12, 2, 2]`), it doesn't matter what order they're in. Binary search worked for this problem because we knew the subarray with the single number is always odd-lengthed, not because the array was fully sorted numerically. We commonly call this an *invariant*, something that is always true (i.e. "The array containing the single element is always odd-lengthed"). Be on the lookout for invariants like this when solving array problems, as binary search is very flexibile!

Complexity Analysis

- Time complexity : $O(\log n)$. On each iteration of the loop, we're *halving* the number of items we still need to search.
- Space complexity : $O(1)$. We are only using constant space to keep track of where we are in the search.

Approach 3: Binary Search on Evens Indexes Only

It turns out that we only need to binary search on the even indexes. This approach is more elegant than the last, although both are good solutions.

Intuition

The single element is at the *first* even index *not* followed by its pair. We used this property in the linear search algorithm, where we iterated over all of the *even indexes* until we encountered the first one not followed by its pair.

Instead of linear searching for this index though, we can binary search for it.

After the single element, the pattern changes to being odd indexes followed by their pair. This means that the single element (an even index) and all elements after it are even indexes *not* followed by their pair. Therefore, given any even index in the array, we can easily determine whether the single element is to the left or to the right.

Algorithm

We need to set up the binary search variables and loop so that we are only considering *even* indexes. The last index of an odd-lengthed array is always even, so we can set `lo` and `hi` to be the start and end of the array.

We need to make sure our `mid` index is *even*. We can do this by dividing `lo` and `hi` in the usual way, but then *decrementing* it by `1` if it is odd. This also ensures that if we have an even number of even indexes to search, that we are getting the *lower middle* (*incrementing* by 1 here would not work, it'd lead to an infinite loop as the search space would not be reduced in some cases).

Then we check whether or not the `mid` index is the same as the one after it. - *If it is*, then we know that `mid` is not the single element, and that the single element must be at an even index *after* `mid`. Therefore, we set `lo` to be `mid + 2`. It is `+2` rather than the usual `+1` because we want it to point at an even index. - *If it is not*, then we know that the single element is either at `mid`, or at some index *before* `mid`. Therefore, we set `hi` to be mid.

Once `lo == hi`, the search space is down to 1 element, and this must be the single element, so we return it.

C++ Java Python Copy

```python
def singleNonDuplicate(self, nums: List[int]) -> int:
    lo = 0
    hi = len(nums) - 1
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if mid % 2 == 1:
            mid -= 1
        if nums[mid] == nums[mid + 1]:
            lo = mid + 2
        else:
            hi = mid
    return nums[lo]
```

Complexity Analysis

- Time complexity : $O(\log \frac{n}{2}) = O(\log n)$. Same as the binary search above, except we are only binary searching half the elements, rather than all of them.
- Space complexity : $O(1)$. Same as the other approaches. We are only using constant space to keep track of where we are in the search.

Rate this article:

Previous Next

Comments: 13 Sort By

Type comment here... (Markdown is supported)

Preview Post

himanshusingh11 ★ 97 January 2, 2020 11:51 PM
@Hai_dee Very good explanation! :)
6 Share Reply

Nitkau ★ 8 October 31, 2019 5:24 PM Report
We can use XOR operation and find the odd value out
8 Share Reply
SHOW 4 REPLIES

alexander1089 ★ 12 May 13, 2020 6:38 AM
In binary search I am always confused if to use while(lo < hi) or while(lo <= hi).
How do we determine which one to use ?
2 Share Reply
SHOW 2 REPLIES

BlueJP ★ 53 April 8, 2020 6:14 AM
How to determine halvesAreEven is check left side or right side?
2 Share Reply
SHOW 1 REPLY

soumyajitchatterjee73 ★ 15 November 1, 2019 9:36 AM
Solution O(log N):

```
 ```
private static int getSingleElement(int [] a, int l, int h) {
        if(h >= l) {
```

Read More
2 Share Reply

joanromano ★ 106 October 30, 2019 5:26 PM
Really well explained, thank you so much for the detailed explanation on binary search
2 Share Reply

s961206 ★ 753 May 15, 2020 7:45 PM Report
Could u give a simple proof of correctness for approach 3 rather than just giving a magic algorithm...
0 Share Reply

ftwpker ★ 15 May 13, 2020 5:45 AM
Why do we have to return nums[lo] at the end? Wouldn't it be caught before?
0 Share Reply
SHOW 1 REPLY

warrenmo ★ 0 May 13, 2020 1:10 AM
Great write-up!
In reference to the quote "*incrementing* by 1 here would not work, it'd lead to an infinite loop...," can someone (@Hai_dee or anyone really) provide some guidance on how to preemptively prevent such infinite loops? Is it just a matter of "diagramming" in the beginning? Doing more binary search problems? I've read up on the wikipedia page and completed a handful of binary search problems w/o
Read More
0 Share Reply
SHOW 3 REPLIES

Squirtle_21lbs ★ 0 April 30, 2020 10:14 AM
Great explanation! For approach 3, can you pls give a sample for below pls? on what case the incrementing approach will lead to an infinite loop? assuming that if we use incrementing, we will change to compare nums[mid] vs nums[mid-1] and set hi = mid -2 if they equal

"This also ensures that if we have an even number of even indexes to search, that we are getting the
Read More
0 Share Reply

< 1 2 >
```