动态规划

曾乘风　　2021.11.10

什么是动态规划？

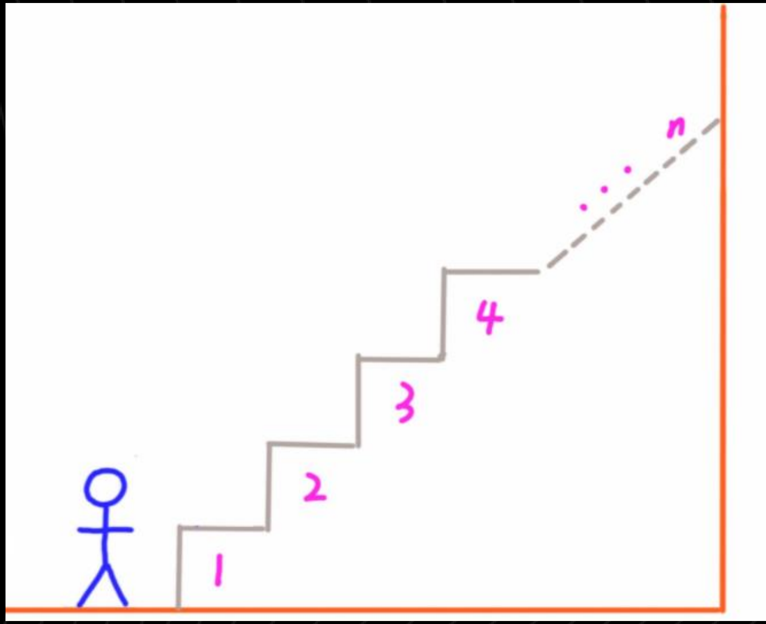
动态规划算法是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推的方式去解决

4个步骤

- 问题拆解
- 状态定义
- 递推方程推导
- 实现

例1：爬楼梯

- 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
- 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
- 注意：给定 n 是一个正整数。



示例1：

输入：2
输出：2
解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶

示例2：

输入：3
输出：3
解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

- **问题拆解**：
  我们到达第 n 个楼梯可以从第 n - 1 个楼梯和第 n - 2 个楼梯到达，因此第 n 个问题可以拆解成第 n - 1 个问题和第 n - 2 个问题，第 n - 1 个问题和第 n - 2 个问题又可以继续往下拆，直到第 0 个问题，也就是第 0 个楼梯 (起点)

- **状态定义**
  “问题拆解”中已经提到了，第 n 个楼梯会和第 n - 1 和第 n - 2 个楼梯有关联，那么具体的联系是什么呢？你可以这样思考，第 n - 1 个问题里面的答案其实是从起点到达第 n - 1 个楼梯的路径总数，n - 2 同理，从第 n - 1 个楼梯可以到达第 n 个楼梯，从第 n - 2 也可以，并且路径没有重复，因此我们可以把第 i 个状态定义为“**从起点到达第 i 个楼梯的路径总数**”，状态之间的联系其实是相加的关系。

- **递推方程**
  “状态定义”中我们已经定义好了状态，也知道第 i 个状态可以由第 i - 1 个状态和第 i - 2 个状态通过相加得到，因此递推方程就出来了 dp[i] = dp[i -1] + dp[I -2]

- **实现**：设置初始值，dp[0] = 0, dp[1] = 1, dp[2] = 2

```javascript
/**
 * @param {number} n
 * @return {number}
 */
var climbStairs = function(n) {
    let dp = [];
    dp[0] = 0;
    dp[1] = 1;
    dp[2] = 2;
    for (let index = 3; index <= n; index++) {
      dp[index] = dp[index - 1] + dp[index - 2];
    }

    return dp[n];
};
```

例2：最长上升子序列

- 给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例1：

输入：[10,9,2,5,3,7,101,18]

输出：4

解释：最长的上升子序列是 [2,3,7,101]，

它的长度是 4

示例2：

输入：[7,7,7,7,7,7]

输出：1

- **问题拆解**：
  我们要求解的问题是“数组中最长递增子序列”，一个子序列虽然不是连续的区间，但是它依然有起点和终点，如果我们确定终点位置，然后去 **看前面 i - 1 个位置中，哪一个位置可以和当前位置拼接在一起**，这样就可以把第 i 个问题拆解成思考之前 i - 1 个问题，注意这里我们并不是不考虑起始位置，在遍历的过程中我们其实已经考虑过了

- **状态定义**
  问题拆解中我们提到“第 i 个问题和前 i - 1 个问题有关”，也就是说“如果我们要求解第 i 个问题的解，那么我们必须考虑前 i - 1 个问题的解”，我们定义 **dp[i] 表示以位置 i 结尾的子序列的最大长度**，也就是说 dp[i] 里面记录的答案保证了该答案表示的子序列以位置 i 结尾。

- **递推方程**
  对于 i 这个位置，我们需要考虑前 i - 1 个位置，看看哪些位置可以拼在 i 位置之前，如果有多个位置可以拼在 i 之前，那么必须选最长的那个，这样一分析，递推方程就有了：
  `dp[i] = Math.max(dp[j],...,dp[k]) + 1`

- **实现**

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var lengthOfLIS = function(nums) {
    let length = nums.length
    let dp = Array.from({length},()=> 1)
    let maxLength = 1
    for(let i=1; i<length; i++){
        for(let j=0;j<i;j++){
            if(nums[i]>nums[j]){
                dp[i] = Math.max(dp[j]+1,dp[i])
            }
        }
     maxLength = Math.max(dp[i],maxLength)
    }
    return maxLength
};
```

使用场景

能采用动态规划求解的问题的一般要具有 3 个性质：

- **最优化**：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。子问题的局部最优将导致整个问题的全局最优。换句话说，就是问题的一个最优解中一定包含子问题的一个最优解。
- **无后效性**：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关，与其他阶段的状态无关，特别是与未发生的阶段的状态无关。
- **重叠子问题**：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）