Leetcode→[Array] 1

485.[Array] find maximum consecutive numbers in Array.

Method: 双指针 two pointers

```
func findMaxConsecutiveOnes(_ nums:[Int]) -> Int {

     var counter = 0
     var pointer = 0


        for i in 0..<nums.count{
            pointer = (nums[i] == 1) ? (counter + 1) : 0
            counter = (counter > pointer) ? counter : pointer
        }

        return counter
}
```

Time Complexity: O(N), where N is the number of elements in the array.
Space Complexity: O(1). We do not use any extra space.

1295.[Array] find numbers with Even Number of digits.

method 1:String

```
func findNumbers(_nums:[Int])->Int{
    return nums.filter({String($0).count.isMultiple(of: 2)}).count
}
```

method 2:

```
func findNumbers(_ nums: [Int]) -> Int {
var count = 0
for i in nums {
count += String(i).count % 2 == 0 ? 1 : 0
}
return count
}
```

977. [Array] Squares of a sorted() array.

3 methods :sort()

```
func sortedSquares(_ nums: [Int])->[Int]{
        var myArray = [Int]()

        for i in 0..<nums.count{
            myArray.append(nums[i]*nums[i])}

        myArray.sort()
        return myArray
}
```

//Time Complexity: O(N logN), where N is the length of A.
//Space complexity : O(N) or O(logN). The space complexity of the sorting algorithm,depends on implementation of each programming language.For instance, in Python, list.sort() is using Timesort() algorithm, which space complexity is O(N). In Java, sorting() is using quicksort() algorithm, which space complexity is O(logN).

```
func sortedSquares(_ nums: [Int]) -> [Int] {
        guard nums.count > 0 else {
            return []
        }

        let squared = nums.map({ $0 * $0 })
        return squared.sorted()
    }
```

```
func sortedSquares(_ nums: [Int]) -> [Int] {
        return nums.map({ $0 * $0 }).sorted()
    }
```

1089. [Array]→Duplicate Zeros

```
func duplicateZeros(_ arr: inout [Int]) {

          var index = 0

          while (index < arr.count) {
            if arr[index] == 0 {
                index += 1
                arr.insert(0, at: index)
                arr.removeLast()
            }
            index += 1
        }

    }
```

Complexity Analysis

- Time Complexity: O(N), where *N* is the number of elements in the array. We do two passes through the array, one to find the number of `possible_dups` and the other to copy the elements. In the worst case we might be iterating the entire array, when there are less or no zeros in the array.
- Space Complexity: O(1). We do not use any extra space.

88.[Array]→Merge Sorted Array

```
func merge(_ nums1: inout [Int], _ m: Int, _ nums2: [Int], _ n: Int){

nums1 = (nums1[0..<m] + nums2[0..<n]).sorted(by: <)
}
```

Complexity Analysis

- Time complexity : O((n + m) log(n + m).)
- Space complexity: O(n).

27 .Remove element.

```
class Solution {
    func removeElement(_ nums: inout [Int], _ val: Int) -> Int {
       
       nums = nums.filter { $0 != val }
       return nums.count
        
    }
}
```

time complexity: O(n)

space complexity: O(1)

26.Remove duplicates from sorted array .

解法1，直接用swift的set功能。

```
class Solution {
    func removeDuplicates(_ nums: inout [Int]) -> Int {
        
        if nums.count == nil {return 0}
        nums = Array(Set(nums)).sorted()
        return nums.count
        
    }
}
```

time complexity:O(n)

space complexity: O(1)

解法2，双指针（推荐）。

```
class Solution {
    func removeDuplicates(_ nums: inout [Int]) -> Int {
        
     if nums.count == 0{
         return 0
     }
        var slow = 1
        for fast in 1..<nums.count{
            if nums[fast-1] != nums[fast]{
                nums[slow]=nums[fast]
                slow+=1
            }
        }
        return slow

    }
}
```

1346. Check If N and Its Double Exist

```
func checkIfExist(_ arr: [Int]) -> Bool {

            var numSet = Set<Int>()

        for i in 0..<arr.count {
            let x = arr[i] * 2
            let y = arr[i]/2


            if numSet.contains(x) || (numSet.contains(y) && arr[i] % 2 == 0) {
                return true
            } else {
                numSet.insert(arr[i])
            }
        }

        return false

    }
```

941. Valid Mountain Array.

```
// （测试通过了）

let count = A.count
  if count < 3 { return false }

  var index = 1
  while index < A.count && A[index - 1] < A[index] {
    index += 1
  }

  if index == 1 || index == count { return false } // did not find a valid peak
  while index < A.count && A[index - 1] > A[index] {
    index += 1
  }

  return index == count // after 2 interactions, index should be at the end
```

```
// pseudocode. (测试没通过)



class Solution {
  func validMountainArray(_ arr: [Int]) -> Bool {
    var peak = 0
    // First iteration to find the peak
    for i in 0..<(arr.count-1) {
      // if ever encounter the next value is equal to the current one, return false
      if arr[i+1] == arr[i] {
        return false
      } else if arr[i+1] < arr[i] {
        // if the next one is smaller than the current one, then we find the peak
        peak = i
        break
      }
    }
    // we need to make sure the peak is neither the first one nor the last one
    // [1,2,3] -> false
    // [3,2,1] -> false
    if peak == 0 || peak == arr.count-1 {
      return false
    }
    // starting with the peak, the following values should getting smaller
    for j in peak..<(arr.count-1) {
      // if the enxt value is greater or equal to the current one, return false
      if arr[j+1] >= arr[peak] {
        return false
      }
      // otherwise move the current to the next one
      peak += 1
    }
    // Default to true
    return true
  }


  func validMountainArray2(_ arr: [Int]) -> Bool {
    var i = 0
    let j = arr.count
    // keep one index, if next one is greater than the current one, increase the index
    while i+1 < j && arr[i] < arr[i+1] {
      i += 1
    }
    // we need to make sure the peak is neither the first one nor the last one
    // [1,2,3] -> false
    // [3,2,1] -> false
    if i == 0 || i == j-1 {
      return false
    }
    // at this point i should be the peak, values following should be in a decreasing order
    while i+1 < j && arr[i] > arr[i+1] {
      i += 1
    }
    // only iterating through the whole array that we consider it's a valid mountain array
    return i == j-1
  }
}
```

Time complexity: O(n)

Space complexity: O(1)

1299.Replace Elements with Greatest Element on Right Side.

method 1

```
func replaceElements(_ arr: [Int]) -> [Int] {

  let lastIndex = arr.count - 1
  var nums = arr
  var curMax = nums[lastIndex]

  nums[lastIndex] = -1
  for i in stride(from: lastIndex - 1, through: 0, by: -1) {
    let temp = curMax
    curMax = max(curMax, nums[i])
    nums[i] = temp
  }

  return nums
}
```

method 2

```

func replaceElements(_ arr: [Int]) -> [Int] {

var answer = [Int](repeating: 0, count: arr.count)

        var currentMax = -1
        for i in stride(from: arr.count - 1, through: 0, by: -1) {
            answer[i] = currentMax
            if arr[i] > currentMax {
                currentMax = arr[i]
            }
        }

        return answer
}
```

283. Move Zeros.

```
func moveZeroes(_ nums: inout [Int]) {

var lastNonZero = 0
        for i in 0..<nums.count {
            if nums[i] != 0 {
                nums.swapAt(i, lastNonZero)
                lastNonZero += 1
            }
        }
}
```

```
func moveZeroes(_ nums: inout [Int]) {

        var i = 0, zero = 0
        while i < nums.count - zero {
            if nums[i] == 0 {
                zero += 1
                nums.append(0)
                nums.remove(at: i)
            } else {
                i += 1
            }
        }


    }
```

905. Sort array by parity.

```
func sortArrayByParity(_ A: [Int]) -> [Int] {
        var A = A
        var lastEven = 0
        for i in 0 ..< A.count {
            if A[i] % 2 == 0 {
                A.swapAt(i, lastEven)
                lastEven += 1
            }
        }
        return A
    }
```

ExplanationNote the line `var A = A`. Since arrays are value types in Swift we cannot modify the array without making a copy. If you saw this problem in an interview, it would be good to mention that we could improve the peformance by using `inout [Int]` as the parameter type since this would allow us to modify the array in place.

The `lastEven` variable will always point to the next available index for an even number. If we swap all the even numbers with `lastEven` then all the even numbers will be moved to the front of the array. At the same time the odd numbers will be moved to the end of the array.

1051. Height Checker

2 methods.

```
func heightChecker(_ heights: [Int]) ->Int{

        var expected = heights.sorted()
        var mismatches = 0

        for i in 0..<heights,count{
            if heights[i] != expected[i]{
                    mismatches+=1
}
}

         return mismatches
```

```
func heightChecker( _ heights: [Int])->Int{

   let expected = heights.sorted()
   return heights.indices.filter {heights[$0] != expected[$0]}.count
}
```

487.maximum consecutive ones 2

sliding window法：看youtube→[今天比昨天厉害]

```
func findMaxConsecutiveOnes(_ nums: [Int]) -> Int {
        var result = 0
        var left = 0
        var right = 0
        var zeros = 0
        while right < nums.count {
            if nums[right] == 0 {
                zeros += 1
            }

            while zeros > 1 {
                if nums[left] == 0 {
                    zeros -= 1
                }
                left += 1
            }

            result = max(result, right - left + 1)
            right += 1
        }

        return result
    }
```

```
func findMaxConsecutiveOnes(_ nums: [Int]) -> Int {
        var longest = 0
        var slow = 0
        var fast = 0
        var numberOfZeros = 0

        while fast < nums.count {
            if nums[fast] == 0 {
                numberOfZeros += 1
            }

            while (numberOfZeros == 2) {
                if nums[slow] == 0 {
                    numberOfZeros -= 1
                }
                slow += 1
            }

            longest = max(longest, fast-slow + 1)
            fast += 1
        }

        return longest
    }
```

```
func findMaxConsecutiveOnes(_ nums: [Int]) -> Int {


         var beforeFlip = 0
        var afterFlip = 0
        var maxLength = 0
        var current = 0

        for i in nums {
            if i == 1 {
             afterFlip += 1
            }

            if i == 0 {
                   // +1 here to include the flip
             beforeFlip = afterFlip + 1
             afterFlip = 0
            }

            current = beforeFlip + afterFlip
            maxLength = max(maxLength, current)
        }

        return maxLength

    }
```

414. Third maximum number.

2 methods.

```
func thirdMax(_ nums: [Int]) -> Int {

        return Set(nums).count < 3 ? nums.max()! : Set(nums).sorted { $0 > $1 }[2]
                                                               means排倒叙
    }
```

```
func thirdMax(_ nums: [Int]) -> Int {
        var values = Set<Int>()
        for num in nums {
            values.insert(num)
        }
        if values.count < 3 {
            return values.max()!
        } else {
            values.remove(values.max()!)
            values.remove(values.max()!)
            return values.max()!
        }
    }
```

448. Find all numbers disappears in the array.

```
func findDisappearedNumbers(_ nums: [Int]) -> [Int] {
```

```
var solution = Array(repeating: 0, count: nums.count)
        // O(n)
        for i in 1..<nums.count+1 {
            solution[i-1] = i
        }
        // O(n)
        for num in nums {
            solution[num-1] = -1
        }
        // O(n) https://developer.apple.com/documentation/swift/array/3017530-removeall
        solution.removeAll { $0 == -1 }
    // O(n) + O(n) + O(n) = O(n)
        return solution

}
}
```

```
看懂了。

func findDisappearedNumbers(_ nums: [Int]) -> [Int] {
    let n = nums.count
    var missing: [Int] = []
    let set = Set(nums)
    for num in 1...n {
        if !set.contains(num) {
            missing.append(num)
        }
    }
    return missing
}
```

```
func findDisappearedNumbers(_ nums: [Int]) -> [Int] {
        var nums = nums
        for i in 0..<nums.count {
            let n = abs(nums[i])
            if nums[n - 1] >= 0 {
                nums[n - 1] = -nums[n - 1]
            }
        }
        var missing = [Int]()
        for i in 0..<nums.count {
            if nums[i] >= 0 {
                missing.append(i + 1)
            }
        }
        return missing
    }
```

```
func findDisappearedNumbers(_ nums: [Int]) -> [Int] {
        (1...nums.count).filter { !nums.contains($0) }
    }
```