Solution: Single Element in a Sorted Array

Let's solve the Single Element in a Sorted Array problem using the Modified Binary Search pattern.

We'll cover the following

Statement

You are given a sorted array of integers, `nums`, where all integers appear twice except for one. Your task is to find and return the single integer that appears only once.

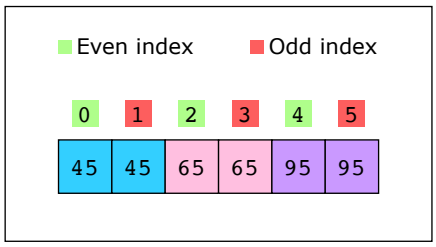
The solution should have a time complexity of $O(\log n)$ or better and a space complexity of $O(1)$.

Constraints:

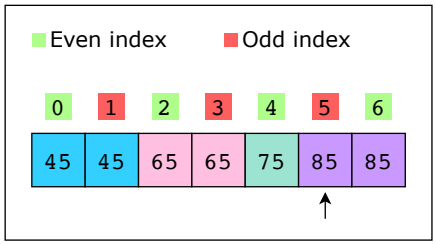
- $1 \leq$ `nums.length` $\leq 10^3$
- $0 \leq$ `nums[i]` $\leq 10^3$

Solution

Let's make a couple of observations before we go to the actual solution. Suppose that all the elements of `nums` appear twice. We can say that all the elements are in pairs. In each pair, the first element of the pair is at the even index of `nums`, and the second element is at the odd index of `nums`.



The pattern of having the first elements of pairs at the even index and the second elements at the odd index breaks when we have an extra element in `nums` that only exists once.



Keeping these observations in mind, we use binary search to find the single non-duplicate element in `nums`. The solution involves three pointers: `left`, `mid`, and `right`. If, at any point, `mid` is an odd numbered index, we change it to an even numbered index. This makes it easier to check whether `mid` is part of a pair of integers in

`nums`. Then, we conditionally move the `left` and `right` pointers until we reach to the non-duplicate element in `nums`.

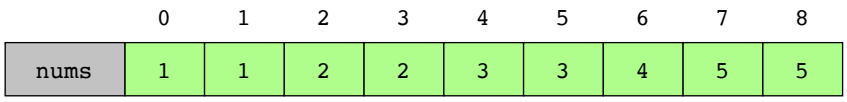
The observations discussed above imply that if `mid` is even and all the elements until `mid` appear in pairs, then a new pair of identical elements will be found at `mid` and `mid + 1`. However, if an unpaired element appears before `mid`, then the elements at `mid` and `mid + 1` will be different.

The algorithm works as follows:

1. Initialize `left` to the leftmost index (0) and `right` to the rightmost index (`nums.length - 1`).
2. Calculate `mid` using the formula $mid = left + \lfloor \frac{(right - left)}{2} \rfloor$. We use this formula to avoid any integer overflow during the calculation.
    - If `mid` is odd, decrement it by 1 to make it an even index.
3. Check whether `nums[mid]` is the same as `nums[mid + 1]`.
    - If both are the same, it means that all elements up to this point were in pairs, and the single element must appear after `mid`. Therefore, move the `left` pointer toward the right.
    - If both are different, it means that the single element must have appeared before `mid`. Therefore, move the `right` pointer toward the left.
4. Repeat steps 2 to 4 until `left` becomes equal to `right`.

Eventually, all the pointers will be pointing to the same element, so return that element as the output.

The slides below illustrate the steps of the solution in detail:



Given the following sorted array, nums, find the single element.

1 of 6

Let's look at the code for this solution:

Java

```java
class SingleElement {
    public static int singleNonDuplicate(int[] nums) {

        // initilaize the left and right pointer
        int l = 0;
        int r = nums.length - 1;
        while (l < r) {

            // if mid is odd, decrement it to make it even
            int mid = l + (r - l) / 2;
            if (mid % 2 == 1) mid--;

            // if the elements at mid and mid + 1 are the same, then the single element must appear after
```

```
            if (nums[mid] == nums[mid + 1]) {
                l = mid + 2;
            }
                r = mid;
            }
        }
        return nums[l];
    }

    // driver code
    public static void main(String[] args) {
        int[][] inputs = {
```

Single Element in a Sorted Array

Time complexity

Since we conduct a binary search on the elements of `nums`, the time complexity of this solution is $O(\log n)$, where n is the number of elements in `nums`.

Space complexity

The space complexity of this solution is $O(1)$.

Back

Single Element in a So...

Next

Search in Rotated Sort...

Mark as Completed