

清风"数学建模算法讲解"全套视频, 单人购买观看仅需58元

注: 本课件配套的视频可在bilibili网站上面免费观看, 这是数学建模清风老师讲解的公开课系列。

视频观看地址: <https://www.bilibili.com/video/BV1tp4y167c5>

动态规划入门系列课程全集 (数学建模清风主讲)

4991播放 · 31弹幕 · 2020-11-01 11:41:47



由于作者水平所限, 课件中存在的不妥之处, 敬请读者不吝赐教。
如需在博客或者论坛中引用本课件中的内容, 请注明来源, 格式如下:

参考资料: 数学建模清风: 动态规划课程 <https://www.bilibili.com/video/BV1tp4y167c5>

 数学建模学习交流

关注微信公众号"数学建模学习交流"获取更多优质资料

本课件配套视频: <https://www.bilibili.com/video/bv1tp4y167c5>

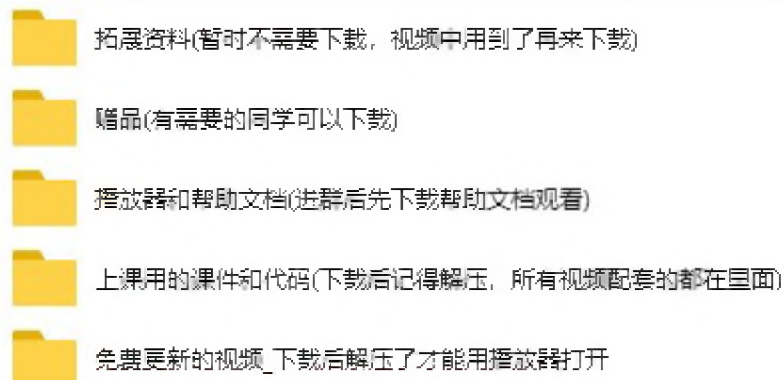
动态规划

(Dynamic Programming, 简称DP)

动态规划是运筹学的一个分支, 通常用来解决多阶段决策过程最优化问题。动态规划的基本想法就是将原问题转换为一系列相互联系的子问题, 然后通过逐层递推来求得最后的解。目前, 动态规划常常出现在各类计算机算法竞赛(例如ACM, 蓝桥杯)或者程序员笔试面试中, 在数学建模比赛中出现的较少, 但这个算法的思想在生活中非常实用, 学完本节后对你解决实际问题的思维方式一定很有启发。

温馨提示

- (1) 视频中提到的附件可在**售后群的群文件**中下载。
包括讲义、代码、我视频中推荐的资料等。



(2) 关注我的**微信公众号《数学建模学习交流》**, 后台发送**“软件”**两个字, 可获得常见的建模软件下载方法; 发送**“数据”**两个字, 可获得建模数据的获取方法; 发送**“画图”**两个字, 可获得数学建模中常见的画图方法。另外, 也可以看看公众号的历史文章, 里面发布的都是对大家有帮助的技巧。

(3) **购买更多优质精选的数学建模资料**, 可关注我的微信公众号《数学建模学习交流》, 在后台发送**“买”**这个字即可进入店铺进行购买。

(4) 视频价格不贵, 但价值很高。单人购买观看只需要**58元**, 和另外两名队友一起购买人均仅需**46元**, 视频本身也是下载到本地观看的, 所以请大家**不要侵犯知识产权**, 对视频或者资料进行二次销售。

动态规划

动态规划 (Dynamic Programming, DP) 是运筹学的一个分支, 通常用来求解决策过程最优化问题。

本节我们将主要以例题的形式来进行讲解, 通过例题来带大家体会动态规划这种思想, 并建立起求解动态规划问题的框架。

本节涉及到的例题如下:

1. 斐波那契数列
2. 打家劫舍 (力扣198)
3. 礼物的最大价值 (剑指 Offer 47)
4. 零钱兑换 (力扣 322)
5. 01背包问题(01 Knapsack problem)
6. 求硬币兑换的方案数 (国赛1992 年真题: 实验数据分解)

最后, 我也留了几道课后习题并提供了相应的参考答案供大家练习。



动态规划总结

学完后面的内容后再回过头来看这一页

动态规划是用来解决最优问题（也可以用来求方案数）的一种方法，或者说是一种思想。而解决问题的过程，需要经历多个阶段，每一阶段都可以看成一个子问题，每个子问题都对应着一组状态。

使用动态规划一般会经历四个步骤：

(1) 定义原问题和子问题。子问题是和原问题相似但规模较小的问题。

(2) 定义状态。这里的状态大家可以认为就是某个函数的自变量，根据状态中包含的参数个数的不同，我们在编程时设置的DP数组的维度就不同。每个状态中的参数通常都能对应DP数组中某个元素的下标，而DP数组的元素就是这个状态对应的子问题的求解结果。

(3) 寻找状态转移方程。这一步往往是最难的，大家需要找到关于状态之间的某种转移关系，这个关系往往是一个递推式子，根据这个递推式我们才能一步一步计算出DP数组里面的元素。另外别忘了确定边界条件，也就是我们递推的初始条件。另外，如果一个问题能用动态规划方法求解，需要满足最优子结构和无后效性，我在讲解例题时回避掉了这一点，但这不代表它们并不重要，因为这一块严格证明非常困难，后续如果你有兴趣可以学习专门的算法课程。

(4) 编程实现。如果前三步大家逻辑都理顺了，那么编程不是大的问题。我们无非就是先初始化一个DP数组，再结合边界条件计算DP数组中的初始值，最后再利用循环来对DP数组进行迭代。一般DP数组中最后那个元素就是我们要解决的原问题的答案。

最后，动态规划问题需要多练，我这一讲涉及到的问题不算困难，只起到抛砖引玉的作用。大家真的对算法这一块感兴趣的话可以去B站搜索“数据结构”或者“算法”这种关键词，能学到很多计算机科班同学的课程。尽管在数模比赛中动态规划很少用到，但我相信大家学完这一节后思维一定很有启发，有兴趣的同学可以在“力扣”网站上找到更多关于动态规划的练习题。

从斐波那契数列开始

斐波那契数列（又称为兔子数列）没有涉及到优化，所以严格来说不是动态规划问题，这里我们希望通过这个例子来引出动态规划中的一些重要概念。

百度百科上的定义：

斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：

0、1、1、2、3、5、8、13、21、34、55.....

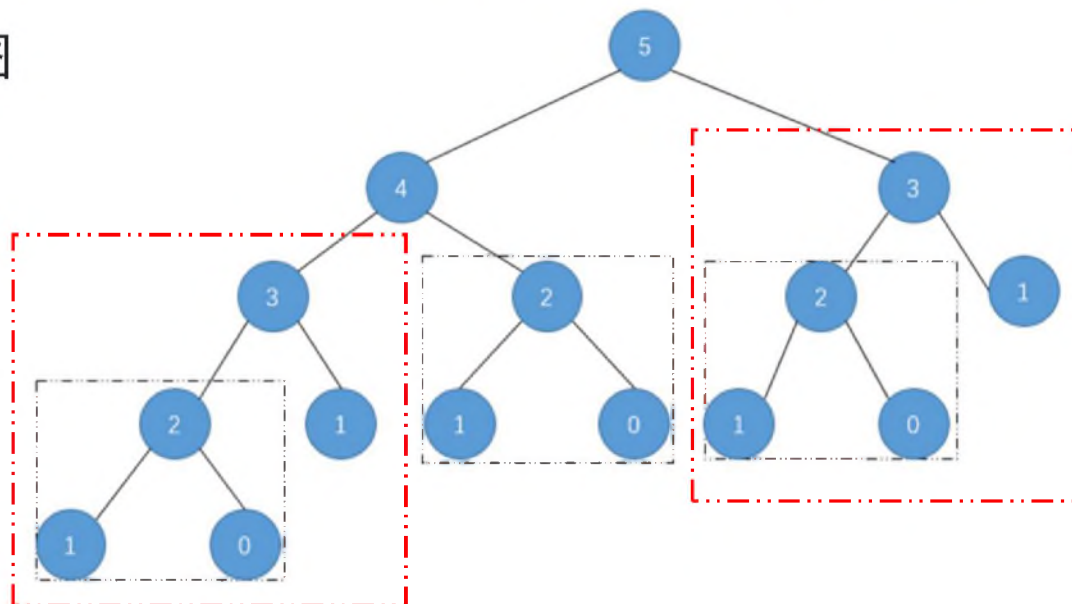
斐波那契数列可以用下面的方法定义：

$$\begin{cases} F(0) = 0, F(1) = 1 \\ F(n) = F(n-1) + F(n-2), n \text{ 为 } \geq 2 \text{ 的整数} \end{cases}$$

计算 $F(5)$

$$\begin{cases} F(0) = 0, F(1) = 1 \\ F(n) = F(n-1) + F(n-2), n \text{ 为 } \geq 2 \text{ 的整数} \end{cases}$$

$n = 5$ 时的递归图



虚线框中 $F(2)$ 的计算用到了三次, 同样的 $F(3)$ 的计算用到了两次, 显然我们执行了非常多的重复运算, 直接递归计算的时间复杂度达到了 2^n .

补充知识: 算法复杂度分为时间复杂度和空间复杂度。时间复杂度衡量了执行算法所需要的计算工作量; 而空间复杂度是指执行这个算法所需要的内存空间。感兴趣的同学可以看数据结构这门课, 非计算机科班的同学了解即可。

补充: 递归的概念

程序调用自身的编程技巧称为递归(recursion)。递归做为一种算法在程序设计语言中广泛应用。它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题(子问题)来求解, 递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算, 大大地减少了程序的代码量。

构成递归需具备的条件:

1. 子问题须与原始问题为同样的事, 且更为简单;
2. 不能无限制地调用本身, 须有个出口, 化简为非递归状况处理。

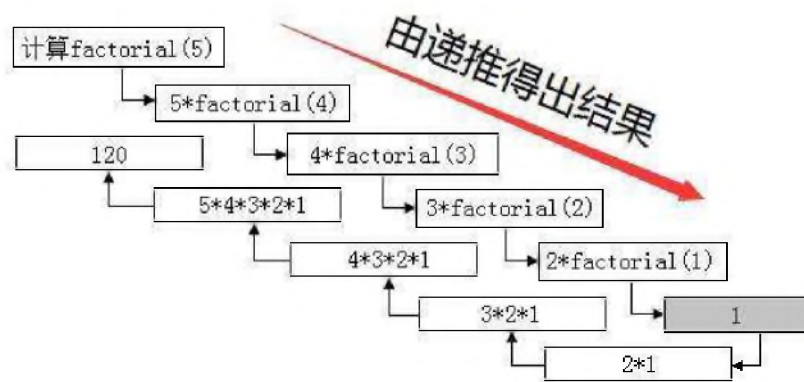
例子: 计算正整数 n 的阶乘

原问题: 计算 $n!$

子问题: 计算 $(n-1)!$

联系: $n! = n * (n-1)!$

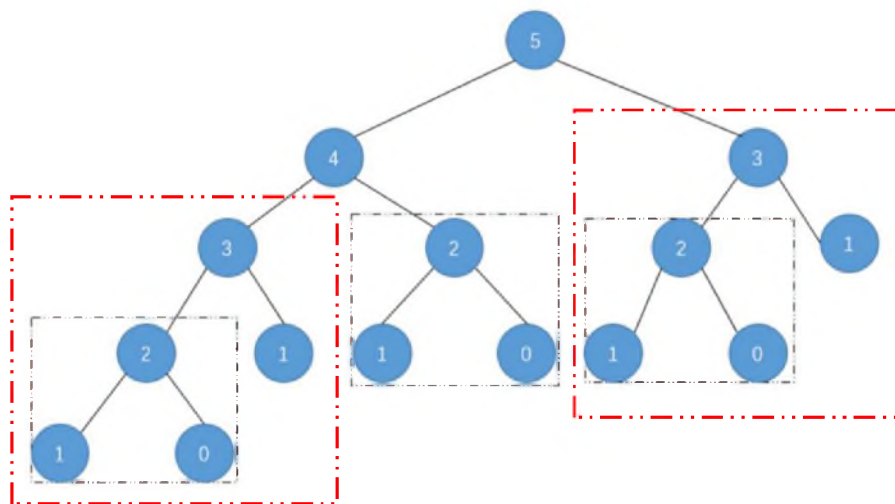
出口: $n=1$ 时, 结果为1



根据递归的思想利用Matlab编写函数:

```
function F = factorial(n)
% 利用递归求正整数n的阶乘
if n == 1 % 递归的出口
    F = 1;
else
    F = n * factorial(n-1);
end
end
```


斐波那契数列的递归代码



$$\begin{cases} F(0) = 0, F(1) = 1 \\ F(n) = F(n-1) + F(n-2), n \text{ 为 } \geq 2 \text{ 的整数} \end{cases}$$

```
function F = fib_dg(n)
% 利用递归求解斐波那契数列
if n == 0 % 递归的第一个出口
    F = 0;
elseif n == 1 % 递归的第二个出口
    F = 1;
else
    F = fib_dg(n-1) + fib_dg(n-2);
end
end
```

因为存在大量的重复计算, 所以上面代码的时间复杂度达到了 2^n , 那么我们去解决这个问题呢?

两种思路: (1) 带有备忘录的递归算法 (2) 自底向上法

注意, 递归一般是自顶向下的形式编写的, 我们上面的递归图是从上向下延伸的, 都是从一个规模较大的原问题比如说 $f(5)$, 向下逐渐分解规模, 直到到达了 $f(1)$ 和 $f(0)$ 这两个出口, 然后逐层返回答案。因此, 带有备忘录的递归算法也被称为带有备忘录的自顶向下法。

带有备忘录的递归算法

直接利用递归求解耗时的原因是重复计算, 那么我们可以建立一个「备忘录」, 每次算出某个子问题的答案后别急着返回, 先记到「备忘录」里再返回; 每次遇到一个子问题先去「备忘录」里查一查, 如果发现之前已经解决过这个问题了, 直接把答案拿出来用, 不要再耗时去计算了。

因为matlab里面向量(一维数组)的下标是从1开始取的, 所以为了方便我们将斐波那契数列变成从1开始:

$$\begin{cases} F(1) = 1, F(2) = 1 \\ F(n) = F(n-1) + F(n-2), n \text{ 为 } \geq 3 \text{ 的整数} \end{cases}$$

(c/c++/java/python
等多数语言数组下
标是从0开始哦)

新建一个m文件后再调用右边的子函数:

```
n = 40;
```

```
global memo % 备忘录定义成全局变量
```

```
memo = -1*ones(1,n); % 初始化备忘录全为-1
```

```
fib_dg_memo(n)
```

% 利用带有备忘录的递归求解斐波那契数列

```
function F = fib_dg_memo(n)
```

```
global memo % 备忘录定义成全局变量
```

```
if n == 1 || n == 2
```

```
    F = 1;
```

```
elseif memo(n) ~= -1
```

```
    F = memo(n);
```

```
else
```

```
    memo(n) = fib_dg_memo(n-1) + fib_dg_memo(n-2);
```

```
    F = memo(n);
```

```
end
```

```
end
```

参考: <https://zhuanlan.zhihu.com/p/78220312>



数学建模学习交流

自底向上法

$$\begin{cases} F(1) = 1, F(2) = 1 \\ F(n) = F(n-1) + F(n-2), n \text{ 为 } \geq 3 \text{ 的整数} \end{cases}$$

啥叫「自底向上」？它和「自顶向下」刚好反过来，我们直接从最底下，最简单，问题规模最小的 $F(1)$ 和 $F(2)$ 开始往上推，直到推到我们想要的答案 $F(n)$ 。

```
function F = fib_zdxx(n)
% 利用自底向上的思想求解斐波那契数列
FF = ones(1,n); % 初始化保存中间结果的数组(向量)全为1
if n <= 2
    F = 1;
else
    for i = 3:n
        FF(i) = FF(i-1) + FF(i-2);
    end
    F = FF(n);
end
end
```

没有Matlab基础的同学可以看B站UP主[正月点灯笼](#)的Matlab入门教程
安装Matlab的话可以在公众号《数学建模学习交流》发送“软件安装”

从斐波那契数列到动态规划

计算斐波那契数列的例子严格来说不算动态规划问题, 因为动态规划一般用来求解优化问题, 但在这个例子中我们实际上能看到动态规划的影子:

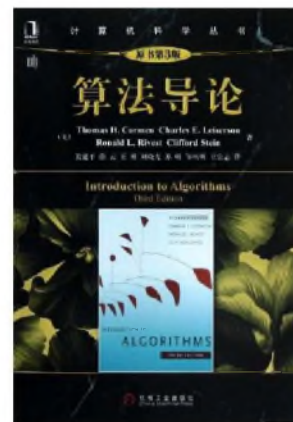
动态规划通过组合子问题的解来求解原问题, 一般来说, 动态规划应用于**重叠子问题**的情况, 即不同的子问题具有公共的子子问题。动态规划算法对每个子子问题只求解一次, 将其解保存在一个表格中, 从而无需每次求解一个子子问题时都重新计算, 避免了这种不必要的计算工作。

动态规划有两种等价的实现方法:

第一种方法称为带备忘的自顶向下法(top-down with memoization)。此方法仍按自然的递归形式编写过程, 但过程会保存每个子问题的解(通常保存在一个数组或散列表中)。当需要一个子问题的解时, 过程首先检查是否已经保存过此解。如果是, 则直接返回保存的值, 从而节省了计算时间;否则, 按通常方式计算这个子问题。我们称这个递归过程是带备忘的(memoized), 因为它“记住”了之前已经计算出的结果。

第二种方法称为自底向上法(bottom-up method)。这种方法一般需要恰当定义子问题“规模”的概念, 使得任何子问题的求解都只依赖于“更小的”子问题的求解。因而我们可以将子问题按规模排序, 按由小至大的顺序进行求解。当求解某个子问题时, 它所依赖的那些更小的子问题都已求解完毕, 结果已经保存。每个子问题只需求解一次, 当我们求解它(也是第一次遇到它)时, 它的所有前提子问题都已求解完成。

——摘自《算法导论第三版中文翻译版》P204和P207的内容



《算法导论》(Introduction to Algorithms)是麻省理工学院出版社出版的关于计算机中数据结构与算法的图书, 作者是托马斯·科尔曼(Thomas H. Cormen)、查尔斯·雷瑟尔森(Charles E. Leiserson)、罗纳德·李维斯特(Ronald L. Rivest)、克利福德·斯坦(Clifford Stein)。

绝大多数时候我们都可以使用第二种方法实现动态规划, 而且实现起来比第一种方法也更加容易。



数学建模学习交流

动态规划中常见到的概念

这里我们还是以求解斐波那契数列来举例子, 尽管它不算严格的动态规划问题:

(1) 子问题和原问题

原问题就是你要求解的这个问题本身, 子问题是和原问题相似但规模较小的问题(原问题本身就是子问题的最复杂的情形, 即子问题的特例)。

例如: 要求 $F(10)$, 那么求出 $F(10)$ 就是原问题, 求出 $F(k)(k \leq 10)$ 都是子问题。

(2) 状态

状态就是子问题中会变化的某个量, 可以把状态看成我们要求解的问题的自变量。

例如: 我们要求的 $F(10)$, 那么这里的自变量10就是一个状态。

(3) 状态转移方程

能够表示状态之间转移关系的方程, 一般利用关于状态的某个函数建立起来。

例如: $F(n) = F(n-1) + F(n-2)$, 当 n 为 >2 的整数时; 当 $n=1$ 或 2 时, $F(n)=1$, 这种最简单的初始条件一般称为边界条件, 也被称为基本方程。

(4) DP数组 (DP就是动态规划的缩写)

DP数组也可以叫"子问题数组", 因为DP数组中的每一个元素都对应一个子问题的结果, DP数组的下标一般就是该子问题对应的状态。

例如: 使用自底向上法编程求解时, 我们定义的向量FF就可以看成一个DP数组, 数组下标从1取到 n , 对应的元素从 $F(1)$ 取到 $F(n)$ 。

打家劫舍 (力扣198)

你是一个小偷, 现在有一排相邻的房屋等着你去偷窃。这些房子装有相互连通的防盗系统, 如果两间相邻的房屋在同一晚上被小偷闯入, 系统会自动报警。给定一个代表每个房屋存放金额的正整数数组, 计算你不触动警报装置的情况下, 一夜之内能够偷窃到的最高金额。(不用考虑偷窃时间)

示例 1:

输入: [1,2,3,1,3]

输出: 7

解释: 偷窃 1 号房屋 (金额 = 1), 然后偷窃 3 号房屋 (金额 = 3), 接着偷窃 5 号房屋 (金额 = 3)。偷窃到的最高金额 = $1 + 3 + 3 = 7$ 。

示例 2:

输入: [2,7,2,3,8]

输出: 15

解释: 偷窃 2 号房屋 (金额 = 7), 然后偷窃 5 号房屋 (金额 = 8)。偷窃到的最高金额 = $7 + 8 = 15$ 。

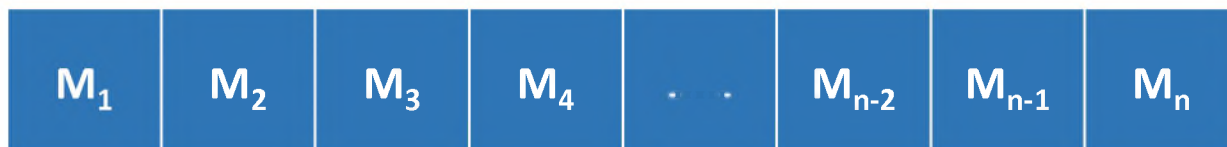
题目来源: 力扣198. 打家劫舍 链接: <https://leetcode-cn.com/problems/house-robber>



数学建模学习交流

分析思路

假设一共有 n 间房子, 第 i 间房子中有 M_i 金额的钱财, 如下图所示:



步骤一: 定义原问题和子问题

原问题: 从全部 n 间房子中能够偷到的最大金额;

子问题: 从前 k 间($k \leq n$)房子中能够偷到的最大金额。

步骤二: 定义状态

我们记 $f(k)$ 为小偷能从前 k 间房子中偷到的最大金额, 这里的 k 就表示这个小偷在子问题时的一个状态, 即小偷仅偷前 k 间房子 (注意: 这里的 k 有点像函数中的自变量, $f(k)$ 实际上也可以视为一个关于状态 k 的函数)。特别地, 当 $k=n$ 时, $f(n)$ 就是要求解的原问题。

前提条件: 不能偷两个相邻的房子



数学建模学习交流

分析思路

$f(k)$: 小偷能从前 k 间房子中偷到的最大金额

步骤三: 寻找状态转移方程

所谓的寻找状态转移方程, 本质上就是寻找子问题之间的一个递推关系, 我们这里先给出答案, 然后再进行解释:

$$f(k) = \begin{cases} M_1, & k = 1 \\ \max\{M_1, M_2\}, & k = 2 \\ \max\{f(k-1), f(k-2) + M_k\}, & k \text{ 是 } \geq 3 \text{ 的整数} \end{cases}$$

情况(1):

当 $k=1$ 时, 只有第1间房子可以偷, 因此 $f(1)$ 应该等于这间房子的金额。

情况(2):

当 $k=2$ 时, 只有第1间房子和第2间房子可以偷, 又因为小偷不可以偷两个相邻的房子, 因此我们只能选择金额较大的那间房子偷。

注: 这里情况(1)和情况(2)就是动态规划问题中的边界条件, 也称为基本方程。

情况(3):

当 $k \geq 3$ 时, 分两种情况讨论:

(1) 小偷已经偷了第 $k-1$ 间房子, 这时候小偷不能偷第 k 间房子, 此时最多能获得 $f(k-1)$ 的钱财。

(2) 小偷没有偷第 $k-1$ 间房子, 根据定义小偷在前 $k-2$ 间房子能够偷到的最大钱财为 $f(k-2)$, 又因为没有偷第 $k-1$ 间房子, 那么小偷一定会去偷第 k 间房, 此时最多能获得 $f(k-2) + M_k$ 的钱财。

因此, 小偷采取哪种情况行事取决于这两种情况下最多能偷到的金额大小, 即 $f(k) = \max\{f(k-1), f(k-2) + M_k\}$ 。

编程求解

步骤四: 编程求解

```
%% 题目来源: 力扣198. 打家劫舍
function f = house_robber1(M)
    % 输入的M就是每间房子里面的金额
    n = length(M); % 一共多少间房子
    if n == 1 % 只有第1间房子可以偷时, f应该等于这间房子的金额
        f = M(1);
    elseif n == 2 % 只有前两间房子可以偷时, f应该等于较大的金额
        f = max(M(1),M(2));
    else % 超过三间房可以偷
        FF = zeros(1,n); % DP数组, 保存f(k)
        FF(1) = M(1); % 边界条件
        FF(2) = max(M(1),M(2)); % 边界条件
        for i = 3:n % 利用状态转移方程循环计算
            FF(i) = max(FF(i-1),FF(i-2)+M(i));
        end
        f = FF(n); % 输出FF中最后一个元素, 也就是原问题的解
    end
end
```

状态压缩(非必须步骤)

前面我们提到过, 计算机科班同学都要学的一门专业课叫做《数据结构》, 里面有一个很重要的分析算法效率的工具: 算法复杂度。

算法复杂度可以分为时间复杂度和空间复杂度, 事实上对比穷举法搜索而言, 动态规划方法能够显著的降低计算的时间成本, 即降低时间复杂度; 另外, 在空间复杂度方面, 我们可以通过设计合适的程序来减少内存的开销。

状态压缩就是用来减少空间复杂度的一种方法, 其基本思想是很多时候我们并不需要始终持有全部的 DP 数组 (有些地方把这种处理技巧称为滚动数组)。对于前面讲的打家劫舍问题而言, 我们发现, 最后一步计算 $f(n)$ 的时候, 实际上只用到了 $f(n-1)$ 和 $f(n-2)$ 的结果, 再往之前的子问题的结果实际上早就已经用不到了。因此, 我们可以利用两个变量保存前两个子问题的结果, 这样既可以依次计算出所有的子问题, 又能节省计算机内存的消耗。



利用状态压缩改进后的代码

```
%% 通过状态压缩来节省空间消耗
function f = house_robber2(M)
    % 输入的M就是每间房子里面的金额
    n = length(M); % 一共多少间房子
    if n == 1 % 只有第1间房子可以偷时, f应该等于这间房子的金额
        f = M(1);
    elseif n == 2 % 只有前两间房子可以偷时, f应该等于较大的金额
        f = max(M(1),M(2));
    else
        pre2 = M(1); % 保存F(i-2)的值
        pre1 = max(M(1),M(2)); %保存F(i-1)的值
        for i = 3:n
            cur = max(pre1,pre2+M(i)); % 即FF(i) = max(FF(i-1),FF(i-2)+M(i));
            pre2 = pre1; % F(i-2)往前进1位变成F(i-1)
            pre1 = cur; % F(i-1)往前进1位变成F(i)
        end
        f = cur; % 输出FF(i), 此时i=n, 也就是原问题的解
    end
end
```

怎么输出偷窃的房屋编号?

注意: 产生最大偷窃金额的方案不唯一, 我们这里只要求输出任意一个方案即可。例如 $M=[1,3,2]$ 就有两种方案。

```
function [f, IND] = house_robber3(M)
% 输入的M就是每间房子里面的金额
n = length(M);
if n == 1 % 只有一间房子
    f = M(1); IND = 1;
elseif n == 2 % 只有两间房子
    [f, IND] = max([M(1), M(2)]);
else % 大于两间房子的情况
    FF = zeros(1, n); % DP数组, 保存f(k)
    FF(1) = M(1);
    FF(2) = max(M(1), M(2));
    for i = 3:n
        FF(i) = max(FF(i-1), FF(i-2)+M(i));
    end
    f = FF(i);
    % IND可以通过DP数组FF推算出来:
    IND = [];
    ind = find(FF == FF(end), 1);
    IND = [IND, ind];
    while FF(ind) > M(ind)
        ind = find(FF == (FF(ind) - M(ind)), 1);
        IND = [IND, ind];
    end
    IND = IND(end:-1:1); % 翻转一下
end
end
```

这里我们只需要在最开始写的函数加上一个返回值IND, 表示我们盗窃的房屋编号, IND的计算方法在下方红色方框内, 这里不是很好理解, 我们下面给一个实例:

给定每间房子的金额向量 $M = [1, 4, 1, 2, 5, 6, 3]$ (M 中元素都大于0) 根据左边代码计算得到的DP数组 $FF = [1, 4, 4, 6, 9, 12, 12]$, FF 中第 k 个元素表示的含义是小偷能从前 k 间房子中偷到的最大金额。最终求得的 $f = 12$, 表示从所有房子中能偷到的最大金额为12。IND = [2, 4, 6], 表示小偷偷的房子编号为2, 4, 6, 可以验证2, 4, 6三间房子对应的金额和恰好12(4+2+6)。

根据FF的定义容易证明下面两个结论: (1) 对于任意的 i 均有 $FF(i) \geq M(i)$, 当且仅当小偷只偷了一间房子时等号成立; (2) FF 递增(不要求严格)。

那么, 怎么从FF中推出IND呢? 我们先初始化IND为一个空向量。

第1步: 找到FF中第一次出现最大值的位置, 此时是小偷偷的最后一间房子。反证: 假设小偷还偷了后面的房子, 那么FF一定会增加, 矛盾。因此FF中第一个最大值对应的下标就是小偷偷的最后一间房子的标号。比如在我们上面这个例子里面, FF的最大值是12, 第一次出现的下标为6, 这表示6号房子是小偷最后偷的一间房子, 因此我们可以把6添加到向量IND内。

第2步: 记第一步找到的下标为ind, 判断 $FF(ind)$ 与 $M(ind)$ 的大小, 根据结论(1), $FF(ind) \geq M(ind)$. 所以可以分为两种情况讨论:

(1) 如果 $FF(ind)$ 等于 $M(ind)$, 这意味着小偷在之前没有偷任何房子, 这可以根据上面的结论(1)得到, 那么我们就可以直接返回IND。

(2) 如果 $FF(ind)$ 大于 $M(ind)$, 则说明小偷还偷了其他房子, 在我们这个例子中: $ind=6, FF(ind)=12, M(ind)=6$, 那么我们可以用 $FF(ind)$ 减去 $M(ind)$, 得到的结果就是在 $ind=6$ 号房子之前能够偷得的最大金额, 在我们这个例子里面, $FF(ind)-M(ind)=6$. 然后我们在FF里面找到第一个出现这个6的位置, 这个位置就是小偷倒数第二次选择的房子, 本例中 $FF(4)=6$, 因此可以把4也添加到IND中, 这样不断循环下去, 直到 $FF(ind)$ 等于 $M(ind)$ 为止。例如本例中, $FF(4)=6$ 而 $M(4)=2$, 因为 $6-2=4>0$, 所以小偷在4号房子之前最多能偷到的金额为4, 接下来我们再在FF里面找到第一个出现这个4的位置, 得到了2号房子, 则把2添加到IND内, 然后判断出 $FF(2)=M(2)$, 这时候返回IND即可。

动态规划两个条件（了解）

我们一般是用动态规划来解决最优问题。而解决问题的过程, 需要经历多个决策阶段。每个决策阶段都对应着一组状态。然后我们寻找一组决策序列(动态转移方程), 经过这组决策序列, 能够产生最终期望求解的最优值。

一般来说, 如果一个问题能用动态规划方法求解, 需要满足最优子结构和无后效性。

最优子结构指的是, 问题的最优解包含子问题的最优解。反过来说就是, 我们可以通过子问题的最优解, 推导出问题的最优解。如果我们把最优子结构, 对应到我们前面定义的动态规划问题模型上, 那我们也可以理解为, **后面阶段的状态可以通过前面阶段的状态推导出来。**

无后效性有两层含义, 第一层含义是, 在推导后面阶段的状态的时候, 我们只关心前面阶段的状态值, 不关心这个状态是怎么一步一步推导出来的。第二层含义是, 某阶段状态一旦确定, 就不受之后阶段的决策影响。**无后效性是一个非常“宽松”的要求。**只要满足前面提到的最优子结构性质, 基本上都会满足无后效性。

很抽象的概念, 需要结合实际题目来加以理解, 大家知道有这个东西就行。

参考: <https://time.geekbang.org/column/article/75702>

礼物的最大价值 (剑指 Offer 47)

在一个 $m \times n$ (m 和 n 都大于1)的棋盘的每一格都放有一个礼物, 每个礼物都有一定的价值 (价值大于 0) 。你可以从棋盘的左上角开始拿格子里的礼物, 并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值, 请计算你最多能拿到多少价值的礼物?

示例:

输入:

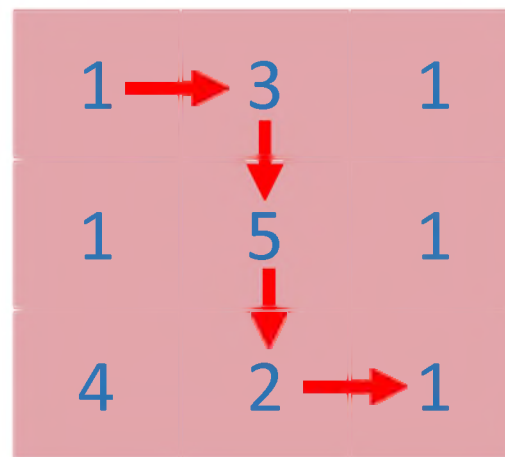
[1,3,1;
1,5,1;
4,2,1]

输出:


12

解释:

路径 $1 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 1$ 可以拿到最多价值的礼物



题目来源: <https://leetcode-cn.com/problems/li-wu-de-zui-da-jie-zhi-lcof/>

 数学建模学习交流

分析思路

假设棋盘大小为 $m * n$, 第 i 行第 j 列格子中有价值为 M_{ij} 的礼物, 如下图所示:

M_{11}	M_{12}	M_{13}	...	$M_{1,n-1}$	M_{1n}
M_{21}	M_{22}	M_{23}	...	$M_{2,n-1}$	M_{2n}
M_{31}	M_{32}	M_{33}	...	$M_{3,n-1}$	M_{3n}
...
$M_{m-1,1}$	$M_{m-1,2}$	$M_{m-1,3}$...	$M_{m-1,n-1}$	$M_{m-1,n}$
M_{m1}	M_{m2}	M_{m3}	...	$M_{m,n-1}$	M_{mn}

题目告诉我们: 从棋盘的左上角开始拿格子里的礼物, 并每次向右或者向下移动一格、直到到达棋盘的右下角。因此, **棋盘中的某一个单元格只能从它的上边一个单元格或左边一个单元格到达。**

分析思路

步骤一：定义原问题和子问题

原问题：从棋盘的左上角开始直到到达棋盘的右下角所能获得的最大礼物价值；

子问题：从棋盘的左上角开始直到到达棋盘的第 i 行第 j 列的格子所能获得的最大礼物价值(这里 $1 \leq i \leq m, 1 \leq j \leq n$).

步骤二：定义状态

记 $f(i, j)$ 为从棋盘左上角走至第 i 行第 j 列的格子所能获得的最大礼物价值, 这里可认为 (i, j) 就是上述子问题所对应的状态; 特别的当 $i = m, j = n$ 时, 该状态对应的 $f(m, n)$ 就是我们要求解的原问题的答案。

M_{11}	M_{12}	M_{13}	...	$M_{1,n-1}$	M_{1n}
M_{21}	M_{22}	M_{23}	...	$M_{2,n-1}$	M_{2n}
M_{31}	M_{32}	M_{33}	...	$M_{3,n-1}$	M_{3n}
...
$M_{m-1,1}$	$M_{m-1,2}$	$M_{m-1,3}$...	$M_{m-1,n-1}$	$M_{m-1,n}$
M_{m1}	M_{m2}	M_{m3}	...	$M_{m,n-1}$	M_{mn}

分析思路

步骤三：寻找状态转移方程

$f(i, j)$: 从棋盘左上角走至第*i*行第*j*列的格子所能获得的最大礼物价值。

$$f(i, j) = \begin{cases} M_{11} & , i, j = 1 \\ f(i, j-1) + M_{ij} & , i = 1 \text{ 且 } j > 1 \\ f(i-1, j) + M_{ij} & , i > 1 \text{ 且 } j = 1 \\ \max\{f(i, j-1), f(i-1, j)\} + M_{ij} & , i > 1 \text{ 且 } j > 1 \end{cases}$$

情况(1):

当*i**j*都等于1时, 这时候在起点, 所以 $f(1,1)=M_{11}$.

情况(2):

当*i*=1时, 这时候在棋盘的第一行, 此时只能从左边的格子到达, 那么 $f(1,j)=f(1,j-1)+M_{1j}$

情况(3):

当*j*=1时, 这时候在棋盘的第一列, 此时只能从上面的格子到达, 那么 $f(i,1)=f(i-1,1)+M_{i1}$

注: 这里情况(1)-(3)就是动态规划问题中的边界条件, 也称为基本方程。

情况(4):

当*i*和*j*都大于1时, 这时候在棋盘的右下方, 此时可从左边或上面的格子到达, 因此可以分两种情况讨论: 从左边格子到达的话, 能够获得的最大礼物价值为 $f(i,j-1)+M_{ij}$; 从上面的格子到达的话, 能够获得的最大礼物价值为 $f(i-1,j)+M_{ij}$. 因此, 采取哪种情况到达取决于这两种情况下最多能获得的礼物价值大小, 也就是说: $f(i,j)=\max\{f(i,j-1)+M_{ij}, f(i-1,j)+M_{ij}\}=\max\{f(i,j-1), f(i-1,j)\}+M_{ij}$.



编程求解

步骤四: 编程求解

```
function f = max_gift_value1(M)
    % 输入的M就是棋盘每个礼物的价值
    [m,n] = size(M); % 棋盘m行n列
    FF = M; % 初始化DP数组和M完全相同, 用来保存f(i,j)
    % 计算FF的第一列
    FF(:,1) = cumsum(M(:,1)); % cumsum函数用来计算累加和
    % 计算FF的第一行
    FF(1,:) = cumsum(M(1,:));
    % 循环计算右下部分的元素
    for i = 2:m
        for j = 2:n
            tem1 = FF(i,j-1) + M(i,j); % 从左边来
            tem2 = FF(i-1,j) + M(i,j); % 从上面来
            FF(i,j) = max(tem1,tem2);
        end
    end
    f = FF(m,n);
end
```


怎么输出所走的路线?

注意: 产生最大礼物的路线不唯一, 我们这里只要求输出任意一个路线即可。

我们就以示例给的数据为例子, 我们可以根据程序中得到的DP数组(即FF)来推出对应的路径:

M: 礼物价值矩阵

1	→ 3	1
1	↓ 5	1
4	↓ 2	→ 1

FF: 编程求得的DP数组

1	4	5
2	9	10
6	11	12

$FF(i, j)$: 从棋盘左上角走至第*i*行第*j*列的格子所能获得的最大礼物价值。

第1步: 找到FF最右下角格子的值, 将其减去M的同位置格子的值; 然后找到这个格子的左边一格和上面一格 (如果存在的话), 看哪个格子的FF值和这个结果相同, 那么上一步就是从这个方向过来的。(如果两个都一样的话说明任意一个方向都可以)

例如: 在左边这个例子里面, FF最右下角格子的值为12, M同位置的值为1, 将12减去1得到11, 然后看这个格子的左边一个格子和上面一个格子哪个格子的FF值是11, 很明显左边是11, 因此最后一步是从左边这个格子过来的。

第2步: 根据上面找到的这个新的格子, 循环第1步的步骤, 直到最终返回到了起始位置为止。

例如, 第3行第2列的格子对应的FF值为11, M值为2, 计算 $11-2=9$, 然后发现这个格子的上面一个格子对应的FF值也是9, 因此上一步是从上面一格下来的; 接着因为 $9-5=4$, 而它的上一个格子FF值是4, 所以是从上面一格下来的; 最后, 因为这个格子左边就是起点, 因此我们找到了完整的路径, 结束循环。

(还有一种更简单的思路: 直接比较FF中某个格子的左边元素和上面元素, 哪个元素大就说明是从哪个方向来的)

输出所走的路线参考代码

注意：产生最大礼物的路线不唯一，我们这里只要求输出任意一个路线即可。

```
function [f,path] = max_gift_value2(M)
% 输入的M就是棋盘中每个礼物的价值
[m,n] = size(M); % 棋盘m行n列
FF = M; % 初始化DP数组和M完全相同, 用来保存f(i,j)
% 计算FF的第一列
FF(:,1) = cumsum(M(:,1)); % cumsum函数用来计算累加和
% 计算FF的第一行
FF(1,:) = cumsum(M(1,:));
% 循环计算右下部分的元素
for i = 2:m
    for j = 2:n
        tem1 = FF(i,j-1) + M(i,j); % 从左边来
        tem2 = FF(i-1,j) + M(i,j); % 从上面来
        FF(i,j) = max(tem1,tem2);
    end
end
f = FF(m,n);
% 根据程序中得到的DP数组(FF)来推出对应的路径path
path = zeros(m,n); % path全为0和1组成, 1表示经过该格子
i = m; j = n;
while i ~= 1 || j ~= 1 % 只要没有回到原点
    path(i,j) = 1; % 把path矩阵的第i行第j列变成1 (表示访问了这个格子)
    if i == 1 % 如果到了第一行
        path(1,1:j) = 1; % 剩余的路径沿着左边一直走就可以了
        break % 退出循环
    end
    if j == 1 % 如果到了第一列
        path(1:i,1) = 1; % 剩余的路径沿着上方一直走就可以了
        break % 退出循环
    end
    tmp1 = FF(i-1,j); % 上方单元格FF的值
    tmp2 = FF(i,j-1); % 左边单元格FF的值
    ind = find([tmp1,tmp2] == (FF(i,j)-M(i,j)),1); % 看哪个值等于FF(i,j)-M(i,j)
    if ind == 1 % 如果上方单元格FF的值等于FF(i,j)-M(i,j)
        i = i-1; % 说明上一步沿着上方来的
    else % 如果左边单元格FF的值等于FF(i,j)-M(i,j)
        j = j-1; % 说明上一步沿着左边来的
    end
end
end
```

例如： M =

5	1	5	1
3	5	3	5
4	2	1	1
1	5	1	3
1	5	3	4

path =

f =	1	0	0	0
32	1	1	0	0
	0	1	0	0
	0	1	0	0
	0	1	1	1

path全为0和1组成, 1表示经过该格子

零钱兑换 (力扣 322)

给定不同面值的硬币 `coins` 和一个总金额 `S`。编写一个函数来计算可以凑成总金额所需的**最少**的硬币个数。如果没有任何一种硬币组合能组成总金额, 返回-1。(每种硬币的数量是无限的, `S`以及`coins`中的元素都是正整数)

示例 1:

输入: `coins = [1, 2, 5], S = 11`

输出: 3

解释: $11 = 5 + 5 + 1$

示例 2:

输入: `coins = [2,5], S = 3`

输出: -1

示例 3:


输入: `coins = [1, 5, 11, 20], S = 15`

输出: 3

解释: $15 = 5 + 5 + 5$



题目来源: 力扣[322. 零钱兑换](https://leetcode-cn.com/problems/coin-change/) 链接: <https://leetcode-cn.com/problems/coin-change/>

 数学建模学习交流

分析思路

步骤一：定义原问题和子问题

原问题：可以凑成总金额 S 所需的最少的硬币个数；

子问题：可以凑成目标金额 $x(x \leq S)$ 所需的最少的硬币个数。

步骤二：定义状态

我们记 $f(x)$ 为凑成目标金额 x 所需的最少的硬币个数，这里的目标金额 x 就表示子问题所对应的状态。

步骤三：寻找状态转移方程

这个问题的状态转移方程不容易寻找，我们下面用一个具体的例子来说明：假设 $\text{coins} = [1, 2, 5, 20]$, $S = 11$, 我们现在要求 $f(11)$ 。

现在大家可以考虑下，状态11可以由哪些状态经过1步就转移过来？

答案：可以由状态10、状态9和状态6经过1步转移过来。

分析思路

$f(x)$ 为凑成目标金额 x 所需的最少的硬币个数

步骤三：寻找状态转移方程

假设 $coins = [1, 2, 5, 20]$, $S = 11$, 我们现在要求 $f(11)$.

分析：状态11可以由状态10、状态9和状态6经过1步转移过来。因为 $10+1 = 9+2 = 6+5 = 11$, 我们能够猜到： $f(11) = \min\{f(10), f(9), f(6)\} + 1$.

(注意, 因为 $11 - 20 < 0$, 所以面值20的硬币没有作用)

现在考虑一般的情形, 已知 $coins = [c_1, c_2, \dots, c_m]$, 目标金额为 x , 那么:

$$f(x) = \min\{f(x - c_1), f(x - c_2), \dots, f(x - c_m)\} + 1$$

这里要注意:

- (1) 如果 $x - c_i < 0$, 我们令 $f(x - c_i)$ 取成正无穷, 这样取最小值后就相当于排除了这种可能(因为这个硬币面值大于了目标金额, 不会起到作用).
- (2) 如果 $x - c_i = 0$, 那么 $f(x - c_i) = 0$, 即 $f(0) = 0$ (目标金额为0时不需要硬币).

题目中说: 如果没有任何一种硬币组合能组成总金额 S 就返回-1, 因此最后我们要判断下 $f(S)$ 是否为正无穷, 如果是就让它返回-1.

一个具体的例子

coins = [1, 2, 5], S = 11

现在考虑一般的情形, 已知 $coins = [c_1, c_2, \dots, c_m]$, 目标金额为 x , 那么:

$$f(x) = \min\{f(x - c_1), f(x - c_2), \dots, f(x - c_m)\} + 1$$

这里要注意:

- (1) 如果 $x - c_i < 0$, 那么 $f(x - c_i)$ 取成正无穷.
- (2) 如果 $x - c_i = 0$, 那么 $f(x - c_i) = 0$, 即 $f(0) = 0$.

x	最少硬币数量 $f(x)$	结果
1	$f(1) = \min\{f(1-1), f(1-2), f(1-5)\} + 1 = \min\{0, +\infty, +\infty\} + 1$	1
2	$f(2) = \min\{f(2-1), f(2-2), f(2-5)\} + 1 = \min\{1, 0, +\infty\} + 1$	1
3	$f(3) = \min\{f(3-1), f(3-2), f(3-5)\} + 1 = \min\{1, 1, +\infty\} + 1$	2
4	$f(4) = \min\{f(4-1), f(4-2), f(4-5)\} + 1 = \min\{2, 1, +\infty\} + 1$	2
5	$f(5) = \min\{f(5-1), f(5-2), f(5-5)\} + 1 = \min\{2, 2, 0\} + 1$	1
6	$f(6) = \min\{f(6-1), f(6-2), f(6-5)\} + 1 = \min\{1, 2, 1\} + 1$	2
7	$f(7) = \min\{f(7-1), f(7-2), f(7-5)\} + 1 = \min\{2, 1, 1\} + 1$	2
8	$f(8) = \min\{f(8-1), f(8-2), f(8-5)\} + 1 = \min\{2, 2, 2\} + 1$	3
9	$f(9) = \min\{f(9-1), f(9-2), f(9-5)\} + 1 = \min\{3, 2, 2\} + 1$	3
10	$f(10) = \min\{f(10-1), f(10-2), f(10-5)\} + 1 = \min\{3, 3, 1\} + 1$	2
11	$f(11) = \min\{f(11-1), f(11-2), f(11-5)\} + 1 = \min\{2, 3, 2\} + 1$	3

编程求解

步骤四: 编程求解

```
function f = coin_change1(coins, S)
% coins:不同面值的硬币, S:总金额
%  $f(x) = \min\{f(x-c_i) \text{ for every } c_i \text{ in coins}\} + 1$ 
%  $x < 0$ 时,  $f(x) = +\infty$ ;  $x = 0$ 时,  $f(x) = 0$ .
FF = +inf * ones(1,S+2); % 初始化DP数组全为正无穷
% 注意, 这个DP数组长度为S+2, 前面S个才是我们有用的
% FF(x)为凑成目标金额 $x(x \leq S)$ 所需的最少的硬币个数
% 为什么这里还要在后面再加上两个元素呢?
% 后面大家就知道这样做的妙处了~
FF(S+2) = 0; % 最后一个元素改为0
for x = 1:S % 注意, FF只需要更新前S个元素
    tmp = x - coins; % 计算出 ' $x - c_i$ ' for every  $c_i$  in coins
    tmp(tmp < 0) = S+1; % FF下标为S+1的元素为+inf, 所以我们将tmp < 0的位置变成S+1
    tmp(tmp == 0) = S+2; % FF下标为S+2的元素为0, 所以我们将tmp = 0的位置变成S+2
    FF(x) = min(FF(tmp))+1; %  $f(x) = \min\{f(x-c_i) \text{ for every } c_i \text{ in coins}\} + 1$ 
end
if FF(S) < +inf
    f = FF(S);
else
    f = -1; % 如果没有任何一种硬币组合能组成总金额S就返回-1
end
end
```

怎么得到具体的硬币组合

注意：凑成目标金额所需的最少的硬币个数的组合不唯一，我们这里只要求输出任意一个组合即可。（例如用面值为1 2 3 4的硬币凑成 5）

coins = [1, 2, 5], S = 11

x	最少硬币数量f(x)	结果
1	$f(1)=\min\{f(1-1),f(1-2),f(1-5)\}+1 = \min\{0,+\infty,+\infty\}+1$	1
2	$f(2)=\min\{f(2-1),f(2-2),f(2-5)\}+1 = \min\{1,0,+\infty\}+1$	1
3	$f(3)=\min\{f(3-1),f(3-2),f(3-5)\}+1 = \min\{1,1,+\infty\}+1$	2
4	$f(4)=\min\{f(4-1),f(4-2),f(4-5)\}+1 = \min\{2,1,+\infty\}+1$	2
5	$f(5)=\min\{f(5-1),f(5-2),f(5-5)\}+1 = \min\{2,2,0\}+1$	1
6	$f(6)=\min\{f(6-1),f(6-2),f(6-5)\}+1 = \min\{1,2,1\}+1$	2
7	$f(7)=\min\{f(7-1),f(7-2),f(7-5)\}+1 = \min\{2,1,1\}+1$	2
8	$f(8)=\min\{f(8-1),f(8-2),f(8-5)\}+1 = \min\{2,2,2\}+1$	3
9	$f(9)=\min\{f(9-1),f(9-2),f(9-5)\}+1 = \min\{3,2,2\}+1$	3
10	$f(10)=\min\{f(10-1),f(10-2),f(10-5)\}+1 = \min\{3,3,1\}+1$	2
11	$f(11)=\min\{f(11-1),f(11-2),f(11-5)\}+1 = \min\{2,3,2\}+1$	3

FF是DP数组，FF(x)为凑成目标金额x所需的最少的硬币个数

令IND表示我们选择的硬币组合，初始化为空向量。

令x先从S开始取，例如左边表格中，x先取11，然后判断FF(x)是否为-1，如果为-1说明没有任何一种硬币组合能组成总金额，这时候直接输出IND即可；然后判断FF(x)是否为1，如果为1就可以将x添加到IND中，然后输出IND；

在我们这个表中，FF(11)=3，我们将3减去1得到2，然后在表格中找到：f(11-1),f(11-2),f(11-5)中哪一个是2，显然，f(10)=2,f(9)=3,f(6)=2，这里出现了两个2，说明可能出现了两个不同的方案，我们只需要选择任意一个就行，比如我们选择f(6)，从x=6变成x=11需要一枚面值为5(11-6)的硬币，因此我们将5添加到IND中。（注意：在这一步的时候我们也可以直接判断f(10),f(9)和f(6)哪个最小，因为我们始终选择的是f(x-c_i)较小的那种情况）

下一步令x=6，再重复上面的步骤，直到到达了某个x'使得f(x')等于1，这时候将x'也添加到IND后，再输出IND即可。

例如f(6)=2>1，因此我们将2减去1得到1，然后在表格中找到：f(6-1),f(6-2),f(6-5)中哪一个是1，显然f(5)和f(1)都是1，那么我们不妨选择f(1)，从x=1变成x=6需要一枚面值为5(6-1)的硬币，因此将5添加到IND中，这时候的x变成了1，而f(1)刚好等于1，于是我们将1添加到IND后再输出IND即可。这时候的IND=[5,5,1]。

得到具体的硬币组合代码

```
function [f, IND] = coin_change2(coins, S)
    FF = +inf * ones(1,S+2);
    FF(S+2) = 0; % 最后一个元素改为0
    for x = 1:S
        tmp = x - coins;
        tmp(tmp<0) = S+1;
        tmp(tmp==0) = S+2;
        FF(x) = min(FF(tmp))+1;
    end
    % 利用FF来计算IND
    IND = []; % IND表示我们选择的硬币组合, 初始化为空向量
    if FF(S) < +inf % 存在能凑成S的组合
        f = FF(S);
        ind = S; % ind先指向最后一个位置S
        while FF(ind) > 1 % 如果FF(ind) = 1时就不用寻找了
            indd = ind; % 保存前一个位置
            tmp = ind - coins;
            tmp(tmp<0) = S+1; % FF下标为S+1的元素为+inf
            tmp(tmp==0) = S+2; % FF最后一个元素为0
            % 找到新的位置
            ind = tmp(find(FF(tmp) == (FF(ind) - 1),1));
            % 两个位置之差就是我们要添加的硬币
            IND = [IND,indd-ind];
        end
        IND = [IND,ind]; % FF(ind) = 1时, 把ind也放入到IND中
    else % 如果没有任何一种硬币组合能组成总金额S就返回-1
        f = -1;
    end
end
```

```
>> coins = [1, 2, 5, 10, 20, 50, 100];
S = 67;
[f, IND] = coin_change2(coins, S)
```

f =

4

IND =

2 5 10 50

```
>> coins = [2, 5, 8, 15, 60];
S = 130;
[f, IND] = coin_change2(coins, S)
```

f =

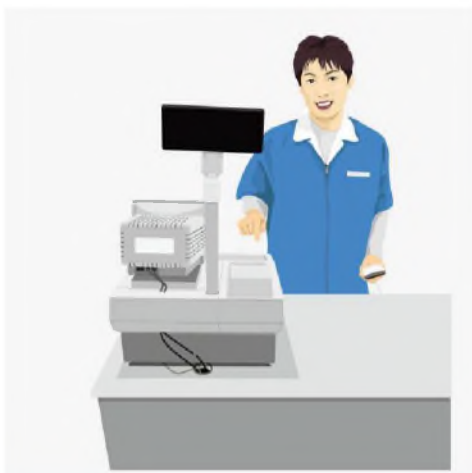
4

IND =

2 8 60 60

现实中的贪心算法

想象这样一个现实中的情景：你是某超市的一名收银员，某顾客买了63元钱的東西，并掏出了一张百元大钞给你，你需要找给他37元钱，你会选择怎么找零（假设每种金额的钞票都有足够多）



$37 = 20 + 10 + 5 + 1 + 1$
(我们优先选择较大面值的钞票找给顾客)

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法一般得到的是在某种意义上的**局部最优解**。

还是考虑使用最少钞票张数这个问题，在现实生活中，因为人民币票面金额设计的很巧妙，贪心算法得到的结果和前面用动态规划得到的结果相同。

如果我们换一组钞票的面值，例如换成[1,5,11,20]，要凑出15元钱，那么使用动态规划得到的结果是3($15 = 5 + 5 + 5$)，贪心算法得到的结果是5($15 = 11 + 1 + 1 + 1 + 1$)。

01 背包问题(01 Knapsack problem)

有10件货物要从甲地运送到乙地, 每件货物的重量(单位: 吨)和利润(单位: 元)如下表所示:

物品	1	2	3	4	5	6	7	8	9	10
重量	6	3	4	5	1	2	3	5	4	2
利润	540	200	180	350	60	150	280	450	320	120

由于只有一辆最大载重为30t的货车能用来运送货物, 所以只能选择部分货物配送, 要求确定运送哪些货物, 使得运送这些货物的总利润最大。

分析: 在更新12中我们讲过, 该问题是一个典型的线性01规划问题:

$$\text{记 } x_i = \begin{cases} 1, & \text{运送了第 } i \text{ 件物品} \\ 0, & \text{没有运送第 } i \text{ 件物品} \end{cases}, i = 1, 2, \dots, 10$$

记 w_i 表示第 i 件物品的重量, p_i 表示第 i 件物品的利润

则目标函数为: $\sum_{i=1}^{10} p_i x_i$ 我们要求目标函数的最大值

$$\text{约束条件: } \begin{cases} \sum_{i=1}^{10} w_i x_i \leq 30 \\ x_i = 0 \text{ 或者 } 1 \end{cases}$$

利用 *Matlab* 的 *intlinprog* 函数可以直接求出上述问题的解

(答案: 除了不运送物品3和物品5外, 运送其他物品能获得最大利润2410元)



数学建模学习交流

分析思路

不妨假设一共有 $m(\geq 2)$ 件物品, 其中第 i 件物品的利润为 v_i , 重量为 w_i , 背包能容纳的总重量为 W .

步骤一: 定义原问题和子问题

原问题: 在满足重量约束的条件下, 将这 m 件物品选择性的放入容量为 W 的背包中所能获得的最大利润.

子问题: 在满足重量约束的条件下, 将前 i ($i \leq m$)件物品选择性的放入容量为 j ($j \leq W$)的背包中所能获得的最大利润.

步骤二: 定义状态

我们不妨记 $f(i, j)$ 为前 i ($i \leq m$)件物品选择性的放入容量为 j ($j \leq W$)的背包中所能获得的最大利润, 那么 $f(m, W)$ 就是我们要求的原问题的答案。

这里 i 和 j 的数值构成的一个组合就是对应的子问题的状态, 显然, 因为这里存在两个参数, 所以我们在编程时要定义一个二维的DP数组。

分析思路

不妨假设一共有 $m(\geq 2)$ 件物品, 其中第 i 件物品的利润为 v_i , 重量为 w_i , 背包能容纳的总重量为 W .

$f(i, j)$: 前 i ($i \leq m$)件物品选择性的放入容量为 j ($j \leq W$)的背包中所能获得的最大利润。

步骤三: 寻找状态转移方程

考虑到要利用Matlab来编程, 我们需要**结合实际的应用情形来做两点简化**。

第一: **假设所有物品重量都是正整数**, 实际应用中如果不是整数的话, 我们可以向上取整, 也可以对现有的重量单位进行换算 (例如由吨转换为千克);

第二: **假设背包能够容纳的总重量也是一个正整数**, 实际应用中我们也可以通过单位换算或者向下取整来进行调整。

在用Matlab编程时, 我们可以把中间结果 $f(i, j)$ 保存在一个矩阵内(即DP数组), 我们最终要求的原问题就是 $f(m, W)$, 也就是这个矩阵最右下角位置的元素。下面我们就先来考虑**边界条件**, 然后再来找一般情况下的状态转移方程:

(1) 矩阵的第一行: $f(1, j)$ 表示把第1件物品放入容量为 j 的背包所能获得的最大利润, 显然只要 $j \geq w_1$, 即只要背包的容量大于或者等于第1件物品的重量时, 就有 $f(1, j) = v_1$; 否则 $f(1, j) = 0$ 。

(2) 矩阵的第一列: $f(i, 1)$ 表示把前 i 件物品放入容量为1的背包所能获得的最大利润, 根据我们上面的简化: 所有物品重量都是正整数, 因此我们先找到前 i 件物品中重量为1的那些物品, 然后找到其中利润最大的那件物品, 这样 $f(i, 1)$ 就等于这件物品的利润; 如果前 i 件物品中不存在重量为1的物品时, $f(i, 1) = 0$ 。

分析思路

不妨假设一共有 $m(\geq 2)$ 件物品, 其中第 i 件物品的利润为 v_i , 重量为 w_i , 背包能容纳的总重量为 W .

$f(i, j)$: 前 i ($i \leq m$) 件物品选择性的放入容量为 j ($j \leq W$) 的背包中所能获得的最大利润。

步骤三: 寻找状态转移方程

前面我们考虑了边界条件, 即DP数组的第一行和第一列, 下面我们再来考虑一般的情况, 即 $i > 1$ 且 $j > 1$ 的情况。

我们现在不妨考虑这样一个情景: 有一个容量大小为 j 的背包, 前面 $i - 1$ 件物品你已经规划好了要装的方案, 现在你需要考虑装不装第 i 件物品。

对于这个问题可分为两种情况考虑:

(1) 第 i 件物品的重量 w_i 比背包的容量 j 还要大: 这时候我们只能放弃物品 i , 那么根据定义: $f(i, j) = f(i - 1, j)$, 即「前 i 件物品选择性的放入容量为 j 的背包中所能获得的最大利润」与「前 $i - 1$ 件物品选择性的放入容量为 j 的背包中所能获得的最大利润」一样大。

(2) 第 i 件物品的重量 w_i 小于等于背包的容量 j : 这时候你有两种选择:

第一种选择就是不放, 这时候我们的利润还是等于 $f(i - 1, j)$;

第二种选择是放进去, 如果你把第 i 件物品放进去了, 那么我们至少能获得 v_i 的利润, 并且, 这时候背包剩余的容量为 $j - w_i$, 如果剩余容量大于0的话, 你完全可以认为它就是一个新的背包, 这个新的背包的容量就是 $j - w_i$, 而且将前 $i - 1$ 件物品选择性的放入这个背包中能够获得的最大利润应该等于 $f(i - 1, j - w_i)$, 因此选择放进去能够获得的利润应该为 $v_i + f(i - 1, j - w_i)$, 那么我们是否放进去取决于这两种选择带来的利润大小, 即 $f(i, j) = \max\{f(i - 1, j), v_i + f(i - 1, j - w_i)\}$. (注意当 $j = w_i$ 时, $f(i - 1, j - w_i) = 0$)

编程求解

步骤四: 编程求解

```
function f = knapsack01problem1(p,w,W)
% 输入: p: 物品的利润 w: 物品的重量 W: 背包的容量
% 为了编程方便, 假设W是大于等于2的正整数; w中每个元素都是大于等于1的正整数
m = length(p); % 物品个数
FF = zeros(m,W); % 初始化DP数组
% FF(i,j): 前i件物品选择性的放入容量为j的背包中所能获得的最大利润
if w(1) <= W % 初始化第一行
    FF(1,w(1):end) = p(1);
end
for i = 2:m % 初始化第一列
    FF(i,1) = max([p(w(1:i) == 1),0]);
end
% i,j>1的情况
for i = 2:m
    for j = 2:W
        if w(i) > j % 第i件物品的重量w(i)比背包的容量j还要大
            FF(i,j) = FF(i-1,j);
        elseif w(i) == j % 第i件物品的重量w(i)等于背包的容量j
            FF(i,j) = max(FF(i-1,j), p(i)); % 不放进去和放进去取较大的值
        else % 第i件物品的重量w(i)小于背包的容量j
            FF(i,j) = max(FF(i-1,j), p(i)+FF(i-1,j-w(i))); % 不放进去和放进去取较大的值
        end
    end
end
f = FF(m,W);
end
```

怎么得到选择的物品编号

我们来看下面这个例子, 假设一共6个物品:

编号	1	2	3	4	5	6
利润	11	12	10	26	14	16
重量	3	2	2	5	1	3

背包容量为10, 我们可以计算得到DP数组FF:

容量 \ 物品	1	2	3	4	5	6	7	8	9	10
1	0	0	11	11	11	11	11	11	11	11
2	0	12	12	23	23	23	23	23	23	23
3	0	12	22	23	23	33	33	33	33	33
4	0	12	22	26	26	38	38	48	49	49
5	0	14	26	26	36	40	40	52	52	62
6	0	14	26	26	36	42	42	52	56	62

$f(i, j)$: 前 i ($i \leq m$)件物品选择性的放入容量为 j ($j \leq W$)的背包中所能获得的最大利润。

令IND表示我们选择的物品, 初始化为空向量。

(1) 背包的初始容量为10, 我们先找到矩阵FF中容量为10的这一列, 然后再找到第一个最大值的位置对应的物品, 将这个物品放入IND中。例如左边例子中, 物品5和物品6的FF值都是62, 均是这一列的最大值, 但是物品5是这一列的第一个最大值, 因此物品5被放入到IND中。

(2) 如果上面那一步得到的物品为1号物品, 就可以直接输出IND, 不需要进行下面的步骤了, 否则用上一步的容量减去该物品的重量, 得到了一个新的容量, 如果这个容量恰好为0, 也可以直接输出IND; 否则我们选择这个新容量所在的一列, 并重复上一步的操作, 但是这时候找这一列的FF最大值时, 我们只能选择前面的物品。例如, 物品5的重量为1, 那么 $10-1=9$, 因此我们找到容量为9的这一列, 然后在前面4个物品中找到FF值第一次出现最大值的位置, 也就是物品4, 我们把物品4也被放入到IND中; 然后我们用9减去物品4的重量5, 得到新的容量为4, 于是我们选择第4列, 然后在前面3个物品中找到FF值第一次出现最大值的位置, 也就是物品3, 我们把物品3也被放入到IND中; 然后我们用4减去物品3的重量2, 得到新的容量为2, 于是我们选择第2列, 然后在前面2个物品中找到FF值第一次出现最大值的位置, 也就是物品2, 我们把物品2也被放入到IND中; 这时候剩余的容量为0, 所以返回IND值, 此时IND中有四个元素, 分别是5,4,3,2, 我们可以对其排序后再输出。

得到选择的物品编号的代码

```

%% 怎么得到选择的物品编号?
function [f, IND] = knapsack01problem2(p,w,W)
% 输入: p: 物品的利润 w: 物品的重量 W: 背包的容量
% 为了编程方便, 假设W是大于等于2的正整数; w中每个元素都是大于等于1的正整数
m = length(p); % 物品个数
FF = zeros(m,W); % 初始化DP数组
% FF(i,j): 前i件物品选择性的放入容量为j的背包中所能获得的最大利润
if w(1) <= W % 初始化第一行
    FF(1,w(1):end) = p(1);
end
for i = 2:m % 初始化第一列
    FF(i,1) = max([p(w(1:i) == 1),0]);
end
% i,j > 1的情况
for i = 2:m
    for j = 2:W
        if w(i) > j % 第i件物品的重量w(i)比背包的容量j还要大
            FF(i,j) = FF(i-1,j);
        elseif w(i) == j % 第i件物品的重量w(i)等于背包的容量j
            FF(i,j) = max(FF(i-1,j), p(i)); % 不放进去和放进去取较大的值
        else % 第i件物品的重量w(i)小于背包的容量j
            FF(i,j) = max(FF(i-1,j), p(i)+FF(i-1,j-w(i))); % 不放进去和放进去取较大的值
        end
    end
end
f = FF(m,W);
IND = []; % 选择的物品编号IND初始化为空
if f > 0 % 只要有利润, 就可以利用FF来计算选择的物品编号IND
    ww = W; % 初始化背包的剩余容量为整个背包的容量W
    tmp = FF(:,ww); % 取出最后一列
    while 1 % 不断循环下去, 后面通过条件判断来退出循环
        ind = find(tmp == max(tmp),1); % 找到装入背包的那个物品
        ww = ww - w(ind); % 更新背包的剩余容量
        IND = [IND,ind]; % 更新IND里面的元素
        if ind > 1 && ww > 0 % 只要不是第一个物品或者背包容量为空
            tmp = FF(1:ind-1,ww); % 重新取出剩余容量的那一列 (只保留前面的物品)
        else
            break % 跳出循环
        end
    end
    IND = sort(IND); % 排序下, 输出好看点
end
end

```

```

>> p = [11,12,10,26,14,16];
w = [3,2,2,5,1,3];
W = 10;
[f, IND] = knapsack01problem2(p,w,W)

```

```
f =
```

```
62
```

```
IND =
```

```
2 3 4 5
```

```

>> p = [540,200,180,350,60,150,280,450,320,120];
w = [6,3,4,5,1,2,3,5,4,2];
W = 30;
[f, IND] = knapsack01problem2(p,w,W)

```

```
f =
```

```
2410
```

```
IND =
```

```
1 2 4 6 7 8 9 10
```

课后拓展

除了我们前面学习的「01背包问题」外, 背包问题还有也很常见的两类, 它们分别是「完全背包问题」和「多重背包问题」。

在背包容量、单个物品利润和重量都给定的情况下, 这三个问题的主要区别可以用下面这张表说明:

问题	区别
01背包问题	每种物品只有一件, 要么拿要么不拿
完全背包问题	每种物品都有无穷件
多重背包问题	部分或全部物品有数量限制 (例如物品A最多能拿5件)

大家如果遇到了后两种背包问题, 可以先将其转换为01背包问题, 然后再来求解。当然, 网上也能搜索到后两种背包问题的状态转移方程, 感兴趣的同学可以百度下相应的博客。

求硬币兑换的方案数

因为这个问题没有涉及到优化, 所以严格来说不是动态规划问题, 但我们仍然可以用动态规划的思想去求解, 大家不需要过度纠结动态规划的定义, 我们以应用为主。

给定不同面值的 m 种硬币 coins 和一个总金额 S , 请编写一个函数来计算用这些硬币可以凑成总金额 S 的方案数。(每种硬币的数量是无限的, S 以及 coins 中元素都是正整数, 且不考虑每种方案中硬币的顺序)

示例 1:

输入: $S = 4$, $\text{coins} = [1, 2, 3]$

输出: 4

解释: 有四种方案: $[1, 1, 1, 1]$, $[1, 1, 2]$, $[2, 2]$ 和 $[1, 3]$

示例 2:

输入: $S = 10$, $\text{coins} = [2, 3, 5, 6]$

输出: 5

解释: 有五种方案: $[2, 2, 2, 2, 2]$, $[2, 2, 3, 3]$, $[2, 2, 6]$, $[2, 3, 5]$ 和 $[5, 5]$

分析思路

步骤一：定义原问题和子问题

原问题：能够使用所有面值的硬币来凑出总金额 S 的方案数目。

子问题：只能使用前 i ($i \leq m$) 种面值的硬币来凑出总金额 j ($j \leq S$) 的方案数目。

步骤二：定义状态

根据子问题的定义，我们可以看出每种状态包含两个参数：第一个参数就是我们可以使用前多少种面值的硬币；第二个参数就是要凑出的总金额数。

因此，我们记 $f(i, j)$ 为只能使用前 i ($i \leq m$) 种面值的硬币来凑出总金额 j ($j \leq S$) 的方案数目，当 $i = m$ 且 $j = S$ 时就是原问题的解。

注意：有同学看到本题一定会想到前面讲解的“零钱兑换”那一道题，在那道题里面，我们求的是凑成总金额所需的最少的硬币个数，在定义状态时，我们的状态只有一个参数：需要凑出的总金额数是多少。因此我们当时令 $f(x)$ 为凑成目标金额 x 所需的最少的硬币个数，在编程求解时我们使用的也是一维的DP数组。那为什么在本题里面我们定义了两个状态呢？如果你尝试和之前一样只定义一个状态的话，你会发现在下一步寻找状态转移方程时困难重重，因此我们这里采用了和背包问题类似的策略：状态中包含两个参数，构成一个二维的DP数组，在后面大家就会看到这样定义的好处。那有同学会想到，之前“零钱兑换”那题不能也像本题一样定义成有两个参数的状态呢？答案是可以的，如果定义成二维数组的话，我们的状态转移矩阵会更加好写，但这样会增加运算的时间和空间成本。因此，能用一维DP数组解决的问题我们尽量用一维DP数组解决，如果使用一维DP数组时状态转移方程很难写的话，我们再来考虑使用二维DP数组。

分析思路

$f(i, j)$: 只能使用前 i ($i \leq m$)种面值的硬币来凑出总金额 j ($j \leq S$) 的方案数目

步骤三: 寻找状态转移方程

		总金额为j							
		1	2	3	4	.	S-2	S-1	S
使用前 i 种硬币	1								
	2								
	3								
	...								
	m-1								
	m								

既然每种状态包含两个参数, 那么我们会用到一个二维DP数组 (在Matlab中用矩阵表示), 大家学了前面那么多例题这一点应该知道了。

我们先来考虑边界条件: DP数组的第一行和第一列。

分析思路

$f(i, j)$: 只能使用前 i ($i \leq m$)种面值的硬币来凑出总金额 j ($j \leq S$) 的方案数目

步骤三: 寻找状态转移方程

我们先来看矩阵的第一行怎么填, 矩阵的第一行就是 $f(1, j)$, $j = 1, 2, \dots, S$, 即只能使用第1种面值的硬币能够凑出总金额为 j 的方案数是多少。

我们举几个具体的例子再来总结规律:

(1) $S = 4$, $\text{coins} = [1, 2, 3]$

根据定义, $f(1, j) = 1$ ($j = 1, 2, 3, 4$), 因为不管总金额 j 是多少, 都能用 j 个面值为1的硬币凑齐, 而且有且仅有这一种方案 (注意, 我们只能用第一种面值的硬币)。

(2) $S = 10$, $\text{coins} = [2, 3, 5, 6]$

根据定义, 当 $j=2, 4, 6, 8, 10$ 时: $f(1, j) = 1$; 当 $j=1, 3, 5, 7, 9$ 时: $f(1, j) = 0$. 因为我们现在只能使用第一种面值的硬币, 即面值为2的硬币, 那么使用面值为2的硬币能够凑出的总金额 j 也只能是2的倍数, 而且方案数也只有一个, 即用 $j/2$ 个面值为2的硬币凑齐总金额 j 。

假设 coins 中第一种硬币的面值为 c ,

$$\text{规律: } f(1, j) = \begin{cases} 1, & j \text{ 能被 } c \text{ 整除} \\ 0, & j \text{ 不能被 } c \text{ 整除} \end{cases}$$

		总金额为j							
		1	2	3	4	...	S-2	S-1	S
使用前i种硬币	1								
	2								
	3								
	...								
	m-1								
	m								

分析思路

$f(i,j)$: 只能使用前 i ($i \leq m$)种面值的硬币来凑出总金额 j ($j \leq S$) 的方案数目

步骤三：寻找状态转移方程

我们再来看矩阵的第一列怎么填，矩阵的第一列就是 $f(i, 1)$, $i = 1, 2, \dots, m$ ，即只能使用前 i 种面值的硬币能够凑出总金额为1的方案数是多少。

		总金额为j							
		1	2	3	4	...	S-2	S-1	S
使用前 i 种硬币	1								
	2								
	3								
	...								
	m-1								
	m								

这个结果很容易猜到，假设coins中元素是按照从小到大排列的（我们可以通过排序来达到这个要求），那么只有coins中第一个元素为1时， $f(i,1)$ 才等于1，否则的话 $f(i,1)$ 等于0。

例如：当coins = [1,2,3]时， $f(i,1)=1$ ，因为第一个硬币的面值就是1，所以我们选择一枚面值为1的硬币就能得到总金额1。（注意：根据定义：当 $i>1$ 时，我们也能使用面值为1的硬币哦！）

再比如，当coins = [2,3,5,6]时， $f(i,1)=0$ ，因为这些硬币的面值都比1大，所以不可能凑出1。

（注意：coins 中元素要小到大排列才能得到上述的结论，例如coins = [2,1,3]时，因为第一种硬币的面值为2，比1要大，所以 $f(1,1)=0$ ；但 $f(2,1)=f(3,1)=1$ ，这是因为前2种硬币和前3种硬币中都包含coins中的面值为1的硬币）

分析思路

$f(i, j)$: 只能使用前 i ($i \leq m$) 种面值的硬币来凑出总金额 j ($j \leq S$) 的方案数目

步骤三: 寻找状态转移方程

最后我们再给出当 i 和 j 都大于1时, $f(i, j)$ 的一个递推公式:

$$f(i, j) = \begin{cases} f(i-1, j) & , j - \text{coins}(i) < 0 \\ f(i-1, j) + 1 & , j - \text{coins}(i) = 0 \\ f(i-1, j) + f(i, j - \text{coins}(i)) & , j - \text{coins}(i) > 0 \end{cases}$$

总金额 j

	1	2	3	4	...	S-2	S-1	S
1								
2								
3								
...								
m-1								
m								

使用前 i 种硬币

下面我们来解释这个公式的含义, 一共有三种情况:

(1) 要凑的总金额 j 小于第 i 种硬币的面值, 那么第 i 种硬币不会起到任何作用, 因此「只能使用前 i 种面值的硬币来凑出总金额 j 的方案数目」等于「只能使用前 $i-1$ 种面值的硬币来凑出总金额 j 的方案数目」, 即 $f(i, j) = f(i-1, j)$.

例如 $\text{coins} = [1, 2, 5]$, $j = 4$, 求 $f(3, 4)$, 那么根据这个公式, $f(3, 4) = f(2, 4)$.

(2) 要凑的总金额 j 等于第 i 种硬币的面值, 那么结果取决于我们是否选择使用第 i 种硬币: 假设我们选择使用第 i 种硬币, 那么只有一种方案, 即我们使用1枚第 i 种硬币就能凑出总金额 j ; 假设我们选择不用第 i 种硬币的话, 那么方案数等价于「只能使用前 $i-1$ 种面值的硬币来凑出总金额 j 的方案数目」, 因此把这两种不同的选择结果相加就能得到总的方案数。

例如 $\text{coins} = [1, 2, 5]$, $j = 5$, 求 $f(3, 5)$, 那么根据这个公式, $f(3, 5) = f(2, 5) + 1$.

(3) 要凑的总金额 j 大于第 i 种硬币的面值(不妨将其面值记为 f), 由于硬币的个数可以无限选取, 因此对于第 i 种硬币来说, 我们可以依次选取 0 枚, 1 枚, 2 枚, 以此类推, 直到选取到第 k 枚时, 第 i 种硬币凑的总金额大于或者等于需要的总金额 j 为止。举个具体的例子, 例如 $j = 8$, $\text{coins} = [1, 2, 4]$, 要计算 $f(3, 8)$, 那么我们可以使用 0 枚、1 枚、2 枚第 3 种硬币, 即 $f(3, 8) = f(2, 8) + f(2, 4) + f(2, 0)$, 注意这里出现的 $f(2, 0)$ 等于 1, 因为使用前 2 种硬币凑出 0 的方案数只有 1 种, 那就是不使用它们, 只使用 2 枚面值为 4 的硬币。接下来我们考虑一般形式, 容易推出: $f(i, j) = f(i-1, j - 0 * f) + f(i-1, j - 1 * f) + \dots + f(i-1, j - k * f)$, 注意, 这个式子可以化简, 我们将 j 替换成 $j - f$, 那么 $f(i, j - f) = f(i-1, j - f - 0 * f) + f(i-1, j - f - 1 * f) + \dots + f(i-1, j - f - k' * f)$, 这里的 $k' = k - 1$, 可以发现, $f(i, j - f)$ 的等号右侧就是 $f(i, j)$ 等号右侧从第二个式子开始的那些求和项, 因此 $f(i, j) = f(i-1, j) + f(i, j - f)$, 这里的 f 就是第 i 种硬币的面值, 即 $f = \text{coins}(i)$.

编程求解

步骤四: 编程求解

```

%% 求硬币兑换的方案数
function [f, FF]= coin_change_numbers(coins, S)
% 这里也可以返回FF, FF就是DP数组, 1992年国赛那一题有用
coins = sort(coins); % 先排个序
m = length(coins); % 一共m种硬币
FF = zeros(m,S); % 初始化DP数组
% FF(i,j): 只能使用前i种面值的硬币来凑出总金额j 的钱的方案数目
% 初始化第一行(只有第1种硬币, 需要凑出金额为j的钱)
FF(1,:) = (mod(1:S,coins(1))==0);
% 初始化第一列(只能使用前i种面值的硬币, 需要凑出金额为1的钱的方法数)
FF(:,1) = (coins(1) == 1);
% ij>1的情况
for i = 2:m % 使用了前i种面值的硬币
    for j = 2:S % 凑出金额为j的钱
        if j-coins(i) < 0
            % 如果第i个硬币的面值比j还要大, 那第i个硬币没起到作用
            FF(i,j) = FF(i-1,j);
        elseif j-coins(i) == 0
            % 如果第i个硬币的面值等于j, 那有了第i个硬币后相当于多了1种方法
            FF(i,j) = FF(i-1,j) + 1;
        else
            % 如果第i个硬币的面值小于j, 那就可以等价于两部分之和:
            FF(i,j) = FF(i-1,j) + FF(i,j-coins(i));
        end
    end
end
f = FF(m,S);
end

```

```

>> coins = [1 2 3];
S = 4;
f = coin_change_numbers(coins, S)

```

```

f =

    4

```

```

>> coins = [2 3 5 6];
S = 10;
f = coin_change_numbers(coins, S)

```

```

f =

    5

```

国赛1992年真题

1992 年题 B 实验数据分解

组成生命蛋白质的若干种氨基酸可以形成不同的组合。通过质谱实验测定分子两来分析某个生命蛋白质分子的组成时, 遇到的首要问题就是如何将它的分子量 X 分解为几个氨基酸的已知分子量 $a[i]$ ($i = 1, 2, \dots, n$) 之和。某实验室所研究的问题中:

$$n = 18$$

$$a[1:18] = 57, 71, 87, 97, 99, 101, 103, 113, 114, 115, 128, 129, 131, 137, 147, 156, 163, 186.$$

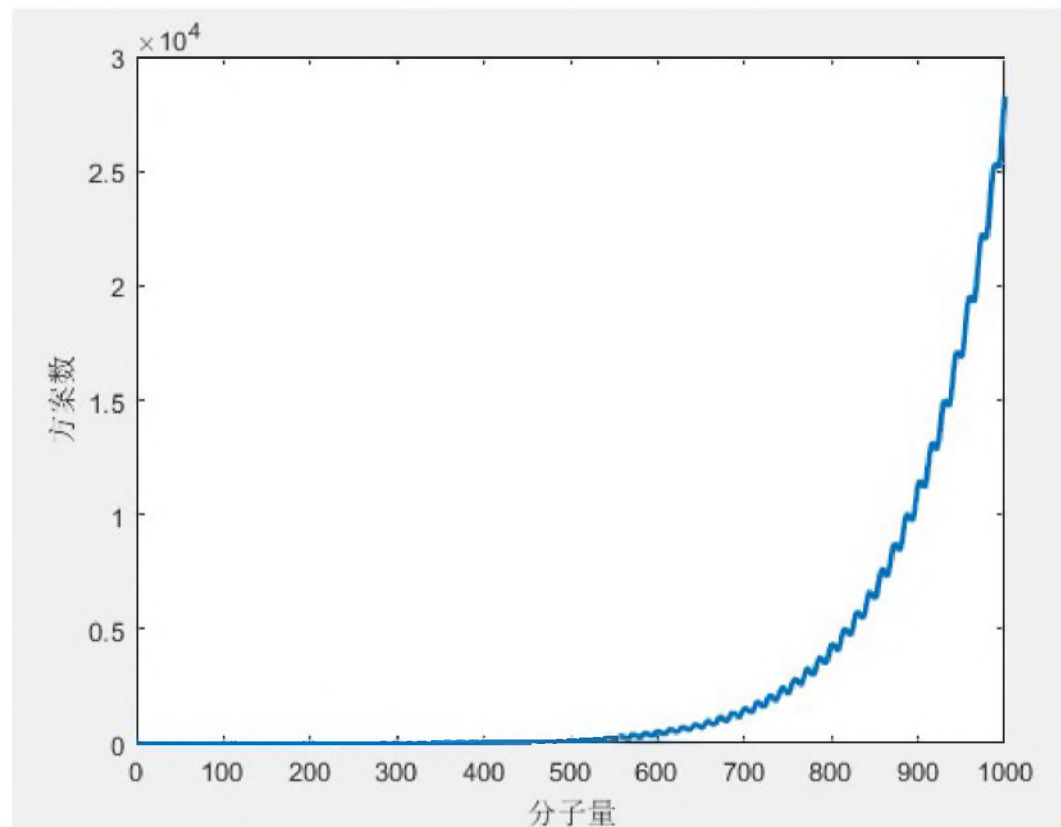
X 为正整数 ≤ 1000

要求针对该实验室拥有或不拥有微型计算机的情况, 对上述问题提出你们的解答, 并就你所研讨的数学模型与方法在一般情形下进行讨论。

参考论文: 程龙, 张云军, 赵蕊, 胡云芳, 龙永红. 蛋白质氨基酸的组合问题[J]. 数学的实践与认识, 1993(03): 86-94.

求解结果

分子量	方案数
200	4
300	14
400	45
500	158
600	522
700	1508
800	4291
900	11249
1000	28268



如果没有计算机怎么求解? 那就需要根据化学专业知识多假设一些条件, 例如: 在实际的蛋白质一级结构测定中, 通常可以对蛋白质经过充分水解后所得到的氨基酸混合液作离子交换层析、纸层析或薄层层析, 定性研究的结果可以确定该蛋白质所含的全部或部分氨基酸种类。

作业1: 爬楼梯 (简单)

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

示例 1:

输入: 2

输出: 2

解释: 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

示例 2:

输入: 3

输出: 3

解释: 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶

2. 1 阶 + 2 阶

3. 2 阶 + 1 阶



作业1参考答案

步骤一：定义原问题和子问题

原问题：爬到第 n 层的方法数；

子问题：爬到第 k ($k \leq n$)层的方法数。

步骤二：定义状态

我们记 $f(k)$ 为爬到第 k ($k \leq n$)层的方法数， k 就是子问题对应的一个状态。

步骤三：寻找状态转移方程

爬到第 k ($k \geq 3$)阶楼梯的方法数量，等于两部分之和：

(1) 爬上 $k-1$ 阶楼梯的方法数量，因为再爬1阶就能到第 k 阶。

(2) 爬上 $k-2$ 阶楼梯的方法数量，因为再爬2阶就能到第 k 阶。

注意，有同学可能会觉得，我在 $k-2$ 阶的时候能不能先爬1阶再爬1阶？实际上这种情况包含在了上面那个情况里面。（如果不相信的话可以用 $k=3$ 验证下）

因此，我们能够得到递推关系： $f(k) = f(k-1) + f(k-2)$

同时别忘了边界条件， $f(1) = 1, f(2) = 2$

作业1参考答案

步骤四: 编程求解

```
function F = homework1(n)
% 利用自底向上的思想求解爬楼梯问题
if n == 1
    F = 1;
elseif n == 2
    F = 2;
else
    FF = ones(1,n); % 初始化DP数组
    FF(2) = 2;
    for i = 3:n
        FF(i) = FF(i-1) + FF(i-2);
    end
    F = FF(n);
end
end
```

作业2: 机器人走格子 (简单)

有一个大小为 $m * n$ 的网格, 一个机器人每次只能向右或向下走一步。现在这个机器人要从左上角走到右下角。请计算机器人有多少种走法。

(假设 m 和 n 都是大于1的数)



例如: 上面这个图中, $m=3$, $n=7$, 有28种走法。

注意: 高中我们学过排列组合问题, 机器人一定会走 $m+n-2$ 步, 那么从 $m+n-2$ 步中挑出 $m-1$ 步向下走就能得到最后的结果: C_{m+n-2}^{m-1}

Matlab中组合数的计算函数为`nchoosek(m+n-2, m-1)`

当然, 这里希望大家自己使用动态规划的思想来计算出最终的结果, 你可以用上面的公式验证你的结果是否正确。

作业2参考答案

$f(i, j)$: 从棋盘左上角走到第*i*行第*j*列的格子的方法数。

$$f(i, j) = \begin{cases} 1 & , i = 1 \text{ 或者 } j = 1 \\ f(i-1, j) + f(i, j-1) & , i > 1 \text{ 且 } j > 1 \end{cases}$$

解释如下:

- (1) 当机器人当前位置在第一行或者在第一列时, 只可能有一种走法。
- (2) 当机器人位于第*i*行第*j*列(*i, j*都大于1)时, 它的上一步可能有两个方向的来源: 要么从第*i*-1行第*j*列来(上方), 要么从第*i*行第*j*-1列来(左侧), 因此路径数目也就等于两种方式之和。



作业2参考答案

```
function f = homework2(m,n)
    % 格子有m行n列
    FF = ones(m,n); % 初始化DP数组全为1
    % 循环计算右下部分的元素
    for i = 2:m
        for j = 2:n
            tem1 = FF(i,j-1); % 左侧过来
            tem2 = FF(i-1,j); % 上面过来
            FF(i,j) = tem1+tem2;
        end
    end
    f = FF(m,n);
end
```

```
>> homework2(3,7)
```

```
ans =
```

```
28
```

```
>> nchoosek(3+7-2, 3-1)
```

```
ans =
```

```
28
```

作业3: 机器人走有障碍的格子 (中等)

有一个大小为 $m * n$ 的网格, 一个机器人每次只能向右或向下走一步。另外, 这个网格中某些格点存在障碍物, 机器人不能通行; 现在这个机器人要从左上角走到右下角。请计算机器人有多少种走法。(假设 m 和 n 都大于1)

网格中的障碍物和空位置分别用 1 和 0 来表示。

0	0	0
0	1	0
0	0	0

输入:[0 0 0;
0 1 0;
0 0 0]

输出: 2

解释: 3x3 网格的正中间有一个障碍物。
从左上角到右下角一共有 2 条不同的路径:

1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右

作业3参考答案

$f(i, j)$: 从棋盘左上角走至第*i*行第*j*列格子的方法数。

我们可以分为下面**两大类情况**讨论:

(1) 第*i*行第*j*列的格子_上有障碍物, 此时走到该格子的方法数: $f(i, j) = 0$

(2) 第*i*行第*j*列的格子_上没有障碍物, 又可以分为**四种情况**:

情况1: $i = j = 1$ 时, $f(i, j) = 1$;

情况2: $i = 1$ 且 $j > 1$ 时, 位于第一行上, 这时候只能从左边到达, 在遇到障碍物前, $f(i, j) = 1$, 一旦遇到了障碍物, 那么障碍物所处的位置以及这个障碍物右边的位置对应的 $f(i, j) = 0$;

情况3: $i > 1$ 且 $j = 1$ 时, 位于第一列上, 这时候只能从上方到达, 在遇到障碍物前, $f(i, j) = 1$, 一旦遇到了障碍物, 那么障碍物所处的位置以及这个障碍物下面的位置对应的 $f(i, j) = 0$;

情况4: $i > 1$ 且 $j > 1$ 时, 和作业2的解释一样, 即 $f(i, j) = f(i - 1, j) + f(i, j - 1)$.

(有同学在情况4中会认为, 万一左边的路或者上方的路不通怎么办? 实际上不用担心, 因为 $f(i - 1, j)$ 和 $f(i, j - 1)$ 的值在之前已经算好了, 不通的话会取为0)

作业3参考答案

```
function f = homework3(obstacle)
    % obstacle是障碍物矩阵, 全为0和1组成, 1表示有障碍物
    [m,n] = size(obstacle);
    FF = ones(m,n); % 初始化DP数组
    % 处理第一列
    for i = 1:m
        if obstacle(i,1) == 1 % 发现了障碍物
            FF(i:end,1) = 0; % 障碍物所处的位置以及下方的位置对应的f(i,j)=0
            break
        end
    end
    % 处理第一行
    for j = 1:n
        if obstacle(1,j) == 1
            FF(1,j:end) = 0; % 障碍物所处的位置以及右边的位置对应的f(i,j)=0
            break
        end
    end
    % 循环计算右下部分的元素
    for i = 2:m
        for j = 2:n
            if obstacle(i,j) == 1
                FF(i,j) = 0;
            else
                FF(i,j) = FF(i,j-1)+FF(i-1,j);
            end
        end
    end
    f = FF(m,n);
end
```


作业4: 掷骰子的N种方法 (中等)

有 m 个完全相同的骰子, 每个骰子上都有 f 个面, 分别标号为 $1, 2, \dots, f$ 。我们约定: 掷骰子得到的总点数为各骰子面朝上的数字的总和, 且骰子每个面出现的概率相同。如果需要掷出的总点数为 S , 请你计算出有多少种不同的组合情况。

示例 1:

输入: $m = 1, f = 6, S = 3$

输出: 1

解释: 因为只有一个骰子, 要得到的点数为3, 而这个骰子能够得到1-6这六个不同的结果, 因此只需要骰出3就可以了, 只有这一种情况, 所以输出1。

示例 2:

输入: $m = 2, f = 6, S = 7$

输出: 6

解释: 因为有两个骰子, 要得到的点数为7, 而每个骰子都能得到1-6的点数, 因此有六种不同的组合情况: (1,6), (2,5), (3,4), (4,3), (5,2), (6,1)。

示例 3:

输入: $m = 1, f = 3, S = 4$

输出: 0

解释: 每个骰子只能得到1-3的点数, 不可能得到4。



作业4参考答案

每个骰子上都有 f 个面, 分别标号为 $1, 2, \dots, f$

记 $f(i, j)$ 为前 $i (i \leq m)$ 个骰子掷出总点数等于 $j (j \leq S)$ 的方法数。

容易得到下面的状态转移方程:

$$f(i, j) = f(i-1, j-1) + f(i-1, j-2) + \dots + f(i-1, j-f) = \sum_{k=1}^f f(i-1, j-k)$$

即前 i 个骰子掷出总点数等于 j 有下面这些可能:

(1) : 前 $i-1$ 个骰子掷出总点数等于 $j-1$, 第 i 个骰子掷出 1;

(2) : 前 $i-1$ 个骰子掷出总点数等于 $j-2$, 第 i 个骰子掷出 2;

.....

$(j-f)$: 前 $i-1$ 个骰子掷出总点数等于 $j-f$, 第 i 个骰子掷出 f ;

注意: 这个公式里面不一定每一项都存在, 如果出现了 $j \leq k$, 我们就令这一项为 0.

最后大家别忘了考虑边界条件, 也就是 DP 数组的第 1 行和第 1 列:

第 1 行: 前 f 个元素(如果有的话)为 1, 其余位置元素为 0.

第 1 列: 第一个元素为 1, 其余位置元素为 0.

作业4参考答案

```
function f = homework4(m, f, S)
% m个骰子, 每个骰子f个面, 需要掷出总点数为S
FF = zeros(m,S); % 初始化DP数组
% 处理第一列
FF(1,1) = 1; % 第一个元素为1, 其余位置元素为0.
% 处理第一行
for j = 1:S
    if j <= f
        FF(1,j) = 1; % 前f个元素(如果有的话)为1, 其余位置元素为0.
    end
end
% 循环计算右下部分的元素
for i = 2:m
    for j = 2:S
        for k = 1:f
            if j > k
                FF(i,j) = FF(i,j)+FF(i-1,j-k);
            else
                break
            end
        end
    end
end
f = FF(m,S);
end
```

作业5: 编辑距离 (困难)

给你两个小写的英文单词 word1 和 word2, 请你计算出将 word1 转换成 word2 所使用的最少操作数 (定义为word1和word2的编辑距离)。你可以对任意一个单词进行如下三种操作:

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

示例 1:

输入: word1 = "horse", word2 = "ros"

输出: 3 (即word1和word2的编辑距离为3)

解释:

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

提示: 本题难度很大, 你可以记 $f(i, j)$ 表示 word1 的前 i 个字母转换成 word2 的前 j 个字母所使用的最少操作数

示例 2:

输入: word1 = "intention", word2 = "execution"

输出: 5 (即word1和word2的编辑距离为5)

解释:

intention -> inention (删除 't')

inention -> enention (将 'i' 替换为 'e')

enention -> exention (将 'n' 替换为 'x')

exention -> exection (将 'n' 替换为 'c')

exection -> execution (插入 'u')

题目来源: 力扣 [72. 编辑距离](https://leetcode-cn.com/problems/edit-distance/) 链接: <https://leetcode-cn.com/problems/edit-distance/>

作业5参考答案

假设word1一共有 m 个字母, word2一共有 n 个字母, 记 $f(i, j)$ 表示word1的前 i 个字母转换成word2的前 j 个字母所使用的最少操作数, 那么 $f(m, n)$ 就是我们要计算的word1和word2的编辑距离。

根据以往的经验, 我们要定义的DP数组是一个 m 行 n 列的矩阵。下面我们先来考虑边界条件, 也就是DP数组的第1行和第1列:

1. DP数组的第1行 $f(1, j)$, 即word1的第1个字母转换成word2的前 j 个字母所使用的最少操作数。

下面我们先举几组具体的例子:

- (1) 假设word1第1个字母为'a', word2='mean'
 $f(1, 1)=1$, 因为'a'转换为'm'只需要经过一次替换;
 $f(1, 2)=2$, 因为'a'转换为'me'需要经过一次替换和一次插入;
 $f(1, 3)=2$, 因为'a'转换为'mea'需要经过两次插入;
 $f(1, 4)=3$, 因为'a'转换为'mean'需要经过三次插入。
- (2) 假设word1第1个字母为'a', word2='add'
 $f(1, 1)=0$, 因为'a'转换为'a'不需要操作;
 $f(1, 2)=1$, 因为'a'转换为'add'需要经过一次插入;
 $f(1, 3)=2$, 因为'a'转换为'add'需要经过两次插入。

作业5参考答案

- (3) 假设word1第1个字母为'a', word2='her'
 $f(1,1)=1$, 因为'a'转换为'h'只需要经过一次替换;
 $f(1,2)=2$, 因为'a'转换为'he'需要经过一次替换和一次插入;
 $f(1,3)=3$, 因为'a'转换为'her'需要经过一次替换和两次插入。
- (4) 假设word1第1个字母为'a', word2='ada'
 $f(1,1)=0$, 因为'a'转换为'a'不需要操作;
 $f(1,2)=1$, 因为'a'转换为'ad'需要经过一次插入;
 $f(1,3)=2$, 因为'a'转换为'ada'需要经过两次插入。
- (5) 假设word1第1个字母为'a', word2='cada'
 $f(1,1)=1$, 因为'a'转换为'c'只需要经过一次替换;
 $f(1,2)=1$, 因为'a'转换为'ca'需要经过一次插入;
 $f(1,3)=2$, 因为'a'转换为'cad'需要经过两次插入;
 $f(1,4)=3$, 因为'a'转换为'cada'需要经过三次插入。

总结规律:

如果word1的第1个字母不在word2中, 那么 $f(1,j)=j$;

如果word1的第1个字母在word2中, 那么找到这个字母首次出现在word2的位置, 如果位置为1(即word1和word2的第1个字母相同), 那么 $f(1,j)=j-1$; 如果位置不为1, 那么该位置之前的 $f(1,j)=j$, 该位置以及该位置之后的 $f(1,j)=j-1$ 。



作业5参考答案

2. DP数组的第1列 $f(i, 1)$, 即word1的前 i 个字母转换成word2的第1个字母所使用的最少操作数。注意, 两个单词之间的编辑距离具有对称性, 因此 $f(i, 1)$ 也等于word2的第1个字母转换成word1的前 i 个字母所使用的最少操作数。

那么这个规律就可以套用第1行的规律解决。

(1) 如果word2的第1个字母不在word1中, 那么 $f(i, 1)=i$;

(2) 如果word2的第1个字母在word1中, 那么找到这个字母首次出现在word1的位置, 如果位置为1(即word1和word2的第1个字母相同), 那么 $f(i, 1)=i-1$; 如果位置不为1, 那么该位置之前的 $f(i, 1)=i$, 该位置以及该位置之后的 $f(i, 1)=i-1$ 。

word1的前 i 个字母	word2的第1个字母	最少操作数
a b c	a	0 1 2
c d e	a	1 2 3
c a s d	a	1 1 2 3
c d a d a b	a	1 2 2 3 4 5

作业5参考答案

记 $f(i, j)$ 表示word1的前 i 个字母转换成word2的前 j 个字母所使用的最少操作数

最后我们再给出当 i 和 j 都大于2时,
 $f(i, j)$ 的一个递推公式:

$$\text{记 } tmp1 = f(i-1, j-1) + \begin{cases} 0, & \text{word1}(i) = \text{word2}(j) \\ 1, & \text{word1}(i) \neq \text{word2}(j) \end{cases}$$

$$\text{记 } tmp2 = f(i-1, j) + 1$$

$$\text{记 } tmp3 = f(i, j-1) + 1$$

$$\text{那么 } f(i, j) = \min\{tmp1, tmp2, tmp3\}$$

怎么理解上面的公式呢? 大家可以看下面三个问题: (记符号word[1..p]表示word的前p个字母)

问题1: 如果 word1[1..i-1] 到 word2[1..j-1] 的变换需要消耗 a 步, 那 word1[1..i] 到 word2[1..j] 的变换需要消耗几步呢?

答: 先把 word1[1..i-1] 变换到 word2[1..j-1], 消耗掉 a 步, 再把 word1[i] 改成 word2[j], 就行了。这里分为两种情况: 如果 $\text{word1}[i] == \text{word2}[j]$, 什么也不用做, 一共消耗 a 步; 否则需要替换最后一个字符, 一共消耗 $a + 1$ 步。

问题2: 如果 word1[1..i-1] 到 word2[1..j] 的变换需要消耗 b 步, 那 word1[1..i] 到 word2[1..j] 的变换需要消耗几步呢?

答: 先把 word1[1..i-1] 变换到 word2[1..j], 消耗掉 b 步, 再把 word1[i] 删除, 这样, word1[1..i] 就完全变成了 word2[1..j] 了。一共消耗 $b + 1$ 步。

问题3: 如果 word1[1..i] 到 word2[1..j-1] 的变换需要消耗 c 步, 那 word1[1..i] 到 word2[1..j] 的变换需要消耗几步呢?

答: 先把 word1[1..i] 变换成 word2[1..j-1], 消耗掉 c 步, 接下来, 再插入一个字符 word2[j], word1[1..i] 就完全变成了 word2[1..j] 了。一共消耗 $c + 1$ 步。

从上面三个问题来看, word1[1..i] 变换成 word2[1..j] 主要有三种操作, 哪种操作步数最少就用哪种。

参考: 力扣上Allen的评论: <https://leetcode-cn.com/problems/edit-distance/comments/>



数学建模学习交流

作业5参考答案

带注释的代码太长了课件放不下, 这里删除了注释, 如果看不明白可以看m文件里面的代码

```
function f = homework5(word1,word2)
m = length(word1);
n = length(word2);
FF = ones(m,n);
ind = strfind(word2, word1(1));
for j = 1:n
    if isempty(ind)
        FF(1,j)=j;
    elseif ind(1)==1
        FF(1,j)=j-1;
    else
        if j < ind(1)
            FF(1,j)=j;
        else
            FF(1,j)=j-1;
        end
    end
end
ind = strfind(word1, word2(1));
for i = 1:m
    if isempty(ind)
```

```
        FF(i,1)=i;
    elseif ind(1)==1
        FF(i,1)=i-1;
    else
        if i < ind(1)
            FF(i,1)=i;
        else
            FF(i,1)=i-1;
        end
    end
end
for i = 2:m
    for j = 2:n
        tmp1 = FF(i-1,j-1) + (word1(i) ~= word2(j))*1;
        tmp2 = FF(i-1,j) + 1;
        tmp3 = FF(i,j-1) + 1;
        FF(i,j) = min([tmp1,tmp2,tmp3]);
    end
end
f = FF(m,n);
end
```