Hash

1. 两数之和

找出 和为目标值的, 那两个整数的数组下标。

const nums = [2,7,11,15], target = 9 // [0,1]

const nums = [3, 2, 4], target = 6 // [1,2]

/\*\*

\* map{值：位置}

\* 遍历

\* map有补数 -> 立刻返回[map里这个补数的位置，当前位置]

\* map无补数 -> 更新map{当前值：当前位置}

\*/

49. 字母异位词分组

字母异位词 是由重新排列源单词的所有字母得到的一个新单词。

const strs = ["eat", "tea", "tan", "ate", "nat", "bat"] // [ [ 'eat', 'tea', 'ate' ], [ 'tan', 'nat' ], [ 'bat' ] ]

/\*\*

\* map{特定字符传:对应的[]}

\* 遍历

\* 当前字符串转数组 -> sort -> 恢复成字符串

\* map 有 此字符串，放进list[]

\* map 无 此字符串，新建list[]

\* 更新map{此字符串：list[]}

\*/

128. 最长连续序列

找出数字连续的最长序列的长度。

const nums = [100,4,200,1,3,2] //4

const nums = [0,3,7,2,5,8,4,6,0,1]//9

双指针

11. 盛最多水的容器

有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i]) 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

const height = [1,8,6,2,5,4,8,3,7] // 49

/\*\*

\* 左右，往中间指

\* 找下标之差 \* 值之差 的最大

\*/

15. 三数之和

给你一个整数数组 nums ，

判断是否存在三元组 [nums[i], nums[j], nums[k]]

满足 i != j、i != k 且 j != k ，

还满足 nums[i] + nums[j] + nums[k] == 0 。

返回所有和为 0 且不重复的三元组。

const nums = [-1,0,1,2,-1,-4] // [ [ -1, -1, 2 ], [ -1, 0, 1 ] ]

/\*\*

\* let ans[]

\* sort

\* 遍历

\* L：当前索引+1，

\* R：末尾

\* 只要(L < R)

\* sum = 当前值 + L值 + R值

\* 1. sum = 0

\* 放入ans[]

\* 左移 -> 去重: 只要(L<R && 左值 = 左前一个值)

左移

\* 右移 -> 去重: 只要(L<R && 右值 = 右前一个值)

右移

\* 2. sum < 0 左移

\* 3. sum > 0 右移

\*

\*\*/

/\*\*

\* 42. 接雨水

\*

\* 有高有低的柱子，接多少水

\*/

const height = [0,1,0,2,1,0,1,3,2,1,2,1] // 6

console.log(trap(height))

/\*\*

\* 变量：ans, leftMax, rightMax, 左右指针

\* 两指针居中循环

\* 更新左、右两边max高度

\* 当前位置比较

\* 左大 -> ans +=（当前高度-左max），左移

\* 右大 -> ans +=（当前高度-右max），右移

\*/

动态规划

1. 接雨水

有高有低的柱子，接多少水

const height = [0,1,0,2,1,0,1,3,2,1,2,1] // 6

/\*\*

\* 变量：左max[]，右max[], ans

\*

\* 子问题1： 某个位置左max高度[]（for第二个开始：当前左max值 = 选大的(当前高度,前一个max值)）

\* 子问题2： 某个位置右max高度[]（for倒二个开始：当前右max值 = 选大的(当前高度,后一个max值)）

\*

\* for: ans += 选小的（左max值 - 右max值) - 当前值

\*

\* 状态转移方程：leftMax[i]和rightMax[i]的计算就是状态转移的过程。

\* 重叠子问题：在计算leftMax和rightMax时，每个位置的计算都依赖于之前（或之后）已经计算过的位置

\* 最优子结构：每个位置能够接住的雨水量：取决于其左侧最大高度和右侧最大高度中的较小值 - 当前高度

\*\*/

198. 打家劫舍

每间房内都藏有一定的现金，

如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

不警报的情况下 ，一夜之内能够偷窃到的最高金额。

const nums = [2,7,9,3,1, 2] // 13

console.log(rob(nums))

/\*\*

\*

\* 子问题的划分：即从第i间房屋开始，能够偷窃到的最大金额。

\* 最优子结构：对于每个位置i，其能够偷窃到的最大金额取决于两个选择：

\* 不偷当前房屋（即选择dp[i-1]），

\* 或者偷当前房屋（即选择nums[i] + dp[i-2]，其中dp[i-2]是从i+1间房屋开始的最大金额）。

\* 状态转移方程：dp[i] = Math.max(dp[i - 1], nums[i] + dp[i - 2])。

\* 重叠子问题：在计算dp[i]时，需要用到之前计算过的dp[i-1]和dp[i-2]

\*\*/

1. 杨辉三角

const numRows = 5 // [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

/\*\*

子问题的划分：每一行的元素的计算，

最优子结构：每一行的元素（除了第一个和最后一个）都是上一行相邻两个元素的和。

状态转移方程：即row[j] = ans[i - 1][j - 1] + ans[i - 1][j]。

重叠子问题：在计算每一行时，需要用到上一行的元素

\*/

var generate = function(numRows) {

const ans = [];

for(let i = 0; i< numRows; i++) {

const row = new Array(i + 1).fill(1);

for (let j = 1; j < row.length - 1; j++) {

row[j] = ans[i - 1][j - 1] + ans[i - 1][j];

}

ans.push(row)

}

return ans;

};

322. 零钱兑换

给你一个整数数组 coins ，表示不同面额的硬币；以及一个整数 amount ，表示总金额。

计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 -1 。

你可以认为每种硬币的数量是无限的。

const coins = [1, 2, 5], amount = 11 // 3

// const coins = [2], amount = 3 // -1

// const coins = [1], amount = 0 // 0

console.log(coinChange(coins, amount))

/\*\*

\* 子问题： dp[i] 为凑成金额 i 所需的最少硬币数

\* 最优子结构：要凑成金额 i 的最优解依赖于：凑成金额 i-coin 的最优解（其中 coin 是小于等于 i 的硬币面额）。

\* 这意味着，如果我们知道了较小金额的最少硬币数，就可以用它来帮助确定较大金额的最少硬币数。

\*

\*/