Assignment 3: Searching

1. Find Peak in a Array

Description

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] = -∞`.

Examples

Example 1

> input: [1,8,9,5]
>
> output:9

> Explaination: The number `9` is the peak becuase it is greater than its neighbors. So return the index of `9`.

Example 2

> input: [9,8,5,1]
>
> output: 9
>
> Explaination: The number `9` is the peak becuase it is greater than its neighbors (its left is -∞).

Example 3

> input: [1,5,8,9]
>
> output: 9
>
> Explaination: The number `9` is the peak becuase it is greater than its neighbors (its right is -∞).

Solutions

Solution 1: Use intuition

Algorithm

The idea is simple: go though the array and find the max number in flight.

Implemenation

(This is very straightforward so I would rather spend my time on other high effenciecy solutions.)

Complexity

Time: $O(n)$

Space: $O(1)$

Solution 2: Use 'binary search'

Algorithm

This could be improved to a complexity of $O(log(n))$ by using 'binary search' strategy.

- use two pointers low and high, one of which from the start and the other one from the end
- compare the middle element between the two pointers with its next
- if the next is bigger, set the low pointer to mid+1
- otherwise if it is smaller, set the high pointer to mid
- continue to do when low < high
- when eixt, low equals high and either of nums[low] or nums[high] is the result

Implemenation

Java

```java
    public static int findPeakElement(int[] nums) {
        int low = 0, high = nums.length - 1;

        while (low < high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] < nums[mid + 1]) {
                low = mid + 1;
            } else if (nums[mid] > nums[mid + 1]) {
                high = mid; // trick
            }
        }
        return nums[low];
    }
```

Comoplexity

Time: $O(log(n))$. Each time the array is devided to two, one of which is droped.

Space: $O(1)$. Constant space is used.

References

Find Minimum in Rotated Sorted Array

2. Find Pair Where Sum is Closest to 0

Description

Given an integer array, you need to find the two elements such that their sum is closest to zero and print them in ascending order.

Note: If there are more than two sums equal close to 0, output any pair if fine.

Examples

> *Example 1*
>
> input: -8 -66 -60
>
> output: -60 -8

> *Example 2*
>
> input: -21 -67 -37 -18 4 -65
>
> output: -18 4

> *Example 3*
>
> input: -24 -73
>
> output: -73 -24

Algorithm

Sort this array in asending order and use two pointers and move following this rule:

- if the sum of the low index and high index number is 0, return these two numbers directly
- if the sum is greater than 0, move the high pointer left if the new sum is closer to 0
- otherwise, move the low right if the new sum is closer to 0
- if the new sum is further from 0, then stop anytime in the previous two steps

Solution: Two pointers

Java

```
    public static int[] findClosestToZero(int[] nums) {
        if (nums == null || nums.length < 2)
            throw new IllegalArgumentException();
        Arrays.sort(nums);

        int low = 0, high = nums.length - 1;
        while (low < high - 1) {
            int sum = nums[low] + nums[high];
            if (sum == 0)
                break;
            else if (sum < 0) {
                if (Math.abs(nums[low + 1] + nums[high] - 0) > Math.abs(sum
- 0))
                    break;
                low++;
            } else {
                if (Math.abs(nums[low] + nums[high - 1] - 0) > Math.abs(sum
- 0))
                    break;
                high--;
            }
        }
        return new int[] { nums[low], nums[high] };
    }
```

Complexity

Time: $O(n)$. In the worst case (such as the middle two elements are the result), all the number in this array will be visted.

Space: $O(1)$. The sorting is in space and the used sapce is constant.

References

Two numbers with sum closest to zero