# 动态规划

> 动态规划（英语：Dynamic programming，简称DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。
>
> 动态规划常常适用于有重叠子问题[1]和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。
>
> 动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。
>
> 通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

使用动态规划需要满足的条件：

1. 最优子结构性质。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。最优子结构性质为动态规划算法解决问题提供了重要线索。
2. 无后效性。即子问题的解一旦确定，就不再改变，不受在这之后、包含它的更大的问题的求解决策影响。
3. 子问题重叠性质。子问题重叠性质是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的效率，降低了时间复杂度。

## 动态规划的实现步骤

1. 定义状态
2. 设置初值
3. 设置状态转移方程

## 实例

### 1.硬币问题：

> 给你一个整数数组 nums ， 表示不同面额的硬币；以及一个整数 money ， 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。

步骤

> 定义状态： dp(n)是凑成n金额所需要的最少硬币个数。 定义初值: dp(i) == 1, i是coins内给定的的硬币面值 状态转移方程： dp(n) = dp(n - i) + 1 。i是coins中给定的值。取dp(n)最小值

```
/// 自己传入目标硬币和硬币种类
    /// - Parameters:
    ///   - n: <#n description#>
    ///   - nums: <#nums description#>
    /// - Returns: <#description#>
    func minCoins4(money n: Int, nums: [Int]) -> Int {
        if n == 0 { return 0 }
        if n < 0 || nums.count == 0 {
            return -1
        }
        var dp = Array(repeating: 0, count: n + 1)
        for i in 1...n {
            var min = Int.max
            for num in nums {
                if i < num { continue }
                let v = dp[i - num]
                if v == -1 || v > min { continue }
                min = v
            }
            if min == Int.max {
                dp[i] = -1
            }else {
                dp[i] = min + 1
            }
        }
        return dp[n]
    }
```

## 2. 最长公共子序列

> 给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。
>
> 一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺

> 序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。
>
> 例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。 两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

步骤：

1. 定义状态： dp[i][j]是nums1[0-i]和nums2[0-j]的最长公共子序列的长度
2. 初始值： dp[0][j] 和dp[i][0] = 0
3. 状态转移方程： if num1[i - 1] == num2[j - 1] dp[i][j] = dp[i - 1][j - 1] + 1 。 else num1[i - 1] != num2[j - 1] dp[i][j] = max(dp[i][ j - 1],dp[ i - 1 ][j])

```
    func longestCommonSubsequence(_ text1: String, _ text2: String) -> I
        var nums1 = Array(text1)
        var nums2 = Array(text2)
        var dp = Array(repeating: Array(repeating: 0, count: nums2.count
        for i in 1...nums1.count {
            for j in 1...nums2.count {
                dp[i][0] = 0
                dp[0][j] = 0
                if nums1[i - 1] == nums2[j - 1] {
                    dp[i][j] = dp[i - 1][j - 1] + 1
                }else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
                }
            }
        }
        return dp[nums1.count][nums2.count]
    }
```

## 3.最长公共子串

> 子串是连续的子序列 求两个字符串的最长公共子串长度 ABCBA 和 BABCA 的最长公共子串是 ABC，长度为 3

步骤：

1. 定义状态：dp[i][j]是str1[i - 1]和str2[j - 1]的最长公共子串
2. 初始值：str1[0] && str2[n] str1[n]&&str2[0] 初始值为0
3. 状态转移方程： 如果str1[i - 1] str2[j - 1]相等 dp[i][j] = dp[i - 1][j - 1] + 1

```
    func longgestCommonSubString(_ str1: String, _ str2:String) -> Int {
        guard str1.count > 0 && str2.count > 0 else { return 0 }
        let list1 = Array(str1)
        let list2 = Array(str2)
        var dp = Array(repeating: Array(repeating: 0, count: list2.count
        var maxValue = 0
        for i in 1...list1.count {
            for j in 1...list2.count {
                if list1[i - 1] != list2[j - 1] { continue }
                dp[i][j] = dp[i - 1][j - 1] + 1
                maxValue = max(maxValue, dp[i][j])
            }
        }

        return maxValue
    }
```

### 4. 背包问题

> 有 n 件物品和一个最大承重为 W 的背包，每件物品的重量是 $wi$、价值是 $vi$ 在保证总重量不超过 W 的前提下，选择某些物品装入背包，背包的最大总价值是多少? 注意:每个物品只有 1 件，也就是每个物品只能选择 0 件或者 1 件
>
> 假设 values 是价值数组，weights 是重量数组
>
> 编号为 k 的物品，价值是 values[k]，重量是 weights[k]，k ∈ [0, n)

步骤：

1. 定义状态：dp(i,j) 是第前i个物品中 最大能承受 j 重量 的背包的最大价值
2. 初始状态： dp[0][n] dp[n][0] == 0
3. 状态转移： 如果第 i 个物品的重量 > 背包能承受的重量 dp(i,j) = dp(i - 1,j), 如果第 i 个物品的重量 < 背包能承受的重量 两种情况 1. 不选最后一件物品： dp(i, j) = dp(i - 1, j) 2. 选最后一件： dp(i, j) = prices[i - 1] + max(i - 1, j - weights[i]) 选1，2的最大值

```
func packageBiggestPrice(weights: [Int], prices: [Int], target: Int) -
> Int {
        guard weights.count == prices.count && weights.count > 0 && price
        var dp = Array(repeating: Array(repeating: 0, count: target + 1)
        var maxValue = 0
        for i in 1...weights.count {
            for j in 1...target {
                if j < weights[i - 1] {
                    dp[i][j] = dp[i - 1][j]
                }else {
                    dp[i][j] = max(dp[i - 1][j], prices[i - 1] + dp[i -
                }
                maxValue = dp[i][j]
            }
        }
        return maxValue
    }
```