第5章

扫一扫

视频讲解

动态规划

扫一扫

思政教学案例

本章学习要点

- 动态规划的基本思想
- 适合用动态规划求解的问题具有的两个重要性质
  - 最优子结构性质
  - 重叠子问题性质
- 动态规划的解题步骤
  - 找出最优解的性质，并刻画其结构特征
  - 递归地定义最优值
  - 以自底向上的方式计算出最优值
  - 根据计算最优值时得到的信息，构造最优解
- 通过典型应用学习动态规划算法设计策略
  - 斐波那契数列
  - 数字三角形问题
  - 0-1 背包问题
  - 矩阵连乘问题
  - 最长公共子序列
  - 最长不上升子序列
  - 编辑距离问题
  - 最优二叉搜索树

前面学习了递归和分治，两种算法设计策略的共同点是把问题转化或分解为规模较小的子问题进行求解。对于某些问题，分解得到的子问题有重复的情况，此时使用分治法会导致子问题重复计算，影响算法效率。

对于此类问题，如何避免子问题的重复计算以降低算法时间复杂度？本章讨论的动态规划(Dynamic Programming，DP)算法设计策略将给出解决这类问题的一般方法。首先通过两个引例阐述动态规划的基本思想和解题步骤，然后介绍若干典型应用，借助典型应用的问题分析和求解进一步加深对动态规划的理解和运用。

扫一扫

视频讲解

5.1 引例一：兔子繁殖问题

问题描述 一对兔子从出生后第 3 个月开始，每月生一对小兔子。小兔子到第 3 个月开始生下一代小兔子。如果兔子只生不死，1 月抱来一对刚出生的小兔子，问一年中每个月

各有多少对兔子?

问题分析 这个问题是依据兔子的繁殖规律,推算每个月兔子的数量。根据题目描述可以得到前两个月都是1对;第3个月加上繁殖的1对小兔子,共2对;第4个月,最开始的1对依旧会繁殖1对小兔子,而其他兔子暂没有繁殖能力,共3对;第5个月,除了最开始的1对会繁殖1对外,第3个月出生的小兔子也可以繁殖1对,共5对,以此类推,如图5-1所示。这种方法随着月份的增长,分析越来越困难。下面从兔子的繁殖规律进行分析,推算兔子数量的计算公式。

扫一扫

看彩图

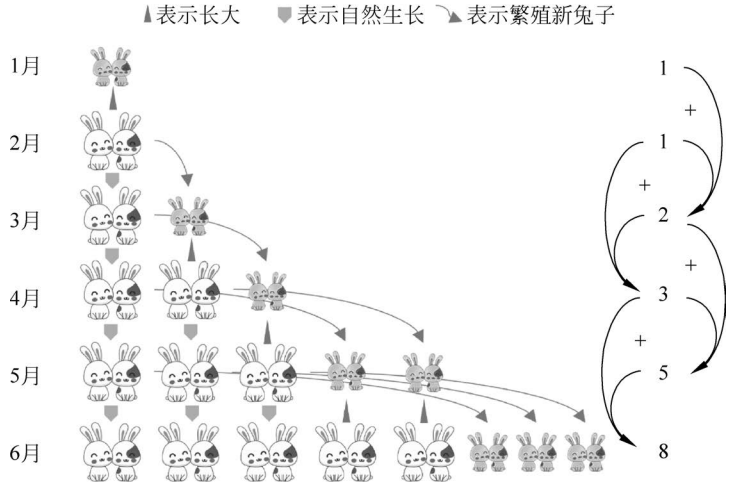


图5-1 兔子繁殖过程

由于兔子到第3个月才具备繁殖能力,因此,第n个月的兔子对数$F(n)$由两部分组成,一部分是上个月,即第$n-1$个月的兔子保持不变,另一部分是新出生的兔子,因为第$n-2$个月的兔子到第n个月具备繁殖能力,即新出生的兔子数量为第$n-2$个月的兔子数量,因此,可以得到$F(n)=F(n-1)+F(n-2)$。这个递推关系是一个非常有名的数列——斐波那契数列(Fibonacci),该数列的完整描述如下。

$$F(n)=\begin{cases}1, & n=0,1\\ F(n-1)+F(n-2), & n\text{ 取其他值}\end{cases} \tag{5-1}$$

下面讨论斐波那契数列的求解方法。

1. 递归方法

$F(n)=F(n-1)+F(n-2)$具有明显的递归特点,也可以看作基于分治策略的一种解决方法。不难写出递归实现方法,如代码5-1所示。

代码5-1:斐波那契数列的递归实现

```
def F(n):
    if n == 0 or n == 1:                    #边界条件,当 n = 0 或 n = 1 时 F(n) = 1
        return 1
    else:
        return F(n - 1) + F(n - 2)          #递归式

if __name__ == '__main__':
```

```
n = int(input())
print(F(n))
```

在计算过程中，$F(n)$分为 $F(n-1)$和 $F(n-2)$两部分，$F(n-1)$需要计算 $F(n-2)$和 $F(n-3)$，$F(n-2)$需要计算 $F(n-3)$和 $F(n-4)$，以此类推，如图 5-2 所示。

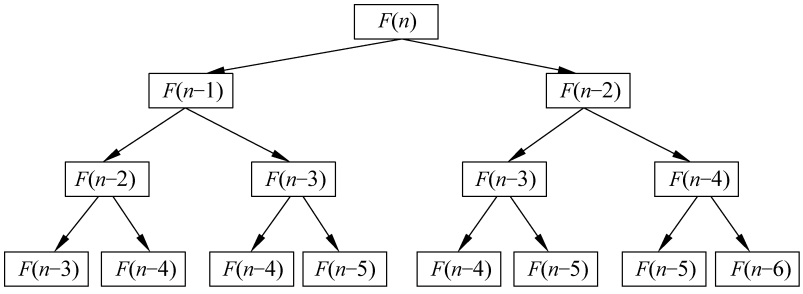


图 5-2　递归法求解斐波那契数列

从图 5-2 可以看出，在求解 $F(n)$的过程中，有很多子问题被多次重复计算，如 $F(n-3)$要计算 3 次，$F(n-4)$要计算 5 次等，重复计算必然导致算法效率的降低。运用递归算法的时间复杂度分析方法，该算法的时间复杂度可表示为

$$T(n)=T(n-1)+T(n-2)+O(1) \tag{5-2}$$

运用递归树方法求解式(5-2)，得到 $T(n)=O(2^n)$。

可见，递归实现的代码 5-1 虽然具有代码简洁、容易理解的优点，但子问题多次重复计算导致算法效率很低，实用性差。为了优化算法，需要寻找避免问题重复计算的方法。

2. 带记忆的递归

基于朴素的思想，如果把问题结果保存起来，需要使用时直接查询，就可以避免问题的重复计算。这种实现方法称为带记忆的递归实现，也称为备忘录方法。代码 5-2 为斐波那契数列的备忘录实现。

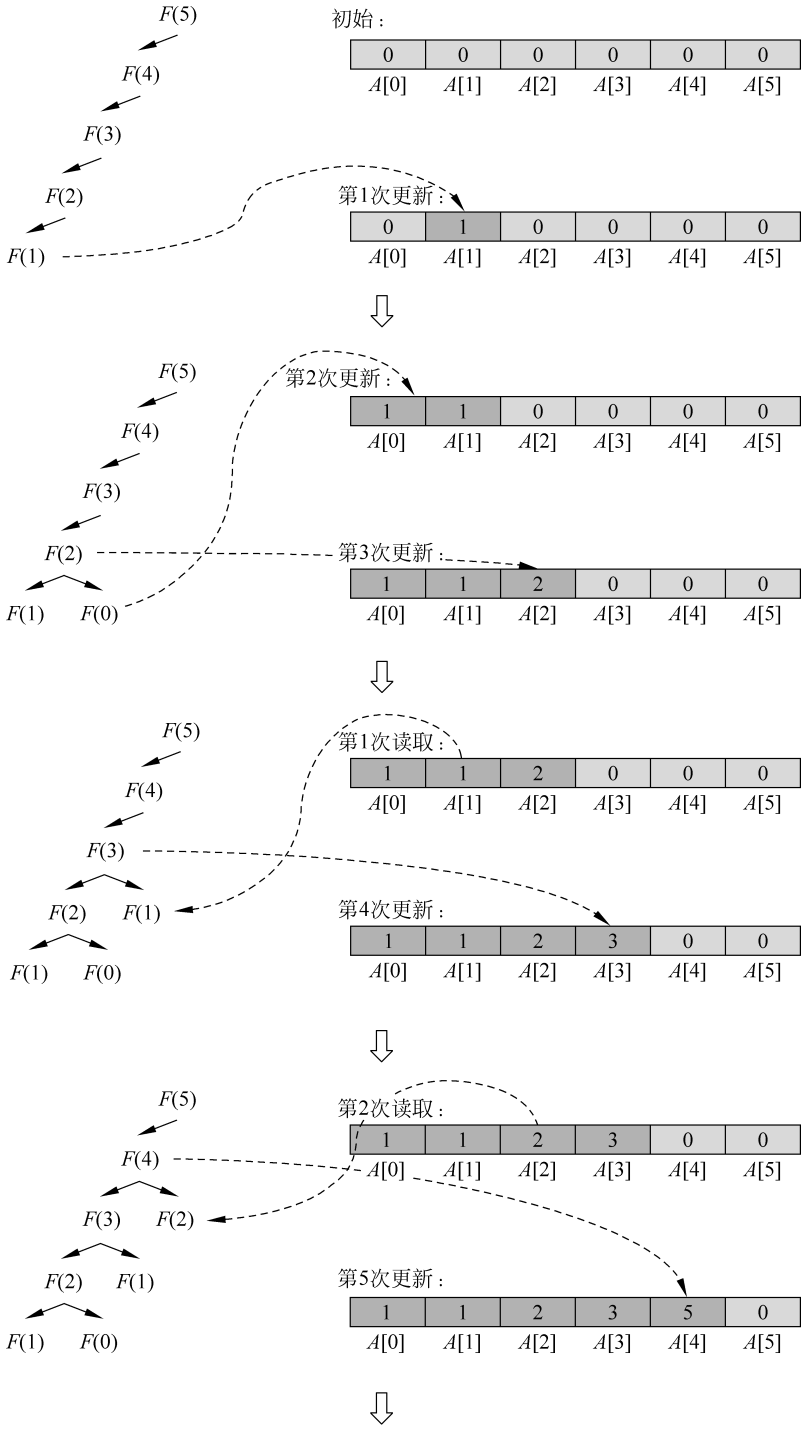
代码 5-2：斐波那契数列的备忘录实现

```
def F(n):
    global A
    if n == 0 or n == 1:                    #初始条件
        A[n] = 1
    elif A[n] == 0:
        A[n] = F(n-1) + F(n-2)              #记录 F(n)的值到 A(n)
    return A[n]

if __name__ == '__main__':
    n = int(input())
    A = [0 for i in range(n+1)]
    print(F(n))
```

与代码 5-1 不同，代码 5-2 增加数组 A 保存问题结果，初始时数组元素为 0。递归调用前，首先判断 $A[n]$是否为 0，如果为 0 则表示 $F(n)$还没有计算，此时进行递归调用，同时把结果存入 $A[n]$；如果 $A[n]$不为 0，则表示问题 $F(n)$已经计算完毕，直接返回结果 $A[n]$；

边界条件处理中，增加把结果存入 A 中的处理($A[n]=1$)。该算法中，由于每个问题只计算了一次，因此，代码 5-2 的时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。具体的计算过程如图 5-3 所示。



扫一扫

看彩图

图 5-3　斐波那契数列备忘录方法的执行过程示意图

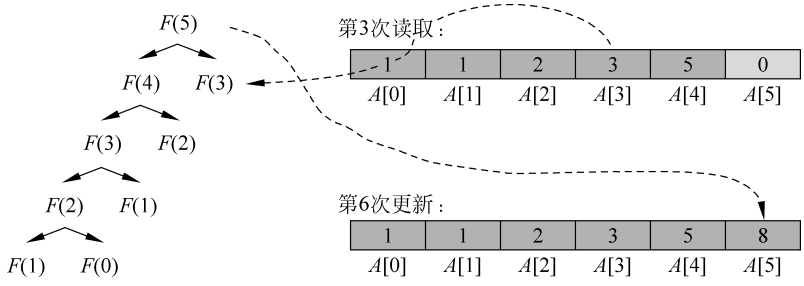


图 5-3　（续）

3. 递推实现一

代码 5-1 和代码 5-2 都是运用递归技术，从大问题先递归求解小问题，再从小问题结果得到大问题结果。

结合斐波那契数列的定义 $F(n)=F(n-1)+F(n-2)$，可以采用更简单的方法——递推法，从小问题依次推出大问题，具体实现见代码 5-3。

代码 5-3：斐波那契数列的递推实现一

```
def F(n):
    A[0] = A[1] = 1                          #初始值
    for i in range(2, n+1):
        A[i] = A[i-1] + A[i-2]               #递推公式
    return A[n]

if __name__ == '__main__':
    n = int(input())
    A = [0 for i in range(n+1)]
    print(F(n))
```

4. 递推实现二

代码 5-3 仍然会用到数组 A，时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。如果只需要得到第 n 个月的兔子对数，可以进一步降低空间需求，具体实现见代码 5-4。

代码 5-4：斐波那契数列的递推实现二

```
def F(n):
    a = 1
    b = 1
    c = 0                        #初始化,用3个变量滚动存储数列的值
    for i in range(2, n+1):
        c = a + b                #递推公式
        a = b                    #滚动更新值
        b = c
    return c

if __name__ == '__main__':
    n = int(input())
    print(F(n))
```

代码 5-4 只需 3 个变量 a、b、c，依次表示前两个月、前一个月以及本月的兔子数，每次计算后，更新 a 为 b，更新 b 为 c。算法时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

以上讨论了斐波那契数列问题的4种不同的实现方法,表5-1列出了每种算法的时间复杂度和空间复杂度。

表5-1 求解斐波那契数列问题的算法实现比较

比 较 项	递归实现	带记忆的递归实现	递推实现一	递推实现二
对应代码	代码5-1	代码5-2	代码5-3	代码5-4
时间复杂度	$O(2^n)$	$O(n)$	$O(n)$	$O(n)$
空间复杂度	$O(1)$	$O(n)$	$O(n)$	$O(1)$

代码5-1由于有重复计算,因此时间复杂度高,其他3种方法通过保存问题结果消除了重复计算,降低了时间复杂度,但代价是需要一个长度为n的一维数组或若干变量。此外,在代码5-1中,空间复杂度未考虑递归调用时系统栈空间的开销。

本节从兔子繁殖问题引出斐波那契数列,并就该问题讨论了多种时空复杂度不同的求解方法,从中可以发现:在问题分析中,找规律很重要,可以使问题更清晰;在问题求解中,通过分析算法的时空复杂度,找到影响算法效率的关键问题,对其改进可以实现算法的优化。

引申:时空转换

时间复杂度和空间复杂度是评价算法的重要方面。在算法设计中,经常会使用时空转换方法优化算法。例如,本节中通过保存问题结果以避免重复计算就是一种用空间换时间的方法。在空间资源相对紧缺的应用中,还可以用时间换空间的方法以达到减少空间的目的。

扫一扫

视频讲解

5.2 引例二:数字三角形问题

问题描述 如图5-4所示,由若干数字构成一个三角形,上面第1层是塔顶,最下面一层是塔底。除塔底外,每个数字可沿左下或右下方向到达下一层。求一条从塔顶到塔底的路径,使该路径上所经过节点值的和最大。

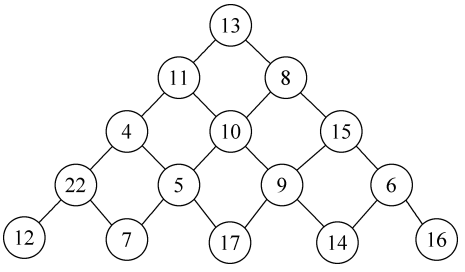


图5-4 数字三角形问题

问题分析 从塔顶13号节点出发,每向下走一层,都有两个方向可以选择。如果塔高为n层(塔顶为一个元素,塔底为n个元素),采用穷举法可得到共2^{n-1}种可能的路径,时间复杂度为$O(2^{n-1}n)$,效率很低。

运用前面学习的递归与分治策略,从问题是否可以分解为规模更小的子问题的角度出发,分析该问题。从塔顶13号节点出发,究竟该往左下(经过11号节点)还是右下走(经过8号节点)取决于左下方11号节点到塔底的最大路径长度与右下方8号节点到塔底的最大

路径长度的较大者。进一步分析，左下方 11 号节点到塔底的最大路径长度取决于其左下方 4 号节点与右下方 10 号节点到塔底的最大路径长度的较大者；右下方 8 号节点到塔底的最大路径长度取决于其左下方 10 号节点与右下方 15 号节点到塔底的最大路径长度的较大者…直至塔底，而塔底位置到塔底的最大路径长度是数据本身，如塔底 12 号节点到塔底的最大路径长度是 12。因此，数字三角形问题满足某个数到塔底的最长路径等于该数加上其左下方的数到塔底的最长路径和右下方的数到塔底的最长路径中的大者。

把数字三角形最左边一列对齐后可以看作一个下三角矩阵，假设塔顶位于 1 行 1 列，定义第 i 行 j 列的数 $a[i][j]$ 到塔底的最长路径长度为 $f(i,j)$，那么有

$$\begin{cases} f(i,j)=a[i][j]+\max\{f(i+1,j),f(i+1,j+1)\}, i<n(n\text{ 为高度}) \\ f(n,j)=a[n][j] \end{cases} \tag{5-3}$$

很明显，式(5-3)具有明显的递归特点，可以采用递归法求解。

1. 递归法

根据式(5-3)，当 $i=n$ 时，直接返回，否则递归调用 $f(i+1,j)$ 和 $f(i+1,j+1)$。具体实现见代码 5-5。

代码 5-5：数字三角形问题的递归实现

```
def f(i, j):                    #f(i,j)表示 a[i][j]到塔底的最长路径长度
    global a, n
    if i == n:                  #边界条件,a[i][j]存储数字三角形相应位置的值,塔顶在 a[1][1]
        return a[i][j]
    return a[i][j] + max(f(i+1, j), f(i+1, j+1))     #a[i][j]到塔底的最长路径长度

if __name__ == '__main__':
    n = int(input())
    a = [[None for i in range(n+1)] for j in range(n+1)]

    data = list(map(int, input().split()))
    num = 0
    for i in range(1, n+1):
        for j in range(1, i+1):
            a[i][j] = data[num]
            num += 1
    print(f(1, 1))
```

如果用 $T(n)$ 表示高度为 n 的数字三角形问题的时间复杂度，那么 $T(n)=2T(n-1)+O(1)$，应用迭代法求解，可以得到 $T(n)=O(2^n)$。进一步分析可知，复杂度高的原因是在递归计算过程中有很多问题需要重复计算，如 8 号节点和 11 号节点到塔底的最长路径，都要用到 10 号节点到塔底的最长路径。因此，需要计算两次 10 号节点到塔底的最长路径。

根据 5.1 节介绍的方法，为了避免问题的重复计算，可以采用带记忆的递归实现，具体代码实现，在此不再详述，读者可自行完成。

2. 递推法自底向上求解

除了递归实现方法外，也可以采用先求解小规模问题，再求解大规模问题的方法。这里的自底向上表示自小规模问题求解大规模问题的含义，具体实现见代码 5-6。

代码 5-6：数字三角形问题的自底向上求解

```
def fun():
    global f
    for j in range(1, n + 1):
        f[n][j] = a[n][j]
    for i in range(n - 1, 0, -1):                    #从下向上加,更新 f[i][j]的值,取较大者
        for j in range(1, i + 1):
            f[i][j] = a[i][j] + max(f[i + 1][j], f[i + 1][j + 1])
    return f[1][1]

if __name__ == '__main__':
    n = int(input())
    f = [[0 for i in range(n + 1)] for j in range(n + 1)]      #f[i][j]记录 f(i,j),初始为 0
    #a[i][j]存储数字三角形相应位置的值,塔顶在 a[1][1]
    a = [[0 for i in range(n + 1)] for j in range(n + 1)]

    data = list(map(int, input().split()))
    num = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            a[i][j] = data[num]
            num += 1
    print(fun())
```

在代码 5-6 中，函数的第 1 个 for 循环计算塔底的最长路径并把结果存入 $f[n][j]$，接下来两重 for 循环从倒数第 2 层开始，自左向右依次求解每个位置(i,j)到塔底的最长路径并存入 $f[i][j]$。由于 $f[i][j]$依赖于 $f[i+1][j]$和 $f[i+1][j+1]$，因此整个计算采取从下而上、从左至右的次序，如图 5-5 所示。

以图 5-4 的具体数据为例，代码 5-6 执行后对应 $f[i][j]$的结果如图 5-6 所示。

代码 5-6 需要开辟一个二维数组 f 保存问题结果，所以其空间复杂度为 $O(n^2)$，时间主要花费在两重循环中，时间复杂度为 $O(n^2)$，优于穷举法和代码 5-5。

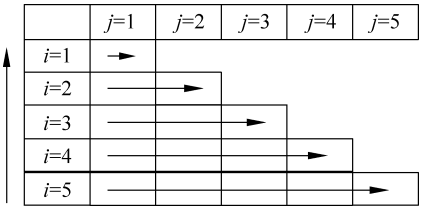


图 5-5　问题的计算次序

	j=1	j=2	j=3	j=4	j=5
i=1	62				
i=2	49	49			
i=3	38	36	41		
i=4	34	22	26	22	
i=5	12	7	17	14	16

图 5-6　数组 f 的结果

从兔子繁殖和数字三角形问题的求解过程中可以发现它们具有以下共性。

(1) 都是最优化问题。数字三角形问题是在若干路径中找到一条长度最大的路径；兔子繁殖问题虽然不是严格意义上的最优化问题，但也可以把每天的兔子数量看作一个最大值的问题。

(2) 问题的最优值和子问题的最优值有一定关系。数字三角形问题中，某个位置(i,j)到塔底的最大路径长度与其左下位置$(i+1,j)$和右下位置$(i+1,j+1)$到塔底的最大路径长度有关；兔子繁殖问题中，第 i 个月的兔子数量与第 $i-1$ 个月和第 $i-2$ 个月的兔子数量有关。

(3) 子问题需多次使用，通过保存子问题的结果避免重复计算。数字三角形问题中，除了边缘位置($j=1$ 或 $i=j$)外，某个位置(i,j)到塔底的最大路径长度需要使用多次；兔子繁殖问题中，第 i 个月的兔子数量在求第 $i+1$ 个月的兔子数量和求第 $i+2$ 个月的兔子数量时需要使用。对于需要多次使用的子问题，为了避免重复计算，在子问题计算完毕后，把子问题的结果保存起来，需要使用时直接查询。

以上 3 点是逐步深入的，在最优化问题中，如果问题和子问题的最优值具有一定的关系，而且子问题会重复使用，那么可以先求解子问题并存储子问题的结果，利用问题和子问题的最优值关系，逐步得到较大规模问题的最优值，直至得到原问题的最优值。这种解题方法其实就是本章要学习的一种新的算法设计策略——动态规划，下面介绍动态规划的基本思想和解题步骤。

扫一扫

视频讲解

5.3　动态规划基本思想

5.3.1　动态规划与分治法的区别

动态规划与分治法类似，其基本思想也是将待求解问题分解成若干子问题，然后对这些子问题分别求解，如果子问题的规模仍然不够小，再继续划分为若干规模更小的子问题，直到问题规模足够小，很容易求出其解为止；最后合并子问题的结果得到原问题的结果，这是动态规划与分治的共同点。

但与分治法不同的是，适合用动态规划求解的问题，经分解得到的子问题往往不是互相独立的，不同子问题的数目常常只有多项式量级。这里的子问题互相独立是指子问题间没有共同的计算部分，或者说某个子问题不会被重复使用多次。

例如，归并排序采取的是一种分治策略，每次把待排序序列从中间位置分为左、右两个子问题，这两个子问题的排序过程没有重叠部分，所以这两个子问题是互相独立的；又如，快速排序也是基于分治的思想，通过每个元素和基准元素的比较，把待排序序列分为比基准小的一部分和比基准大的一部分，这两部分的排序也是互相独立的；而对于数字三角形问题，通过 5.2 节的分析可知，有些子问题需要重复使用多次，子问题不是互相独立的。对于这种情形，如果采用分治法，会导致子问题的重复计算，算法时间复杂度较高。此时，适合使用动态规划方法，通过保存子问题结果，避免子问题重复计算，从而降低时间复杂度，优化算法性能。因此，可以通过判断子问题是否相互独立，选择使用分治法还是动态规划。

5.3.2　适合用动态规划求解的问题具有的两个重要性质

一般来说，适合用动态规划求解的问题一般具有两个重要性质：最优子结构性质和重叠子问题性质。

1. 最优子结构性质

最优子结构性质是指对于最优化问题，问题的最优解包含其子问题的最优解。以数字三角形问题为例，如果塔顶到塔底的最长路径为 $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \cdots \rightarrow a_n$，那么，$a_2$ 到塔底的最长路径一定为 $a_2 \rightarrow a_3 \rightarrow \cdots \rightarrow a_n$。因此，数字三角形问题具有最优子结构性质。

判断一个问题是否具有最优子结构性质，通常使用反证法。首先，假设由问题的最优解导出的子问题的解不是最优的；其次，说明在假设下可构造出比原问题最优解更好的解，从而导出矛盾。

仍以数字三角形问题为例，如果 a_2 到塔底的最长路径不是 $a_2 \rightarrow a_3 \rightarrow \cdots \rightarrow a_n$，而是 $a_2 \rightarrow b_3 \rightarrow \cdots \rightarrow b_n$，那么 $a_1 \rightarrow a_2 \rightarrow b_3 \rightarrow \cdots \rightarrow b_n$ 的路径长度大于 $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \cdots \rightarrow a_n$ 的路径长度，与已知 $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \cdots \rightarrow a_n$ 是塔顶到塔底的最长路径矛盾，因此假设不成立，从而得到数字三角形问题满足最优子结构性质。

利用问题的最优子结构性质，可以自底向上的方式从子问题的最优解逐步构造出整个问题的最优解。

2. 重叠子问题性质

重叠子问题性质是指问题依赖的子问题并不总是新问题，有些子问题会重复出现多次。

动态规划算法对每个子问题只求解一次，并将解保存在数组(表格)中，当需要再次计算该子问题时，可以用常数时间查表获得结果，避免重复计算。通常，不同子问题的个数随问题的大小呈多项式增长，因此动态规划算法只需多项式时间即可获得较高的解题效率。

5.3.3 动态规划的解题步骤

在判断问题满足最优子结构性质和重叠子问题性质后，使用动态规划方法求解的步骤如下。

(1) 分析问题和子问题的关系，找出最优解的性质，并刻画其结构特征。

(2) 递归地定义最优值。

(3) 以自底向上的方式计算出最优值。

(4) 根据计算最优值时得到的信息，构造最优解。

步骤(1)和步骤(2)中，对问题进行合适的定义、确定问题和子问题的关系是关键，对问题的定义不同，问题和子问题的关系也有所不同。这里的关系是指问题最优值和子问题最优值之间的递归关系，即问题的最优子结构性质。步骤(3)给出了具体的实现方法，可以采用自底向上方法(从小问题逐步推出大问题)，也可以采用备忘录方法(带记忆的递归实现)。步骤(4)是可选步骤，在需要求最优解时使用，求解最优解通常需要记录问题最优值对应的最优解的相关信息，然后从整个问题逐步倒推到子问题，得到最优解。

扫一扫

视频讲解

5.4 动态规划应用：0-1 背包问题

0-1 背包问题是一个经典的动态规划算法求解问题，有多种不同的求解方法，在本书的很多章节都有出现。

问题描述 给定 n 个物品和一个背包，物品 i 的重量为 w_i，价值为 v_i，背包的容量为 C。现在需要从 n 个物品中选择若干物品装入背包，使得在不超过背包容量的前提下，装入背包中的物品价值之和最大。规定物品不能分割，即对每个物品只有两种选择：要么装入，要么不装，因此称为 0-1 背包问题。

为了对问题有更清楚的认识，下面给出 0-1 背包问题的形式化描述。

给定 $C>0$，$w_i>0$，$v_i>0$，$1 \leqslant i \leqslant n$，要求找出一个 n 元 0-1 向量 $(x_1, x_2, \cdots, x_n)$，$x_i \in \{0,1\}$，使得在满足 $\sum_{i=1}^{n} w_i x_i \leqslant C$ 的条件下，$\sum_{i=1}^{n} v_i x_i$ 达到最大。

问题分析 0-1 背包问题比较容易理解，在实际生活中的应用也比较多。例如，外出旅行时，想拿的东西可能很多，但行李箱的承重或空间有限，因此只能选择部分物品装入。由

于每个物品有两种选择，所以 n 个物品的所有装包方案共 2^n 种。针对每种方案，在符合物品重量总和不大于背包容量的条件下，找出价值总和最大的一种。这是一种穷举法的解题思路，由于每种方案最多对 n 个物品求解重量和以及价值和，因此，穷举法的时间复杂度为 $O(n2^n)$，达到了指数级，当 n 较大时，穷举法的效率很低，无法实现有效求解。

0-1 背包问题是否可以运用动态规划求解呢？这需要 0-1 背包问题满足动态规划的两个性质。若满足最优子结构性质，则在 0-1 背包问题的最优解中，如果把物品 j 从背包中拿出来，剩下的物品一定是取自 $n-1$ 个物品且不超过背包容量为 $C-w_j$ 的价值最高的解。为了证明这一点，给出如下命题。

命题　已知 $(x_1,x_2,\cdots,x_n)$ 是 0-1 背包问题的最优解，则 $(x_2,x_3,\cdots,x_n)$ 是满足 $\sum_{i=2}^{n}w_ix_i\leqslant C-w_1x_1$，且 $\sum_{i=2}^{n}v_ix_i$ 达到最大的最优解。

根据该命题，已知 $(x_1,x_2,\cdots,x_n)$ 是 0-1 背包问题的最优解，若 $x_1=0$，说明第 1 个物品不放入背包，那么 $(x_2,x_3,\cdots,x_n)$ 是 0-1 背包问题的最优解；若 $x_1=1$，说明第 1 个物品放入背包，那么 $(x_2,x_3,\cdots,x_n)$ 是在背包容量为 $C-w_1x_1$ 时的最优解。以上两种情况说明，问题的最优解包含子问题的最优解。因此，如果该命题成立，说明 0-1 背包问题满足最优子结构性质。

下面用反证法证明该命题的正确性。

假设 $(x_2,x_3,\cdots,x_n)$ 不是满足 $\sum_{i=2}^{n}w_ix_i\leqslant C-w_1x_1$，且 $\sum_{i=2}^{n}v_ix_i$ 达到最大的最优解，其对应的最优解为 $(z_2,z_3,\cdots,z_n)$，那么有 $\sum_{i=2}^{n}v_iz_i>\sum_{i=2}^{n}v_ix_i$ 成立，得 $v_1x_1+\sum_{i=2}^{n}v_iz_i>\sum_{i=1}^{n}v_ix_i$，即 $(x_1,z_2,\cdots,z_n)$ 是 0-1 背包问题的最优解，这与已知条件 $(x_1,x_2,\cdots,x_n)$ 是 0-1 背包问题的最优解矛盾。因此，假设不成立，从而命题得证。下面结合一个具体实例说明 0-1 背包问题的最优子结构性质。

假设背包容量 $C=7$，物品个数 $n=3$，重量 w_i 依次为 $(3,4,5)$，价值 v_i 依次为 $(5,6,10)$。很明显，该问题的最优解为 $(1,1,0)$，对应的最大价值为 11。根据最优子结构性质，若去除物品 3，那么 $x_1=1,x_2=1$ 一定是问题（背包容量 $C=7$，物品数 $n=2$，物品重量 $(w_1,w_2)=(3,4)$，物品价值 $(v_1,v_2)=(5,6)$）的最优解。同样，若去除物品 2，那么 $x_1=1,x_3=0$ 一定是问题（背包容量 $C=7-x_2w_2=7-1\times4=3$，物品数 $n=2$，物品重量 $(w_1,w_3)=(3,5)$，物品价值 $(v_1,v_3)=(5,10)$）的最优解。

扫一扫

视频讲解

5.4.1　动态规划求解 0-1 背包问题

步骤 1　分析问题和子问题的关系，找出最优解的性质，并刻画其结构特征。

根据对最优子结构性质的分析可知，0-1 背包问题中影响最优值的因素有物品个数（包含物品的重量和价值）和背包容量。因此，定义 $f(i,j)$ 表示可选物品为 $i,i+1,\cdots,n$，背包容量为 j 时 0-1 背包问题对应的最大价值，$f(1,C)$ 即是整个问题的最优值。

首先考虑规模最小的子问题 $f(n,j)$，表示背包容量为 j，只有第 n 个物品可选的最大价值。此时，问题比较简单，如果 $w_n\leqslant j$，物品 n 装入背包；如果 $w_n>j$，物品 n 不能装入背包，因此有

$$f(n,j)=\begin{cases}v_n, & w_n \leqslant j\\ 0, & w_n > j\end{cases} \tag{5-4}$$

在此基础上,增加第 $n-1$ 个物品,考虑子问题 $f(n-1,j)$的求解。$f(n-1,j)$表示背包容量为 j,可选物品为第 $n-1$ 个和第 n 个时的最大价值。对于第 $n-1$ 个物品,有以下两种选择。

(1) 如果 $w_{n-1}\leqslant j$,该物品可以装入背包,此时得到的价值为该物品的价值 v_{n-1} 加上物品为 n、背包容量为 $j-w_{n-1}$ 时的价值 $f(n,j-w_{n-1})$;该物品也可以选择不装入背包,此时得到的价值为 $f(n,j)$。

(2) 如果 $w_{n-1}>j$,该物品无法装入背包,此时得到的价值为 $f(n,j)$。

根据以上分析,可以得到

$$f(n-1,j)=\begin{cases}\max(v_{n-1}+f(n,j-w_{n-1}),f(n,j)), & w_{n-1}\leqslant j\\ f(n,j), & w_{n-1}>j\end{cases} \tag{5-5}$$

可见,问题 $f(n-1,j)$与子问题 $f(n,j)$和 $f(n,j-w_{n-1})$有关。以此类推,可以得到 $f(n-2,j),f(n-3,j),\cdots,f(1,j)$。

步骤 2 递归地定义最优值。

步骤 1 分析了 $f(n-1,j)$与 $f(n,j)$的关系,不难得到更一般的情况即 $f(i,j)$的求解方法,即

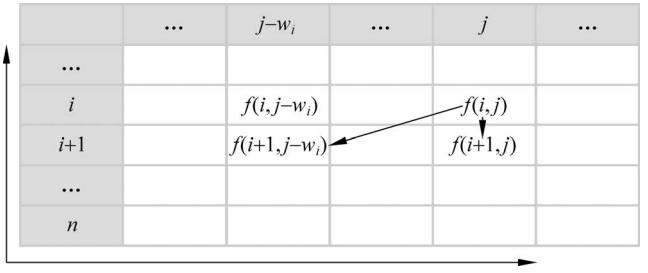
$$f(i,j)=\max(v_i+f(i+1,j-w_i),f(i+1,j)),i<n \tag{5-6}$$

$$f(n,j)=\begin{cases}v_n, & w_n \leqslant j\\ 0, & w_n > j\end{cases}$$

这里,对于第 i 个物品装与不装两种情况分别求解,取较大值作为 $f(i,j)$的结果。

步骤 3 以自底向上的方式计算出最优值。

根据最优值的递归关系,$f(i,j)$依赖 $f(i+1,j)$和 $f(i+1,j-w_i)$,如图 5-7 所示。因此,正确的计算次序应该是依次增加物品的个数和背包的容量,按照从下到上、从左到右的顺序计算,先求解规模小的问题,再求解规模大的问题。



扫一扫

看彩图

图 5-7 0-1 背包问题中问题与子问题的依赖关系

从图 5-7 中可以看出,对于子问题 $f(i+1,j-w_i)$,不仅计算 $f(i,j)$时需要,计算 $f(i,j-w_i)$时也需要,因此,0-1 背包问题具有子问题重叠性质,需要开辟一个二维数组存储 $f(i,j)$的结果。

步骤 4 根据计算最优值时得到的信息,构造最优解。

$f(i,j)$得到的是最优值,如果想获得最优解(每个物品是否选择),有多种方法。这里给出一种直观的方法。从最优值 $f(1,C)$反推,判断 $f(1,C)$是否等于 $f(2,C)$,如果等于,

说明物品 1 未装入背包，继续从 $f(2,C)$ 反推；否则，说明物品 1 装入背包，则继续从 $f(2,C-w_1)$ 反推；直至得到每个物品的装入情况。

根据以上求解思路，结合具体实例详细介绍求解过程。

实例分析　有 $n=4$ 个物品，重量依次为(2,1,3,2)，价值依次为(12,10,20,15)，背包容量 $C=5$，如图 5-8 所示。

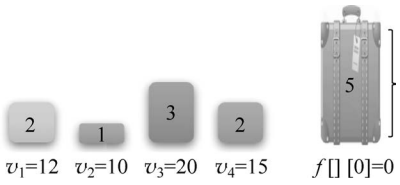


图 5-8　0-1 背包问题

(1) 初始化。首先构造一个二维数组 $f[n+1][C+1]$，$f[i][j]$存储可选物品为 $i,i+1,\cdots,n$，背包容量为 j 时可以得到的最大价值。初始化数组 $f[n+1][C+1]$，置 $f[i][0]=0$，表示当背包容量为 0 时，无法装入任何物品，对应的最大价值为 0。此时，数组 f 的取值如表 5-2 所示。

表 5-2　动态规划求解 0-1 背包问题：数组 f 的初始化

i	$f[i][j]$					
	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	0					
$i=2$	0					
$i=3$	0					
$i=4$	0					

(2) 计算 $f(4,j)$。当 $j<w_4$ 时，物品 4 不能装入，此时价值为 0；当 $j\geqslant w_4$ 时，物品 4 装入可以得到最大价值 v_4。此时，数组 f 的取值如表 5-3 中 $i=4$ 行所示，装包过程如图 5-9 所示。

表 5-3　动态规划求解 0-1 背包问题：数组 f 的结果

i	$f[i][j]$					
	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	0	10	15	25	30	37
$i=2$	0	10	15	25	30	35
$i=3$	0	0	15	20	20	35
$i=4$	0	0	15	15	15	15

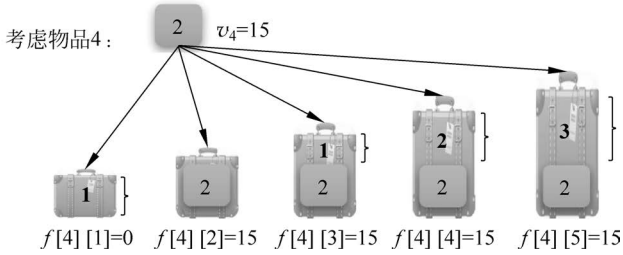


图 5-9　$i=4$ 时的背包情况

(3) 计算 $f(3,j)$。当 $j<w_3$ 时,物品 3 不能装入,此时价值为 $f(4,j)$;当 $j\geqslant w_3$ 时,比较装入物品 3 得到的价值 $v_3+f(4,j-w_3)$和不装入物品 3 得到的价值 $f(4,j)$,选择较大值。此时,数组 f 的取值如表 5-3 中 $i=3$ 行所示,装包过程如图 5-10 所示。

扫一扫 看彩图

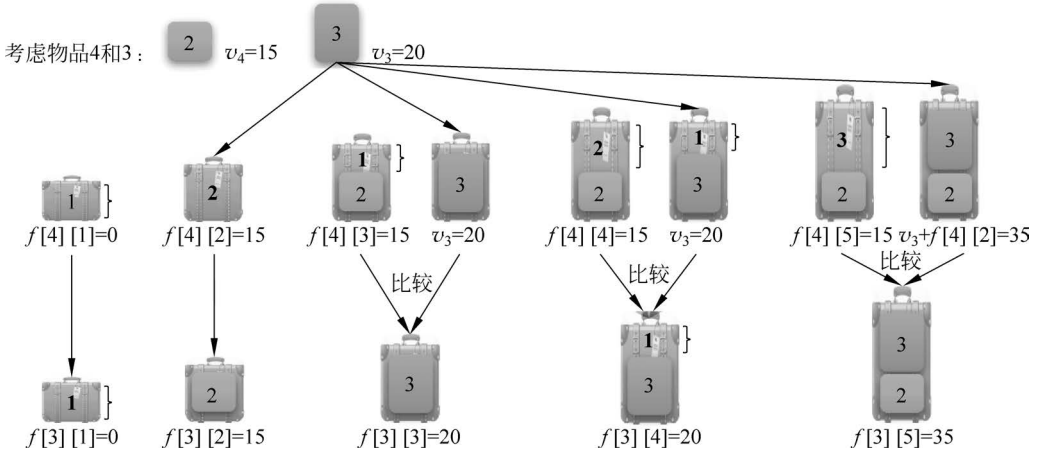


图 5-10 $i=3$ 时的背包情况

(4) 计算 $f(2,j)$。当 $j<w_2$ 时,物品 2 不能装入,此时价值为 $f(3,j)$;当 $j\geqslant w_2$ 时,比较装入物品 2 得到的价值 $v_2+f(3,j-w_2)$和不装入物品 2 得到的价值 $f(3,j)$,选择较大值。此时,数组 f 的取值如表 5-3 中 $i=2$ 行所示,装包过程如图 5-11 所示。

扫一扫 看彩图

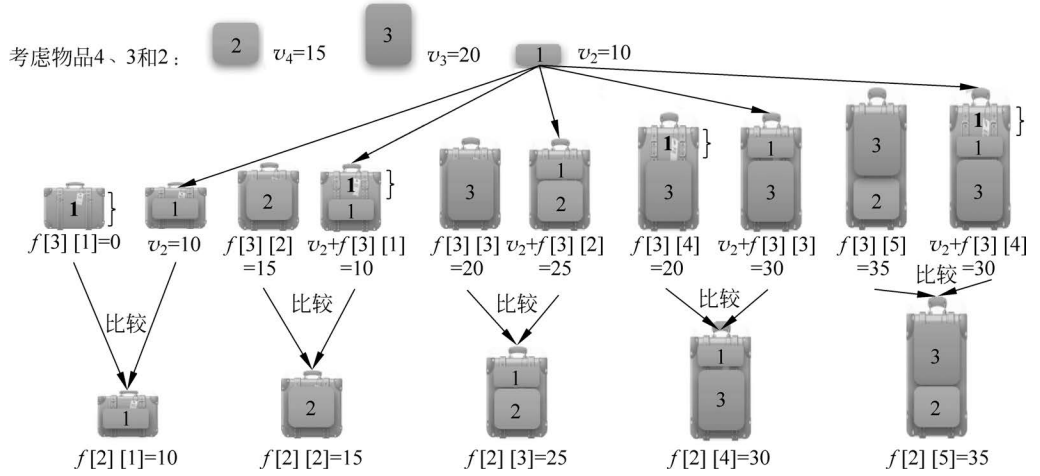


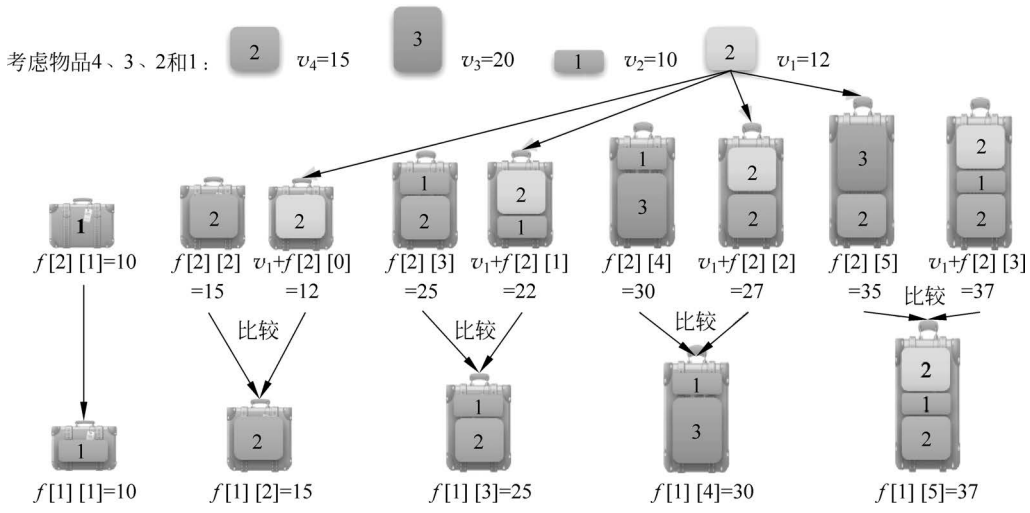
图 5-11 $i=2$ 时的背包情况

(5) 计算 $f(1,j)$。当 $j<w_1$ 时,物品 1 不能装入,此时价值为 $f(2,j)$;当 $j\geqslant w_1$ 时,比较装入物品 1 得到的价值 $v_1+f(2,j-w_1)$和不装入物品 1 得到的价值 $f(2,j)$,选择较大值。此时,数组 f 的取值如表 5-3 中 $i=1$ 行所示,装包过程如图 5-12 所示。

(6) 求最优解。

在表 5-3 中,由于 $f[1][5]\neq f[2][5]$,可知物品 1 装入背包,$x_1=1$,此时由于 $f[1][5]=v_1+f[2][5-w_1]=v_1+f[2][3]$,因此接下来判断 $f[2][3]$;

由于 $f[2][3]\neq f[3][3]$,可知物品 2 装入背包,$x_2=1$,此时由于 $f[2][3]=v_2+f[3][3-w_2]=v_2+f[3][2]$,因此接下来判断 $f[3][2]$;



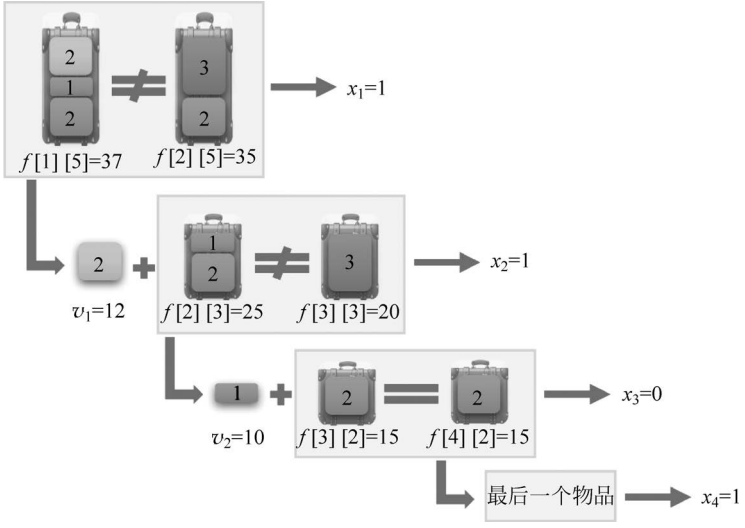
扫一扫　看彩图

图 5-12　$i=1$ 时的背包情况

由于 $f[3][2]=f[4][2]$，可知物品 3 未装入背包，$x_3=0$，继续判断 $f[4][2]$；

由于 $f[4][2]>0$，且已是最后一个物品，可知物品 4 装入背包，$x_4=1$。

最终，得到最优解为(1,1,0,1)，求解过程如图 5-13 所示。



扫一扫　看彩图

图 5-13　最优解的求解过程

算法实现　根据以上具体实例的计算过程，不难写出 0-1 背包问题的实现代码。主函数详见代码 5-7，两种实现方法(自底向上的求解和备忘录方法)详见代码 5-8 和代码 5-9。

代码 5-7：0-1 背包问题的数据读入、初始化和主函数

```
if __name__ == '__main__':
    M, n = list(map(int, input("输入背包容量和物品个数: ").split())) #n 为物品总数
                                                                      #M 为背包容量
    w = list(map(int, input("输入物品重量: ").split()))          #w 存储物品的重量
    v = list(map(int, input("输入物品价值: ").split()))          #v 存储物品的价值
    w.insert(0, None)                                            #舍弃 w[0]位置
    v.insert(0, None)                                            #舍弃 v[0]位置
```

```
    #f[i][j]表示已知物品 i,i+1,…,n 的情况下,背包容量为 j 对应的最大价值
    f = [[0 for j in range(M + 1)] for i in range(n + 1)]
    #x 记录最优解,x[i] = 1 表示物品 i 选择,否则表示不选
    x = [0 for i in range(n + 1)]

    knap_DP()                                          #动态规划求解最大价值
    print(f[1][M])

    for i in range(n + 1):
        for j in range(M + 1):
            print(f[i][j], end = '\t')
        print()
    knap_Demo(1, M)                                    #备忘录求解最大价值
    print(f[1][M])

    getResult()                                        #递推法求最优解
    for i in range(1, n + 1):
        print(x[i], end = ' ')
    print()
    getR_n(1, M)                                       #递归法求最优解
    for i in range(1, n + 1):
        print(x[i], end = ' ')
```

代码 5-8：自底向上求解 0-1 背包问题的最大价值

```
def knap_DP():                          #从最后一个物品开始考虑
    global n, M, w, v, f, x             #声明全局变量
    #初始化
    for i in range(n + 1):
        f[i][0] = 0
    for i in range(w[n], M + 1):
        f[n][i] = v[n]
    #动态规划,自底向上求解
    for i in range(n - 1, 0, -1):
        for j in range(1, M + 1):
            if j < w[i]:
                f[i][j] = f[i + 1][j]
            else:
                f[i][j] = max(f[i + 1][j - w[i]] + v[i], f[i + 1][j])
```

算法首先进行初始化，当背包容量为 0 时，对应价值为 0；接着考虑只有一个物品(第 n 个物品)的情况，当背包容量小于 $w[n]$时，背包中对应价值为 0，否则为第 n 个物品的价值 $v[n]$；后面的两层 for 循环中，依次从第 $n-1$ 个物品开始直到第 1 个物品，分别计算在背包容量为 $1\sim M$ 时可装入物品的最大价值；最后输出整个问题的最优值。很明显，算法时间主要耗费在两层 for 循环中，因此，最坏情况下的时间复杂度为 $O(nM)$。

代码 5-9：备忘录方法求解 0-1 背包问题的最大价值

```
#参数 k、c 分别表示物品的开始编号和背包容量
def knap_Demo(k, c):
    global n, M, w, v, f, x                    #声明全局变量
    if f[k][c] != 0:
        return f[k][c]
    temp = 0
```

```
    if k == n:
        if c < w[k]:
            f[k][c] = 0
        else:
            f[k][c] = v[k]
        return f[k][c]
    if c < w[k]:
        temp = knap_Demo(k + 1, c)
    else:
        temp = max(knap_Demo(k + 1, c), knap_Demo(k + 1, c - w[k]) + v[k])
    f[k][c] = temp
    return temp
```

在备忘录方法实现中，首先判断对应问题的结果是否已经计算完毕，由于 $f[k][c]$ 的初值为 0，如果 $f[k][c]\neq 0$，说明该问题已经求解，直接返回；接下来处理边界条件，把只有一个物品（第 n 个物品）时的价值存入 $f[k][c]$ 中；然后进行递归调用保存结果并返回。

递推方式求解最优解的实现详见代码 5-10。

代码 5-10：递推方式求解 0-1 背包问题的最优解

```
def getResult():
    global n, M, w, v, f, x                    #声明全局变量
    c = M
    for i in range(1, n):
        if f[i][c] == f[i + 1][c]:
            x[i] = 0
        else:
            x[i] = 1
            c = c - w[i]
    if f[n][c] == 0:
        x[n] = 0
    else:
        x[n] = 1
```

同样可以使用递归方法求解最优解，见代码 5-11。

代码 5-11：递归方式求解 0-1 背包问题的最优解

```
def getR_n(k, c):
    global n, M, w, v, f, x                     #声明全局变量
    if k == n:
        return
    if f[k][c] == f[k + 1][c]:
        x[k] = 0
    else:
        x[k] = 1
        c = c - w[k]
    getR_n(k + 1, c)
```

5.4.2　算法空间优化

扫一扫

视频讲解

上述实现的代码使用一个二维数组记录问题的解。如果 0-1 背包问题只要求得到最优值，不要求得到最优解，可以对空间进行优化。下面介绍两种优化方法。

1. 使用滚动数组

从图 5-7 中可以看出,在求解 $f(i,j)$时,只与它下面一行的 $f(i+1,j)$和 $f(i+1,j-w_i)$有关,而与其他行无关,因此,只需要使用两个一维数组的空间即可。如果使用两个一维数组 $A[M]$和 $B[M]$,用数组 A 存放已知值,利用数组 A 计算数组 B,那么就需要在计算完毕数组 B 后再把其复制到数组 A 中,非常麻烦。可以采用滚动数组,定义一个只有两行的二维数组 $f[2][M]$,初始时使用 $f[0][M]$,利用 $f[0][M]$求解 $f[1][M]$,下次利用 $f[1][M]$求解 $f[0][M]$,从而避免数组之间的多次复制。实现时,只需要借助一个控制变量 k(k 取 0 或 1),让 $f[k][M]$和 $f[1-k][M]$轮流使用,如此反复,像一个滚动的桶一样,具体实现见代码 5-12。

代码 5-12:采用滚动数组求解 0-1 背包问题的最优解

```
def knap_DP_Roll():
    global n, M, w, v                                          #定义同代码 5-7
    k = 0                                                      #控制滚动的变量
    f = [[0 for i in range(M + 1)] for j in range(2)]          #定义滚动数组
    #初始化
    for i in range(w[n], M+1):
        f[0][i] = v[n]

    #动态规划,自底向上求解
    for i in range(n - 1, 0, -1):
        for j in range(M+1):
            if j < w[i]:                                       #滚动更新数组
                f[1 - k][j] = f[k][j]
            else:
                f[1 - k][j] = max(f[k][j - w[i]] + v[i], f[k][j])
        k = 1 - k                                              #每更新完一行,更新 k
    print(f[k][M])

if __name__ == '__main__':
    M, n = list(map(int, input("输入背包容量和物品个数: ").split()))   #n 为物品总数
                                                                        #M 为背包容量
    w = list(map(int, input("输入物品重量: ").split()))        #w 存储物品的重量
    v = list(map(int, input("输入物品价值: ").split()))        #v 存储物品的价值
    w.insert(0, None)                                          #舍弃 w[0]位置
    v.insert(0, None)                                          #舍弃 v[0]位置

    knap_DP_Roll()
```

优化后,空间需求从之前的 nM 降低为 $2M$。进一步,可以继续优化空间,使用大小为 M 的空间即可。

2. 使用一维数组

由图 5-7 可知 $f(i,j)$与 $f(i+1,j)$、$f(i+1,j-w_i)$有关,在使用一维数组时,需要自右向左计算(j 从 M 开始递减),此时 $f(i+1,j)$即是 $f(i,j)$,$f(i+1,j-w_i)$即是 $f(i,j-w_i)$,通过 $f(i,j)=\max(f(i,j),f(i,j-w_i)+v_i)$可以得到 $f(i,j)$,具体实现见代码 5-13。这里有一个问题,必须要自右向左计算吗?请读者思考。

代码 5-13：使用一维数组求解 0-1 背包问题的最优解

```
def knap_DP_Single():
    global n, M, w, v #全局变量,同代码 5-7
    f = [0 for i in range(M+1)]                    #定义一维数组
    for j in range(w[n], M+1):                     #初始化
        f[j] = v[n]
    for i in range(n-1, 0, -1):
        for j in range(M, w[i]-1, -1):             #自右向左求解
            f[j] = max(f[j-w[i]] + v[i], f[j])
    print(f[M])

if __name__ == '__main__':
    M, n = list(map(int, input("输入背包容量和物品个数: ").split()))   #n 为物品总数
                                                                        #M 为背包容量
    w = list(map(int, input("输入物品重量: ").split()))        #w 存储物品的重量
    v = list(map(int, input("输入物品价值: ").split()))        #v 存储物品的价值
    w.insert(0, None)                                          #舍弃 w[0]位置
    v.insert(0, None)                                          #舍弃 v[0]位置

    knap_DP_Single()
```

应用扩展　某同学为参加 ACM 程序设计竞赛准备购买一批书籍，预算只有 500 元，他为不同书籍的重要性进行了评级(1～5，5 表示最重要)。在预算有限的情况下，如何选择才能使购买的所有书籍总的重要性最大？

显然，这是一个 0-1 背包问题，预算对应背包容量，书籍价格对应物品重量，书籍重要性对应物品价值。经过转换后，该问题可以直接使用 0-1 背包问题的解题方法求解。类似的问题还有很多，学习时，需要重点掌握 0-1 背包问题的一般特征，继而灵活应用到其他具体问题中。

本节学习了动态规划的典型应用——0-1 背包问题。它代表了一类问题，具有以下特征：已知 n 个可选择的物品，每个物品有若干属性，如重量和价值，有一个容器(如背包)以及其某个方面的限制(如容量)，问题是：在容器限制下，如何挑选物品(每个物品要么选，要么不选)才能使另一个属性值之和达到最大值？

在用动态规划求解 0-1 背包问题时，首先分析最优子结构性质，列出最优值的递归关系，然后采用自底向上或备忘录方法进行求解。整个求解过程与动态规划求解问题的 4 个步骤一致。

引申：更多类型的背包问题

除了 0-1 背包问题，还有完全背包、多重背包、满背包问题等。

完全背包问题与 0-1 背包问题相似，不同的是每种物品有无限件，也就是每种物品不是只有取和不取两种可能，而是可以取任意件，直至超出背包容量为止。

多重背包问题中，第 i 种物品最多有 M_i 件可用，即可以取 0～M_i 件。

满背包问题是求解选择哪些物品刚好可以装满背包且价值最大。

以上问题可采用与 0-1 背包问题相似的求解思路进行求解。更多的背包问题还有分组背包问题、有依赖的背包问题、二维费用的背包问题等，有兴趣的读者可以查阅相关资料进一步学习。

5.5 动态规划应用：矩阵连乘问题

扫一扫

视频讲解

扫一扫

视频讲解

问题描述 给定 n 个矩阵$\{\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_n\}$，其中 $\boldsymbol{A}_i$ 与 $\boldsymbol{A}_{i+1}$ 可乘，$i=1,2,\cdots,n-1$。如何确定计算矩阵连乘的计算次序，使得依此次序计算这 n 个矩阵连乘需要的乘法次数最少？

为了更好地理解该问题，这里对矩阵连乘进行两点说明。

(1) 两个矩阵可乘的条件是前一个矩阵的列等于后一个矩阵的行，如图 5-14 所示，矩阵 $\boldsymbol{A}$ 为 p 行 q 列，矩阵 $\boldsymbol{B}$ 为 q 行 j 列，因此，矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$ 是可乘的。

(2) 矩阵 $\boldsymbol{A}$ 与矩阵 $\boldsymbol{B}$ 相乘需要加法操作和乘法操作。由于相乘得到的矩阵中有 $p\times j$ 个元素，其中每个元素是由矩阵 $\boldsymbol{A}$ 的每行的 q 个元素与矩阵 $\boldsymbol{B}$ 每列的 q 个元素相乘然后相加得到，需要 q 次乘法和 $q-1$ 次加法，因此两个矩阵相乘共需要执行 pqj 次乘法和 $p(q-1)j$ 次加法，加法和乘法次数相当，可以用乘法次数反映矩阵乘问题的复杂度。

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1q} \\ a_{21} & a_{22} & \cdots & a_{2q} \\ \vdots & \vdots & & \vdots \\ a_{p1} & a_{p2} & \cdots & a_{pq} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1j} \\ b_{21} & b_{22} & \cdots & b_{2j} \\ \vdots & \vdots & & \vdots \\ b_{q1} & b_{q2} & \cdots & b_{qj} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1j} \\ c_{21} & c_{22} & \cdots & c_{2j} \\ \vdots & \vdots & & \vdots \\ c_{p1} & c_{p2} & \cdots & c_{pj} \end{bmatrix}$$

图 5-14 矩阵 $\boldsymbol{A}$ 与矩阵 $\boldsymbol{B}$ 相乘

n 个矩阵连乘，不同的计算次序所需要的矩阵乘法的次数是不一样的。例如，有 4 个矩阵 $\boldsymbol{A}_1$、$\boldsymbol{A}_2$、$\boldsymbol{A}_3$ 和 $\boldsymbol{A}_4$，$\boldsymbol{A}_1$ 是 50 行 10 列，$\boldsymbol{A}_2$ 是 10 行 40 列，$\boldsymbol{A}_3$ 是 40 行 30 列，$\boldsymbol{A}_4$ 是 30 行 5 列。这 4 个矩阵共有 5 种不同的计算次序，根据前面分析的结果，矩阵 $\boldsymbol{A}_{p\times q}$ 与矩阵 $\boldsymbol{B}_{q\times j}$ 相乘需要 pqj 次乘法，可得到每种计算次序对应的乘法次数如下。

(1) $(\boldsymbol{A}_1(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4)))$：$40\times30\times5+10\times40\times5+50\times10\times5=10500$ 次。

(2) $(\boldsymbol{A}_1((\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4))$：$10\times40\times30+10\times30\times5+50\times10\times5=16000$ 次。

(3) $((\boldsymbol{A}_1\boldsymbol{A}_2)(\boldsymbol{A}_3\boldsymbol{A}_4))$：$50\times10\times40+40\times30\times5+50\times40\times5=36000$ 次。

(4) $((\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))\boldsymbol{A}_4)$：$10\times40\times30+50\times10\times30+50\times30\times5=34500$ 次。

(5) $(((\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3)\boldsymbol{A}_4)$：$50\times10\times40+50\times40\times30+50\times30\times5=87500$ 次。

可以看出，5 种计算次序虽然得到的结果矩阵是相同的，但所花费的乘法次数有很大区别。其中，计算次序(1)是乘法次数最少的计算方案，求解效率最高。矩阵连乘问题就是寻找所需乘法次数最少的计算方案。

问题分析 先考虑穷举法的求解方法。计算所有可能方案的乘法次数，再从中选择一种最优的方案。假设用 $P(n)$ 表示 n 个矩阵相乘的计算方案总数，由于最后一次是两个矩阵相乘，而这两个矩阵分别来自左边连续 k 个矩阵的连乘结果和右边连续 $n-k$ 个矩阵的连乘结果，因此，$P(n)$ 的递推关系为

$$P(n)=\begin{cases}1, & n=1\\ \displaystyle\sum_{k=1}^{n-1}P(k)P(n-k), & n>1\end{cases} \tag{5-7}$$

利用卡特兰数和斯特林公式，求解式(5-7)可以得到 $P(n)=\Omega(4^n/n^{3/2})$。可见，采用穷举法求解矩阵连乘问题的复杂度很高，有必要寻找更好的算法。

引申：卡特兰数和斯特林公式

卡特兰数(Catalan Number)又称为卡塔兰数、明安图数，是组合数学中一种常出现于各种计数问题中的数列。设$h(n)$为卡特兰数的第n项，令$h(0)=1$，$h(1)=1$，卡特兰数满足

$$h(n)=h(0)h(n-1)+h(1)h(n-2)+\cdots+h(n-1)h(0),n\geqslant 2 \tag{5-8}$$

式(5-8)的解为

$$h(n)=\frac{C_{2n}^{n}}{n+1},\quad n=0,1,2,\cdots \tag{5-9}$$

卡特兰数可以用于求解n对括号的正确匹配数目、n个节点构成的二叉搜索树总数、n个数的出栈序列总数、凸多边形三角划分等问题。

斯特林公式(Stirling's Approximation)是一个用来取n的阶乘的近似值的数学公式，即

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n} \tag{5-10}$$

斯特林公式在理论和应用上都具有重要的价值，对于概率论的发展也有着重大的意义。

从上述分析可以看出，n个矩阵的连乘问题不管用什么计算次序，最后一次都是两个矩阵相乘，而这两个矩阵分别是n个矩阵中左边若干矩阵连乘的结果和右边若干矩阵连乘的结果，也就是说，矩阵连乘问题可以通过两个规模更小的矩阵连乘问题来求解，即问题和子问题存在一定的关系，那么矩阵连乘问题是否符合最优子结构性质呢？

在前面的实例中，$(\boldsymbol{A}_1(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4)))$是4个矩阵连乘问题的最优解，那么$(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4))$是否是$\boldsymbol{A}_2$、$\boldsymbol{A}_3$、$\boldsymbol{A}_4$这3个矩阵连乘问题的最优解呢？可以用反证法证明。

假设$(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4))$不是$\boldsymbol{A}_2$、$\boldsymbol{A}_3$、$\boldsymbol{A}_4$这3个矩阵连乘问题的最优解，那么一定存在一个最优的计算次序，如$((\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4)$，得到$(\boldsymbol{A}_1((\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4))$对应的乘法次数一定比$(\boldsymbol{A}_1(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4)))$对应的乘法次数要少(原因是两种计算次序花费的乘法次数的差别仅在$\boldsymbol{A}_2$、$\boldsymbol{A}_3$、$\boldsymbol{A}_4$这3个矩阵连乘问题中，其他方面都是相同的)，继而得到$(\boldsymbol{A}_1((\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4))$是最优解，与已知条件$(\boldsymbol{A}_1(\boldsymbol{A}_2(\boldsymbol{A}_3\boldsymbol{A}_4)))$是$\boldsymbol{A}_1\sim\boldsymbol{A}_4$这4个矩阵连乘问题的最优解相矛盾。同样的方法，对于n个矩阵连乘问题，也可以通过反证法证明矩阵连乘问题满足最优子结构性质。

此外，矩阵连乘问题具有重叠子问题性质。例如实例中，$\boldsymbol{A}_2$、$\boldsymbol{A}_3$这两个矩阵的连乘问题在计算次序(2)和(4)中都要使用到。因此，矩阵连乘问题具有最优子结构和重叠子问题两个性质，可以用动态规划求解。

动态规划求解矩阵连乘问题的步骤如下。

步骤1　分析问题和子问题的关系，找出最优解的性质，并刻画其结构特征。

矩阵连乘问题的最少乘法次数与矩阵个数、每个矩阵的行列大小以及矩阵乘法的计算次序有关，将从$\boldsymbol{A}_i$开始到$\boldsymbol{A}_j$结束的若干矩阵连乘问题$\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j$的结果矩阵简记为$\boldsymbol{A}[i:j]$，$i\leqslant j$，最优解即为求$\boldsymbol{A}[i:j]$的最优计算次序。设这个计算次序中最后一次是在矩阵$\boldsymbol{A}_k$和$\boldsymbol{A}_{k+1}$之间将矩阵链断开，其中$i\leqslant k<j$，则其对应的计算次序用完全加括号方式可表示为$(\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}\boldsymbol{A}_{k+2}\cdots\boldsymbol{A}_j)$。因此，$\boldsymbol{A}[i:j]$的计算量$=\boldsymbol{A}[i:k]$的计算量$+\boldsymbol{A}[k+1:j]$的计算量$+\boldsymbol{A}[i:k]\times\boldsymbol{A}[k+1:j]$的计算量。

上述等式中,左边 $\boldsymbol{A}[i:j]$是一个规模较大(矩阵个数为 $j-i+1$)的矩阵连乘问题,右边是两个规模较小(矩阵个数分别为 $k-i+1$、$j-k$)的矩阵连乘问题。由于最优解要求乘法次数最少的计算次序,且断开位置 k 有多种可能,因此 $\boldsymbol{A}[i:j]$的最少乘法次数是在 k 的所有可能中取乘法次数最少的。根据矩阵连乘问题的最优子结构性质,可以得到 $\boldsymbol{A}[i:j]$的最少乘法次数$=\min\{\boldsymbol{A}[i:k]$的最少乘法次数$+\boldsymbol{A}[k+1:j]$的最少乘法次数$+\boldsymbol{A}[i:k]\times\boldsymbol{A}[k+1:j]$的乘法次数,$i\leqslant k<j\}$。

步骤 2 递归地定义最优值。

定义 $m(i,j)$为计算 $\boldsymbol{A}[i:j]$所需要的最少乘法次数,则原问题的最优值可表示为 $m(1,n)$。最小规模的矩阵连乘问题可以看作只有一个矩阵的情况,此时不需要做乘法,$m(i,i)=0$;当 $i<j$ 时,根据步骤 1 的分析,有

$$m(i,j)=\min_{i\leqslant k<j}\{m(i,k)+m(k+1,j)+p_{i-1}p_kp_j\} \tag{5-11}$$

其中,$\boldsymbol{A}_i$ 的行和列分别为 p_{i-1} 和 p_i。

综合以上两种情况,得到 $m(i,j)$的递归定义为

$$m(i,j)=\begin{cases}0, & i=j\\ \min\limits_{i\leqslant k<j}\{m(i,k)+m(k+1,j)+p_{i-1}p_kp_j\}, & i<j\end{cases} \tag{5-12}$$

步骤 3 以自底向上的方式计算出最优值。

根据最优值的递归关系,先求解规模较小的矩阵连乘问题,再求解规模较大的矩阵连乘问题,因此,按照自底向上的方式,依次求解长度为 $1,2,\cdots,n$ 的矩阵连乘问题的最优值。

步骤 4 根据计算最优值时得到的信息,构造最优解。

$m(i,j)$得到的是最优值,如果想获得最优解(最优值对应的计算次序),需要记录 $m(i,j)$获得最优值时的断开位置 k。为此,定义一个二维数组 $s(i,j)$记录 k。求解最优解从 $s(1,n)$开始,找到最后一次断开位置 k,再依次通过 $s(1,k)$和 $s(k,n)$继续寻找断开位置,递归下去,直到只剩下一个矩阵,即可得到最优解。

实例分析 假设共有 6 个矩阵 $\boldsymbol{A}_1$、$\boldsymbol{A}_2$、$\boldsymbol{A}_3$、$\boldsymbol{A}_4$、$\boldsymbol{A}_5$、$\boldsymbol{A}_6$,矩阵规模如图 5-15 所示,对应的行列规模如表 5-4 所示。由于矩阵连乘要求前一个矩阵的列数等于后一个矩阵的行数,因此,6 个矩阵的行列值只需用 7 个数值表示,并存放在数组 P 中,其中,矩阵 $\boldsymbol{A}_i$ 的行、列值存储在 P_{i-1}、P_i 中。数组 P 如表 5-5 所示。

扫一扫

看彩图

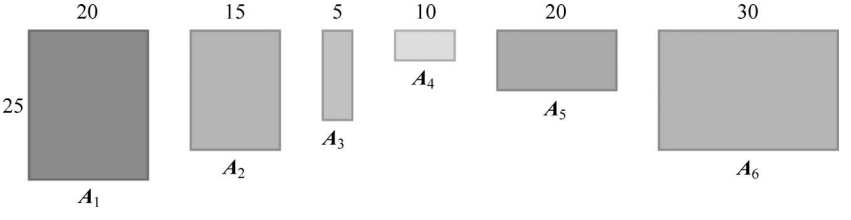


图 5-15 矩阵规模示意图

表 5-4 矩阵的行列信息

$\boldsymbol{A}_1$	$\boldsymbol{A}_2$	$\boldsymbol{A}_3$	$\boldsymbol{A}_4$	$\boldsymbol{A}_5$	$\boldsymbol{A}_6$
25×20	20×15	15×5	5×10	10×20	20×30

表 5-5　矩阵的行列信息存储在一维数组中

下标 i	0	1	2	3	4	5	6
$P[i]$	25	20	15	5	10	20	30

定义二维数组 $m[n+1][n+1]$，$m[i][j]$存储从 $\boldsymbol{A}_i$ 到 $\boldsymbol{A}_j$ 的若干连续矩阵连乘问题的最少乘法次数；定义二维数组 $s[n+1][n+1]$，$s[i][j]$存储 $m[i][j]$对应的断开位置。

(1) 初始化 $m[i][i]=0$，$i=1,2,\cdots,6$，$s[i][i]=0$(表示无须断开)。此时，数组 m 和 s 的内容如表 5-6 所示。

表 5-6　矩阵连乘实例：初始化

i	$m[i][j]$						$s[i][j]$					
	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
$i=1$	0						0					
$i=2$		0						0				
$i=3$			0						0			
$i=4$				0						0		
$i=5$					0						0	
$i=6$						0						0

(2) 计算相邻两个矩阵的最小乘法次数，放入 $m[1][2]$、$m[2][3]$、$m[3][4]$、$m[4][5]$和 $m[5][6]$。对应数组 m、数组 s 的内容如表 5-7 所示，计算过程如图 5-16 所示。

表 5-7　矩阵连乘实例：计算数组 m 和 s

i	$m[i][j]$						$s[i][j]$					
	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
$i=1$	0	7500	4000	5250	7500	11750	0	1	1	3	3	3
$i=2$		0	1500	2500	4500	8500		0	2	3	3	3
$i=3$			0	750	2500	6250			0	3	3	3
$i=4$				0	1000	4000				0	4	5
$i=5$					0	6000					0	5
$i=6$						0						0

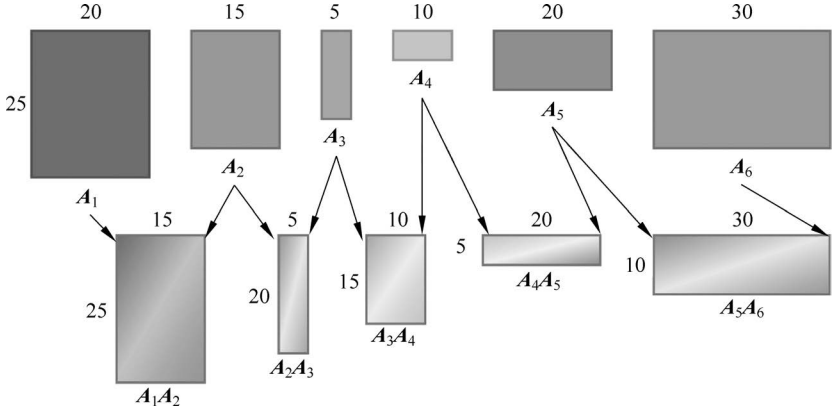
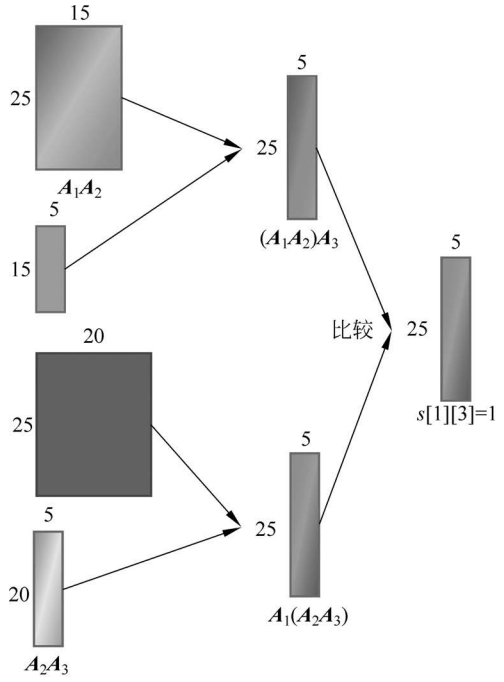


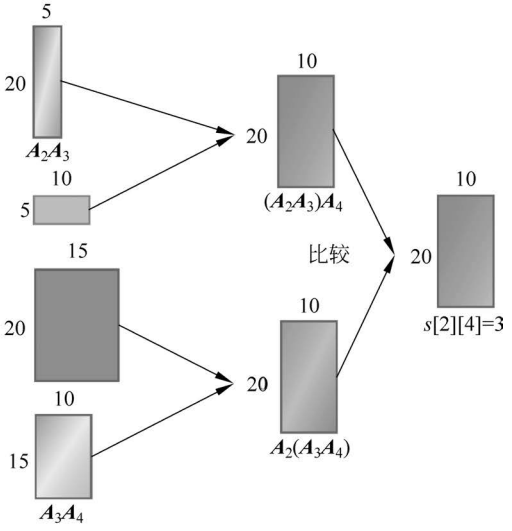
图 5-16　相邻两个矩阵相乘示意图

扫一扫
看彩图

(3) 计算相邻 3 个矩阵的最小乘法次数,结果记录在 $m[1][3]$、$m[2][4]$、$m[3][5]$和 $m[4][6]$中,如表 5-7 所示。计算过程如图 5-17 所示。以 $m[3][5]$的计算为例,$m[3][5]=\min\{m[3][3]+m[4][5]+P_2P_3P_5, m[3][4]+m[5][5]+P_2P_4P_5\}=2500$,$s[3][5]=3$。



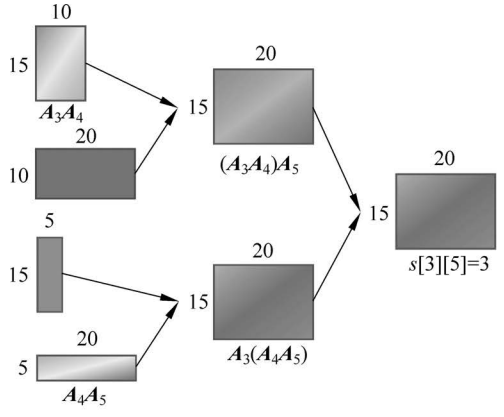
(a) 计算$m[1][3]$



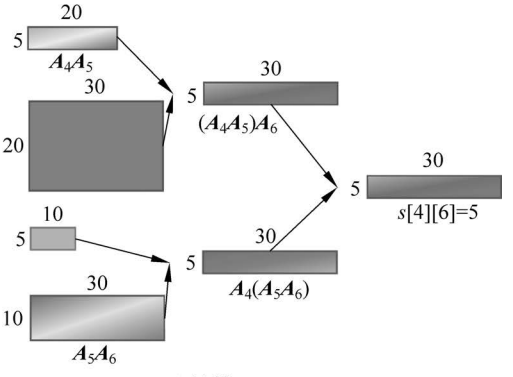
(b) 计算$m[2][4]$

扫一扫

看彩图



(c) 计算$m[3][5]$

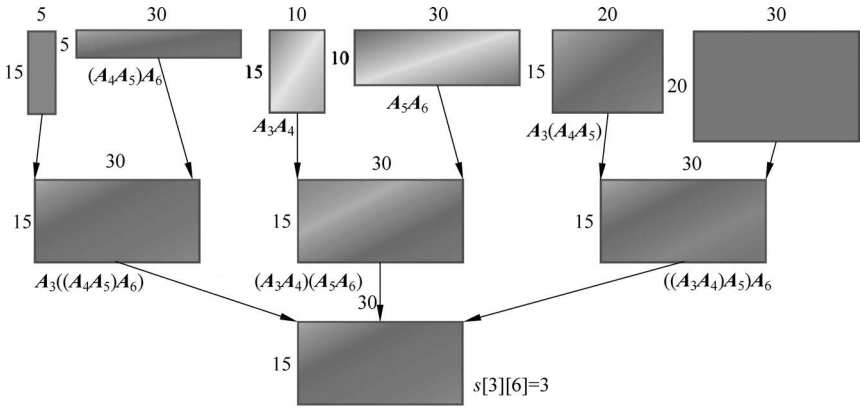


(d) 计算$m[4][6]$

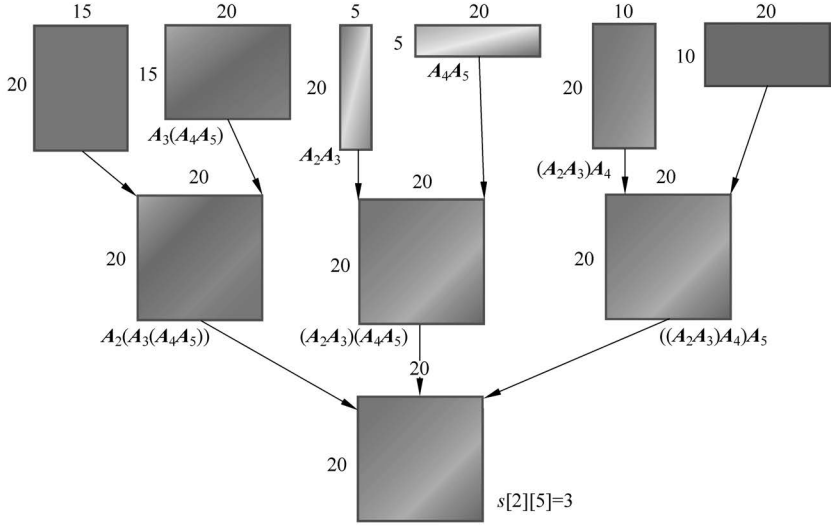
图 5-17 相邻 3 个矩阵相乘示意图

(4) 计算相邻 4 个矩阵的最小乘法次数,结果记录在 $m[1][4]$、$m[2][5]$和 $m[3][6]$中,如表 5-7 所示。计算过程如图 5-18 所示。以 $m[3][6]$的计算为例,$m[3][6]=\min\{m[3][3]+m[4][6]+P_2P_3P_6, m[3][4]+m[5][6]+P_2P_4P_6, m[3][5]+m[6][6]+P_2P_5P_6\}=6250$,$s[3][6]=3$。

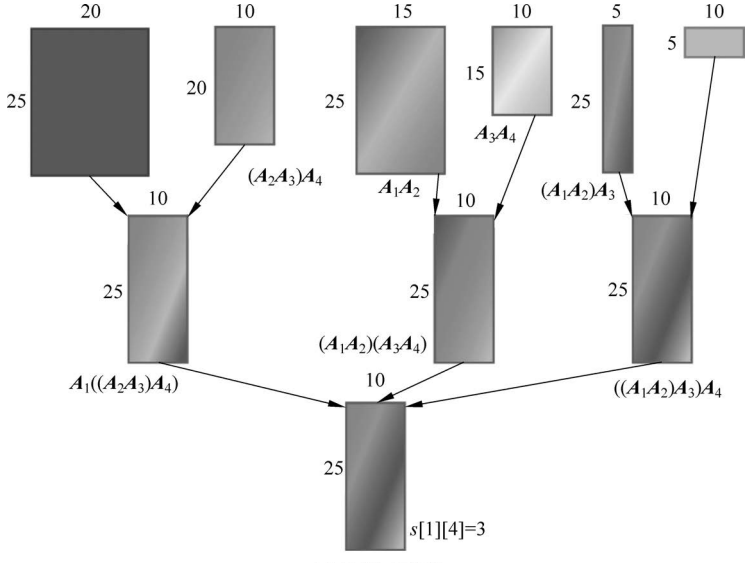
(5) 计算相邻 5 个矩阵的最小乘法次数,结果记录在 $m[1][5]$和 $m[2][6]$中,如表 5-7 所示。计算过程如图 5-19 所示。以 $m[2][6]$的计算为例,$m[2][6]=\min\{m[2][2]+m[3][6]+P_1P_2P_6, m[2][3]+m[4][6]+P_1P_3P_6, m[2][4]+m[5][6]+P_1P_4P_6, m[2][5]+m[6][6]+P_1P_5P_6\}=8500$,$s[2][6]=3$。



(a) 计算$m[3][6]$



(b) 计算$m[2][5]$

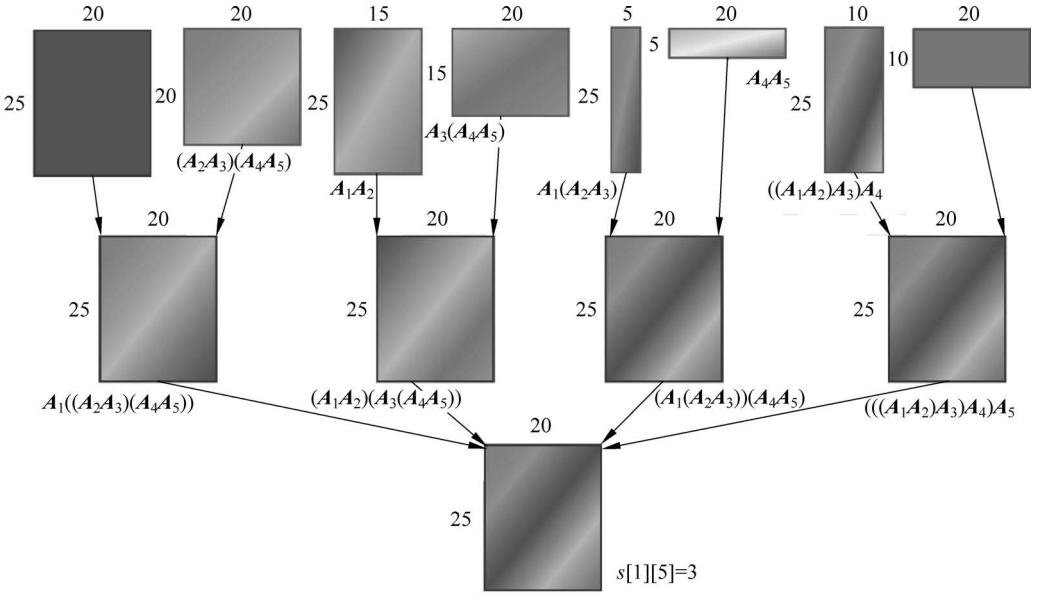


(c) 计算$m[1][4]$

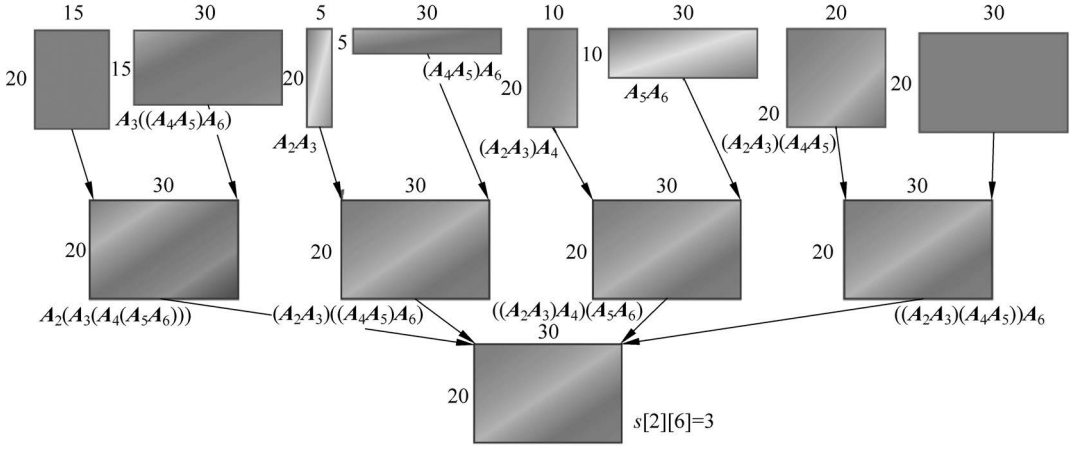
图 5-18　相邻 4 个矩阵相乘示意图

扫一扫

看彩图



(a) 计算$m[1][5]$



(b) 计算$m[2][6]$

图 5-19 相邻 5 个矩阵相乘示意图

扫一扫

看彩图

(6) 计算相邻 6 个矩阵的最小乘法次数,记录在 $m[1][6]$中,此结果即为所求问题的最优值,计算过程如图 5-20 所示。$m[1][6]=\min\{m[1][1]+m[2][6]+P_0P_1P_6, m[1][2]+m[3][6]+P_0P_2P_6, m[1][3]+m[4][6]+P_0P_3P_6, m[1][4]+m[5][6]+P_0P_4P_6, m[1][5]+m[6][6]+P_0P_5P_6\}=11750, s[1][6]=3$。

(7) 计算最优解。由于 $s[1][6]=3$,可知最后一次断开位置在 $\boldsymbol{A}_3$ 处,即$((\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3)(\boldsymbol{A}_4\boldsymbol{A}_5\boldsymbol{A}_6))$,$(\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3)$的计算次序根据 $s[1][3]=1$,得$(\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))$,$(\boldsymbol{A}_4\boldsymbol{A}_5\boldsymbol{A}_6)$的计算次序根据 $s[4][6]=5$,得$((\boldsymbol{A}_4\boldsymbol{A}_5)\boldsymbol{A}_6)$,因此该实例的最优解用完全加括号形式可表示为$((\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))((\boldsymbol{A}_4\boldsymbol{A}_5)\boldsymbol{A}_6))$。

从该实例的具体求解过程可以看到,求解过程是沿着 m 和 s 数组的对角线方向依次向上进行,直至得到最终解,如图 5-21 所示。

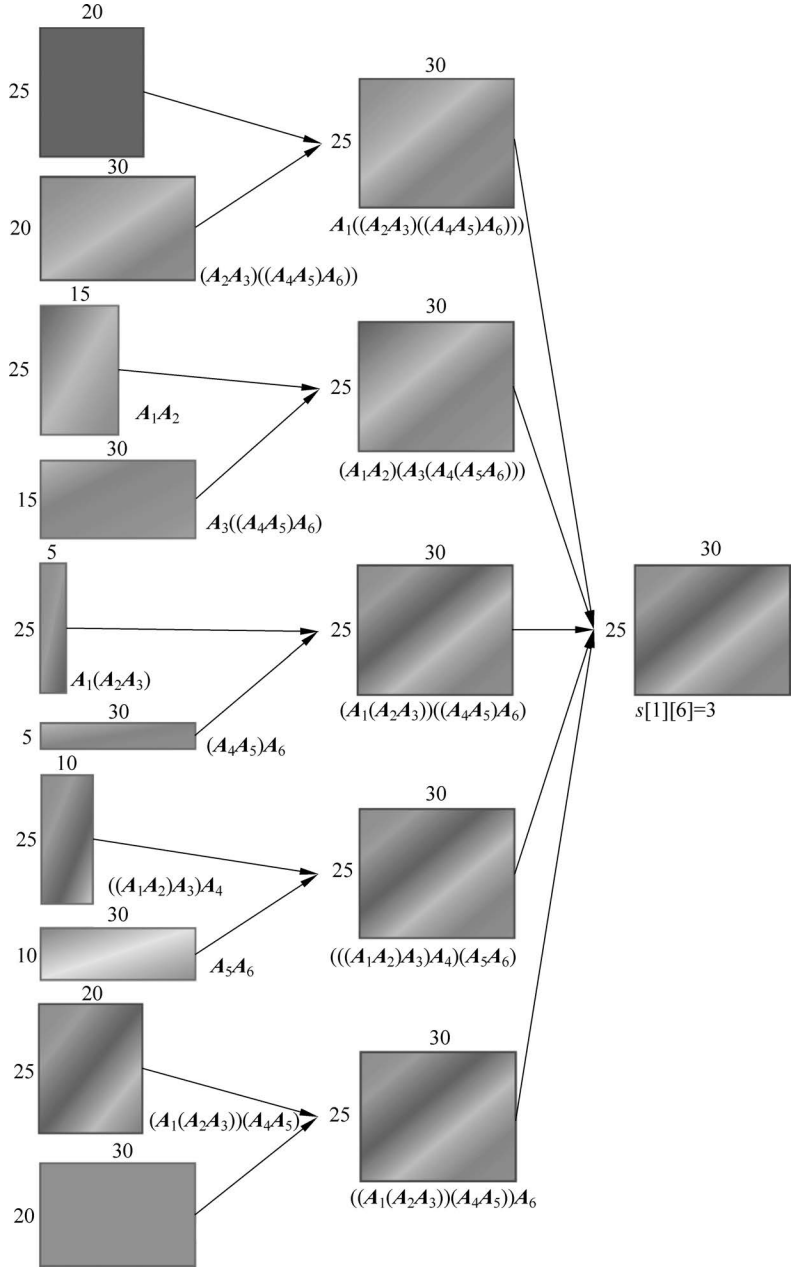


图 5-20　相邻 6 个矩阵相乘示意图

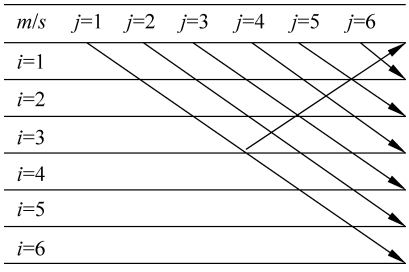


图 5-21　矩阵连乘问题中问题的求解次序

扫一扫

看彩图

算法实现 根据前面的分析和具体实例的求解过程,给出动态规划求解矩阵连乘问题的代码实现。数据结构定义、数据读入和主函数见代码 5-14。

代码 5-14:数据读入和主函数

```
import sys

if __name__ == '__main__':
    n = int(input())                          #n 表示矩阵的个数
    p = list(map(int, input().split()))       #p[i-1]和 p[i]存储第 i 个矩阵的行和列
    #m[i][j]存储 Ai…Aj 矩阵连乘的最小乘法次数,初始化 m 的元素值为最大值
    m = [[sys.maxsize for i in range(n + 1)] for j in range(n + 1)]
    #s[i][j]存储 m[i][j]对应的矩阵断开位置
    s = [[0 for i in range(n + 1)] for j in range(n + 1)]

    MatrixMultiply(p, n, m, s)
    #print(s)
    traceback(s, 1, n)
```

以自底向上的方式求解最优值。从长度 $r=2$ 的矩阵连乘问题开始,依次计算长度 r 为 $3,4,\cdots,n$ 的矩阵连乘问题。对于长度为 r 的矩阵连乘问题有多个,根据起点 i 和长度 r,可以得到终点 j;确定起点 i 和终点 j 后,枚举所有可能的断点位置 $k=i,i+1,\cdots,j-1$,得到最小的乘法次数,存入数组 $m[i][j]$,同时对应的断开位置 k 存入数组 $s[i][j]$。具体实现见代码 5-15。

代码 5-15:计算最优值并记录断开位置

```
def MatrixMultiply(p, n, m, s):
    '''
    p[i]和 p[i-1]存储第 i 个矩阵的行和列
    n 表示矩阵的个数
    m[i][j]存储 Ai…Aj 矩阵连乘的最少乘法次数
    s[i][j]存储 m[i][j]对应的矩阵断开位置
    '''
    for i in range(1, n + 1):                 #初始化长度为 1 的矩阵
        m[i][i] = 0
    for r in range(2, n + 1):                 #矩阵连乘问题规模从 2 到 n
        for i in range(1, n - r + 2):         #i 表示规模为 r 的矩阵连乘问题的起点
            j = i + r - 1                     #j 表示规模为 r 的矩阵连乘问题的终点
            #依次计算断开位置在 i,i+1,i+2,…,k 处的最少乘法次数,并记录断开位置
            for k in range(i, j):
                #断开位置在 i 处的乘法次数
                t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]
                if t < m[i][j]:               #找到较小乘法次数,更新最优值和断开位置
                    m[i][j] = t
                    s[i][j] = k

    print(m[1][n])
```

求解最优解。假设最优解用完全加括号形式表示,根据前面的示例,从最优值记录的断开位置 $k=s[i][j]$ 开始,递归求解其左边 $s[i][k]$ 和右边 $s[k+1][j]$,直到 $i=j$ 时为止。具体实现见代码 5-16。

代码 5-16：构造最优解

```
def traceback(s, i, j):
    if i == j:
        print('A{}'.format(i), end = '')
        return 0
    print('(', end = '')
    traceback(s, i, s[i][j])
    traceback(s, s[i][j] + 1, j)
    print(')', end = '')
```

时间复杂度分析　使用动态规划求解时，算法的主要计算量在于算法中对 r、i 和 k 进行操作的三重循环。循环体内的计算量为 $O(1)$，而三重循环的总次数为 $O(n^3)$。因此，算法的计算时间上界为 $O(n^3)$。相比穷举法需要指数时间复杂度，算法复杂度降低为多项式时间复杂度。算法所占用的空间是存储最优值和最优解所需要的二维数组，空间复杂度为 $O(n^2)$。

应用扩展　矩阵连乘问题是动态规划的典型应用，类似的问题还有石子合并问题。

石子合并问题：有 n 堆石子排成一排，编号依次为 $1\sim n$，每堆石子由若干石子构成。现要将石子有次序地合并为一堆。规定每次只能选相邻的两堆石子合并成新的一堆，并将合并后的石子数记为本次合并的得分。试设计一个算法，计算将 n 堆石子合并成一堆的最大得分。

分析：要求每次只能选相邻的两堆石子合并，因此最后一次合并是左边若干堆石子合并后与右边若干堆石子合并后的结果进行合并，与矩阵连乘问题类似。进一步分析可知，石子等价于矩阵，石子合并的方案等价于矩阵连乘的方案，石子合并的得分等价于矩阵相乘需要的乘法次数。因此，石子合并问题的求解方法与矩阵连乘问题求解方法一样。假设第 i 堆石子数量用 $p[i]$ 表示，$\mathrm{sum}(i,j)$ 表示第 i 堆石子到第 j 堆石子的石子数量和，$m(i,j)$ 表示从第 i 堆石子到第 j 堆石子合并成一堆的最大得分，则有

$$m(i,j)=\begin{cases}0, & i=j\\ \max\limits_{i\leqslant k<j}\{m(i,k)+m(k+1,j)+\mathrm{sum}(i,j)\}, & i<j\end{cases}$$

类似地，使用同样的方法也可以求 n 堆石子合并的最小得分。

矩阵连乘问题代表了一类问题：存在一个分割点把问题分为两部分，问题的最优解可以通过这两部分的最优解得到，解决方法是枚举分割点，利用最优子结构性质，运用动态规划方法求解。

以自底向上方式求解的代码通常包括：

(1) 问题规模从小到大；

(2) 枚举问题的起点，算出终点；

(3) 枚举问题的分割点；

(4) 利用最优值的递归关系求解最优值和最优解。

扫一扫

视频讲解

5.6　动态规划应用：最长公共子序列

扫一扫

视频讲解

首先介绍一些相关的基本概念。

(1) 序列：由若干字符构成的字符串或文本。

(2) 子序列：序列删去若干字符后得到的序列，子序列与序列中字符的相对次序不变。

(3) 公共子序列：给定两个序列 X 和 Y，当另一序列 Z 既是 X 的子序列，又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列。

(4) 最长公共子序列(Longest Common Subsequence，LCS)：公共子序列中长度最长的子序列。

问题描述 给定两个序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$，找出 X 和 Y 的一个最长公共子序列。

例如，有两个序列，$X=$(B,D,A,B,A,C)，$Y=$(B,A,D,B,C,D,A)，两者的公共子序列有很多，如(B,D,B,A)、(B,D,B,C)、(B,A,C)等，其中(B,D,B,A)、(B,D,B,C)是最长公共子序列，长度为 4。从这个例子也可以看出，最长公共子序列可能有多个，但其长度是唯一的。

显然，最长公共子序列反映了两个序列的相似程度，这个问题可以广泛应用在许多领域，如 DNA 序列比对、论文查重等。因此，寻找一种高效的求解方法具有重要的实际应用价值。

问题分析 首先考虑穷举法，对于序列 X(长度为 m)的每个子序列，验证它是否是序列 Y(长度为 n)的子序列。因为长度为 m 的序列 X 共有 2^m 个子序列，对每个子序列，通过遍历序列 Y 的每个字符，可以判断是否是 Y 的子序列，因此，最坏情况下，穷举法的时间复杂度为 $O(n2^m)$，效率很低。而通过动态规划求解该问题，则可以实现较高效率的求解。

动态规划求解最长公共子序列问题的步骤如下。

步骤 1 分析问题和子问题的关系，找出最优解的性质，并刻画其结构特征。

已知序列 $X_m=(x_1,x_2,\cdots,x_m)$，定义 X_m 的第 i 个前缀为 $X_i=(x_1,x_2,\cdots,x_i)$，$i=1,2,\cdots,m$。定义 $c[i,j]$为序列 $X_i=(x_1,x_2,\cdots,x_i)$和 $Y_j=(y_1,y_2,\cdots,y_j)$的最长公共子序列的长度，则 $c[m,n]$表示序列 X_m 和 Y_n 的最长公共子序列的长度。

通过反证法，很容易证明以下命题。

设序列 $X_m=(x_1,x_2,\cdots,x_m)$和 $Y_n=(y_1,y_2,\cdots,y_n)$的一个最长公共子序列为 $Z_k=(z_1,z_2,\cdots,z_k)$，有：

若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列；

若 $x_m\neq y_n$ 且 $z_k\neq x_m$，则 Z_k 是 X_{m-1} 和 Y_n 的最长公共子序列；

若 $x_m\neq y_n$ 且 $z_k\neq y_n$，则 Z_k 是 X_m 和 Y_{n-1} 的最长公共子序列。

以上 3 种情况，说明问题的最优解包含子问题的最优解。因此，最长公共子序列问题具有最优子结构性质。

步骤 2 递归地定义最优值。

本步骤重点分析问题和子问题最优值之间的关系，给出 $c[i,j]$的定义。首先考虑问题规模最小时的情况。

若 $i=0$ 或 $j=0$，说明某个序列为空，任何一个序列和空序列的最长公共子序列长度都为 0，因此有 $c[i,j]=0$。

若 $x_i=y_j$，那么 x_i 或 y_j 一定在某个最长公共子序列中，即 $c[i,j]=c[i-1,j-1]+1$。

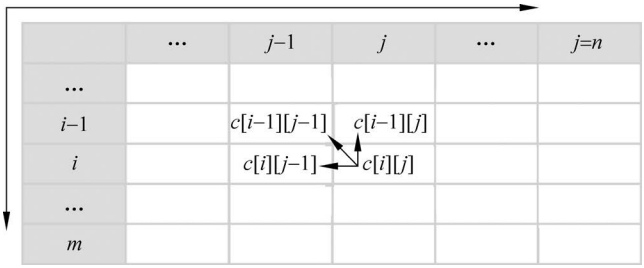
若 $x_i\neq y_j$，那么 $c[i,j]$的值为 $c[i,j-1]$和 $c[i-1,j]$中的较大者。

综合以上情况，得到 $c[i,j]$的递归定义，即

$$c[i,j]=\begin{cases}0, & i=0 \text{ 或 } j=0\\ c[i-1,j-1]+1 & i,j>0 \text{ 且 } x_i=y_j\\ \max(c[i-1,j],c[i,j-1]) & i,j>0 \text{ 且 } x_i\neq y_j\end{cases} \tag{5-13}$$

步骤 3　以自底向上的方式计算出最优值。

根据最优值的递归关系，$c[i,j]$的值与 $c[i-1,j-1]$、$c[i-1,j]$、$c[i,j-1]$有关，需要先求出 $c[i-1,j-1]$、$c[i-1,j]$、$c[i,j-1]$，然后才能求解 $c[i,j]$。因此，$c[i,j]$的求解次序应该是从上到下、从左到右，如图 5-22 所示。



扫一扫

看彩图

图 5-22　动态规划求解最长公共子序列问题的计算顺序

步骤 4　根据计算最优值时得到的信息，构造最优解。根据 $c[i,j]$的取值是来自其上($c[i-1,j]$)、左上($c[i-1,j-1]$)还是其左($c[i,j-1]$)，可以得到最优解。也可以另外定义一个二维数组 $s[m,n]$，记录 $c[i,j]$的 3 种取值信息，得到最优解。

实例分析　给定 $X=(B,D,A,B,A,C)$，$Y=(B,A,D,B,C,D,A)$，求 X 和 Y 的最长公共子序列。定义二维数组 $c[m+1][n+1]$，$c[i][j]$存储序列 $X_i=(x_1,x_2,\cdots,x_i)$和 $Y_j=(y_1,y_2,\cdots,y_j)$的最长公共子序列的长度；定义二维数组 $s[m+1][n+1]$，$s[i][j]$存储 $c[i][j]$取值的 3 种情况。为了同时显示最长公共子序列的长度和最优解的情况，这里用数字表示长度，用符号↑表示 $c[i][j]$的值取自 $c[i-1][j]$，用符号←表示 $c[i][j]$取自 $c[i][j-1]$，用符号↖表示 $c[i][j]$取自 $c[i-1][j-1]+1$，如果 $c[i-1][j]=c[i][j-1]$，优先选择 $c[i-1][j]$。

(1) 初始化。当 $i=0$ 或 $j=0$ 时，$c[i,j]=0$。此时，数组 c 取值如图 5-23 所示。

		0	1 B	2 A	3 D	4 B	5 C	6 D	7 A
	0	0	0	0	0	0	0	0	0
B	1	0							
D	2	0							
A	3	0							
B	4	0							
A	5	0							
C	6	0							

扫一扫

看彩图

图 5-23　动态规划求解最长公共子序列问题：初始化

(2) 计算 $c[1][j]$。首先计算 $i=1$ 时的情况，依次计算 $j=1,2,\cdots,7$，数组 c、s 取值如图 5-24 中的 $i=1$ 行所示。例如，计算 $c[1][4]$时，由于 $x_1=y_4$，因此 $c[1][4]=c[0][3]+1=1$；计算 $c[1][6]$时，由于 $x_1\neq y_6$，因此 $c[1][6]=\max\{c[0][6],c[1][5]\}=\max\{0,1\}=1$。

(3) 计算 $c[2][j]$，依次计算 $j=1,2,\cdots,7$，数组 c、s 取值如图 5-24 中的 $i=2$ 行所示。

(4) 计算 $c[3][j]$，依次计算 $j=1,2,\cdots,7$，数组 c、s 取值如图 5-24 中的 $i=3$ 行所示。

(5) 计算 $c[4][j]$，依次计算 $j=1,2,\cdots,7$，数组 c、s 取值如图 5-24 中的 $i=4$ 行所示。

(6) 计算 $c[5][j]$，依次计算 $j=1,2,\cdots,7$，数组 c、s 取值如图 5-24 中的 $i=5$ 行所示。

扫一扫

看彩图

		0	1 B	2 A	3 D	4 B	5 C	6 D	7 A
	0	0	0	0	0	0	0	0	0
B	1	0	1 ↖	1 ←	1 ←	1 ↖	1 ←	1 ←	1 ←
D	2	0	1 ↑	1 ↑	2 ↖	2 ←	2 ←	2 ↖	2 ←
A	3	0	1 ↑	2 ↖	2 ↑	2 ↑	2 ↑	2 ↑	3 ↖
B	4	0	1 ↖	2 ↑	2 ↑	3 ↖	3 ←	3 ←	3 ↑
A	5	0	1 ↑	2 ↖	2 ↑	3 ↑	3 ↑	3 ↑	4 ↖
C	6	0	1 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ←	4 ↑

图 5-24　动态规划求解最长公共子序列问题：计算数组 c

(7) 计算 $c[6][j]$，依次计算 $j=1,2,\cdots,7$，数组 c、s 取值如图 5-24 中的 $i=6$ 行所示。最终结果记录在 $c[6][7]$中，得到 X 和 Y 的最长公共子序列长度为 4。

(8) 计算最优解。最优解从 $s[6][7]$开始判断：

$s[6][7]=\uparrow$，说明 $c[6][7]=c[5][7]$，继续查看 $s[5][7]$；

$s[5][7]=\nwarrow$，说明 $c[5][7]=c[4][6]+1$，$x_5=y_7=$'A','A'在最长公共子序列中，继续查看 $s[4][6]$；

$s[4][6]=\leftarrow$，说明 $c[4][6]=c[4][5]$，继续查看 $s[4][5]$；

$s[4][5]=\leftarrow$，说明 $c[4][5]=c[4][4]$，继续查看 $s[4][4]$；

$s[4][4]=\nwarrow$，说明 $c[4][4]=c[3][3]+1$，$x_4=y_4=$'B','B'在最长公共子序列中，继续查看 $s[3][3]$；

$s[3][3]=\uparrow$，说明 $c[3][3]=c[2][3]$，继续查看 $s[2][3]$；

$s[2][3]=\nwarrow$，说明 $c[2][3]=c[1][2]+1$，$x_2=y_3=$'D','D'在最长公共子序列中，继续查看 $s[1][2]$；

$s[1][2]=\leftarrow$，说明 $c[1][2]=c[1][1]$，继续查看 $s[1][1]$；

$s[1][1]=\nwarrow$，说明 $c[1][1]=c[0][0]+1$，$x_1=y_1=$'B','B'在最长公共子序列中，由于 $c[i][j]$中如果 $i=0$ 或 $j=0$，说明一个子序列为空，可以终止，因此得到最优解为 BDBA，计算过程如图 5-25 所示。

扫一扫

看彩图

		0	1 B	2 A	3 D	4 B	5 C	6 D	7 A
	0	0	0	0	0	0	0	0	0
B	1	0	1 ↖	1	1 ←	1 ↖	1 ←	1 ←	1 ←
D	2	0	1 ↑	1 ↑	2 ↖	2 ←	2 ←	2 ↖	2 ←
A	3	0	1 ↑	2 ↖	2	2 ↑	2 ↑	2 ↑	3 ↖
B	4	0	1 ↖	2 ↑	2 ↑	3 ↖	3	3	3 ↑
A	5	0	1 ↑	2 ↖	2 ↑	3 ↑	3 ↑	3 ↑	4
C	6	0	1 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ←	4

图 5-25　动态规划求解最长公共子序列问题的最优解

从上述计算过程可以看出，在多个 $c[i][j]$处有 $c[i-1][j]=c[i][j-1]$，因此，最优解不唯一，但它们的长度都是 4。例如，如果 $c[6][7]$的取值从其左边 $c[6][6]$开始推导，按照上述类似的过程，可以得到最优解为 BDBC。

算法实现　数据结构定义、数据读入和主函数见代码 5-17。

代码 5-17：数据结构定义、数据读入和主函数

```
if __name__ == '__main__':
    m, n = list(map(int, input().split()))                  #m 和 n 表示两个序列的长度
    x, y = input().split()                                  #x 和 y 存储序列内容
    c = [[0 for j in range(n+1)] for i in range(m+1)]       #c[i][j]存储最长公共子序列长度
    s = [[0 for j in range(n + 1)] for i in range(m + 1)]   #s[i][j]存储 c[i][j]对应的取值情况
    x = ''+x
    y = ''+y                                                #把序列存储在下标为 1 开始的位置
    print(LCSLength(m, n, x, y, c, s))
    traceback(m, n, x, s)
```

自底向上求最优值。首先初始化，子序列长度为 0 时，设置 $c[0][j]=0$，$c[i][0]=0$；依次计算 $i=1,2,\cdots,m$ 时 $c[i][j]$ 的值，同时用 $s[i][j]=1,2,3$ 分别记录 $c[i][j]$ 取自 $c[i-1][j-1]+1$、$c[i-1][j]$、$c[i][j-1]$ 3 种情况。具体实现见代码 5-18。

代码 5-18：动态规划求解长度为 *m* 的序列 *x* 和长度为 *n* 的序列 *y* 的最长公共子序列长度

```
#数组 c 存储最优值，数组 s 用来记录最优解的相关信息
def LCSLength(m, n, x, y, c, s):
    for i in range(1, m+1):
        for j in range(1, n+1):
            if x[i] == y[j]:
                c[i][j] = c[i-1][j-1] + 1       #当 x[i] = y[j]时最长公共子序列长度加 1
                s[i][j] = 1                     #s[i][j] = 1 表示 x[i]或 y[j]出现在最长
                                                #公共子序列中
            #其他两种情况 c[i][j]不变
            elif c[i-1][j] >= c[i][j-1]:
                c[i][j] = c[i-1][j]
                s[i][j] = 2
            else:
                c[i][j] = c[i][j-1]
                s[i][j] = 3

    return c[m][n]
```

求最优解。从 $s[m][n]$ 开始倒推，如果 $s[i][j]=1$，说明 x_i（或 y_j）出现在最长公共子序列中，i 和 j 同时递减；否则减小 i 或 j，继续寻找，直到 $i=0$ 或 $j=0$ 时为止。具体实现见代码 5-19。

代码 5-19：求最优解

```
def traceback(i, j, x, s):
    if i == 0 or j == 0:                    #边界条件
        return 0
    if s[i][j] == 1:  #s[i][j] = 1 表示 x[i] 或 y[j]出现在最长公共子序列中，进行输出
        traceback(i-1, j-1, x, s)
        print(x[i], end='')
        #其他两种情况减小 i 或 j，继续寻找，直至 i=0 或 j=0
    elif s[i][j] == 2:
        traceback(i - 1, j, x, s)
    else:
        traceback(i, j - 1, x, s)
```

算法复杂度分析 显然,算法的主要计算量在二重循环中,算法的计算时间上界为$O(mn)$。相比穷举法需要指数时间复杂度,算法复杂度降低为多项式时间复杂度。算法所占用的空间是存储最优值和最优解所需要的二维数组,空间复杂度为$O(mn)$。

算法空间优化 对于数组s,前面已经介绍过,只根据数组c的内容也可以获得最优解,因此,数组s可以省略,从而节省一个二维数组。

如果最长公共子序列问题只要求求解最优值,那么和0-1背包问题类似,由于$c[i][j]$只与其左上方、上方和左方的3个子问题有关,也就是只与上一行的子问题相关,因此从空间优化角度,可以把二维数组(空间复杂度为$O(mn)$)降低为两个一维数组的空间(使用滚动数组,空间复杂度为$O(m)$或$O(n)$)。由于序列X和Y无论哪个作为行或列都可以,从节省空间考虑,选择长度小的作为列即可。因此,可以进一步把空间降低到$2\min\{m,n\}$。需要注意的是,空间优化后,时间复杂度保持不变(计算量并没有减少)。

应用扩展 给定两个单词word1和word2,找到使word1和word2相同所需的最小步数,每步可以删除任意一个单词中的一个字符。例如,两个单词分别为sea和eat,最少需要两步,第1步将sea变为ea,第2步将eat变为ea。

由于只能进行删除操作,那么使两个单词相同的最少删除步数,就是找到两个单词的最大公共部分,即两个单词的最长公共子序列,然后再删除其余的字符。因此,利用本节学习的动态规划求解方法,先求出两个单词的最长公共子序列长度(假定为k),那么最小步数为word1的长度+word2的长度$-2k$。

最长公共子序列问题在很多实际应用中发挥着重要作用,采用动态规划方法可以在多项式时间复杂度内求解该问题。求解的重点是分析规模较小的问题最优值和规模较大的问题最优值之间的关系,巧妙的方法是对两个序列的最后一个字符进行判断,根据可能的两种情况:相等或不等,可以找到问题最优值和子问题最优值之间的关系。

扫一扫

视频讲解

5.7 动态规划应用:最长不上升子序列

首先看一个问题:某国为了防御敌国的导弹袭击,开发出一种导弹拦截系统。但是,这种导弹拦截系统有一个缺陷:虽然它的第1发炮弹能够到达任意的高度,但是以后每发炮弹都不能高于前一发的高度。某天,雷达捕捉到敌国的导弹来袭,由于还在试用阶段,所以只有一套系统,因此有可能不能拦截所有导弹。输入导弹的枚数和导弹依次飞来的高度(雷达给出的高度数据是不大于30 000的正整数,每个数据之间有一个空格),计算这套系统最多能拦截多少导弹。

根据题意,所能拦截的导弹中,满足每枚导弹高度小于或等于前一枚导弹高度的条件,因此,该问题本质上是在一个序列中寻找一个长度最大的子序列,子序列中每个数都小于或等于前一个数,这个问题称为最长不上升子序列问题。下面给出这个问题的一般性描述。

问题描述 已知一个正整数序列$b_1,b_2,\cdots,b_n$,对于下标i1<i2<…<im,若有$b_{i1}\geqslant b_{i2}\geqslant\cdots\geqslant b_{im}$,则称存在一个长度为$m$的不上升子序列。由于要求i1<i2<…<im,即下标依次递增,意味着要按照序列从左到右的顺序选择若干数构成子序列;$b_{i1}\geqslant b_{i2}\geqslant\cdots\geqslant b_{im}$意味着这个子序列中数值变化特点是不增加(即不上升)。求解满足这两个条件且长度最大的子序列就是最长不上升子序列问题。

举例说明:给定长度为8的正整数序列389,207,155,300,299,170,158,65。对于下标

i1＝0，i2＝3，i3＝4，i4＝5，i5＝6，i6＝7 满足 389＞300＞299＞170＞158＞65 且长度最长，所以存在长度为 6 的最长不上升子序列。

问题分析　如果采用穷举法，基本过程是：找出长度为 n 的所有子序列，逐一判断，选择满足条件的最长的子序列。虽然穷举法简单，但其最坏情况的时间复杂度为 $O(2^n)$。

下面考虑用动态规划求解。

算法设计　给定序列长度为 n，依次存放在 $a[0],a[1],\cdots,a[n-1]$中，定义 $f(i)$为以 $a[i]$结尾的最长不上升子序列的长度，那么整个序列的最优值一定是所有 $f(i)$中值最大的，即 $\max\{f(i)\},i=0,1,\cdots,n-1$。

根据定义，$f(i)$表示以 $a[i]$结尾的最长不上升子序列的长度，在这个子序列中，$a[i]$的前一个元素 $a[j]$一定满足 $a[j]\geqslant a[i]$，由于要求子序列长度是最大的，因此，只需要找出最大的 $f(j)$，即 $f(i)=\max\{f(j)+1\}$，其中 $j<i$ 且 $a[j]\geqslant a[i]$，$0\leqslant j<i<n$。这个式子的含义是：找到 i 之前的某个 j，满足 $a[j]\geqslant a[i]$，且 $f(j)$最大。可以看出，该式中的 $f(i)$和 $f(j)$的关系反映了最优子结构性质。

整个序列的最长不上升子序列长度为 $\max\{f(i)\},i=0,1,\cdots,n-1$。

需要注意的是，该问题的最优值定义与本章前面几个典型应用不同，不能直接给出得到最优值的问题定义，只能间接得到。

实例分析　给定长度为 8 的正整数序列，分别为 389，207，155，300，299，170，158，65，结合 $f(i)$的求解方法，从左到右依次计算 $f(i)$，得到结果如表 5-8 所示。

表 5-8　动态规划求解最长不上升子序列问题

i	0	1	2	3	4	5	6	7
$a[i]$	389	207	155	300	299	170	158	65
$f[i]$	1	2	3	2	3	4	5	6

(1) 首先计算 $f[0]$：由于以 $a[0]$结尾的子序列只有一个元素，因此 $f[0]=1$。

(2) 计算 $f[1]$：由于 $a[0]\geqslant a[1]$，因此 $f[1]=f[0]+1=2$。

(3) 计算 $f[2]$：由于 $a[1]\geqslant a[2]$，$a[0]\geqslant a[2]$，因此 $f[2]=\max\{f[1],f[0]\}+1=3$。

(4) 计算 $f[3]$：由于 $a[2]<a[3]$，$a[1]<a[3]$，只有 $a[0]\geqslant a[3]$，因此 $f[3]=f[0]+1=2$。

(5) 计算 $f[4]$：由于 $a[3]\geqslant a[4]$，$a[0]\geqslant a[4]$，因此，$f[4]=\max\{f[3],f[0]\}+1=3$。

(6) 计算 $f[5]$：除 $a[2]<a[5]$外，$a[0]\sim a[4]\geqslant a[5]$，因此 $f[5]=\max\{f[0],f[1],f[3],f[4]\}+1=4$。

(7) 计算 $f[6]$：$f[6]=\max\{f[0],f[1],f[3],f[4],f[5]\}+1=5$。

(8) 计算 $f[7]$：$f[7]=\max\{f[0],f[1],f[2],f[3],f[4],f[5],f[6]\}+1=6$。

所以，整个序列的最长不上升子序列长度为 $\max\{f[i]\}=6$。

算法实现　根据上述计算过程，求解主要包括 3 步：

(1) 读数据并初始化；

(2) 分阶段求出每步的最优值，通过一个二重循环实现，对于每个 i，依次判断其前面的每个 j 是否满足条件；

(3) 通过一轮遍历找出 $f[i]$中最大值。

具体实现如代码 5-20 所示，而根据最优值如何求解最长不上升子序列问题的最优解，作为习题留给读者。

代码 5-20：动态规划求解最长不上升子序列问题的最优解

```
#函数功能：动态规划求解最长不上升子序列问题的最优解
#参数说明：n 和 a 分别存储序列元素总数和序列元素
def LNIS(n, a):
    f = [0 for i in range(n)]      #f[i]表示以元素 a[i]结尾的最长不上升子序列的最优值
    f[0] = 1                       #初始化

    for i in range(n):             #从前往后依次计算 f[i]
        temp = 1
        for j in range(i):
            if a[j] >= a[i] and f[j] + 1 > temp:   #a[i]依次与从 a[0]到 a[i-1]的数字比较
                                                   #当 a[i]<a[j]且 temp<f[j]+1 时
                                                   #更新 temp,并将 temp 赋给 f[i]
                temp = f[j] + 1
        f[i] = temp

    return max(f)                  #返回 f 最大值

if __name__ == '__main__':
    n = int(input())               #n 表示序列长度
    a = list(map(eval, input().split()))

    print(LNIS(n, a))
```

复杂度分析 显然，该算法的时间复杂度为 $T(n)=O(n^2)$，空间复杂度为 $O(n)$。

应用扩展 和最长不上升子序列问题类似，对于一个给定序列，寻找其最长上升子序列、最长下降子序列、最长不下降子序列等问题的求解方法是相同的，相信读者很快可以得到答案。

本节从拦截导弹这个具体问题抽象出更一般的问题——最长不上升子序列问题，然后给出使用动态规划方法求解的思路、具体过程和实现代码，和穷举法花费指数时间复杂度相比，降低到了多项式时间复杂度，效率提升明显。

扫一扫

视频讲解

5.8 动态规划应用：编辑距离问题

问题描述 设 A 和 B 是两个字符串，现要用最少的字符操作将字符串 A 转换为字符串 B。这里所说的字符操作是指：①删除一个字符；②插入一个字符；③将一个字符改为另一个字符。将字符串 A 转换为字符串 B 所用的最少字符操作数称为字符串 A 到字符串 B 的编辑距离。给定两个字符串 A 和 B，计算出它们的编辑距离。例如，两个字符串分别为 banana 和 panda，则编辑距离为 3。

算法设计 设所给的两个字符串为 $A[1..m]$和 $B[1..n]$，$\delta(A,B)$表示字符串 A 和 B 的编辑距离，定义 $D[i,j]=\delta(A[1..i],B[1..j])$。单个字符 a 和 b 之间的距离定义为

$$\delta(a,b)=\begin{cases}0, & a=b\\ 1, & a\neq b\end{cases}$$

从问题和子问题之间的关系考虑，对于字符串 $A[1..i]$到 $B[1..j]$的最优转换，首先考虑 $A[i]$和 $B[j]$的转换，之后就可以转换为规模更小的字符串转换问题。根据题意，$A[i]$和 $B[j]$的转换可以通过修改、插入、删除 3 种操作完成，而编辑距离要求最少的字符操作数，因此，$D[i,j]$的取值来自以下 3 种情况中最小的一种。

(1) 如果把字符 $A[i]$修改为 $B[j]$,那么 $D[i,j]$转换为求解 $D[i-1,j-1]$的问题,此时 $D[i,j]=D[i-1,j-1]+\delta(A[i],B[j])$。

(2) 如果是删除字符 $A[i]$(需要一次操作),那么 $D[i,j]$转换为求解 $D[i-1,j]$的问题,此时 $D[i,j]=D[i-1,j]+1$。

(3) 如果是插入字符 $B[j]$(需要一次操作),那么 $D[i,j]$转换为求解 $D[i,j-1]$的问题,此时 $D[i,j]=D[i,j-1]+1$。

所以,$D[i,j]$可递归地定义为

$$D[i,j]=\min\{D[i-1,j-1]+\delta(A[i],B[j]),D[i-1,j]+1,D[i,j-1]+1\}$$

$$D[i,0]=i,i=0,1,\cdots,m$$

$$D[0,j]=j,j=0,1,\cdots,n$$

初始状态定义为某个字符串为空,此时,通过对非空字符串的每个字符进行逐个删除,可以用最少的操作次数进行变换。

为了获得最优解,记录具体的字符操作,定义 $C[i,j]$如下。

$C[i,j]=0$ 表示 $A[i]$和 $B[j]$相等,不需要任何操作。

$C[i,j]=1$ 表示把 $A[i]$修改为 $B[j]$。

$C[i,j]=2$ 表示删除 $A[i]$。

$C[i,j]=3$ 表示插入 $B[j]$。

问题的最优值 $\delta(A,B)=D[m,n]$。从 $C[m,n]$开始可反推出 A 到 B 的转换方案。

实例分析　在编辑距离问题中,假定 $A=$banana,$B=$panda,采用自底向上的方法,求解得到数组 D(见表 5-9)、数组 C(见表 5-10)。

表 5-9　动态规划求解编辑距离问题：数组 *D*

i	*D*[*i*][*j*]					
	***j*=0**	***j*=1(p)**	***j*=2(a)**	***j*=3(n)**	***j*=4(d)**	***j*=5(a)**
$i=0$	0	1	2	3	4	5
$i=1$(b)	1	1	2	3	4	5
$i=2$(a)	2	2	1	2	3	4
$i=3$(n)	3	3	2	1	2	3
$i=4$(a)	4	4	3	2	2	2
$i=5$(n)	5	5	4	3	3	3
$i=6$(a)	6	6	5	4	4	3

表 5-10　动态规划求解编辑距离问题：数组 *C*

i	*C*[*i*][*j*]					
	***j*=0**	***j*=1(p)**	***j*=2(a)**	***j*=3(n)**	***j*=4(d)**	***j*=5(a)**
$i=0$	0	3	3	3	3	3
$i=1$(b)	2	1	1	1	1	1
$i=2$(a)	2	1	0	3	3	0
$i=3$(n)	2	1	2	0	3	3
$i=4$(a)	2	1	0	2	1	0
$i=5$(n)	2	1	2	0	1	1
$i=6$(a)	2	1	0	2	1	0

算法实现 具体实现如代码 5-21 所示。

代码 5-21：动态规划求解编辑距离问题

```
def editDistance(A, B, m, n):
    #初始化,当 B 为空时,删除 A 的每个字符
    for i in range(0, m + 1):
        D[i][0] = i
        C[i][0] = 2

    #初始化,当 A 为空时,增加 B 的每个字符
    for j in range(0, n + 1):
        D[0][j] = j
        C[0][j] = 3

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            #当字符相等时,不做任何操作
            if A[i] == B[j]:
                D[i][j] = D[i - 1][j - 1]
                C[i][j] = 0
            else:
                t1 = D[i - 1][j - 1] + 1
                t2 = D[i - 1][j] + 1
                t3 = D[i][j - 1] + 1

                if t1 <= t2 and t1 <= t3:          #把 A[i]修改为 B[j]
                    D[i][j] = t1
                    C[i][j] = 1
                elif t2 <= t1 and t2 <= t3:        #删除 A[i]
                    D[i][j] = t2
                    C[i][j] = 2
                elif t3 <= t1 and t3 <= t2:        #增加 B[j]
                    D[i][j] = t3
                    C[i][j] = 3

if __name__ == '__main__':
    A = list(input("输入数组 A: "))
    B = list(input("输入数组 B: "))
    m = len(A)
    n = len(B)
    A.insert(0, A[0])
    B.insert(0, B[0])

    D = [[0 for i in range(n + 1)] for j in range(m + 1)]#记录 A[1,i]到 B[1,j]的编辑距离
    C = [[0 for i in range(n + 1)] for j in range(m + 1)]  #记录 A[1,i]到 B[1,j]所进行的操作

    editDistance(A, B, m, n)

    print("D:")
    for i in range(0, m + 1):
        for j in range(0, n + 1):
            print("{:<4}".format(D[i][j]), end = '')
```

```
        print()
    print("=======================")
    print("C:")
    for i in range(0, m + 1):
        for j in range(0, n + 1):
            print("{:<4}".format(C[i][j]), end = '')
        print()
```

算法复杂度分析　算法的时间复杂度为 $T(n)=O(mn)$，空间复杂度为 $O(mn)$。

应用场景　两个字符串的编辑距离反映了它们之间的相似程度，编辑距离越短，说明相似程度越高。编辑距离问题通常用于需要对字符串进行匹配的场景，如 DNA 分析、拼写检查、语音识别等。例如，在 DNA 分析中，由于 DNA 序列是由 A、G、C、T 组成的序列，可以类比为字符串，通过编辑距离衡量两个 DNA 序列的相似度，编辑距离越小，说明这两个 DNA 序列越相似。

本节讨论的编辑距离问题是莱文斯坦距离，属于编辑距离的一种。该算法是由弗拉基米尔·莱文斯坦(Levenstein Vladimir I)于 1965 年提出的，有兴趣的读者可以查阅更多的编辑距离问题。

扫一扫

视频讲解

5.9　动态规划应用：最优二叉搜索树

二叉搜索树是在元素发生动态改变的序列中，借鉴二分查找的思想，为了方便对元素进行高效查找而提出的一种二叉树结构，定义如下。

(1) 若它的左子树不为空，则左子树上所有节点的值均小于它的根节点的值。

(2) 若它的右子树不为空，则右子树上所有节点的值均大于它的根节点的值。

(3) 它的左、右子树也分别为二叉搜索树。

图 5-26 所示就是一棵二叉搜索树，其中实线节点表示序列中的元素 a_i，虚线节点 b_i 是为了方便描述搜索过程加上的虚拟叶子节点。

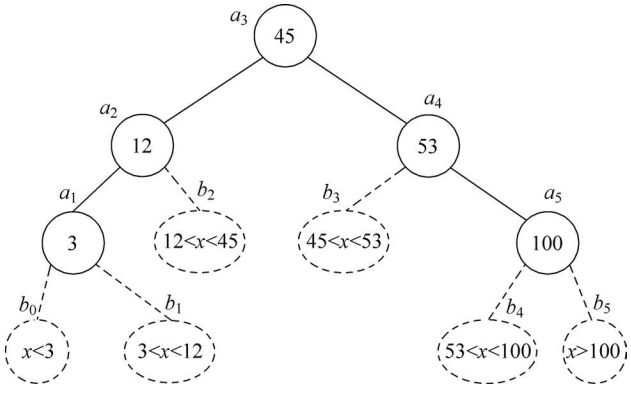


图 5-26　二叉搜索树(1)

在该二叉搜索树上搜索节点 100 的过程是：首先与根节点 45 比较，100＞45，所以在右子树继续查找；与节点 53 比较，100＞53，继续在右子树查找；与节点 100 比较，100＝100，查找成功，结束。

可见，在二叉搜索树上查找成功所需要的比较次数是从根节点到该节点路径上经过的

节点数目,也等于该路径上边的数目加1。

类似地,在该二叉搜索树上搜索节点40的过程是:首先与根节点45比较,40<45,所以在左子树继续查找;与节点12比较,40>12,继续在右子树查找;右子树是一个虚拟叶子节点,表示查找不成功,结束。

可知,在二叉搜索树上查找不成功所需要的比较次数是从根节点到虚拟叶子节点路径上经过的实线节点数目,也等于该路径上边的数目。

由以上搜索过程可知,在二叉搜索树上查找某个节点,如果遇到实线节点终止,表示搜索成功;如果遇到虚拟叶子节点终止,表示搜索失败。根据二叉树的基本性质,$n_0=n_2+1$(n_0表示度为0的节点总数,n_2表示度为2的节点总数),对于由n个元素构成的二叉搜索树,可知有n个实线节点和$n+1$个虚拟叶子节点,分别对应查找成功的n种情况和查找不成功的$n+1$种情况。

问题描述 给定若干元素构成的序列$S=<a_1,a_2,\cdots,a_n>$,该序列中搜索成功和不成功的概率定义为$P=<q_0,p_1,q_1,p_2,q_2,\cdots,p_n,q_n>$,$p_i$表示搜索$a_i$成功的概率,$q_i$表示待搜索元素位于区间$(a_i,a_{i+1})$且搜索失败的概率。其中,$q_0$表示待搜索元素小于$a_1$时的搜索概率,$q_n$表示待搜索元素大于$a_n$时的搜索概率。所有$p_i$和$q_i$满足$\sum_{i=1}^{n}p_i+\sum_{i=0}^{n}q_i=1$。

由序列S构成的二叉搜索树中,$d(a_i)$表示根节点到节点a_i的路径上边的数目,$d(b_i)$表示根节点到虚拟叶子节点b_i的路径上边的数目,在该二叉搜索树上,查找操作对应的平均比较次数为$t=\sum_{i=1}^{n}p_i(1+d(a_i))+\sum_{i=0}^{n}q_id(b_i)$。

在S和P给定的情况下,可以构建多种不同的二叉搜索树,平均比较次数也不尽相同。假定$S=<3,12,45,53,100>$,$P=<0.05,\underline{0.10},0.15,\underline{0.10},0.05,\underline{0.15},0.04,\underline{0.12},0.08,\underline{0.07},0.09>$,如果构建的二叉搜索树如图5-26所示,搜索$a_i(i=1,2,\cdots,5)$成功时对应的比较次数分别为3、2、1、2、3;搜索$b_i(i=0,2,\cdots,5)$不成功时对应的比较次数分别为3、3、2、2、3、3,则平均比较次数$t=(0.1\times3+0.1\times2+0.15\times1+0.12\times2+0.07\times3)+(0.05\times3+0.15\times3+0.05\times2+0.04\times2+0.08\times3+0.09\times3)=1.1+1.29=2.39$。

保持S和P不变,还可以构造如图5-27所示的二叉搜索树,其平均比较次数$t=(0.1\times2+0.1\times1+0.15\times3+0.12\times2+0.07\times3)+(0.05\times2+0.15\times2+0.05\times3+0.04\times3+0.08\times3+0.09\times3)=1.2+1.18=2.38$。

因此,给定序列S和查找概率P,可以构造不同的二叉搜索树,其中,平均比较次数最少的就是最优二叉搜索树。那么,如何找到最优二叉搜索树呢?

问题分析 首先考虑最优二叉搜索树是否满足最优子结构性质。如图5-28所示,假定以a_k为根节点的二叉搜索树是给定S和P对应的最优二叉搜索树T,那么其左子树一定是由序列$S_L=<a_1,a_2,\cdots,a_{k-1}>$以及对应查找概率$P_L=<q_0,p_1,q_1,p_2,q_2,\cdots,p_{k-1},q_{k-1}>$构成的最优二叉搜索树;同理,其右子树一定是由序列$S_R=<a_{k+1},a_{k+2},\cdots,a_n>$以及对应查找概率$P_R=<q_k,p_{k+1},q_{k+1},\cdots,p_n,q_n>$构成的最优二叉搜索树。可以用反证法证明。

假设其左子树L不是S_L和P_L对应的最优二叉搜索树,一定存在一棵最优二叉搜索

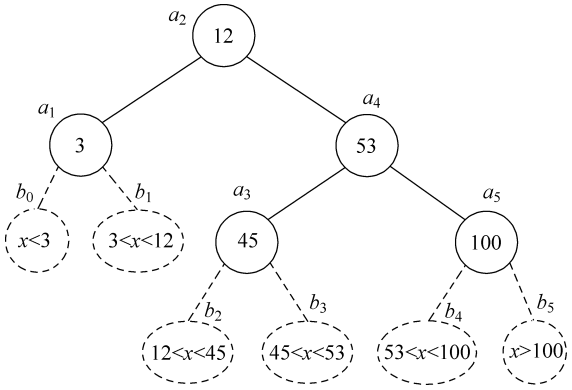


图 5-27　二叉搜索树(2)

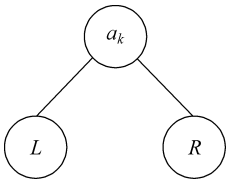


图 5-28　最优二叉搜索树

树 L',使 L'的平均比较次数小于 L 的平均比较次数。用 L'替换 L,得到对应 S 和 P 的二叉搜索树 T',由于 T'与 T 的区别只是其左子树不同,而根节点和右子树保持不变,可知,T'的平均比较次数小于 T 的平均比较次数,这与 T 是最优二叉搜索树矛盾,因此假设不成立,说明最优二叉搜索树 T 的左子树也是一棵最优二叉搜索树。同理,可证明 T 的右子树也是也是一棵最优二叉搜索树。因此,最优二叉搜索树满足最优子结构性质。

利用动态规划算法,考虑以每个节点作为根节点,从中选择最优二叉搜索树。

算法设计　根节点 $a_k(k=1,2,\cdots,n)$把序列和概率分布分为两个连续的部分,已知 $S(i,j)=<a_i,a_{i+1},\cdots,a_j>$,概率分布 $P(i,j)=<q_{i-1},p_i,q_i,\cdots,p_j,q_j>$,$w(i,j)=q_{i-1}+p_i+q_i+\cdots+p_j+q_j$。定义 $m(i,j)$是对于输入 $S(i,j)$和 $P(i,j)$的最优二叉搜索树的平均比较次数,那么 $m(1,n)$就是问题所要求的最优值。计算 $m(i,j)$的递归式为

$$\begin{cases} m(i,j)=w(i,j)+\min\limits_{i\leqslant k\leqslant j}\{m(i,k-1)+m(k+1,j)\}, & i\leqslant j \\ m(i,i-1)=0, & 1\leqslant i\leqslant n \end{cases}$$

为了得到最优解,记录 $q(i,j)=k$,存储使 $m(i,j)$获得最小值时对应的根节点 k。

可以看出,最优二叉搜索树的动态规划求解方法与矩阵连乘问题求解方法类似。采用自底向上的方式,依次计算 $1,2,\cdots,n$ 个节点对应的最优二叉搜索树。下面通过实例给出具体的计算过程。

实例分析　已知 $S=<3,12,45,53,100>$,$P=<q_0,p_1,q_1,p_2,q_2,p_3,q_3,p_4,q_4,p_5,q_5>=<0.05,0.10,0.15,0.10,0.05,0.15,0.04,0.12,0.08,0.07,0.09>$,求对应的最优二叉搜索树。

(1) 首先计算只有一个节点的二叉搜索树的平均比较次数 $m(1,1)$、$m(2,2)$、$m(3,3)$、$m(4,4)$、$m(5,5)$,如图 5-29 所示。

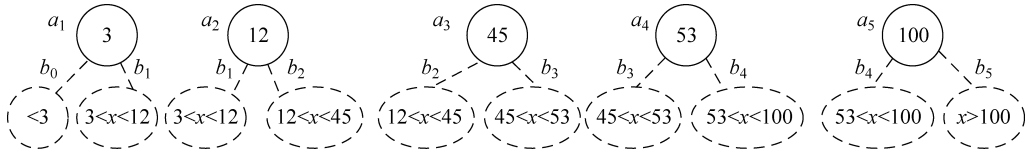


图 5-29　最优二叉搜索树(节点数=1)

$m(1,1)=q_0+p_1+q_1=0.3$，$m(2,2)=q_1+p_2+q_2=0.3$，$m(3,3)=q_2+p_3+q_3=0.24$，$m(4,4)=q_3+p_4+q_4=0.24$，$m(5,5)=q_4+p_5+q_5=0.24$。

(2) 计算有两个节点的二叉搜索树的最少平均比较次数 $m(1,2)$、$m(2,3)$、$m(3,4)$、$m(4,5)$。以 $m(2,3)$的计算为例，如图 5-30 所示。

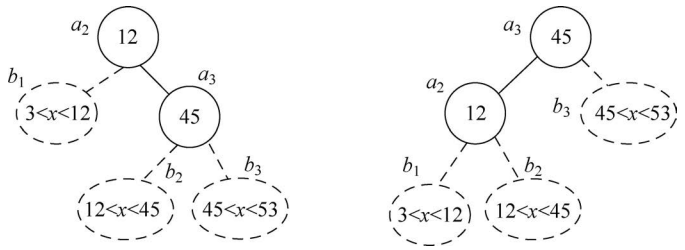


图 5-30　12 和 45 构成的两种形态的二叉搜索树

以 a_2 为根节点时，$m(2,3)=m(3,3)+w(2,3)=0.24+q_1+p_2+q_2+p_3+q_3=0.24+0.49=0.73$。

以 a_3 为根节点时，$m(2,3)=m(2,2)+w(2,3)=0.30+q_1+p_2+q_2+p_3+q_3=0.30+0.49=0.79$。

所以，$m(2,3)=0.73$，即图 5-30 左侧是$<a_2,a_3>$对应的最优二叉搜索树。

(3) 计算有 3 个节点的二叉搜索树的最少平均比较次数 $m(1,3)$、$m(2,4)$、$m(3,5)$。以 $m(2,4)$的计算为例，如图 5-31 所示。

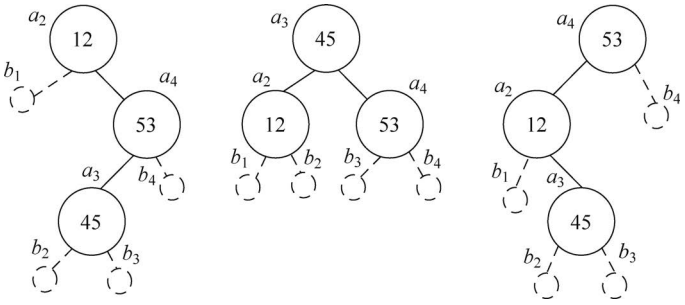


图 5-31　12、45 和 53 构成的 3 种形态的二叉搜索树

以 a_2 为根节点时，$m(2,4)=m(3,4)+w(2,4)=0.68+q_1+p_2+q_2+p_3+q_3+p_4+q_4=0.68+0.69=1.37$。

以 a_3 为根节点时，$m(2,4)=m(2,2)+m(3,3)+w(2,4)=0.30+0.24+q_1+p_2+q_2+p_3+q_3+p_4+q_4=0.54+0.69=1.23$。

以 a_4 为根节点时，$m(2,4)=m(2,3)+w(2,4)=0.73+q_1+p_2+q_2+p_3+q_3+p_4+q_4=0.73+0.69=1.42$。

所以，$m(2,4)=1.23$，即图 5-31 中间的是$<a_2,a_3,a_4>$对应的最优二叉搜索树。

(4) 计算有 4 个节点的二叉搜索树的最少平均比较次数 $m(1,4)$、$m(2,5)$。以 $m(2,5)$的计算为例，共有 4 种形态的二叉搜索树，如图 5-32 所示，其中以 a_3 为根节点的对应最优二叉搜索树，最优值的计算这里不再详述。

(5) 计算有 5 个节点的二叉搜索树的最少平均比较次数 $m(1,5)$。最终的数组 m 如表 5-11 所示，数组 q 如表 5-12 所示。

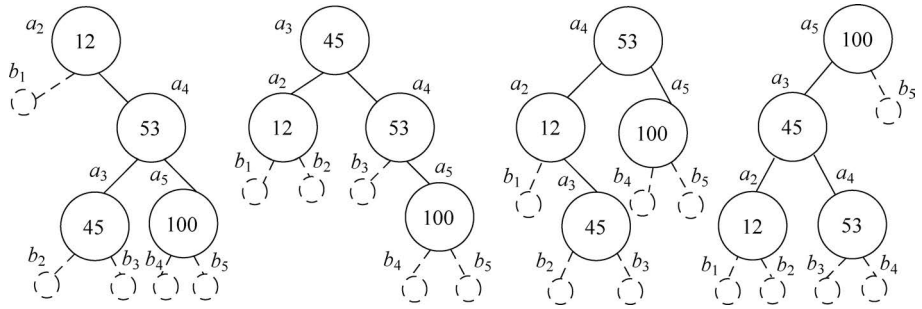


图 5-32　12、45、53 和 100 构成的 4 种形态的二叉搜索树

表 5-11　动态规划求解最优二叉搜索树：最优值(数组 *m*)

i	*m*[*i*][*j*]					
	***j*=0**	***j*=1**	***j*=2**	***j*=3**	***j*=4**	***j*=5**
i=1	0	0.30	0.75	1.18	1.82	2.38
i=2		0	0.30	0.73	1.23	1.79
i=3			0	0.24	0.68	1.08
i=4				0	0.24	0.64
i=5					0	0.24

表 5-12　动态规划求解最优二叉搜索树：最优解(数组 *q*)

i	*q*[*i*][*j*]					
	***j*=0**	***j*=1**	***j*=2**	***j*=3**	***j*=4**	***j*=5**
i=1	0	1	1	2	2	2
i=2			2	2	3	3
i=3				3	4	4
i=4					4	4
i=5						5

得到最优值为 2.38，最优解为以 *s*[2]为根节点(*q*[1][5]=2)，其左子树为 *s*[1](*q*[1][1]=1)，其右子树根节点为 *s*[4](*q*[3][5]=4)，*s*[4]的左子树为 *s*[3](*q*[3][3]=3)，*s*[4]的右子树为 *s*[5](*q*[5][5]=5)，即最优二叉搜索树如图 5-27 所示。

算法实现　具体实现如代码 5-22 所示。

代码 5-22：动态规划求解最优二叉搜索树

```
#计算最优值并记录断开位置
def optimalBT(p, w, n, m, q):
    #计算 w[i]
    for i in range(1, n + 1):
        w[i] = w[i - 1] + p[2 * i - 2] + p[2 * i - 1]

    #初始化节点数目为 0 和 1 的二叉搜索树
    for i in range(1, n + 1):
        m[i][i] = w[i] - w[i - 1] + p[2 * i]
        q[i][i] = i
        m[i][i - 1] = 0

    for r in range(2, n + 1):              #r 表示二叉搜索树节点数目,树中节点数目从 2 到 n
```

```
            for i in range(1, n - r + 2):     #i表示节点的起始编号
                j = i + r - 1                 #j表示节点的终止编号
                w_i_j = w[j] - w[i - 1] + p[2 * j]  #w_i_j表示w(i,j)=w[j]-w[i-1]+p[2*j]
                m[i][j] = m[i + 1][j] + w_i_j      #s[i]作为根节点时的平均比较次数
                q[i][j] = i

                for k in range(i + 1, j + 1):        #k表示根节点的编号
                    t = m[i][k - 1] + m[k + 1][j] + w_i_j

                    #找到较小比较次数,更新最优值和根节点
                    if t < m[i][j]:
                        m[i][j] = t
                        q[i][j] = k

        print(m[1][n])

    def traceback(s, q, i, j):
        if i == j:
            print(s[i], end = '')
            return

        if i > q[i][j] - 1:
            return

        print("(", end = '')
        traceback(s, q, i, q[i][j] - 1)
        print(")", end = '')

        print(s[q[i][j]], end = '')

        if q[i][j] + 1 > j:
            return

        print("(", end = '')
        traceback(s, q, q[i][j] + 1, j)
        print(")", end = '')

    if __name__ == '__main__':
        n = int(input("输入n: "))                          #n表示序列中元素的个数
        s = list(map(eval, input("输入数组s: ").split())) #s[i]存储序列的第i个元素
                                                           #i从1开始
        s.insert(0, 0)
        p = list(map(eval, input("输入数组p: ").split())) #p[i]存储搜索成功和不成功概率
        w = [0 for i in range(n + 1)]                  #w[i]存储 p[0]+p[1]+...+p[2*i-1]
        m = [[0 for i in range(n + 2)] for j in range(n + 2)]  #m[i][j]存储最优值
        q = [[0 for i in range(n + 1)] for j in range(n + 1)]  #q[i][j]存储m[i][j]根节点序号

        optimalBT(p, w, n, m, q)
        traceback(s, q, 1, n)
        print()

        print("最优值矩阵: ")
        for i in range(1, n + 1):
            for j in range(0, n + 1):
                print("{:.2f} ".format(m[i][j]), end = '')
            print()
```

```
    print("===============")
    print("最优解矩阵:")
    for i in range(1, n + 1):
        for j in range(0, n + 1):
            print("{:<4}".format(q[i][j]), end='')
        print()
```

以上述示例为例，输出最优解为 (3)12((45)53(100))，表示 12 为根节点，其右子树的根节点为 53。

算法复杂度分析　由于算法的主要计算量在三重循环上，因此算法的时间复杂度为 $T(n)=O(n^3)$，空间复杂度为 $O(n^2)$。

扫一扫

视频讲解

5.10　小结

动态规划是一种用途很广的问题求解方法，它本身并不是一个特定的算法，不像搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法，而是一种思想，一种手段。

动态规划的实质是分治思想和解决冗余。动态规划是一种将问题分解为更小、相似的子问题，并存储子问题的解而避免计算重复，解决最优化问题的算法策略。

学习动态规划最重要的是学习“一种思想方法和解题过程”。由于问题性质不同，确定最优解的条件也互不相同，因而不存在一种万能的动态规划算法可以解决各类最优化问题。在使用时，除了要对基本概念和方法正确理解外，必须具体问题具体分析，以丰富的想象力建立模型，用创造性的技巧去求解。

学习中，可以透过典型应用的动态规划求解算法，分析典型应用的特征和本质，逐渐学会并掌握这一设计方法。本章的典型应用包括数字三角形、0-1 背包、矩阵连乘、最长公共子序列、最长不上升子序列、编辑距离和最优二叉搜索树。

扩展阅读

动 态 规 划[①]

动态规划最初是运筹学的一个分支，是求解决策过程最优化的过程。20 世纪 50 年代初，美国数学家贝尔曼(R. Bellman)(也是最短路径问题中 Bellman-Ford 算法的发明者之一)等在研究多阶段决策过程的优化问题时，提出了著名的最优化原理，从而创立了动态规划。动态规划的应用极其广泛，包括工程技术、经济、工业生产、军事以及自动化控制等领域，并在背包问题、生产经营问题、资金管理问题、资源分配问题、最短路径问题和复杂系统可靠性问题等很多问题中取得了显著的效果。

贝尔曼对动态规划做出了巨大贡献，而他的成功之路却很艰辛和不易。

贝尔曼是家里唯一的儿子，但他和家人的生活并不那么容易，他反复强调“我们不应该是穷人”。他的父亲把大萧条的影响看得很严重，因此与很多很好的工作失之交臂，每次当他父亲准备好投入一份工作时，都为时已晚。贝尔曼还坦白了他在童年时期不得不面对的两个诅咒：尿床以及对黑暗的恐惧，这与很多孩童都相似。贝尔曼虽然很成功，但是他在中

① 参考来源：Bellman R. Eye of the Hurricane: An Autobiography[M]. WSPC, 1984.

学时对课堂演讲存在深深的恐惧感,每学期的课堂汇报他都要拖到最后一个上台。当他上大学时,这种恐惧仍然存在,但是他强迫自己尽可能多地进行演讲,以克服他的演讲恐惧症。这很符合贝尔曼的处事原则——“一旦我开始了,就会做得很好”。

少年的贝尔曼对心理学很感兴趣,他认为这帮助他处理了青春期的一些问题。此外,他还狂热于科幻小说和文学作品,其中他最喜欢的是马克·吐温和莎士比亚。马克·吐温对贝尔曼的影响最大,马克·吐温的书籍帮助他拓宽了想象力,并培养了他极强的幽默感。也是在这个时候,他开始意识到他已经读够了小说,决定探索一些真实的东西。在他人生的这一时刻,他第一次接触到古生物学、考古学、哲学、历史、传记等学科。

贝尔曼在大学时是数学俱乐部的成员,在那里他偶尔会听到关于数学的讲座。他记得最清楚的是库兰特关于在一个给定的三角形中刻上一个周长最小的三角形问题的讲座。他在大学里待了 4 年半,从布鲁克林学院毕业时,他没有得到任何奖章。但是比起奖章来说,他更喜欢一些书,这就是他开始收集好书的原因。他买的第一本书是《函数论》,然后是《傅里叶积分》。这些书让他对数学有了很好的了解,他甚至在这个时候开始写他的第一篇论文。

1973 年,贝尔曼被诊断患有脑瘤。切除肿瘤的手术是成功的,但由于手术后的并发症,他几乎瘫痪了。尽管他的身体出现了严重的问题,他仍然非常积极地进行数学研究,在他生命的剩余 10 年里,他写了大约 100 篇论文,1984 年去世后,他的书籍还在继续出版。

贝尔曼一生中获奖无数、荣誉无数,但最重要的是,除了贝尔曼传奇的名声之外,他所表现出的勇气和伟大也让他获得了所有他认识的以及认识他的人的钦佩和喜爱。

读到这里,相信大家脑海中动态规划之父贝尔曼的形象将更加鲜活。

从算法思想到立德树人

动态规划与分治的区别在于,当子问题需要多次使用时,通过一种朴素、直观的解决方法——保存子问题的结果来避免重复计算,以空间换时间,提升算法效率。

大道至简。有时,朴素、直观的想法也是求解问题的一把利剑。联想到我们的学习、生活、工作以及与人相处等方面,简单一些,很多问题可能会大而化小,甚至可能不是问题了!

习题 5

扫一扫

习题