【第二十七周】动态规划算法优化

1、714. 买卖股票的最佳时机含手续费

1、状态定义：

dp[i][0]:代表第i天不持有股票的最大收益

dp[i][1]:代表第i天持有股票的最大收益

2、分析动态转移方程：分析两个值的取值范围：

（1）dp[i][0]等于有两种情况，取一个最大值

第一种： i-1 没有 i 没有 收益最大值是：dp[i-1][0]

第二种： i-1有 i卖了没有 收益最大值是：dp[i-1][1] + price[i] - free

总结：dp[i][0] = max(dp[i-1][1] + price[i] - free)

（2）dp[i][1]等于有两种情况，取一个最大值

第一种： i-1有 i有 收益最大值是：dp[i-1][1]

第二种： i-1没有 i 买 收益最大值是：dp[i-1][0] - price[i]

总计：dp[i][1] = max(dp[i-1][1],dp[i-1][0] - price[i])

```
/**
* @param {number[]} prices
* @param {number} fee
* @return {number}
*/
var maxProfit = function(prices, fee) {
    const n = prices.length;
    const dp = new Array(n).fill(0).map(v => new Array(2).fill(0));
    dp[0][0] = 0,// dp[0][0]:代表第0天不持有股票的最大收益 就是0
    dp[0][1] = -prices[0];// dp[0][1]:代表第0天持有股票的最大收益，第0天就是最开始那天的收益，只能是买入股票的收益就是股票的价格
    // 数学归纳法：从k[i] 到 k[i+1]的算法
    for (let i = 1; i < n; ++i) {
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i] - fee);
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
    }
    // 在两者取一个最大值，分别是最后一天持有股票和不持有股票的最大值
    return Math.max(dp[n - 1][0], dp[n - 1][1] );
};
```

2、213. 打家劫舍 II

1、 将一个环形的内容，拆分成两个环，一个是不偷最后一件屋子，一个是不偷最后一间屋子

2、dp[i][0] 第i间房子不偷的最大价值

3、dp[i][1] 第i间房子偷的最大价值

4、dp[n-1][0] 不偷最后一间屋子的最大价值

```
/**
 * @param {number[]} nums
 * @return {number}
 */

var rob = function(nums) {
    let n = nums.length;
    if(n == 1) return nums[0];//如果只有一个房子也可以偷
    const dp = new Array(n).fill(0).map(v => new Array(2).fill(0));
    // 先求不偷最后一间屋子的最大价值
    dp[0][0] = 0,//不偷就是0
    dp[0][1] = nums[0];//第i间房子偷的最大价值
    for(let i = 1;i < n;i++){
        dp[i][0] = Math.max(dp[i-1][1],dp[i-1][0]);//不偷当前房间，那么前一间屋子可偷可不偷
        dp[i][1] = dp[i-1][0] + nums[i];//偷当前房间，那么前一间房间就不能偷；前一间房间不偷的最大价值 + 当前房子
    }
    // 记录第一个最大值：不偷最后一间获得的最大值
    let ans1 = dp[n-1][0];
    // 规定：不偷第一间屋子
    dp[0][0] = 0,
    dp[0][1] = 0;//不管第一间房子偷或者不偷，都让小偷无功而返
    for(let i = 1;i < n;i++){
        dp[i][0] = Math.max(dp[i-1][1],dp[i-1][0]);
        dp[i][1] = dp[i-1][0] + nums[i];
    }
    // 到这，第一件房子没偷，所以最后一件房子可以偷/也可以不偷，所以获取在最后一间房子偷或者不偷的最大值
    let ans2 = Math.max(dp[n-1][0],dp[n-1][1]);
    return Math.max(ans1,ans2);
};
```

3、416. 分割等和子集

可达数组： 在某种情况下，可以达到某个值
1、计算元素组的和值
2、判断原数组的数字是否可以凑出和值的一半，可以凑出来证明可以被分割
3、状态定义：f[i][j]，前i个数字是是否能凑出j值。分成两种情况
4、动态转移方程：f[i][j] = f[i-1][j] / f[i-1][j-num[i]];
f[i-1][j]：没有使用第i个数字
f[i-1][j-num[i]]：使用第i个数字 ，剩余数字等于前i-1个数字，拼凑j-num[i]
这两个状态，只要有一个状态是1， f[i][j] = 1，前i个数字可以拼凑出来j,前i个数字到j是可达的；

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
```

```
var canPartition = function(nums) {
    let sum = 0;
    for(const x of nums) sum += x;
    // 特殊判断，如果数组元素的和值是一个奇数，直接返回false
    if(sum % 2) return false;
    const dp = new Array(sum + 1);
// 一开始所有的状态都是0，都是不可达
    for(let i = 1;i <= sum;i++) dp[i] = 0;
    dp[0] = 1;
    sum = 0;//当前值之前所有值能够拼凑出来的最大值
    for(const x of nums){
        sum += x;
        // 倒着扫描
        for(let j = sum;j >= x;j--){
            dp[j] |= dp[j - x];
        }
    }
    return dp[sum / 2];

};
```

4、474. 一和零

1、在字符串数组中，挑选出来尽可能多的字符串形成一个集合

2、让集合中0和1的数量不能超过m和n

3、本质上是0/1背包问题

4、m和n就是背包容量. 每个字符串就是背包物品，每个物品的价值是多大，因为尽可能多的物品，所以每个物品提供的价值就是1

5、dp[n][j]前n个物品，背包容量是j的情况下，获取的最大价值

6、dp[i][m][n]前i个字符串，m个0，n个1，在这种限制下，我们能取得最大的字符串的数量

7、讨论dp[i][m][n]等于什么?

8、dp[i][m][n] = max(dp[i-1][m][n],dp[i-1][m-1][n-0] + 1)

9、dp[i-1][m][n]：要第i个字符串

10、dp[i-1][m-1][n-0] + 1：不要第i个字符串

没有实现滚动数组优化，同学们下课自己尝试着实现一下

```
 /**
 * @param {string[]} strs
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
var findMaxForm = function(strs, m, n) {
    const dp = new Array(m+1).fill(0).map(v => new Array(n+1).fill(0));
    // 去扫描每一个字符串
    for(const x of strs){
        // 分别计算0的数量和1的数量
```

```
        let cnt0 = 0,cnt1 = 0;
        for(const y of x){
            if(y == "0") cnt0 += 1;//0的数量+1
            else cnt1 += 1; //1的数量+1
        }
        // 采用刷表法：倒着刷表
        for(let i = m;i >= cnt0;--i){
            for(let j = n;j >= cnt1;--j){
                dp[i][j] = Math.max(dp[i][j],dp[i - cnt0][j - cnt1] + 1)
            }
        }
    }
    return dp[m][n];
};
```

5、322. 零钱兑换

1、将这道题：凑成金额的最少硬笔数量 转化成 凑成金额数量的方法总数

2、f(i)(j) = f(i-1)(j) + f(i)(j-x) x:是第i中硬币的金额

3、f(i)(j) 使用前i中硬币，拼凑j元钱的方法总数

4、f(i-1)(j) 用了第i种硬币的方法总数

5、f(i)(j-x)没用第i种硬币的方法总数

6、更新的顺序是从前往后

```
/**
 * @param {number[]} coins
 * @param {number} amount
 * @return {number}
 */
var coinChange = function(coins, amount) {
    let dp = new Array(amount + 1);
    dp[0] = 0;
    // 初始化的时候初始化为-1
    // 因为题意里面说：如果没有任何一种硬币组合能组成总金额，返回 -1 。
    for(let i = 1;i <= amount;i++) dp[i] = -1;
    for(let i = 1;i <= amount;i++){
        for(const  x of coins){
            if(i < x) continue;
            if(dp[i - x] == -1) continue;
            if(dp[i] == -1 || dp[i] > dp[i - x] + 1) dp[i] = dp[i - x] + 1;
        }
    }
    return dp[amount];
};
```

开课吧