【第二十六周】动态规划算法

注意：因为海贼题库不支持JS提交，所以分别在力扣里面选了相似的题。

1、第一题和第二题都是第25课做过的，第三题和第四题是跟船长的海贼题目一样。

2、只有最后一题不同【 分割等和子集 】和海贼题库【#47. 练习题4：0/1背包】。

3、【 分割等和子集 】考的是0/1背包的存在问题：是否存在.....，满足.....；【#47. 练习题4：0/1背包】考的是最值问题：要求最大值/最小值。

1、120. 三角形最小路径和

1、下一层的数据，只能由上一层的下标 - 1 或 下标 处数据过来。

2、我们设置一个 dp[i][j] 的二位数组， i 代表第 i 层，j 代表第 i 层第 j 个数据的最小路径和。

3、所以转移方程就是dp[i][j] = min(dp[i-1][j-1], dp[i-1][j]) + triangle[i][j]
dp[0][0] = triangle[0][0]

4、(1)需要注意到，由于第 i 层的共有 j 个数据，那么第 i-1 层，数据为 j-1
(2) 第 i 层的第 0 个数据，只能由 i-1 层 第 0 个过来
(3)第 i 层最后一个数据，只能由 i-1 层最后一个数据（j-1） 个数据过来

```
/**
 * @param {number[][]} triangle
 * @return {number}
 */
var minimumTotal = function(triangle) {
    const n = triangle.length;
    const dp = new Array(n + 1);
    for(let i = 0; i < n;i++){
        dp[i] = new Array(triangle[i].length);
    }
    // 状态定义 dp[i][j]
    for(let i = 0 ;i < n;i++){
        for(let j = 0;j <= i;j++){
            dp[i][j] = Infinity;
        }
    }
    dp[0][0] = triangle[0][0];
    for(let i = 0; i < n - 1;i++){
        for(let j = 0; j <= i;j++){
            dp[i + 1][j] = Math.min(dp[i + 1][j], dp[i][j] + triangle[i + 1]
[j]);
            dp[i + 1][j + 1] = Math.min(dp[i + 1][j + 1], dp[i][j] + triangle[i
+1][j + 1]);
        }
    }
    let ans = Infinity;
    for(const x of dp[n - 1]) ans = Math.min(ans,x);
    return ans;
};
```

2、 300. 最长递增子序列

1、DP数组用来存储该位置的最长子序列长度。

2、设nums[j] < nums[i], dp[i] = Math.max(dp[i], dp[j] + 1),即上一个比它小的数的最长子序列加1

3、遍历dp数组，找出最大值。

```
/**
 * @param {number[]} nums
 * @return {number}
 */
//  dp[i]
// dp 存储该位置的子序列的长度
// nums[j] < nums[i] dp[i] = Math.max(dp[i],dp[j] + 1);上一个比他小的数据的最长子序列
+1
// 整个原序列遍历完成，
var lengthOfLIS = function(nums) {
    const dp = new Array(nums.length + 1).fill(1);
    for(let i = 0;i < nums.length;i++){
        // i与i前面的元素做比较
        for(let j = 0;j < i;j++){
            // 找到比i小的元素，然后当前序列的最长子序列长度+1
            if(nums[j] < nums[i]){
                dp[i] = Math.max(dp[i],dp[j] + 1);
            }
        }
    }
    // 找出最长子序列
    let res = 0;
    for(let i = 0; i < dp.length;i++){
        res = Math.max(dp[i],res);
    }
    return res;
};
```

3、 1143. 最长公共子序列

1、二维dp的概念。这两个序列里面跳着挑元素，保证元素的相对位置不变，在挑出的序列里面有个长度最长的。

2、dp[i][j]表示text1前i位和text2前j位的最长公共子序列的长度.

3、如果A串里面的第I位和B串里面的第J位不相等， dp[i][j] = Math.max(dp[i - 1][j],dp[i][j - 1])。否则相等的话，就是dp[i][j] = dp[i-1][j-1] + 1。

```
/**
* @param {string} text1
* @param {string} text2
* @return {number}
*/
```

```
var longestCommonSubsequence = function(text1, text2) {
    const m = text1.length,n = text2.length;
    const dp = new Array(m + 1).fill(0).map(() => new Array(n + 1).fill(0));
     // dp[i][j]表示text1前i位和text2前j位的最长公共子序列的长度
    for(let i = 1; i <= m;i++){
        const c1 = text1[i - 1];
        for(let j = 1; j <= n;j++){
            const c2 = text2[j - 1];
            if(c1 === c2){
                dp[i][j] = dp[i-1][j-1] + 1
            }else{
                dp[i][j] = Math.max(dp[i - 1][j],dp[i][j - 1]);
            }
        }
    }
    return dp[m][n];
};
```

4、 132. 分割回文串 II

1、设 f[i] 表示字符串的前缀 s[0..i] 的最少分割次数。要想得出 f[i] 的值，我们可以考虑枚举 s[0..i]分割出的最后一个回文串，这样我们就可以写出状态转移方程：

$$f[i] = \min_{0 \le j < i} \{f[j]\} + 1, \quad \text{其中 } s[j+1..i] \text{ 是一个回文串}$$

2、即我们枚举最后一个回文串的起始位置 j+1，保证 s[j+1..i]是一个回文串，那么 f[i] 就可以从 f[j]转移而来，附加 1 次额外的分割次数。

3、注意到上面的状态转移方程中，我们还少考虑了一种情况，即 s[0..i]本身就是一个回文串。此时其不需要进行任何分割，即：f[i] = 0

4、用下面方法把将字符串 s 的每个子串是否为回文串预先计算出来：

设 $g(i,j)$ 表示 $s[i..j]$ 是否为回文串，那么有状态转移方程：

$$g(i,j) = \begin{cases} \texttt{True}, & i \ge j \\ g(i+1,j-1) \wedge (s[i] = s[j]), & \text{otherwise} \end{cases}$$

其中 $\wedge$ 表示逻辑与运算，即 $s[i..j]$ 为回文串，当且仅当其为空串（$i > j$），其长度为 1（$i = j$）或者首尾字符相同且 $s[i+1..j-1]$ 为回文串。

5、这样一来，我们只需要 O(1) 的时间就可以判断任意 s[i..j] 是否为回文串了。通过动态规划计算出所有的 f 值之后，最终的答案即为 f[n-1]，其中 n 是字符串 s 的长度。

```
 /**
 * @param {string} s
 * @return {number}
 */
//  回文字符串 ：  是从左到右读和从右往左读，完全一致的字符串
// winter老师 7月22日 专业方向课【算法思想】
var minCut = function(s) {
    const n = s.length;
```

```
    const dp = new Array(n).fill(0).map(() => new Array(n).fill(true));

    for(let i = n -1;i >= 0;--i){
        for(let j = i + 1;j < n;++j){
            dp[i][j] = s[i] == s[j] &&  dp[i+1][j-1]
        }
    }

    const f = new Array(n).fill(Number.MAX_SAFE_INTEGER);
    for(let i = 0; i < n; i++){
        if(dp[0][i]){
            f[i] = 0;
        }else{
            for(let j = 0;j < i;j++){
                if(dp[j+1][i]){
                    f[i] = Math.min(f[i],f[j] + 1);
                }
            }
        }
    }
    return f[n - 1];
};
```

5、416. 分割等和子集

1、根据数组的长度 n 判断数组是否可以被划分。如果 n<2，则不可能将数组分割成元素和相等的两个子集，因此直接返回 false

2、计算整个数组的元素和 sum 以及最大元素 maxNum。如果 sum 是奇数，则不可能将数组分割成元素和相等的两个子集，因此直接返回 false。如果 sum 是偶数，则令 target= 2分之一sum，需要判断是否可以从数组中选出一些数字，使得这些数字的和等于 target。如果maxNum>target，则除了 maxNum 以外的所有元素之和一定小于 target，因此不可能将数组分割成元素和相等的两个子集，直接返回 false。

3、创建二维数组 dp，包含 n 行 target+1 列，其中dp[i][j] 表示从数组的 [0,i][0,i] 下标范围内选取若干个正整数（可以是 0 个），是否存在一种选取方案使得被选取的正整数的和等于 j。初始时，dp 中的全部元素都是 false。

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
var canPartition = function(nums) {
    const n = nums.length;
    if(n < 2){
        return false;
    }
    let sum = 0,maxNum = 0;
    for(const num of nums){
        sum += num;
        maxNum = maxNum > num ? maxNum : num;
    }
    if(sum & 1){
        return false;
```

```
    }
    const target = Math.floor(sum / 2);
    if(maxNum > target){
        return false;
    }
    // 谢谢zlTy同学提醒，二维数组放置状态的时候，正确的写法
    const dp = new Array(n).fill(0).map(()=> new Array(target + 1).fill(false));
    // console.log(dp)
    for(let i = 0; i <n;i++){
        dp[i][0] = true;
    }
    dp[0][nums[0]] = true;
    for(let i = 1;i < n;i++){
        const num = nums[i];
        for(let j = 1;j <= target;j++){
            if(j >= num){
                dp[i][j] = dp[i-1][j] | dp[i-1][j-num];

            }else{
                dp[i][j] = dp[i-1][j];
            }
        }
    }
    return dp[n - 1][target];
};
```