【第二十八周】本月刷题测试题

1、343. 整数拆分

1、创建数组 dp，其中 *dp*[*i*] 表示将正整数 *i* 拆分成至少两个正整数的和之后，这些正整数的最大乘积。

2、特别地，0 不是正整数，1 是最小的正整数，0 和 1 都不能拆分，因此 dp[0]=dp[1]=0。

3、当 i≥2 时，假设对正整数 ii 拆分出的第一个正整数是 j（1≤j<i），则有以下两种方案：

a、将 i 拆分成 i−j 的和，且 i−j 不再拆分成多个正整数，此时的乘积是 j×(i−j)；

b、将 i 拆分成 j 和i−j 的和，且 i−j 继续拆分成多个正整数，此时的乘积是 j×dp[i−j]。

4、状态转方程：dp[i]=max(j×(i−j),j×dp[i−j])，最终得到 *dp*[*n*] 的值即为将正整数 *n* 拆分成至少两个正整数的和之后，这些正整数的最大乘积。

```
/**
 * @param {number} n
 * @return {number}
 */
const integerBreak=(n)=>{
    let dp =new Array(n+1).fill(0)
    dp[2]=1
    for(var i=3;i<=n;i++){
        for(var j=1;j<i-1;j++){
            dp[i]=Math.max(dp[i-j]*j,j*(i-j),dp[i]);
        }
    }
    return dp[n]
}
```

2、55. 跳跃游戏

1、状态dp，初始化为全false（true表示能到达，false表示无法到达）

2、 dp[0] = true，第一个点一定能走到

3、 接下来考虑第二个点开始判断能否到达该点?

a.想知道能否到达该点，则要看能否从该点之前的点跳跃过来，从后往前找的原因是越接近它的点越可能能够跳　　过来，可以减少循环次数

b.如果前面有的点到达不了，就不必考虑从那个点跳不跳的过来了，因为那个点自己本身都到不了

c.如果找到了可以跳到当前点的前面的点，则当前点可达到，更新他的状态为true，并退出循环

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

```
 */
var canJump = function(nums) {
    //特殊情况只有一个元素
    if (nums.length == 1) return true
    //状态dp，初始化为全false（true表示能到达，false表示无法到达）
    const dp = new Array(nums.length).fill(false)
    //第一个点一定能走到
    dp[0] = true
    //从第二个点开始判断能否到达该点？
    for (let i = 1; i < nums.length; ++i) {
        //想知道能否到达该点，则要看能否从该点之前的点跳跃过来，从后往前找的原因是越接近它的点越可能能够跳过来，可以减少循环次数
        for (let j = i-1; j >= 0; --j) {
            //如果前面有的点到达不了，就不必考虑从那个点跳不跳的过来了，因为那个点自己本身都到不了
            if (!dp[j]) continue
            //如果前面有点能达到，但是跳不了足够远到此点，也继续找再往前一个点
            if (nums[j] < i-j) continue
            //如果找到了可以跳到当前点的前面的点，则当前点可达到，更新他的状态为true，并退出循环
            dp[i] = true
            break
        }
    }
    //返回最后一个点是否能达到
    return dp[nums.length-1]
};
```

3、413. 等差数列划分

前面已经是等差数列 f(n - 1)，加上当前数字，如果也是等差数列的话，那么可以新增 f(n - 1) + 1 个子数组。新增的这一个子数组是 (nums[n - 2], nums[n - 1], nums[n])

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var numberOfArithmeticSlices = function(nums) {
    const dp = new Array(nums.length).fill(0)
    let sum = 0
    for (let i = 1; i < nums.length - 1; i ++) {
        if (nums[i] - nums[i - 1] === nums[i + 1] - nums[i]) {
            dp[i + 1] = dp[i] + 1
            sum += dp[i + 1]
        }
    }
    return sum
};
```

4、139. 单词拆分

1、dp[i]表示0-i之间的字符串是否可以被拆分并满足题设条件存在于wordDict中。

2、假设拆分点为j，那么状态转移方程为：dp[i] = dp[j] && s.substring(j+1, i+1)存在于wordDict

```
/**
 * @param {string} s
 * @param {string[]} wordDict
 * @return {boolean}
 */
var wordBreak = function(s, wordDict) {
    // dp[i]表示0-i之间的字符串是否可以被拆分并满足题设条件存在于wordDict中
    let dp = new Array(s.length).fill(false);
    let set = new Set(wordDict);
    for (let i = 0; i < s.length; i++) {
        // 检查0-i之间的字符串是否直接存在于wordDict中
        if (set.has(s.substring(0, i+1))) {
            dp[i] = true;
            continue;
        }
        // 这一步是为了检查。假如s.substring(0,i)不直接存在于wordDict中
        // 那么判断拆分之后是否存在于wordDict中
        for (let j = 0; j < i; j++) {
            if (dp[j] && set.has(s.substring(j+1, i+1))) {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[s.length-1]
};
```

5、221. 最大正方形

1、定义 dp[i][j]：以坐标 (i,j) 为右下角的最大正方形边长。

2、(i,j) 为 0 时，无法构成正方形，dp[i][j] = 0

3、(i,j) 为 1 时，dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1

4、一个正方形的最大边长决定于它左方、上方、斜上方的位置所能形成的最大正方形的边长，即：三者的最小值 + 自身的长度 1。

5、为了避免边界条件判断，可以将 dp 数组的长和宽都增加 1。

```
 /**
 * @param {character[][]} matrix
 * @return {number}
 */
// dp[i][j]=x 表示为x,j右上角的正方形边长
var maximalSquare = function(matrix) {
  let m = matrix.length
  let n =matrix[0].length
  const dp = new Array(m).fill(0).map(()=>new Array(n).fill(0))
  let max=0 // 正方形的最大边长
```

```
  for(let i=0;i<m;i++){
    if(matrix[i][0]==='1'){
      // 只有一列
      dp[i][0]=1
      max=Math.max(max,dp[i][0])
    }

  }
  for(let j=0;j<n;j++){
    if(matrix[0][j]==='1'){
      // 只有一行
      dp[0][j]=1
      max=Math.max(max,dp[0][j])
    }
  }
  for(let i=1;i<m;i++){
    for(let j=1;j<n;j++){
      if((matrix[i][j]==='1')){
        dp[i][j]=Math.min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1
        max=Math.max(max,dp[i][j])
      }
    }
  }
  return max*max
};
```

6、650. 只有两个键的键盘

1、设 f[i] 表示打印出 i 个 A 的最少操作次数。

2、由于我们只能使用「复制全部」和「粘贴」两种操作，那么要想得到 i 个 A，我们必须首先拥有 j 个 A，使用一次「复制全部」操作，再使用若干次「粘贴」操作得到 i个 A。

3、这里的 j 必须是 ii 的因数，「粘贴」操作的使用次数即为 i /。j −1。

4、我们可以枚举 j 进行状态转移，这样就可以得到状态转移方程： Max(f[j] + i/ j)，其中i/ j表示 j 可以整除 i，即 j是 i 的因数。

5、动态规划的边界条件为 *f[1]=0*，最终的答案即为 *f[n]*。

```
var minSteps = function(n) {
    const f = new Array(n + 1).fill(0);
    for (let i = 2; i <= n; ++i) {
        f[i] = Number.MAX_SAFE_INTEGER;
        for (let j = 1; j * j <= i; ++j) {
            if (i % j === 0) {
                f[i] = Math.min(f[i], Math.floor(f[j] + i / j));
                f[i] = Math.min(f[i], Math.floor(f[i / j] + j));
            }
        }
    }
    return f[n];
};
```

7、376. 摆动序列

1、序列长度为`0` `1`，摆动序列长度对应`0` `1`。序列长度`>=2`，摆动序列长度`>=1`

2、遍历序列：最长上升摆动序列`up`，最长下降摆动序列`down`

2.1当前数 = 前数：摆动序列长度不变

2.2当前数 > 前数：
当前数放入最长下降摆动序列 down + 1 → up
当前数放入最长上升摆动序列，替换原末位值 或 舍弃 up → up

2.3当前数 < 前数：
当前数放入最长上升摆动序列 up + 1 → down
当前数放入最长下降摆动序列，替换原末位值 或 舍弃 down → down

```
var wiggleMaxLength = function(nums) {
    if (nums.length < 2) return nums.length
    let up = 1, down = 1
    for (let i = 1; i < nums.length; i++)
        if (nums[i] > nums[i - 1])
            up = Math.max(up, down + 1)
        else if (nums[i] < nums[i - 1])
            down = Math.max(down, up + 1)
    return Math.max(up, down)
};
```

8、1039. 多边形三角剖分的最低得分

递推
1、3个点构成1个三角形

2、4个点构成1 + 1个三角形

3、5个点构成1 + 2 或 1 + 1 + 1 = 3个三角形

4、6个点构成1 + 3 或 1 + 1 + 2 = 4个三角形

5、7个点构成1 + 4 或 1 + 1 + 3 或1 + 2 + 2 = 5个三角形

```
/**
 * 这道题目的关键是:
 * 可以把任何顶点数大于3的凸多边形分解为一个三角形和至多两个凸多边形，并且按这种
 * 方式分割，最后n个顶点得到的一定是n - 2个三角形（我不知道最后得到的一定是n - 2个三角形怎么证明
 * ）。
 * 那么这个就可以dp了
 * @param {number[]} A
 * @return {number}
 */
var minScoreTriangulation = function(A) {
    const len = A.length
```

```
    const dp = new Array(len)
    // 因为dp求的是最小值，那么初始化为最大值
    for(let i = 0;i < len;i++) dp[i] = new Array(len).fill(Infinity)
    for(let i = 0;i < len;i++) {
        // 顶点数小于3不能构成三角形，dp中不会有用到顶点数为1的情况。
        //i、j包含顶点数小于2的情况的值都初始化为0,便于后面计算
        dp[i][(i + 1) % len] = 0
    }
    // temp_len + 1代表凸多边形的顶点个数
    // dp先求凸多边形有3个定点的情况，再求有4个顶点的情况下，直至求出有length个顶点的情况
    for(let temp_len = 2;temp_len < len;temp_len++) {
        // 这些顶点中第一个顶点是i，最后一个顶点是j。
        // i和j都有可能为0 ~ (len - 1)（开闭区间）中的任何一个点
        for(let i = 0;i < len;i++) {
            let j = (i + temp_len) % len
            // 虽然是循环的，但可以认为j在i的后面，也就是k永远不可能等于i或j
            for(let k = (i + 1) % len;k !== j;k = (k + 1) % len) {
                dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + A[i] * A[k]
* A[j])
            }
        }
    }
    return dp[0][len - 1]
};
```

9、646. 最长数对链

1、先给pairs以第二位的大小从小到大排好序

2、排好序后取第一个为start节点，然后从index为1遍历

3、初始化count为1（把第一个节点算进去），之后只要候选的第一位大于start节点的第二位，说明可以接到后面

4、可以接的话，count+1，start节点重置为新的节点

5、返回count

```
 /**
 * @param {number[][]} pairs
 * @return {number}
 */
var findLongestChain = function (pairs) {
    //先给pairs以第二位的大小从小到大拍好序
    let orderedpairs = pairs.sort((a, b) => {
        return a[1] - b[1]
    })
    //初始化count为1（把第一个节点算进去），之后只要候选的第一位大于start节点的第二位，说明可
以接到后面
    let count=1
    let start=orderedpairs[0]
   for(let i=1;i<orderedpairs.length;i++){
       if(orderedpairs[i][0]>start[1]){
           //可以接的话，count+1，start节点重置为新的节点
           count++
           start=orderedpairs[i]
```

```
        }
    }
    return count
};
```