Search Single Element in a sorted array

Problem Statement: Given an array of N integers. Every number in the array except one appears twice. Find the single number in the array.

Example 1:

Input Format: arr[] = {1,1,2,2,3,3,4,5,5,6,6}

Result: 4

Explanation: Only the number 4 appears once in the array.

Example 2:

Input Format: arr[] = {1,1,3,5,5}

Result: 3

Explanation: Only the number 3 appears once in the array.

Naive Approach(Brute force):

A crucial observation to note is that if an element appears twice in a sequence, either the preceding or the subsequent element will also be the same. But only for the single element, this condition will not be satisfied. So, to check this the condition will be the following:

- **If arr[i] != arr[i-1] and arr[i] != arr[i+1]:** If this condition is true for any element, arr[i], we can conclude this is the single element.

Edge Cases:

1. **If n == 1:** This means the array size is 1. If the array contains only one element, we will return that element only.
2. **If i == 0:** This means this is the very first element of the array. The only condition, we need to check is: **arr[i] != arr[i+1].**
3. **If i == n-1:** This means this is the last element of the array. The only condition, we need to check is: **arr[i] != arr[i-1].**

So, we will traverse the array and we will check for the above conditions.

Algorithm:

The steps are as follows:

1. At first, we will check if the array contains only 1 element. If it is, we will simply return that element.
2. We will start traversing the array. Then for every element, we will check the following.
3. **If i == 0:** If we are at the first index, we will check if the next element is equal.
    1. **If arr[i] != arr[i+1]:** This means arr[i] is the single element and so we will return arr[i].
4. **If i == n-1:** If we are at the last index, we will check if the previous element is equal.
    1. **If arr[i] != arr[i-1]:** This means arr[i] is the single element and so we will return arr[i].
5. For the elements other than the first and last, we will check:
   If arr[i] != arr[i-1] and arr[i] != arr[i+1]: If this condition is true for any element, arr[i], we can conclude this is the single element. And we should return arr[i].

```
#include <bits/stdc++.h>
using namespace std;

int singleNonDuplicate(vector<int>& arr) {
    int n = arr.size(); //size of the array.
    if (n == 1) return arr[0];

    for (int i = 0; i < n; i++) {

        //Check for first index:
        if (i == 0) {
            if (arr[i] != arr[i + 1])
                return arr[i];
        }
        //Check for last index:
        else if (i == n - 1) {
            if (arr[i] != arr[i - 1])
                return arr[i];
        }
        else {
            if (arr[i] != arr[i - 1] && arr[i] != arr[i + 1])
                return arr[i];
        }
    }

    // dummy return statement:
    return -1;
}
```

```
int main()
{
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    int ans = singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}
```

Complexity Analysis

Time Complexity: O(N), N = size of the given array.
Reason: We are traversing the entire array.

Space Complexity: O(1) as we are not using any extra space.

Optimal Approach(Using Binary Search):

We are going to use the Binary Search algorithm to optimize the approach.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

We need to consider 2 different cases while using Binary Search in this problem. Binary Search works by reducing the search space by half. So, at first, **we need to identify the halves and then eliminate them accordingly**. In addition to that, **we need to check if the current element i.e. arr[mid] is the 'single element'**.

If we can resolve these two cases, we can easily apply Binary Search in this algorithm.

How to check if arr[mid] i.e. the current element is the single element:

A crucial observation to note is that if an element appears twice in a sequence, either the preceding or the subsequent element will also be the same. But only for the single element, this condition will not be satisfied. So, to check this, the condition will be the following:

- **If arr[mid] != arr[mid-1] and arr[mid] != arr[mid+1]:** If this condition is true for arr[mid], we can conclude arr[mid] is the single element.

The above condition will throw errors in the following 3 cases:

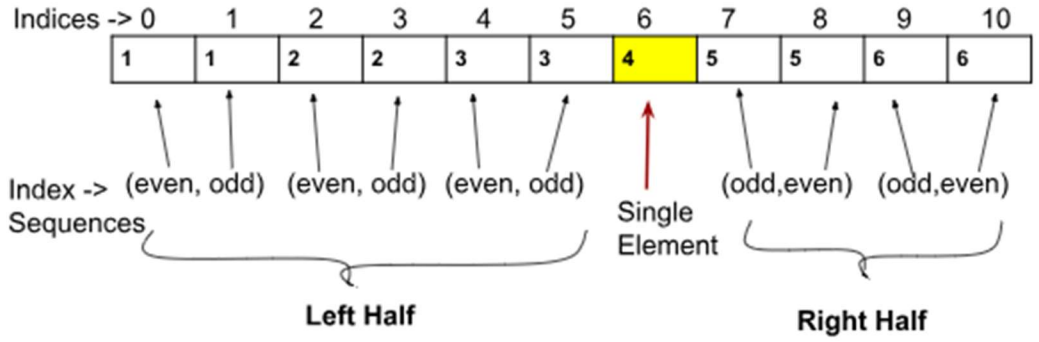
- **If the array size is 1.**
- **If 'mid' points to 0 i.e. the first index.**
- **If 'mid' points to n-1 i.e. the last index.**

Note: *At the start of the algorithm, we address the above edge cases without requiring separate conditions during the check for arr[mid] inside the loop. And the search space will be from index 1 to n-2 as indices 0 and n-1 have already been checked.*

Resolving edge cases:

1. **If n == 1:** This means the array size is 1. If the array contains only one element, we will return that element only.
2. **If arr[0] != arr[1]:** This means the very first element of the array is the single element. So, we will return arr[0].
3. **If arr[n-1] != arr[n-2]:** This means the last element of the array is the single element. So, we will return arr[n-1].

How to identify the halves:



By observing the above image, we can clearly notice a striking distinction between the index sequences of the left and right halves of the single element in the array.

1. The index sequence of the duplicate numbers in the left half is always (even, odd). That means one of the following conditions will be satisfied if we are in the left half:
    1. **If the current index is even, the element at the next odd index will be the same as the current element.**
    2. **Similarly, If the current index is odd, the element at the preceding even index will be the same as the current element.**
2. The index sequence of the duplicate numbers in the right half is always (odd, even). That means one of the following conditions will be satisfied if we are in the right half:
    1. **If the current index is even, the element at the preceding odd index will be the same as the current element.**
    2. **Similarly, If the current index is odd, the element at the next even index will be the same as the current element.**

Now, we can easily identify the left and right halves, just by checking the sequence of the current index, i, like the following:

- **If (i % 2 == 0 and arr[i] == arr[i+1]) or (i%2 == 1 and arr[i] == arr[i-1]),** we are in the left half.

- **If (i % 2 == 0 and arr[i] == arr[i-1]) or (i%2 == 1 and arr[i] == arr[i+1]),** we are in the right half.

Note: *In our case, the index i refers to the index 'mid'.*

How to eliminate the halves:

- If we are in the left half of the single element, we have to eliminate this left half (i.e. low = mid+1). Because our single element appears somewhere on the right side.
- If we are in the right half of the single element, we have to eliminate this right half (i.e. high = mid-1). Because our single element appears somewhere on the left side.

Now, we have resolved the problems and we can use the binary search accordingly.

Algorithm:

The steps are as follows:

1. **If n == 1:** This means the array size is 1. If the array contains only one element, we will return that element only.
2. **If arr[0] != arr[1]:** This means the very first element of the array is the single element. So, we will return arr[0].
3. **If arr[n-1] != arr[n-2]:** This means the last element of the array is the single element. So, we will return arr[n-1].
4. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers excluding index 0 and n-1 like this: low will point to index 1, and high will point to index n-2 i.e. the second last index.
5. **Calculate the 'mid':** Now, inside a loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)

6. **Check if arr[mid] is the single element:**
   If arr[mid] != arr[mid-1] and arr[mid] != arr[mid+1]: If this condition is true for arr[mid], we can conclude arr[mid] is the single element. We will return arr[mid].
7. **If (mid % 2 == 0 and arr[mid] == arr[mid+1])**
   or (mid%2 == 1 and arr[mid] == arr[mid-1]): This means we are in the left half and we should eliminate it as our single element appears on the right. So, we will do this:
   low = mid+1.
8. **Otherwise,** we are in the right half and we should eliminate it as our single element appears on the left. So, we will do this: high = mid-1.

```
#include <bits/stdc++.h>
using namespace std;

int singleNonDuplicate(vector<int>& arr) {
    int n = arr.size(); //size of the array.

    //Edge cases:
    if (n == 1) return arr[0];
    if (arr[0] != arr[1]) return arr[0];
    if (arr[n - 1] != arr[n - 2]) return arr[n - 1];

    int low = 1, high = n - 2;
    while (low <= high) {
        int mid = (low + high) / 2;
```

```
        //if arr[mid] is the single element:
        if (arr[mid] != arr[mid + 1] && arr[mid] != arr[mid - 1]) {
            return arr[mid];
        }

        //we are in the left:
        if ((mid % 2 == 1 && arr[mid] == arr[mid - 1])
                || (mid % 2 == 0 && arr[mid] == arr[mid + 1])) {
            //eliminate the left half:
            low = mid + 1;
        }
        //we are in the right:
        else {
            //eliminate the right half:
            high = mid - 1;
        }
    }

    // dummy return statement:
    return -1;
}

int main()
{
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    int ans = singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}
```

Complexity Analysis

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using the Binary Search algorithm.

Space Complexity: O(1) as we are not using any extra space.