Chapter 5

动态规划

本章我们将介绍一种方法来求解一类问题，这种方法叫作动态规划（Dynamic Programming），当一个最优化问题满足“最优子结构”的性质时，我们便可以构造一个动态规划的求解策略来得到这个最优化问题的解。所谓最优子结构性质就是指当问题的最优解一旦确定，那么最优解中包含的所有子问题的解都是对应子问题的最优解，也就是全局最优蕴含局部最优。但是对于一个问题而言，它会有很多个的子问题，这些子问题可能重叠，也可能不重叠，但是每个子问题都能够演化到原问题，然而我们并不知道全局最优的解决方案是从哪个局部最优演化而来的，因此通常我们需要枚举所有的局部最优，然后把每个局部最优都往下演化成原问题，最后从所有的原问题当中选择最优的那个作为全局的最优解。本章接下来将通过若干个问题来演示如何应用动态规划来解决问题。

5.1 切钢条问题（线性动态规划）

我们接下来考虑这样一个问题，当前市场上不同长度的钢条价格不同，你现在有一个长度为 n 的钢条，问怎样切割能够卖得的总价格最高？

Table 5.1: 当前市场上钢条长度与价格之间的关系

长度	1	2	3	4	5	6	7	8	9	10
价格	1	5	8	9	10	17	17	20	24	30

例如：当前市场上钢条的长度和价格的关系如表5.1所示。如果此时我们有一根长度为 4 的钢条，我们将它切割成两段长度为 2 的小钢条，能够卖 10 元。除此之外，其它的任意一种切割方案，包括根本不切割的方案，均比 10 元要少。

怎么解决这个问题呢？首先我们考虑这样一种蛮力法，假设钢条长度为 4，那么我们最多切 3 刀，将它切为长度为 1 的 4 份等长度的钢条。对于这 3 刀，蛮力算法将枚举每一刀切或者不切两种情况，然后计算能够卖出的价格，最后求取所有方案的最大值就

是最优解。该蛮力法对于一个长度为 n 的钢条，时间复杂度为 $\Theta(2^n)$，是指数阶的，是非常慢的。接下来我们要考虑有没有一种解决方案，能够以更优的时间复杂度来解决该问题。

5.1.1 将解空间划分为子空间

其实不管怎么样去切割，切割的方案无非都会落入这么两种情况：（1）第一块切割的长度为 1，（2）第一块切割的长度不为 1 [1]。而对于第（2）种情况，即第一块切割长度不为 1，又可以划分为两种情况：（1）第一块切割长度为 2，（2）第二块切割长度不为 2。综合下来，就变成了这么三种情况：（1）第一块切割长度为 1，（2）第一块切割长度为 2，（3）第一块切割长度不为 2。紧接着，我们又可以把第一块切割长度不为 2 划分为第一块切割长度为 3 和不为 3。那么以此类推，最终切割的方案就是落入这么几种情况：第一块切割长度为 1，第一块切割长度为 2，第一块切割长度为 3，……，第一块切割长度为 n。如果我们将切割的方案组合成一个集合，即解空间，那么这个解空间就可以划分成上面的 n 个子空间，永远不可能有任何一种方案落入这个 n 个子空间以外，同时也不会有一个子空间是空集。

那么我们接下来要考虑每个子空间的解，我们可以从每个子空间当中挑选子空间内最大的价格的解决方案，再从这 n 个解决方案中选择最大的解决方案，就是原问题的最优解了。那么问题就转化为了：如何找出每个子空间中的最优解？

5.1.2 最优子结构

接下来我们要找每个子空间中的最优解了。首先我们先来看第 1 个子空间，即第一块切割长度为 1，也就是说剩余的钢条长度就为 $n-1$。接下来我们断言：对于第一块切割长度为 1 这个子空间中的所有方案，最优的那个方案是将剩余的长度为 $n-1$ 的钢条进行再切割，得到最高的价格的子方案，再加上第一块切割长度为 1 的钢条构造出的总方案。

我们如何证明这个断言的正确性呢？答案是通过反证法来证明。假设我们有一个第一块切割长度为 1 这个子空间中的最优切割方案，它是由将长度为 $n-1$ 的钢条进行切割但是价格并不是最高的子方案，加上第一块切割长度为 1 的钢条构造出的总方案。在这种情况下，如果我们将这个子方案替换为能够得到最高价格的那个子方案，那么此时再加上第一块长度为 1 的价格，就得到了更高的总价格！这与当前方案是第一块切割长度为 1 的子空间中的最优切割方案相矛盾。因此如果我们有一个第一块切割长度为 1 的子空间中的最优切割方案，那么它一定包含了长度为 $n-1$ 的钢条再切割并取得最高价格的子方案。

上面这种性质被称为*最优子结构*性质。根据这个性质，我们可以求得每一个子空间中的最优方案，最终再在这些最优子方案中选择价格最高的方案，就是总的最优方案了。

[1]这世界上只有两种人，第一种是从北京交通大学毕业的人，另一种是不是从北京交通大学毕业的人。

动态规划策略解决的最优化问题的结构应当具备最优子结构的性质。更加普遍的最优子结构的性质是指如果我们对于某一个问题有一个最优的解决方案，那么这个最优的解决方案当中包含的任意一个子问题的解也应当是最优的子方案。最优子结构性质的证明方法通常使用*剪贴法*，也就是说假设当前方案是最优的方案，但是其中包含的子方案并不是子问题的最优解决方案，那么我们将这个非最优的子方案“剪切”下来，然后“粘贴”上同一子问题的最优解决方案，能够得到比原问题当前解决方案更优的解决方案，于是就推出了矛盾，从而反证出最优方案包含的子问题的解决方案也一定是最优的。

我们再回过头来重新审视一下切钢条问题。假设我们有对一块长度为 n 的钢条的一个最优切割方案，那么该方案当中的任意一刀切下去，就可以得到的两块更小的钢条，这两块更小的钢条再切割的方案必须也是最优的，才能使得总的切割方案最优。否则，我们就可以分别将两块小钢条的切割方案都替换成它们的最优切割方案，从而得到更优的总价格，这与当前方案是最优的切割方案相矛盾了。于是切钢条问题存在最优子结构的性质。

5.1.3 填表法构造最优方案

证明了最优子结构的性质之后，我们就可以构造一个填表算法来求得一个长度为 n 的钢条的最优切割方案了，如 BEST-CUT 所示，该算法接受一个钢条的长度 n 和每个长度 j 对应的价格 $v[j]$。其中 $p[i]$ 维护的是长度为 i 的钢条经过切割所能得到的最大价格，$S[i]$ 维护的是对于长度为 i 的钢条，第一刀应该切多长。

```
BEST-CUT(n, v)
 1: Let p[1..n] be a new array, and set every element to 0.
 2: Let S[1..n] be a new array, and set every element to 0.
 3: for i = 1 to n do
 4:     p[i] = v[i]
 5:     S[i] = i
 6:     for j = 1 to i − 1 do
 7:         if p[i] < v[j] + p[i − j] then
 8:             p[i] = v[j] + p[i − j]
 9:             S[i] = j
10: while n > 0 do
11:     Print(S[n])
12:     n = n − S[n]
```

首先我们先证明算法的正确性，我们将通过强归纳公理来证明。第一步，确定谓词 $P(n)$，

- $P(n)$: 数组 $p[n]$ 中计算得到的是长度为 n 的钢条经过切割后获得的最高价格，同时 $S[n]$ 中记录的是对于长度为 n 的钢条，第一刀应该切多长。

第二步，证明基本情况 $P(1)$，由于长度为 1 的钢条不可再被切割，所以此时 $p[1] = v[1]$，$S[1] = 1$，然后算法的第 6 行不循环，于是命题成立。第三步，证明一般情况 $P(1) \wedge P(2) \wedge P(3) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$, 对于任意一个长度为 $n+1$ 的钢条，我们可以将切割方案划分为第一刀切割长度为 1，第一刀切割长度为 2，……，第一刀切割长度为 $n+1$ 这些子空间，算法第 4 行第 5 行初始化 $p[n+1]$ 和 $S[n+1]$ 为第一刀切割 $n+1$ 长度的钢条，第 6 行的循环则遍历剩余的每一个子空间的最优解，由归纳假设，长度为 i 且 $i < n+1$ 的钢条的最优切割方案的价格被维护在了数组 $p[i]$ 中，且该问题满足最优子结构的性质，于是对于每个第一刀切割长度为 j 的子空间，最优价格为 $v[j] + p[n+1-j]$，算法的第 7 行的判断使得遍历完每个子空间后，最高价格被记录在了 $p[n+1]$ 中，且第一刀的长度被记录在了 $S[n+1]$ 中。于是整个命题得证。

该算法的时间复杂度为 $\Theta(n^2)$，比蛮力法要快不少。

5.1.4 动态规划的基本步骤

接下来我们总结一下对于一个问题应用动态规划策略的基本步骤。

首先第一步我们要判断这个问题是不是适用动态规划策略，判断的依据便是这个问题是不是满足最优子结构的性质，如果该问题满足，那么我们就可以使用动态规划策略。

第二步，由最优子结构的性质，最优的解决方案包含的任意一个子问题的解决方案也必须是最优的，但是我们此时还不知道最优的解决方案是由哪个最优的子问题扩展来的，因此我们需要枚举每一个子问题，在每一个子问题上找到最优的解决方案，最后选取所有子问题的最优方案的最优方案。通常的做法是对解空间进行划分，将解空间划分为若干个子空间，然后递归地求解每个子空间上的最优解，最后求所有子空间最优解的最优解。某些文献中提到的动态规划的*状态转移方程*，便是由这一个步骤得到。

最后一步，就是根据上面的过程构造出具体的解决方案了，通常的做法是递归地记录最优方案是由哪个子空间推广得到的，然后将最终结果综合在一起即可。

5.1.5 自顶向下的解决方案

该问题也可以使用递归的方式来求解，BEST-CUT-RECURSIVE 描述了这一过程。

```
BEST-CUT-RECURSIVE(n, v, S)
1: if n == 1 then
2:     return v[n]
3: p = v[n]
4: for j = 1 to n − 1 do
5:     sub = BEST-CUT-RECURSIVE(n − j, v, S)
```

```
    if p < v[j] + sub then
        p = v[j] + sub
        S[n] = j
return p
```

该算法的正确性可以由强归纳公理得到，这里我们不再展开写了。我们将这种方式称作*自顶向下的解决方案*，因为它是再试图解决长度为 n 的问题时，去求解长度小于 n 的解决方案。填表法与之对应的，称作*自底向上的解决方案*，因为它是先求解长度小于 n 的解决方案，然后再去构造长度为 n 的解决方案。

但是自顶向下的解决方案通常有个弊端，就是会重复求解子问题。就和斐波那契数列相似的，如果我们调用 BEST-CUT-RECURSIVE(n, v, S) 去求解长度为 n 的钢条的切割方案，那么它会求解长度为 $n-1, n-2, n-3, \cdots, 1$ 的解决方案，在求解长度为 $n-1$ 的解决方案时，它会再次重复的求解长度为 $n-2, n-3, \cdots, 1$ 的解决方案。这样递归下来，自顶向下的解决方案重复求解了很多相同的子问题，因此花费了不必要的时间复杂度。然而自底向上的解决方案却不存在这个问题。因此，避免动态规划策略重复的求解子问题的一种解决方案是应用自底向上的填表法，但有的时候填表法仍然不好写，此时我们可以使用自顶向下的递归法，但是需要注意的是，我们需要给自顶向下的递归法提供一个备忘录：每次递归求解一个子问题得到答案后，我们便在备忘录上记录下当前子问题的解，在此之后，如果再次递归求解到了相同的子问题，如果备忘录上存在该子问题的解，我们就直接返回这个解，而不去递归的重复计算了，这样一来，每个子问题便最多只被求解一次，于是它的时间复杂度的上界就和填表法一样了。

5.2 0-1 背包问题

我们接下来介绍 0-1 背包问题，来强化我们之前的动态规划方案的设计方法。假设我们有 n 个物品，其中第 i 个物品的体积是 $w[i]$，它的价值为 $v[i]$，现在给你一个体积为 W 的背包，问你要怎么装才能使得总的价值最大？

例如，假设 $W = 11$，每种物品的体积和价值如表5.2所示。此时，最优的选择方案是选取 3 号和 4 号两种物品，能够获得最大的价值 40，除此之外，任何一种方案的总价值都是比它小的。

Table 5.2: 物品的体积与价值

编号	1	2	3	4	5
体积	1	2	5	6	7
价值	1	6	18	22	28

5.2.1 最优子结构

0-1 背包问题存在最优子结构的性质。假设我们有一个最优的选择方案，那么对于子问题的选择方案一定是最优的。否则我们可以将子问题的方案替换为最优的方案，再拼接上其它不变的子方案，便能够得到更大的价值，这与当前方案是最优方案相矛盾。

例如：对于上面的问题而言，$\{3,4\}$ 是最优解，那么 $\{4\}$ 一定是背包大小为 6 时的最优方案，同时 $\{3\}$ 也一定是背包大小为 5 时的最优方案。否则假设 $\{4\}$ 如果不是背包大小为 6 时的最优方案，那么就一定存在其它方案能够得到总体积在 6 以内但是总价值高于 22，我们将该方案与另一半，即背包大小为 5 的解决方案 $\{3\}$ 组合在一起，便得到了更优的解决方案，这与 $\{3,4\}$ 是最优方案相矛盾。因此，$\{4\}$ 一定是背包大小为 6 时的最优方案，同时 $\{3\}$ 也一定是背包大小为 5 时的最优方案。

5.2.2 将解空间划分为子空间

接下来我们要将问题的解空间划分为子空间，然后对于每个子空间利用最优子结构的性质递归的求解这个子空间的最优方案，最终再将子空间的最优方案们综合起来得到总体的最优方案。

我们可以考虑，所有的选取方案一定能够划分成为这么几个子空间:

(1) 选择物品的最大编号是 n。

(2) 选择物品的最大编号不是 n。

那么原问题的最优方案就是这些子问题分别求出的最优方案中最优的那个。

那么我们假设 $f(n,W)$ 是从前 n 个物品中选择出总体积为 W 的最大价值，结合最优子结构的特性，我们有:

$$\begin{aligned} f(n,W) &= \max\{\text{选择物品的最大编号是}n\text{的最大价值}, \text{选择物品的最大编号不是}n\text{的最大价值}\} \\ &= \max\{f(n-1,W-w[n])+v[n], f(n-1,W)\} \end{aligned}$$

5.2.3 自顶向下的解决方案

有了上面的分析，我们便可以很轻松的构造出如 BACKPACK-RECURSIVE 函数所示的递归算法了，在调用该算法之前，我们需要将备忘录 f 的每个元素都初始化为-1，同时我们要将方案数组 S 中的每个元素初始化为-1，调用 BACKPACK-RECURSIVE(n, W) 即可得到最高的价值。

BACKPACK-RECURSIVE(n, W)

1: if $f[n][W] \neq -1$ then
2: return $f[n][W]$
3: if $n == 0$ then

```
    f[n][W] = 0
    return 0
a = −1
if W ≥ w[n] then
    a = BACKPACK-RECURSIVE(n-1, W-w[n]) + v[n]
b = BACKPACK-RECURSIVE(n-1, W)
if a > b then
    S[n][W] = 1
    ans = a
else
    S[n][W] = 2
    ans = b
f[n][W] = ans
return ans
```

然后我们可以调用 PRINT-BEST-CHOICE(n, W) 打印出最优的选择方案。

```
PRINT-BEST-CHOICE(n, W)
if n == 0 then return
if S[n][W] == 1 then
    PRINT-BEST-CHOICE(n-1, W-w[n])
    print n
else
    PRINT-BEST-CHOICE(n-1, W)
```

5.2.4 自底向上的填表算法

同样地，我们也可以将自顶向下的带备忘录的递归算法改写成自底向上的填表算法。如 BACKPACK 所示，在调用该算法之前，我们需要首先将数组 f 的所有值初始化为 0。

```
BACKPACK(n, W)
for i = 1 to n do
    for j = 1 to W do
        f[i][j] = f[i − 1][j]
        S[i][j] = 2
        if j ≥ w[i] and f[i][j − w[i]] + v[i] > f[i][j] then
```

6: $f[i][j] = f[i][j - w[i]] + v[i]$
7: $S[i][j] = 1$
8: PRINT-BEST-CHOICE(n, W)
9: return $f[n][W]$

该算法的正确性可以由强归纳公理证明，时间复杂度为 $\Theta(nW)$。

5.3 最长上升子序列问题

最长上升子序列问题是指给定一个数组 $a[1..n]$,请找出数组中的最长序列 $a[i_1], a[i_2], \cdots, a[i_k]$，其中 $i_1 < i_2 < \cdots < i_k$ 且 $a[i_1] < a[i_2] < \cdots < a[i_k]$。

例如我们现在有个数组 $a = (1, 8, 2, 9, 3, 10, 4, 5)$，它的最长上升子序列是 $a[1] = 1, a[3] = 2, a[5] = 3, a[7] = 4, a[8] = 5$。

该问题存在最优子结构的性质,即假设我们现在有一个最长上升子序列 $a[i_1], a[i_2], \cdots, a[i_k]$，那么这个子序列的子段 $a[i_1], a[i_2], \cdots, a[i_{k-1}]$ 是子数组 $a[1], a[2], \cdots, a[i_k - 1]$ 的一个最长上升子序列。如果它不是，那么就可以找到最长的那个替换掉这些而使得总的最长上升子序列长度更长，于是推导出了矛盾，因此该问题存在最优子结构的性质。

接下来我们对解空间进行划分。所有的解决方案无非划分成两个部分，即:（1）最后一个元素是 $a[n]$ 和;（2）最后一个元素不是 $a[n]$。再进一步划分可以划分为:（1）最后一个元素是 $a[n]$;（2）最后一个元素是 $a[n-1]$;（3）最后一个元素是 $a[n-2]$; ……;（n）最后一个元素是 $a[1]$。如果我们设 $f[i]$ 代表最后一个元素是 $a[i]$ 的最长上升子序列的长度，那么最长上升子序列的长度就是 $\max_{1 \le i \le n} f[i]$。

由最优子结构的性质，对于 $f[i]$ 的求解，就有:

$$f[i] = \max_{j \in [1,i), a[j] < a[i]} f[j] + 1$$

于是我们可以构造如 LIS 所示的填表算法。

LIS(a)
1: Initialize $S[1..n]$ to 0
2: $f[1] = 1$
3: for $i = 2$ to n do
4: $f[i] = 1$
5: $S[i] = 0$
6: for $j = 1$ to $i - 1$ do
7: if $a[j] < a[i]$ and $f[i] < f[j] + 1$ then
8: $f[i] = f[j] + 1$
9: $S[i] = j$

```
10: ans = 1
11: index = 1
12: for i = 2 to n do
13:     if ans < f[i] then
14:         ans = f[i]
15:         index = i
16: PRINT-LIS(S, index)
17: return ans
```

```
PRINT-LIS(S, index)
1: if index == 0 then return
2: PRINT-LIS(S, S[index])
3: Print(index)
```

该算法的正确性可以由强归纳公理证明，时间复杂度为 $\Theta(n^2)$。

5.4 最大子数组和问题

最大子数组和问题是给定一个数组 $a[1..n]$, 选择一个下标的二元组 (i, j), 使得 $a[i]+a[i+1]+\cdots+a[j]$ 最大。例如对于数组 $a=(-2,1,-3,4,-1,2,1,-5,4)$，最大子数组和是 $\sum a[4..7]=6$。

该问题存在最优子结构的性质，因为对于一个最优方案 (i, j) 而言，对于任意一个 $k, i \le k \le j$ 来说，$a[i..k]$ 一定是 $a[1..k]$ 中包含 $a[k]$ 的和最大的子数组，同时 $a[k+1..j]$ 一定是 $a[k+1..n]$ 中包含 $a[k+1]$ 的和最大的子数组。否则我们就可以分别替换成更大的子数组，加起来使得总的子数组的和更大。

接下来我们来划分解空间，最大的子数组的方案无非有这么几种，（1）空数组，此时和为 0、（2）以 $a[1]$ 结尾的子数组、（3）以 $a[2]$ 结尾的子数组、……、（n+1）以 $a[n]$ 结尾的子数组。设 $f[i]$ 为以 i 结尾的和最大的子数组，则此时总的最大子数组和为 $\max_{i\in[1,n]} f[i]$。以 $a[i]$ 结尾的子数组无非分为这么两种情况，一种是以 $a[i-1]$ 结尾的子数组后面加上 $a[i]$，另一种是不以 $a[i-1]$ 结尾的子数组后面加上 $a[i]$，即仅有 $a[i]$ 一个元素。于是由最优子结构的性质，我们有:

$$f[i] = \max(f[i-1]+a[i], a[i])$$

于是我们便可以构造出一个如 MAX-SUM-SUBARRAY 所示的填表算法来求出最大子数组和的方案。

```
MAX-SUM-SUBARRAY(a)
```

```
1: f[1] = a[i]
2: S[1] = 1
3: for i = 2 to n do
4:     f[i] = a[i]
5:     S[i] = i
6:     if f[i − 1] + a[i] > f[i] then
7:         f[i] = f[i − 1] + a[i]
8:         S[i] = S[i − 1]
9: ans = 0
10: index = −1
11: for i = 1 to n do
12:     if ans < f[i] then
13:         ans = f[i]
14:         index = i
15: if ans ≠ 0 then
16:     Print(S[index], index)
17: return ans
```

该算法的时间复杂度为 $\Theta(n)$。

5.5 最长公共子序列问题

最长公共子序列问题是指给定两个数组 $A[1..n]$ 和 $B[1..m]$，请你找出一个最长的子序列 $C[1..k]$，使得 C 同时是 A 和 B 的子序列。

该问题存在最优子结构。对于任意一个最长公共子序列 $C[1..k]$，其中 $C[k]$ 对应的是 $A[p]$ 和 $B[q]$，那么 $C[1..k-1]$ 一定是 $A[1..p-1]$ 和 $B[1..q-1]$ 的最长公共子序列。否则我们可以找出 $A[1..p-1]$ 和 $B[1..q-1]$ 更长的最长公共子序列，加上 $C[k]$ 得到比当前 $A[1..n]$ 和 $B[1..m]$ 最长公共子序列更长的最长公共子序列，于是形成了矛盾，因此最优子结构性质成立。

下面我们划分解空间。对于解空间中的所有方案，我们可以取任意的 p 和 q，然后分为两个子空间:（1）最后一个公共子序列的元素是 $A[p]$ 和 $B[q]$ 配成了一对，此时必须有 $A[p] = B[q]$。（2）最后一个公共子序列的元素不是 $A[p]$ 和 $B[q]$ 配成一对。子空间（2）可以继续划分:（1）最后一个公共子序列的元素一定没有 $A[p]$ 的参与。（2）最后一个公共子序列的元素一定没有 $B[q]$ 的参与。于是整个解空间就划分为了三个子空间:（1）最后一个公共子序列的元素是 $A[p]$ 和 $B[q]$ 配成了一对，此时必须有 $A[p] = B[q]$;（2）最后一个公共子序列的元素一定没有 $A[p]$ 的参与;（3）最后一个公共子序列的元素一定没有 $B[q]$ 的参与。然后取这三个子空间的最长公共子序列即可。注意，子空间（2）

和（3）之间有重叠，重叠的部分是 “最后一个公共子序列的元素既没有 $A[p]$ 的参与，又没有 $B[q]$ 的参与”，但是这三个子空间取并集以后仍然是完整的解空间，因此即使两个子空间有重叠，仍然不影响原本的解空间的最大值（或者对于有些问题是最小值）的计算。

接下来我们设 $f[p][q]$ 为 $A[1..p]$ 和 $B[1..q]$ 的最长公共子序列，那么综合最优子结构的性质，对于第 1 个子空间，有:

$$f[p][q] = f[p-1][q-1] + 1$$

对于第 2 个子空间，有:

$$f[p][q] = f[p-1][q]$$

对于第 3 个子空间，有:

$$f[p][q] = f[p][q-1]$$

于是我们可以写出例如 LCS 的填表算法。

```
LCS(A, B)
 1: Let f[0..n][0..m] be a new array.
 2: Let S[1..n][1..m] be a new array.
 3: for i = 1 to n do
 4:     for j = 1 to m do
 5:         f[i][j] = f[i − 1][j]
 6:         S[i][j] = 1
 7:         if f[i][j] < f[i][j − 1] then
 8:             f[i][j] = f[i − 1][j − 1]
 9:             S[i][j] = 2
10:         if A[i] == B[j] and f[i − 1][j − 1] + 1 > f[i][j] then
11:             f[i][j] = f[i − 1][j − 1] + 1
12:             S[i][j] = 3
13: PRINT-LCS(S, n, m)
14: return f[n][m]
```

```
PRINT-LCS(S, n, m)
1: if n == 0 or m == 0 then return
2: if S[n][m] == 1 then
3:     PRINT-LCS(S, n − 1, m)
4: else if S[n][m] == 2 then
5:     PRINT-LCS(S, n, m − 1)
```

```
6: else
7:     Print(A[n])
8:     PRINT-LCS(S, n − 1, m − 1)
```

该算法的时间复杂度为 $\Theta(n \times m)$。

5.6 最短编辑距离问题

最短编辑距离是指对于两个字符串 $A[1..n]$ 和 $B[1..m]$，每次操作可以任意选取一个字符串，做以下三种操作之一:

(1) 插入一个字符。

(2) 删除一个字符。

(3) 改变一个字符。

编辑距离是指对字符串 A 和 B 做上面的操作，使得两个字符串相同的操作次数。最短编辑距离问题希望你求出最短的编辑距离。例如: $A = horse, B = ros$，最短编辑距离为 3，因为我们可以做以下三步操作:

(1) $horse \to rorse$，即将第一个 h 改为 r。

(2) $ros \to rose$，即在 ros 后面添加一个 e。

(3) $rorse \to rose$，即删除第二个 r。

我们首先断言: 插入一个字符等价于从另一个字符串中删除一个字符。首先，我们重新排列所有操作，将插入一个字符排在操作的最后。此时我们选取最后一个插入操作，不妨设在 $A[1..n]$ 的位置 k 处插入一个字符 c，使得它变为 $A[1..k-1] + c + A[k..n]$，由于这是最后一个插入操作，那么我们有 $B = B[1..k-1] + c + B[k+1..n+1]$，并且 $A[1..k-1] = B[1..k-1]$，$A[k..n] = B[k+1..n+1]$。此时我们可以将这个插入操作变更为在 B 字符串中删除第 k 个字符，使得两个字符串相同。然后我们以此类推，再选择剩余操作中的最后一个插入操作，它也可以等价于在另一个字符串中删除一个字符，最终所有插入操作都可以被替换为删除操作。

因此，我们的编辑距离便由一开始的三种操作简化为了两种操作，即:

(1) 删除一个字符。

(2) 改变一个字符。

接下来我们证明该问题满足最优子结构的性质。假设我们有一个最优编辑方案，它将 $A[1..n]$ 和 $B[1..m]$ 编辑为相等的两个字符串。那么我们选取针对字符串的操作中的最大下标的那一个，于是我们有两种可能，第一种可能是这是一个删除操作，第二种可能是这是一个修改操作。

首先我们考虑删除操作，如果这是一个删除操作，说明我们接下来要将该字符删除后的字符串与另一个字符串操作后相同的操作次数是最小的。否则我们就有另外的操作，使得删除该字符后的字符串与另一个字符串做若干次操作后相同的操作次数更小，然后我们再删除这个字符，使得总的操作次数变得更小了，于是得到了矛盾。

接下来我们考虑修改操作，如果这时一个修改操作，说明当我们操作到这一步时两个字符串的长度应该是相等的了。那么我们这时候选取修改操作修改的字符和另一个字符串中与之对应的字符，把它们去掉，剩余的两个子串进行操作的次数必须是最小的，否则我们有一种对剩余的两个子串更小的操作方案，操作完后将被去掉的字符放回后修改，使得总的操作次数更小了，于是又得到了矛盾。因此总而言之，该问题满足最优子结构的性质。

再然后我们将解空间划分为若干个子空间。解空间可以划分为:（1）删除 $A[n]$,（2）删除 $B[m]$,（3）修改 $A[n]$ 或者 $B[m]$,（4）对 $A[n]$ 和 $B[m]$ 不操作。其中（3）和（4）两个子空间只存在一个，当 $A[n] \neq B[m]$ 时存在解空间（3），当 $A[n] = B[m]$ 时存在解空间（4）。对于子空间（1），最短编辑距离为 $f[n-1][m]+1$；对于子空间（2），最短编辑距离为 $f[n][m-1]+1$；对于子空间（3），最短编辑距离为 $f[n-1][m-1]+1$；对于子空间（4），最短编辑距离为 $f[n-1][m-1]$。

此时我们就可以构造一个如 EDIT-DISTANCE 所示的填表算法了。

EDIT-DISTANCE(A, B)

```
1:  Let f[0..n][0..m] be a new array.
2:  for i = 1 to n do
3:      f[i][0] = i
4:  for i = 1 to m do
5:      f[0][i] = i
6:  for i = 1 to n do
7:      for j = 1 to m do
8:          f[i][j] = min(f[i − 1][j], f[i][j − 1]) + 1
9:          if A[i] == B[j] then
10:             f[i][j] = min(f[i][j], f[i − 1][j − 1])
11:         else
12:             f[i][j] = min(f[i][j], f[i − 1][j − 1] + 1)
13: return f[n][m]
```

该算法的时间复杂度为 $\Theta(n \times m)$。

接下来我们来研究另一种思考方式。这种思考方式更加抽象，但是如果能够理解，则能够为该问题提供一种更加直观的方法。在这种思考方式中，我们不将插入字符操作等价于从另一个字符串中的删除操作。该问题符合最优子结构性质，如果说对于一个最优方案而言，我们抽取任何一个中间的操作，假设该操作是删除操作，那么除了这个删除操作之外的操作必须是最优的，否则可以通过剪贴法推出矛盾。如果是修改操作和增加操作，那么我们需要把它们对应的另一个字符串中的字符一起去掉，然后剩余的两个字符串必须用最少的操作使得它们相同，否则仍然可以通过剪贴法推出矛盾，因此整个问题符合最优子结构。

接下来我们将解空间划分为子空间，子空间为:（1）删除 A 字符串的最后一个字符或者删除 B 字符串的最后一个字符，（2）在 A 字符串后面添加一个字符或者在 B 字符串后面添加一个字符，（3）修改 A 字符串或者修改 B 字符串，（4）不对 A 字符串和 B 字符串进行操作。设 $f[n][m]$ 为字符串 $A[1..n]$ 和 $B[1..m]$ 的最短编辑距离，那么对于子空间（1），删除 A 字符串的最后一个字符的最短编辑距离应该是 $f[n-1][m]+1$，而删除 B 字符串的最后一个字符的最短距离就是 $f[n][m-1]+1$。对于子空间（2），在 A 字符串后面添加一个字符，等于先将 $A[1..n]$ 和 $B[1..m-1]$ 修改成一样的，然后再在 A 字符串最后添加 $B[m]$，于是最短编辑距离为 $f[n][m-1]+1$，类似的，在 B 字符串后面添加一个字符的最短编辑距离为 $f[n-1][m]+1$。对于子空间（3），最短编辑距离为 $f[n-1][m-1]+1$，而对于子空间（4），最短编辑距离则为 $f[n-1][m-1]$。其中，子空间（3）和子空间（4）只有一种情形下存在。当且仅当 $A[n] \neq B[m]$ 时存在子空间（3），当且仅当 $A[n] == B[m]$ 时存在子空间（4）。于是整个问题的最短编辑距离，便是各个子空间的最小值。

如果你将上面的思路写成代码，会发现代码和 EDIT-DISTANCE 一模一样，这只是另外一种思考方式罢了，但是最后的求解过程都是一样的。

5.7 配对问题（集合上的动态规划）

之前我们讨论的都是线性结构上的动态规划，这一类动态规划的特征是划分子空间时仅针对一个元素。但是当划分子空间时的依据是多个元素时，便不能够使用线性结构的动态规划了。本小节我们举一个例子，它的解空间的划分依据不是针对一个元素，这种情况下，我们将设计一个集合上的动态规划的解决方案。

假设我们在一张地图上有 n 个人和 m 辆自行车，其中 $n \leq m$，我们需要给每个人都分配一辆自行车。人只能沿着平行于坐标轴的方向移动，我们的目标是希望所有人去找到他的自行车的移动距离是最短的。换句话说，我们希望得到一种分配方案，使得每个人和他拥有的自行车的曼哈顿距离之和最短。所谓两点 (x_1, y_1) 和 (x_2, y_2) 之间的曼哈顿距离是指 $|x_1 - x_2| + |y_1 - y_2|$。

如图5.1所示，最优的分配方案是给 0 号人分配 0 号自行车，给 1 号人分配 1 号自

行车。总的曼哈顿距离为 6，是所有分配方案中最短的。

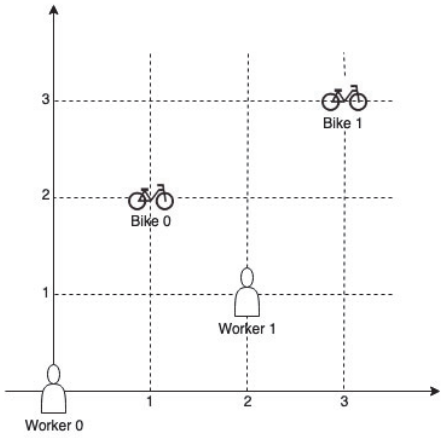

Figure 5.1: 2 个人和 2 辆自行车

我们可以考虑这样一种蛮力法：取 m 辆自行车的全排列，对于每一种排列，第 i 辆自行车分配给第 i 号人，这样枚举完所有分配方案后，选择最小的分配方案即可。这样下来的时间复杂度是 $\Theta(n \times m!)$，我们希望找一个更快一点的算法。

5.7.1 最优子结构

首先我们先来探究这个问题满不满足最优子结构的性质，答案是满足。假设我们使用 A 表示人的集合，B 表示自行车的集合，(a, b) 表示“给第 a 个人分配第 b 辆自行车”，那么对于一个最优分配方案 $P = \{(a_1, b_1), (a_2, b_2), \cdots, (a_n, b_n)\}$，我们选取它的任意一个子方案 $P' = \{(a_{i_1}, b_{i_1}), (a_{i_2}, b_{i_2}), \cdots, (a_{i_k}, b_{i_k})\}$，这个子方案的曼哈顿距离和是固定的。记 $A' = \{a_{i_1}, a_{i_2}, \cdots, a_{i_k}\}$，$B' = \{b_{i_1}, b_{i_2}, \cdots, b_{i_k}\}$。我们将这些人和自行车从集合中刨去，得到的剩余的人和未分配的自行车的集合 $A - A'$ 和 $B - B'$。那么接下来给 $A - A'$ 集合中的人分配 $B - B'$ 集合中的自行车的曼哈顿距离必须是最短的，否则我们可以选择一个使得曼哈顿距离和更短的分配方案，把它和 P' 结合在一起，使得总方案的曼哈顿距离更小了推出了，于是推出了矛盾，因为此时 P 不再是最优的分配方案了。

5.7.2 将解空间划分为子空间

接下来我们将解空间划分为子空间。无论我们如何安排，分配方案无非是落入下列情形之一：

(1) 给第 n 个人分配第 1 辆自行车。

(2) 给第 n 个人分配不是第 1 辆自行车。

其中我们还可以继续展开第（2）个子空间，最终我们可以得到下面这 m 种子空间：（1）给第 n 个人分配第 1 辆自行车、（2）给第 n 个人分配第 2 辆自行车、……、（m）给第 n 个人分配第 m 辆自行车。总的最优分配方案就是这 m 个子空间的最优解中的最优解。

设 $f(i,S)$ 是给前 i 个人从 S 中选一辆自行车分配后的最短曼哈顿距离和，由最优子结构的性质，有:

$$f(i,S) = \min_{k\in[1,m]} f(i-1, S-b_k) + dis(a_i, b_k)$$

最终最小的曼哈顿距离和是 $f(n,B)$。

5.7.3 自顶向下的解决方案

于是我们可以构造一个如 BEST-DISTRIBUTE 所示的自顶向下的递归解决方案，在调用该函数之前需要将 f 的所有值初始化为-1:

```
BEST-DISTRIBUTE(i, B)
 1: if i == 0 then
 2:     return 0
 3: if f[i][B] ≠ −1 then
 4:     return f[i][B]
 5: ans = −1
 6: for each b ∈ B do
 7:     dis = BEST-DISTRIBUTE(i − 1, B − {b})+|a_i.x − b.x| + |a_i.y − b.y|
 8:     if ans == −1 or ans > dis then
 9:         ans = dis
10:         S[i][B] = b
11: f[i][B] = ans
12: return ans
```

```
PRINT-BEST-DISTRIBUTE(i, B, S)
1: if i == 0 then return
2: PRINT-BEST-DISTRIBUTE(i − 1, B − {S[i][B]})
3: Print(i, S[i][B])
```

其中 B 可以使用整数位图的形式来表示，该算法的时间复杂度是 $\Theta(n \times 2^m)$，比蛮力法要快一些。你可以尝试着将自顶向下的解决方案修改为自底向上的。

5.8 矩阵连乘问题（区间上的动态规划）

最后我们再介绍一类问题，这类问题被称作区间上的动态规划。考虑我们有 n 个矩阵相乘，即 $B = \prod_{i=1}^{k} A_i$，问如何给它添加括号，使得总的乘法次数最少?

例如: 假设我们有 3 个矩阵, 分别是 $A_{10\times100}, B_{100\times5}, C_{5\times50}$, 有两种加括号的方法: 第一种是 $((AB)C)$, 这种方法需要的乘法次数是 7500 次, 第二种是 $(A(BC))$, 这种方法需要的乘法次数是 75000 次, 相差 10 倍!

我们可以考虑一种蛮力法, 即枚举所有加括号的方案, 最后选取乘法次数最少的方案即可。这种方案的次数在数学上称为 “卡特兰数”, 是指数阶的, 我们希望找一种更快的算法来求出具有最少的乘法次数的括号方案。

5.8.1 最优子结构

该问题满足最优子结构的性质。我们考虑一种最优的加括号的方案, 它等价于将连乘递归的拆解成左右两段。那么我们考虑最后一次的两个矩阵相乘, 得到这两个矩阵的连乘方案它们的乘法次数必须是最少的, 否则我们就可以将左边的矩阵的计算方案替换成连乘次数更少的方案, 或者将右边的矩阵的计算方案替换成连乘次数更少的方案, 使得总体的连乘次数更少, 于是得到了矛盾。因此该问题满足最优子结构的性质。

5.8.2 将解空间划分为子空间

无论怎样加括号, 最终会落到以下几种情况之一: (1) 最外层的括号将矩阵分成 1 个和 $n-1$ 个两组; (2) 最外层的括号将矩阵分成 2 个和 $n-2$ 个两组; (3) 最外层的括号将矩阵分成 3 个和 $n-3$ 个两组; ……; (n-1) 最外层的括号将矩阵分成 $n-1$ 个和 1 个两组。于是总的最少连乘次数便是这 (n-1) 个子空间的最少炼成次数的最小值。

由最优子结构的性质, 假设 $f[1][n]$ 是从第 1 个矩阵到第 n 个矩阵的最少连乘方案的连乘次数, 那么我们有:

$$f[1][n] = \min_{k\in[1,n-1]}\{f[1][k] + f[k+1][n] + p_1 \times q_k \times q_n\}$$

其中第 j 个矩阵的维度是 $p_j \times q_j$, 并且 $f[i][k]$ 和 $f[k+1][n]$ 可以递归的计算得到。

5.8.3 自底向上的填表算法

由上面的分析, 我们可以得到如果要计算的矩阵的连乘方案中矩阵的个数是 len, 那么我们需要先计算出所有小于 len 的连乘方案。

于是我们可以给出 MATRIX-MULTIPLICATION 所示的自底向上的填表算法。

MATRIX-MULTIPLICATION(n)

```
1: Let f[1..n][1..n] be a new array.
2: for i = 1 to n do
3:     f[i][i] = 0
4: for len = 2 to n do
```

```
    for i = 1 to n do
        j = i + len − 1
        if j > n then
            continue
        f[i][j] = ∞
        for k = i to j − 1 do
            if f[i][k] + f[k + 1][j] + p[i] × q[k] × q[j] < f[i][j] then
                f[i][j] = f[i][k] + f[k + 1][j] + p[i] × q[k] × q[j]
                S[i][j] = k
PRINT-MATRIX-MULTIPLIATION(1, n, S)
return f[1][n]
```

PRINT-MATRIX-MULTIPLICATION(i, j, S)

```
if i == j then return
PRINT-MATRIX-MULTIPLICATION(i, S[i][j])
PRINT-MATRIX-MULTIPLICATION(S[i][j]+1, j)
Print(i, S[i][j])
Print(S[i][j]+1, j)
```

该算法的时间复杂度是 $\Theta(n^3)$，是多项式阶的，比指数阶的要快不少。

（作业：只对 A 字符串做三种操作，使得 A 和 B 相同的最短编辑距离/小偷不能连续偷两个店铺、线性的和环形的）