Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array where `num[i] ≠ num[i+1]`, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that `num[-1] = num[n] = -∞`.

For example, in array `[1, 2, 3, 1]`, 3 is a peak element and your function should return the index number 2.

click to show spoilers.

Note:
Your solution should be in logarithmic complexity.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 1

Consider that each local maximum is one valid peak. My solution is to find one local maximum with binary search. Binary search satisfies the O(logn) computational complexity.

Binary Search: recursion

```
class Solution {
public:

int findPeakElement(const vector<int> &num) {
    return Helper(num, 0, num.size()-1);
}
int Helper(const vector<int> &num, int low, int high)
{
    if(low == high)
        return low;
    else
    {
        int mid1 = (low+high)/2;
        int mid2 = mid1+1;
        if(num[mid1] > num[mid2])
            return Helper(num, low, mid1);
        else
            return Helper(num, mid2, high);
    }
}
};
```

Binary Search: iteration

```
class Solution {
public:
    int findPeakElement(const vector<int> &num)
    {
        int low = 0;
        int high = num.size()-1;

        while(low < high)
        {
            int mid1 = (low+high)/2;
            int mid2 = mid1+1;
            if(num[mid1] < num[mid2])
                low = mid2;
            else
                high = mid1;
        }
        return low;
    }
};
```

Sequential Search:

```
class Solution {
public:
    int findPeakElement(const vector<int> &num) {
        for(int i = 1; i < num.size(); i ++)
        {
            if(num[i] < num[i-1])
            {// <
                return i-1;
            }
        }
        return num.size()-1;
    }
};
```

written by gangan original link here

Solution 2

This problem is similar to Local Minimum. And according to the given condition, num[i] != num[i+1], there must exist a O(logN) solution. So we use binary search for this problem.

- If num[i-1] < num[i] > num[i+1], then num[i] is peak
- If num[i-1] < num[i] < num[i+1], then num[i+1...n-1] must contains a peak
- If num[i-1] > num[i] > num[i+1], then num[0...i-1] must contains a peak
- If num[i-1] > num[i] < num[i+1], then both sides have peak (n is num.length)

Here is the code

```
public int findPeakElement(int[] num) {
    return helper(num,0,num.length-1);
}

public int helper(int[] num,int start,int end){
    if(start == end){
        return start;
    }else if(start+1 == end){
        if(num[start] > num[end]) return start;
        return end;
    }else{

        int m = (start+end)/2;

        if(num[m] > num[m-1] && num[m] > num[m+1]){

            return m;

        }else if(num[m-1] > num[m] && num[m] > num[m+1]){

            return helper(num,start,m-1);

        }else{

            return helper(num,m+1,end);

        }

    }
}
```

written by xctom original link here

Solution 3

I find it useful to reason about binary search problems using invariants. While there are many solutions posted here, neither of them provide (in my opinion) a good explanation about why they work. I just spent some time thinking about this and I thought it might be a good idea to share my thoughts.

Assume we initialize left = 0, right = nums.length - 1. The invariant I'm using is the following:

nums[left - 1] < nums[left] && nums[right] > nums[right + 1]

That basically means that in the current interval we're looking, [left, right] the function started increasing to left and will eventually decrease at right. The behavior between [left, right] falls into the following 3 categories:

1) nums[left] > nums[left + 1]. From the invariant, nums[left - 1] < nums[left] => left is a peak

2) The function is increasing from left to right i.e. nums[left] < nums[left + 1] < .. < nums[right - 1] < nums[right]. From the invariant, nums[right] > nums[right + 1] => right is a peak

3) the function increases for a while and then decreases (in which case the point just before it starts decreasing is a peak) e.g. 2 5 6 3 (6 is the point in question)

As shown, if the invariant above holds, there is at least a peak between [left, right]. Now we need to show 2 things:

I) the invariant is initially true. Since left = 0 and right = nums.length - 1 initially and we know that nums[-1] = nums[nums.length] = -oo, this is obviously true

II) At every step of the loop the invariant gets reestablished. If we consider the code in the loop, we have mid = (left + right) / 2 and the following 2 cases:

a) nums[mid] < nums[mid + 1]. It turns out that the interval [mid + 1, right] respects the invariant (nums[mid] < nums[mid + 1] -> part of the cond + nums[right] > nums[right + 1] -> part of the invariant in the previous loop iteration)

b) nums[mid] > nums[mid + 1]. Similarly, [left, mid] respects the invariant (nums[left - 1] < nums[left] -> part of the invariant in the previous loop iteration and nums[mid] > nums[mid + 1] -> part of the cond)

As a result, the invariant gets reestablished and it will also hold when we exit the loop. In that case we have an interval of length 2 i.e. right = left + 1. If nums[left] > nums[right], using the invariant (nums[left - 1] < nums[left]), we get that left is a peak. Otherwise right is the peak (nums[left] < nums[right] and nums[right] < nums[right + 1] from the invariant).

```java
public int findPeakElement(int[] nums) {
    int N = nums.length;
    if (N == 1) {
        return 0;
    }

    int left = 0, right = N - 1;
    while (right - left > 1) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < nums[mid + 1]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return (left == N - 1 || nums[left] > nums[left + 1]) ? left : right;
}
```

I hope this makes things clear despite the long explanation.

written by cosmin79 original link here