动态规划实战

动态规划简介

基本定义:
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（或者说分治）的方式去解决。

基本思路:

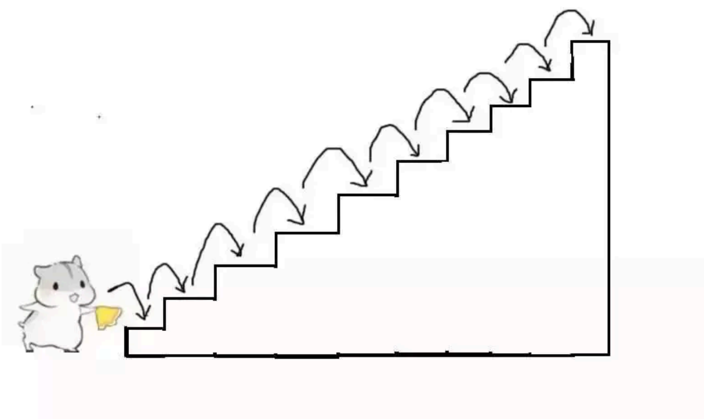
动态规划算法的基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

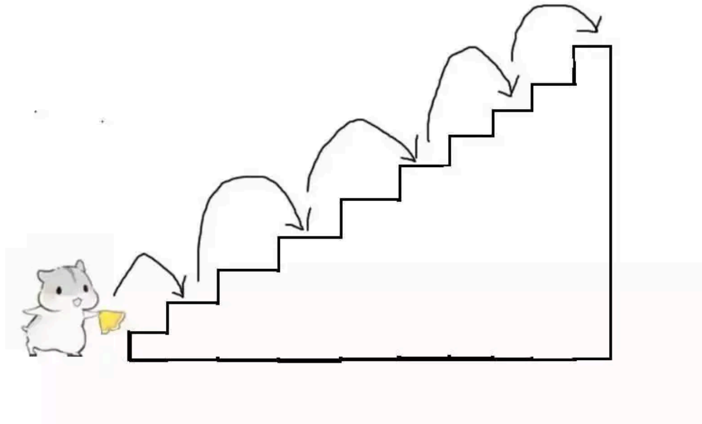
三要素:

（1）问题的阶段-----问题的边界
（2）每个阶段的状态-----最优子结构
（3）从前一个阶段转化到后一个阶段之间的递推关系----转态转移公式

动态规划–爬楼梯

问题描述：
有一座高度是**10**级台阶的楼梯，从下往上走，每跨一步只能向上**1**级或者**2**级台阶。要求用程序来求出一共有多少种走法

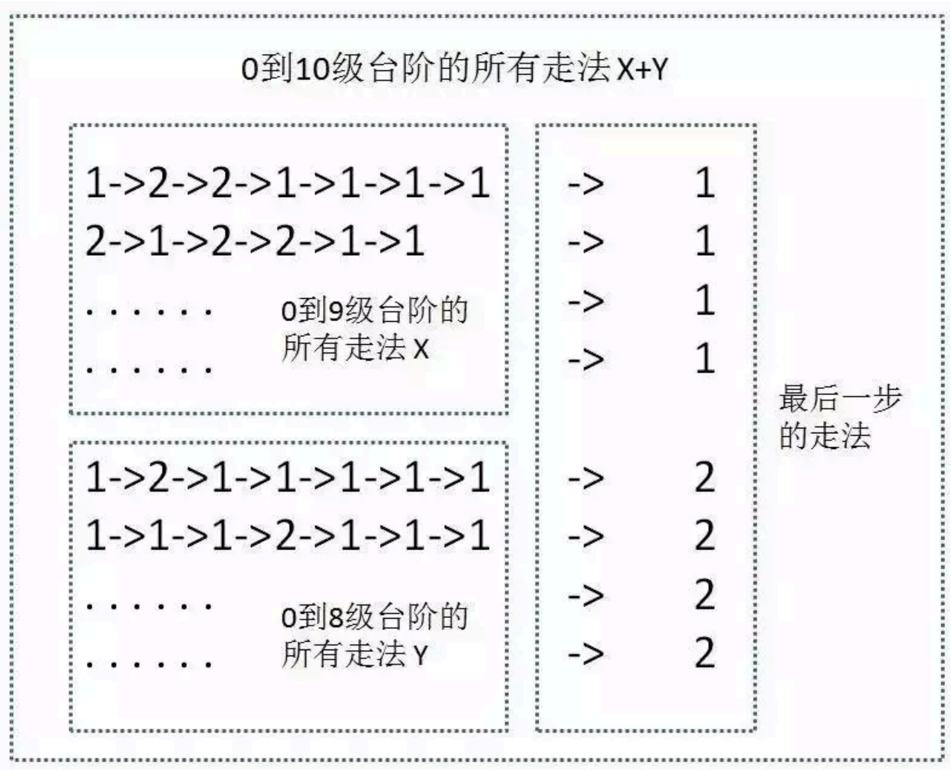




- 最优子结构：F(10)=F(9)+F(8)
- 边界：F(1),F(2)
- 状态转移方程：F(n)=F(n-1)+F(n-2)

https://juejin.im/post/5a29d52cf265da43333e4da7#comment

动态规划-爬楼梯




```python
def climb_stairs(n):
    """
    爬楼梯问题求解
    :param n: n个台阶
    :return:
    """
    # 初始化
    dp = [-1 for i in range(n+1)]
    dp[0] = 1   # n=1
    dp[1] = 2   # n=2
    # 状态方程dp[i] = dp[i - 1] + dp[i - 2]
    for i in range(2, n):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n - 1]
```

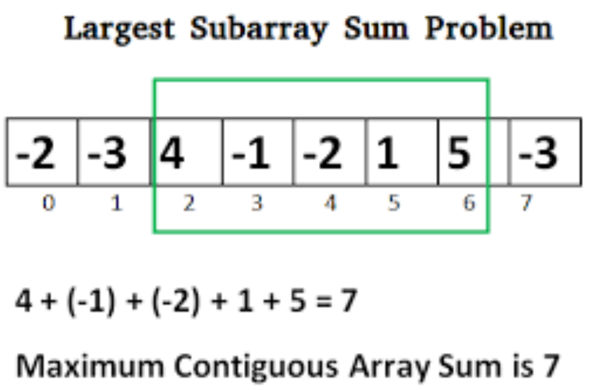
时间复杂度：O(N)

动态规划-最大连续子数组和

问题描述:

Given a sequence of n real numbers A(1) ... A(n), determine a contiguous subsequence A(i) ... A(j) for which the sum of elements in the subsequence is maximized.

给定一个实数数组A， 找到一个具有最大和的连续子数组（子数组最少包含一个元素）， 返回其最大和。



动态规划-最大连续子数组

参考思路：

- 设F(i)为以第i个元素结尾的最大的连续子数组的和
- 假设对于元素i，所有以它前面的元素结尾的子数组的长度都已经求得，那么以第i个元素结尾且和最大的连续子数组实际上，要么是以第i-1个元素结尾且和最大的连续子数组加上这个元素，要么是只包含第i个元素，即F(i)= max(sum[i-1] + a[i], a[i])
- 可以通过判断F(i-1) + a[i]是否大于a[i]来做选择，而这实际上等价于判断F(i-1)是否大于0
- 由于每次运算只需要前一次的结果，因此并不需要像普通的动态规划那样保留之前所有的计算结果，只需要保留上一次的即可，因此算法的时间和空间复杂度都很小。

◆ 最优子结构：F(i-1)+A[i],A[i]

◆ 边界：F(1)

◆ 状态转移方程： F(i)= max(F(i-1) + a[i], a[i])

https://blog.csdn.net/weixin_41958153/article/details/81131379

动态规划-最大连续子数组

```
def max_subarr(arr):
    """
    最大连续子数组求解
  :param arr: 数字列表
  :return:
    """
    length = len(arr)
    for i in range(1, length):
        # 当前值的大小与前面的值之和比较, 若当前值更大, 则取当前值, 舍弃前面的值之和
        max_sum = max(arr[i] + arr[i - 1], arr[i])
        arr[i] = max_sum  # 将当前和最大的赋给nums[i], 新的nums存储的为和值
    return max(arr)
```

```
res = max_subarr([-2, -3, 4, -1, -2, 1, 5, -3])
print(res)

# out:7
```

时间复杂度：O(N)

动态规划–硬币找零问题

问题描述：

给不同面值的硬币若干种（每种硬币个数无限多），用这若干种硬币组合为某种面额amount的钱，使用的硬币的个数最少。比如给定4种面额的硬币1分，2分，5分，6分，如果要找11分的零钱，怎么做才能使得找的硬币数量总和最少。

参考思路：

- 声明一个动态数组dp，其中dp[i]代表表示钱数为i时的最小硬币数的找零
- 初始化dp[0]=0，其他初始化为amount+1　思考下为什么这样做？
- 状态转移方程： dp[i] = min(dp[i], dp[i - coins[j]] + 1);

动态规划-硬币找零问题

```
def coin_change(coins, amount):
    """
    最少找零
  :param coins: 硬币面值
  :param amount: 找零金额
  :return: 返回找零最少硬币数
  """
    if amount < 1:
        return 0

    dp = [amount + 1] * (amount + 1)
    dp[0] = 0

    for amt in range(1, amount + 1):
        for coin in coins:
            if amt - coin >= 0: # 判断当前金额是否大于硬币
                dp[amt] = min(dp[amt], 1 + dp[amt - coin])

    if dp[amount] > amount:
        return -1

    return dp[amount]
```

时间复杂度：O(len(coins)*amount)

https://www.cnblogs.com/grandyang/p/5138186.html

动态规划–最长递增子序列

问题描述：

最长递增子序列（Longest Increasing Subsequence）是指找到一个给定序列nums的最长子序列的长度，使得子序列中的所有元素单调递增。例如：{ 3， 5， 7， 1， 2， 8 } 的 LIS 是 { 3， 5， 7， 8 }， 长度为 4。

参考思路：

- 声明一维dp数组，其中dp[i]表示以nums[i]为结尾的最长递增子串的长度
- 对于每一个nums[i]，我们从第一个数再搜索到i，如果发现某个数小于nums[i]，我们更新dp[i]
- 更新方法为dp[i] = max(dp[i], dp[j] + 1)，即比较当前dp[i]的值和那个小于num[i]的数的dp值加1的大小
- 不断的更新dp数组，到最后dp数组中最大的值就是我们要返回的LIS的长度

动态规划–最长递增子序列

```
def LIS(nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    if not nums:
        return 0
    # 初始化
    dp = [1 for _ in range(len(nums))]
    res = 1
    for i in range(len(nums)):
        for j in range(i):
            if nums[i] > nums[j]: # 判断条件
                dp[i] = max(dp[i], 1 + dp[j]) # 更新
                res = max(res, dp[i])

    return res
```

时间复杂度：O(N^2)

https://www.cnblogs.com/grandyang/p/4938187.html

动态规划-Longest Common Subsequence 最长公共子序列

问题描述:
given two strings x and y, find the longest common subsequence (LCS) and print its length

解题思路：

- Define subproblems
  - Let D_{ij} be the length of the LCS of $x_{1\ldots i}$ and $y_{1\ldots j}$
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS
    - $D_{ij} = D_{i-1,j-1} + 1$
  - Otherwise, either x_i or y_j does not contribute to the LCS, so one can be dropped
    - $D_{ij} = \max\{D_{i-1,j}, D_{i,j-1}\}$
  - Find and solve the base cases: $D_{i0} = D_{0j} = 0$

伪代码：

```
for(i = 0; i <= n; i++) D[i][0] = 0;
for(j = 0; j <= m; j++) D[0][j] = 0;
for(i = 1; i <= n; i++) {
    for(j = 1; j <= m; j++) {
        if(x[i] == y[j])
            D[i][j] = D[i-1][j-1] + 1;
        else
            D[i][j] = max(D[i-1][j], D[i][j-1]);
    }
}
```

https://web.stanford.edu/class/cs97si/04-dynamic-programming.pdf

动态规划-Edit Distance

问题描述：

给定 2 个字符串 a, b. 编辑距离是将 a 转换为 b 的最少操作次数，

操作只允许如下 3 种：
插入一个字符，例如：fj -> fxj
删除一个字符，例如：fxj -> fj
替换一个字符，例如：jxj -> fyj。

参考思路：

- dp[i][j]表示将字符串 **a[0: i-1]** 转变为 **b[0: j-1]** 的最小步骤数
- **边界**：dp[i][0] = i，b 字符串为空，表示将 a[1]-a[i] 全部删除，所以编辑距离为 I;dp[0][j] = j，a 字符串为空，表示 a 插入 b[1]-b[j]，所以编辑距离为 j
- 当 a[i] 等于 b[j] 时，d[i][j] = d[i-1][j-1], 比如 fxy -> fay 的编辑距离等于 fx -> fa 的编辑距离
- 当 a[i] 不等于 b[j] 时，d[i][j] 等于如下 3 项的最小值：
  - d[i-1][j] + 1（删除 a[i]），比如 fxy -> fab 的编辑距离 = fx -> fab 的编辑距离 + 1
  - d[i][j-1] + 1（插入 b[j]），比如 fxy -> fab 的编辑距离 = fxyb -> fab 的编辑距离 + 1 = fxy -> fa 的编辑距离 + 1
  - d[i-1][j-1] + 1（将 a[i] 替换为 b[j]），比如 fxy -> fab 的编辑距离 = fxb -> fab 的编辑距离 + 1 = fx -> fa 的编辑距离 + 1

动态规划-Edit Distance

```
def minDistance(word1, word2):
    """
    :type word1: str
    :type word2: str
    :rtype: int
    """
    m, n = len(word1), len(word2)
    if m == 0:
        return n
    if n == 0:
        return m
    dp = [[0] * (n + 1) for _ in range(m + 1)]  # 初始化dp和边界
    for i in range(1, m + 1):
        dp[i][0] = i
    for j in range(1, n + 1):
        dp[0][j] = j
    for i in range(1, m + 1):  # 计算dp
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(dp[i - 1][j - 1] + 1, dp[i][j - 1] + 1, dp[i - 1][j] + 1)
    return dp[m][n]
```

时间复杂度：O(m*n)