

Leetcode→[Array] 2

1. Two Sum

```
//has table

func twoSum(_ nums: [Int], _ target: Int) -> [Int] {


         var actualSearchNumber:Int;
        for i in 0..<nums.count-1 {
            actualSearchNumber = target - nums[i];
            for j in i+1..<nums.count {
                if nums[j] == actualSearchNumber {
                    return [i, j];
                }
            }
        }
        return [-1, -1];

    }
```

time complex: O(n)

space complex: O(n)

121. best time to sell stock

```
func maxProfit(_ prices: [Int]) -> Int {

        if prices.count < 2 { return  0 }

        var b = 0, p = 0

        for i in 0..<prices.count {
            var t =  prices[i] - prices[b]
            if t > p {
                p = t
            }else if t < 0 {
                b = i
            }
        }
        return p
```

```
    }
```

```
func maxProfit(_ prices: [Int]) -> Int {


       if prices.isEmpty { return 0 }

        var minValue = prices[0]
        var maxProfit = 0

        for price in prices {
            if (price < minValue) {
                minValue = price
            } else if price - minValue > maxProfit {
                maxProfit = price - minValue
            }
        }
        return maxProfit
    }
```

time complex:O(n)

space complex:O(1)

35. search insert position.

binary search (two pointers).

```
var low = 0
        var high = nums.count - 1
        while (low <= high) {
            let m = (low + high)/2
            if(nums[m] == target) {
                return m
            }
            else if (nums[m] > target) {
                high = m - 1
            }
            else if (target > nums[m]) {
                low = m + 1
            }
        }
        return low
    }
```

time complex: O(log n)

space complex: O(1)

11. container with most water

two-pointers.

```
func maxArea(_ height: [Int]) -> Int {


          var left = 0
        var right = height.count - 1
        var max = 0
        while left < right {
            let temp =  min(height[left],height[right]) * (right - left)
            if  temp > max {
                max = temp
            }
            if height[left] < height[right] {
                left += 1
            } else {
                right -= 1
            }
        }
        return max

    }
```

```
func maxArea(_ height: [Int]) -> Int {
        var left = 0
        var right = height.count - 1
        var maxArea = 0

        while left < right {
            maxArea = max(maxArea, min(height[left], height[right]) * (right - left))
            if height[left] < height[right] {
                left += 1
            } else {
                right -= 1
            }
        }

        return maxArea
    }
```

```
//The algorithm is as follows:
```

```
//Start with the edges
//Calculate area using width * min(height)
//if currentArea > maxAreaTill Now, update maxAreaTillNow
//pick the next height by shifting the pointer left or right from the smallest height
//Iterate until leftEdge < rightEdge



func maxArea(_ height: [Int]) -> Int {


        guard height.count > 1 else {
            return 0
        }
        var left = 0, right = height.endIndex - 1

        var maxArea = 0
        while left < right {
            let area = (right - left) * min(height[left], height[right])

            maxArea = max(maxArea, area)

            if height[left] < height[right] {
                left += 1
            } else {
                right -= 1
            }
        }
        return maxArea


    }
```

```
我自己做的， accepted。

func maxArea(_ height: [Int]) -> Int {



        var left = 0
        var right = height.count - 1
        var maxArea = 0

        while left < right{
            maxArea = max(maxArea, (right-left) * min(height[left], height[right]))

            if height[left]<height[right]{
                left += 1
            }else{
                right -= 1
            }
             }
```

```
        return maxArea


    }
```

time compex: O(n)

space complex: O(1)

56. Merge Intervals

sorting.

```
func merge(_ intervals: [[Int]]) -> [[Int]] {


   var result = [[Int]]()
        if intervals.count == 0 {
            return result
        }

        let sortedIntervals = intervals.sorted {$0[0] < $1[0]}

        result.append(sortedIntervals.first!)

        for index in 1..<sortedIntervals.count {
            let prevStart = result.last![0]
            let prevEnd = result.last![1]
            let currStart = sortedIntervals[index][0]
            let currEnd = sortedIntervals[index][1]

            if prevEnd >= currStart && prevEnd <= currEnd {
                let newEntry = [prevStart, currEnd]
                result.removeLast()
                result.append(newEntry)
            }
            else if prevEnd < currEnd {
                let newEntry = [currStart, currEnd]
                result.append(newEntry)
            }
        }

        return result
    }
```

time complex: O(n* log n)

space complex: O(n).

53. Maximum Subarray.

method: **Kadane's Algorithm**

```
//psudocode in Python->


def max_subarray(numbers):
"""Find the largest sum of any contiguous subarray."""
best_sum = 0# or: float('-inf')
current_sum = 0
for x in numbers:
        current_sum = max(0, current_sum + x)
        best_sum = max(best_sum, current_sum)
return best_sum
```

```
class Solution {
    func maxSubArray(_ nums: [Int]) -> Int {

    var max_end_here = nums[0]
    var max_so_far = nums[0]
    for i in 1..<nums.count {
      max_end_here = max(nums[i], max_end_here+nums[i]) //compare 1 and -2 + 1 => 1
      max_so_far = max(max_so_far,max_end_here)//- 2, 1 => 1
    }
    return max_so_far
    }


    }
```

time complex: O(n)

space complex: O(1).

42. Trapping rain water

```
func trap(_ height: [Int]) -> Int {
          guard height.count > 2 else {
            return 0
        }

        var start = 0
        var end = height.count - 1

        // skip zeros on both sides
```

```
        while start < height.count && height[start] == 0 {
            start += 1
        }
        while end > 0 && height[end] == 0 {
            end -= 1
        }

        var leftHighest = 0
        var rightHighest = 0

        var leftSum = 0
        var rightSum = 0

        while start < end {
            let startVal = height[start]
            let endVal = height[end]
            if startVal < endVal {
                if startVal < leftHighest {
                    leftSum += leftHighest - startVal
                }
                else {
                    leftHighest = startVal
                }

                start += 1
            }
            else {
                if endVal < rightHighest {
                    rightSum += rightHighest - endVal
                }
                else {
                    rightHighest = endVal
                }
                end -= 1
            }
        }

        return leftSum + rightSum


    }
```

双指针 two-pointers

```
leftMax = 0
rightMax = 0
leftPointer = 0
rightPointer = len(height)-1
counter = 0

while leftPointer <= rightPointer:
```

```
if leftMax < = rightMax:
res += max(min(leftMax, rightMax)- height[left],0)
leftMax=max(leftMax, height[left])
leftPointer+=1

else:
res += max(min(leftMax, rightMax) - height[right], 0)
rightMax = max(rightMax, height[right])
rightPointer -= 1

return res
```

```
func trap(_ height: [Int]) -> Int {
        var total = 0
        guard var maxLeft = height.first else { return 0 }
        guard var maxRight = height.last else { return 0 }
        var i = 0
        var j = height.count - 1

        while i <= j {
            if maxLeft <= maxRight {
                let water = maxLeft - height[i]
                total += ((water > 0) ? water : 0)
                maxLeft = max(maxLeft, height[i])
                i += 1
            } else {
                let water = maxRight - height[j]
                total += ((water > 0) ? water : 0)
                maxRight = max(maxRight, height[j])
                j -= 1
            }
        }
        return total
    }
```

66. Plus-one.

```
func plusOne(_ digits: [Int]) -> [Int] {


         var result = digits
        for i in (0..<result.count).reversed() {
            if result[i] != 9 {
                result[i] = result[i]+1
                return result
            } else {
                result[i] = 0
            }
        }
        if result.first == 0 {
            result.insert(1, at: 0)
```

```
        }
        return result

    }
```

Explanation: Loop from the end of the array to start. Just add 1 if the digit is not a 9, and return. If digit is a 9, replace it with a 0 and go to the previous digit - rinse an repeat. In the end, check if the first digit of the resulting array is a 0. If it is, that means that the number was of the form 9999... In this case, insert a 1 in the start.

time complex: O(n). it’s not more than one pass along the input list.

space complex: O(n) . Although we perform the operation in-place, in the worst scenario, we would need to allocate an intermediate space for the result, which contains the n+1 elements, hence the overall space complexity is O(n).