目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） **课时安排: 2学时**

第十二讲： 贪心算法 **授课时间 :第十二周周一第1、2节**

教学内容（包括基本内容、重点、难点）：

基本内容： （1）贪心算法：贪心与动态规划算法的异同点；

（2）活动选择问题；

（3）0-1背包问题;

（4）钱币找零问题.

教学重点、难点：重点为贪心算法与动态规划的不同点、贪心算法的原理

教学媒体的选择：本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

第十二讲 贪心算法

一. 贪心算法与动态规划

1. 动态规划

动态规划是用来求解多阶段决策过程最优化问题的一种方法。多阶段决策过程本意是指有这样一类活动，他们可以按照时间顺序分解为若干个互相联系的阶段，称为时段。每一个时段都要做出一个决策，使得整个活动的总体效果最优。

由上述可知，动态规划方法与时间关系很密切,随着时间过程的发展而决定各阶段的决策，产生一决策序列，这就是动态的意思。

2.贪心算法

贪心算法不同于动态规划，贪心算法只考虑本阶段需要作出的最优选择，一般不用从整体最优上进行考虑，所做到的就是局部意义上的最优解，再由局部最优解得到全局最优解。因为每一步都只考虑对自己最优的情况，而忽略整体情况，故称之为贪心。

3.贪心算法与动态规划异同点

贪心算法与动态规划都是用来求解最优化问题的，他们之间有什么相似与相异的性质呢?

动态规划两大性质:

- 最优子结构
- 重叠子问题

贪心算法两大性质:

- 最优子结构
- 贪心选择

二. 活动选择问题

1. 活动选择问题问题描述

假设我们存在这样一个活动集合 $S = a_1, a_2, \cdots, a_n$,其中每一个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i 保证 $(0 \leq s_i < f_i)$,活动a_i 进行时，那么它占用的时间为 $[s_i, f_i)$.现在这些活动占用一个共同的资源，就是这些活动会在某一时间段里面进行安排，如果两个活动 a_i 和 a_j 的占用时间 $[s_i, f_i), [s_j, f_j)$ 不重叠，那么就说明这两个活动是兼容的，也就是说当 $s_i \leq f_j$ 或者 $s_j \leq f_i$,那么活动a_i, a_j 是兼容的。

比如下面的活动集合 S：我们假定在这个活动集合里面，都是按照 f_i 进行升序排序的, 即

$$0 \leq f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$$

活动	i	1	2	3	4	5	6	7	8	9	10	11
开始时间	s_i	1	3	0	5	3	5	6	8	8	2	12
结束时间	f_i	4	5	6	7	9	9	10	11	12	14	16

从上面可见，我们观察可得兼容子集有: $\{a_3, a_9, a_{11}\}$, 但是这个并不是最大兼容子集，因为 $\{a_1, a_4, a_8, a_{11}\}$ 也是这个活动的最大兼容子集，于是我们将活动选择问题描述为：给定一个集合 $S = \{a_1, a_2, a_3, \cdots, a_n\}$, 在相同的资源下，求出最大兼容活动的个数.

2. 动态规划解决过程

（1）活动选择问题的最优子结构

定义子问题解空间 S_{ij} 是 S 的子集，其中的每个获得都是互相兼容的。即每个活动都是在 a_i 结束之后开始，且在 a_j 开始之前结束。

$$S_{ij} = \left\{a_k \in S : f_i \leq s_k < f_k \leq s_j\right\}$$

为了方便讨论和后面的计算，添加两个虚构活动 a_0 和 a_{n+1}，其中 $f_0 = 0$，$s_{n+1} = \infty$。

- **结论：当 $i \geq j$ 时，S_{ij} 为空集。**

如果活动按照结束时间单调递增排序，子问题空间被用来从 S_{ij} 中选择最大兼容活动子集，其中 $0 \leq i < j \leq n+1$，所以其他的 S_{ij} 都是空集。

- **最优子结构为：**假设 S_{ij} 的最优解 A_{ij} 包含活动 a_k，则对 S_{ik} 的解 A_{ik} 和 S_{kj} 的解 A_{kj} 必定是最优的。

通过一个活动 a_k 将问题分成两个子问题，下面的公式可以计算出 S_{ij} 的解 A_{ij}。

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

（2）一个递归解

设 $c[i][j]$ 为 S_{ij} 中最大兼容子集中的活动数目，

- 当 S_{ij} 为空集时，$c[i][j] = 0$；
- 当 S_{ij} 非空时，若 a_k 在 S_{ij} 的最大兼容子集中被使用，则问题 S_{ik} 和 S_{kj} 的最大兼容子集也被使用，故可得到
$$c[i][j] = c[i][k] + c[k][j] + 1$$
- 当 $i \geq j$ 时，S_{ij} 必定为空集，否则 S_{ij} 则需要根据上面提供的公式进行计算，如果找到一个 a_k，则 S_{ij} 非空（此时满足 $f_i \leq s_k$ 且 $f_k \leq s_j$），找不到这样的 a_k，则 S_{ij} 为空集。

$c[i][j]$ 的完整计算公式如下所示：

$$c[i][j] = \begin{cases} 0 & S_{ij} = \varnothing \\ \max\{c[i][k] + c[k][j] + 1\} & S_{ij} \neq \varnothing \end{cases}$$

（3）时间复杂度

动态规划的时间复杂度与问题的个数以及每个问题的选择数有关。

比如这里的 $S(i,j)$一共大约有 N^2 个， 因为 $1 \leq j \leq N$ ， $1 \leq i < j$ ，这里求和大约是 $(N^2)/2$, 每个 $S(i,j)$ 一共有 $j - i + 1$ 种选择

$$\sum_{j=1}^{N} \sum_{i=1}^{j} (j - i + 1) = O(N^3)$$

从而动态规划的时间复杂度为 $O(N^3)$.

动态规划 Python 代码：实现活动选择问题

In [ ]:

```
import numpy as np


# 第一行输入活动的个数，往后每行输入活动的开始时间，结束时间，以及权重
# 输出为选择的活动以及可以获得的最大效益


def input_data():
    data_ = [[0, 0, 0]]
    n_ = int(input("请输入活动的个数"))
    for i in range(0, n_):
        temp = []
        s, f, w = map(int, input().split())
        temp.extend([s, f, w])
        data_.append(temp)
    return n_, data_


def compute_pj(local_data, local_n):
    local_data[0].append(0)
    for i in range(1, local_n + 1):
        for j in range(i - 1, -1, -1):
            if local_data[i][0] >= local_data[j][1]:
                local_data[i].append(j)
                break
    return local_data


def dynamic_programming(local_data, n_):
    opt = np.zeros([n_ + 1], dtype=int)
    select_flag_ = [0]
    for i in range(1, n_ + 1):
        if opt[i - 1] > opt[local_data[i][3]] + local_data[i][2]:  # 没有选jou
            opt[i] = opt[i - 1]
            select_flag_.append(0)
        else:
            opt[i] = opt[local_data[i][3]] + local_data[i][2]
            select_flag_.append(1)
    return opt, select_flag_


if __name__ == '__main__':
    n, data = input_data()
    data = compute_pj(data, n)
    dp, select_flag = dynamic_programming(data, n)
    print(dp[n])
```

```
    while True:
        if n == 0:
            break
        if select_flag[n]:
            print(data[n])
            n = data[n][3]
        else:
            n = n - 1
```

3. 贪心算法解决过程

虽然使用动态规划可以解决问题，但是动态规划的时间复杂度要高些，并且动态规划要解决的子问题比较多些。

贪心选择就是从直觉上选择当前应该是最佳的解决方案，那么两个子问题中因为有了贪心选择就只剩下一个待解决的子问题了。**在活动选择这里，我们做的选择就是结束最早的活动**。那怎么证明最早结束的就一定存在在最优解中呢?

定理：在非空子问题 S_k 中，如果 a_m 是 S_k 中结束最早的活动，那么 a_m 在 S_k 的某个最大兼容活动集中。

证明：设 A_k 是 S_k 的一个最大兼容活动集，且 a_j 是 A_k 中最早结束的活动，若 $a_j == a_m$，则证明了定理，若 $a_j \neq a_m$，令 $A'_k = (A_k - a_j) \cup a_m$，因为 $f(a_m) < f(a_j)$,所以 A'_k 中的活动必然是不相交的，那么就可以得出 A'_k 也是 S_k 的一个最大兼容活动子集。从而证明了定理。

4. 贪心算法时间复杂度

这里从理论上来分析下：因为对于贪心算法而言，每次只有一种选择即贪心选择，而动态规划中每个问题 $S(i,j)$ 中 $j-i+1$ 种选择。

贪心算法做出一次贪心选择后，即选中某个活动后，活动个数减少1，即问题规模减少1。贪心算法的时间复杂度为 $O(N)$.

贪心算法Python代码1：递归实现

In [2]:

```
# k is the activity which has finished and n has't finished
def greedyActivity(s, f, k, n):
    m = k + 1
    while (m <=n and s[m] < f[k]):
        m += 1
    if(m <= n):
        x = [m];
        x += greedyActivity(s, f, m, n)
        return x
    else:
        return []


#in order let the first activity in the result ,we must make up
#an activity as the one  activity finished before the first

s = [-2,1,3,0,5,3,5,6,8,8,2,12]
f = [-1,4,5,6,7,8,9,9,10,11,12,14,16]
k = 0
n = 11
z = greedyActivity(s,f,k,n)
print(z)
```

```
[1, 4, 8, 11]
```

贪心算法Python代码2：非递归实现

In [4]:

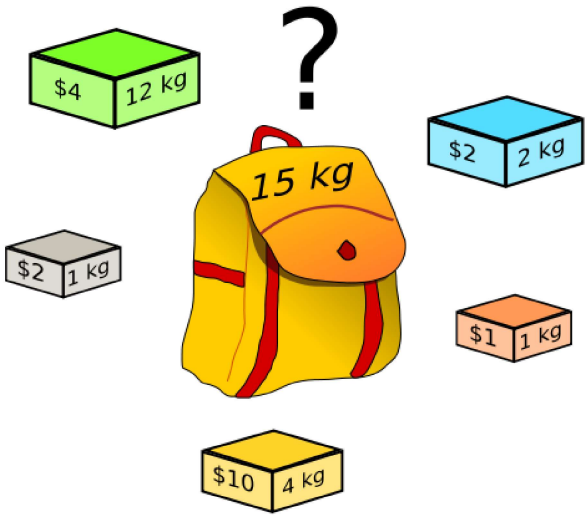
```
#make a for loop and every time findintg the earlist finished activity
# then change the k value
def greedyActivityAdvance(s,f,k,n):
    x = [s[1]]
    k = 1
    for m in range(2,n+1) :
        if( s[m] >= f[k]):
            x.append(m)
            k = m
    return x
s = [-2,1,3,0,5,3,5,6,8,8,2,12]
f = [-1,4,5,6,7,8,9,9,10,11,12,14,16]
k = 0
n = 11
zz = greedyActivityAdvance(s,f,k,n)
print(zz)
```

```
[1, 4, 8, 11]
```

三. 0-1背包问题

1. 0-1背包问题背景

假设我们有 n 件物品，分别编号为 $1,2,\cdots,n$。其中编号为 i 的物品价值为 v_i，它的重量为 w_i。为了简化问题，假定价值和重量都是整数值。现在，假设我们有一个背包，它能够承载的重量是W。现在，我们希望往包里装这些物品，使得包里装的物品价值最大化，那么我们该如何来选择装的东西呢？问题结构如下图所示：



0-1背包问题：假定我们这里选取的物品每个都是独立的，不能选取部分。也就是说我们要么选取某个物品，要么不能选取，不能只选取一个物品的一部分。这种情况，我们称之为0-1背包问题。不能用贪心算法。

部分背包（fractional knapsack)问题：而如果我们可以使用部分的物品的话，这个问题则成为部分背包问题。可以用贪心算法。

动态规划求解0-1背包问题:

我们定义一个函数 $c[i,w]$ 表示到第 i个元素为止，在限制总重量为 w 的情况下我们所能选择到的最优解。那么这个最优解要么包含有 i 这个物品，要么不包含，肯定是这两种情况中的一种。如果我们选择了第 i 个物品，那么实际上这个最优解是 $c[i-1,w-w_i]+v_i$。而如果我们没有选择第i个物品，这个最优解是 $c[i-1,w]$。这样，实际上对于到底要不要取第 i 个物品，我们只要比较这两种情况，哪个的结果值更大就是最优。

还有一个情况我们需要考虑的就是，我们这个最优解是基于选择物品i时总重量还是在 w 范围内的，如果超出了呢？我们肯定不能选择它，这就和 $c[i-1,w]$ 一样。 这里有一点值得注意，这里的 w_i 指的是第i个物品的重量，而不是到第 i 个物品时的总重量。另外，对于初始的情况呢？很明显 $c[0,w]$ 里不管 w 是多少，肯定为 0。因为它表示我们一个物品都不选择的情况。$c[i,0]$ 也一样，当我们总重量限制为 0 时，肯定价值为0。这样，基于我们前面讨论的这3个部分，我们可以得到一个如下的递推公式：

$$c[i,w]=\begin{cases}0 & \text{if } i=0 \text{ or } w=0,\\ c[i-1,w] & \text{if } w_i>w,\\ \max(v_i+c[i-1,w-w_i],\ c[i-1,w]) & \text{if } i>0 \text{ and } w\geq w_i.\end{cases}$$

我们再来考虑一下具体实现的细节。这一组物品分别有价值和重量，我们可以定义两个数组 $int[]v, int[]w$。其中 $v[i]$ 表示第i个物品的价值，$w[i]$ 表示第i个物品的重量。为了表示 $c[i,w]$，我们可以使用一个 $int[i][w]$ 的矩阵。其中i的最大值为物品的数量，而w表示最大的重量限制。按照前面的递推关系，$c[i][0]$ 和 $c[0][w]$ 都是 0。而我们所要求的最终结果是 $c[n][w]$。所以我们实际中创建的矩阵是 $(n+1)\times(w+1)$的规格。

动态规划求解0-1背包问题的时间和空间复杂度均为 $O(N * W)$

动态规划 Python代码：实现0-1背包问题

In [ ]:

```
import numpy as np

def solve(vlist,wlist,totalWeight,totalLength):
    resArr = np.zeros((totalLength+1,totalWeight+1),dtype=np.int32)
    for i in range(1,totalLength+1):
        for j in range(1,totalWeight+1):
            if wlist[i] <= j:
                resArr[i,j] = max(resArr[i-1,j-wlist[i]]+vlist[i],resArr[i-1,j
            else:
                resArr[i,j] = resArr[i-1,j]
    return resArr[-1,-1], resArr

if __name__ == '__main__':
    v = [0,60,100,120]
    w = [0,10,20,30]
    weight = 50
    n = 3
    result, res = solve(v,w,weight,n)
    print("所能装的最大价值为：",result)
    print(res)
```

四. 赫夫曼编码

1. 赫夫曼编码问题的描述

赫夫曼编码是 1952 年由 David A. Huffman 提出的一种无损数据压缩的编码算法。赫夫曼编码先统计出每种字母在字符串里出现的频率，根据频率建立一棵路径带权的二叉树，也就是哈夫曼树，树上每个结点存储字母出现的频率，根结点到结点的路径即是字母的编码，频率高的字母使用较短的编码，频率低的字母使用较长的编码，使得编码后的字符串占用空间最小。

首先统计每个字母在字符串里出现的频率，我们把每个字母看成一个结点，结点的权值即是字母出现的频率，我们把每个结点看成一棵只有根结点的二叉树，一开始把所有二叉树都放在一个集合里。接下来开始如下编码：

- 步骤一：从集合里取出两个根结点权值最小的树 a 和 b，构造出一棵新的二叉树 c，二叉树 c 的根结点的权值为 a 和 b 的根结点权值和，二叉树 c 的左右子树分别是 a 和 b。
- 步骤二：将二叉树 a 和 b 从集合里删除，把二叉树 c 加入集合里。

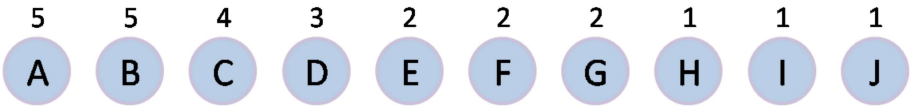
重复以上两个步骤，直到集合里只剩下一棵二叉树，最后剩下的就是哈夫曼树了。

我们规定每个有孩子结点的结点，到左孩子结点的路径为 0，到右孩子结点的路径为 1。每个字母的编码就是根结点到字母对应结点的路径。

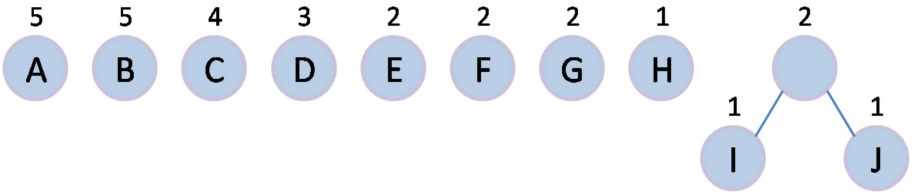
例如有这一个字符串“good good study day day up”，现在我们要对字符串进行哈夫曼编码，该字符串一共有 26 个字符，10 种字符，我们首先统计出每个字符的频率，然后按从大到小顺序排列如下（第二列的字符是空格）：

字符	‘d’	‘ ’	‘o’	‘y’	‘g’	‘u’	‘a’	‘s’	‘t’	‘p’
编号	A	B	C	D	E	F	G	H	I	J
频率	5	5	4	3	2	2	2	1	1	1

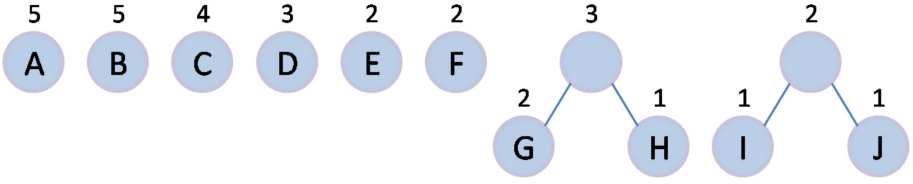
我们把每个字符看成一个结点，权值是字符的频率，每个字符开始都是一棵只有根结点的二叉树，如下图



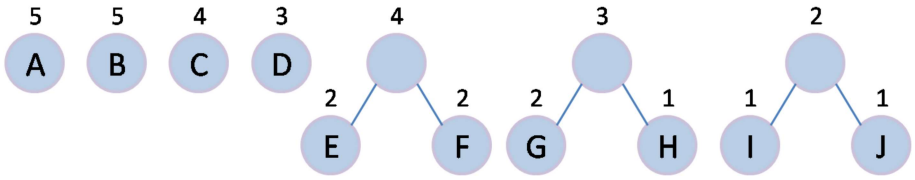
（1）从集合里取出根结点权值最小的两棵树 I 和 J 组成新的二叉树 IJ，根结点权值为 1 + 1 = 2，将二叉树IJ 加入集合，把 I 和 J 从集合里删除，如下图



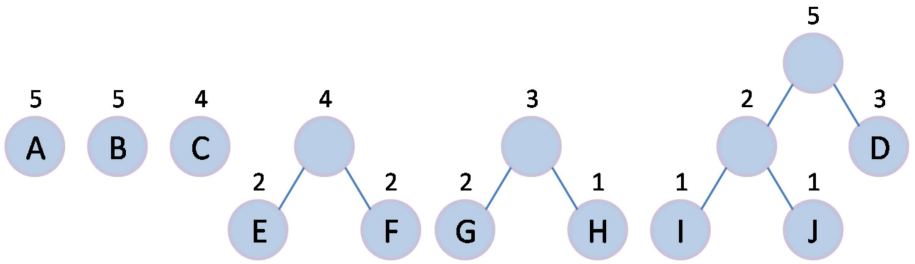
（2）从新集合里取出根结点权值最小的两棵树 H 和 G 组成新的二叉树 HG，根结点权值为 1 + 2 = 3，将二叉树 HG 加入集合，把 H 和 G 从集合里删除，如下图



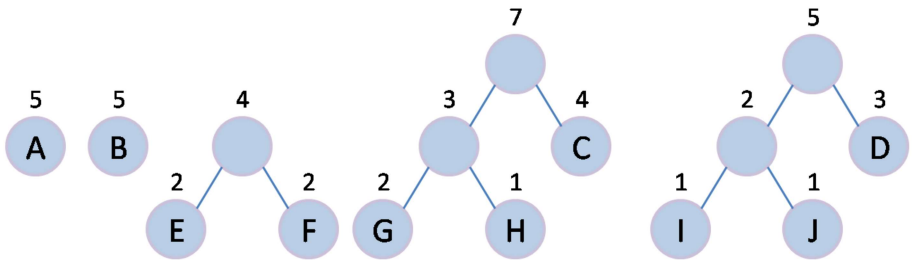
（3）从集合里取出根结点权值最小的两棵树 E 和 F 组成新的二叉树 EF，根结点权值为 2 + 2 = 4，将二叉树 EF 加入集合，把 E 和 F 从集合里删除，如下图



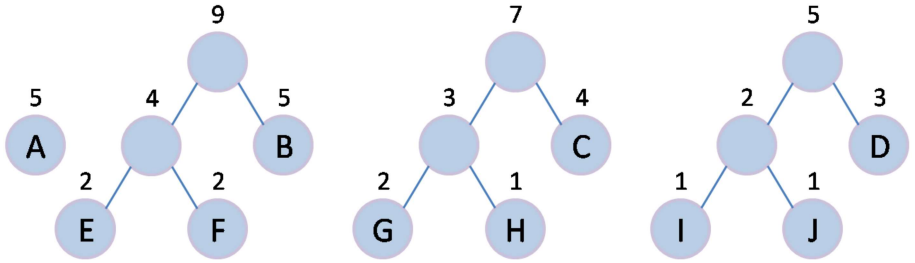
（4）从集合里取出根结点权值最小的两棵树 IJ 和 D 组成新的二叉树 IJD，根结点权值为 2 + 3 = 5，将二叉树 IJD 加入集合，把 IJ 和 D 从集合里删除，如下图



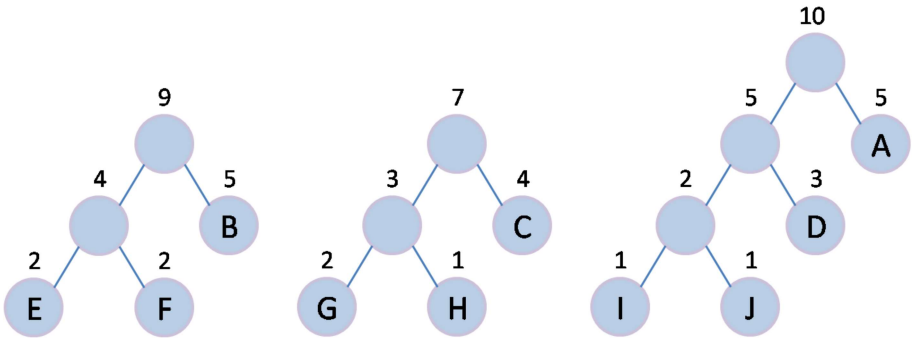
（5）从集合里取出根结点权值最小的两棵树 GH 和 C 组成新的二叉树 GHC，根结点权值为 3 + 4 = 7，将二叉树 GHC 加入集合，把 GH 和 C 从集合里删除，如下图



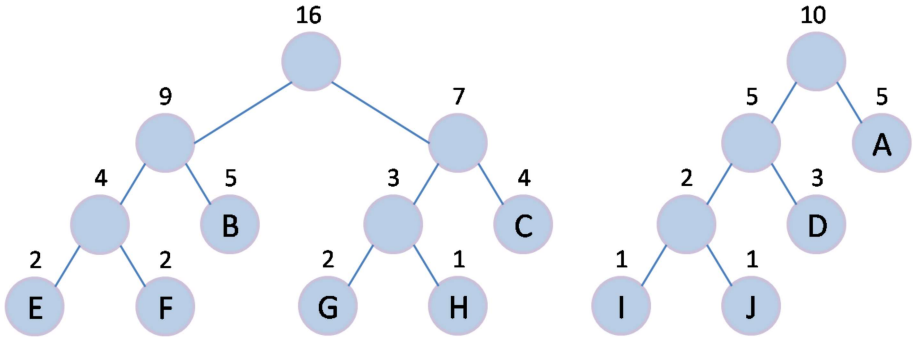
（6）从集合里取出根结点权值最小的两棵树 EF 和 B 组成新的二叉树 EFB，根结点权值为 4 + 5 = 9，将二叉树 EFB 加入集合，把 EF 和 B 从集合里删除，如下图



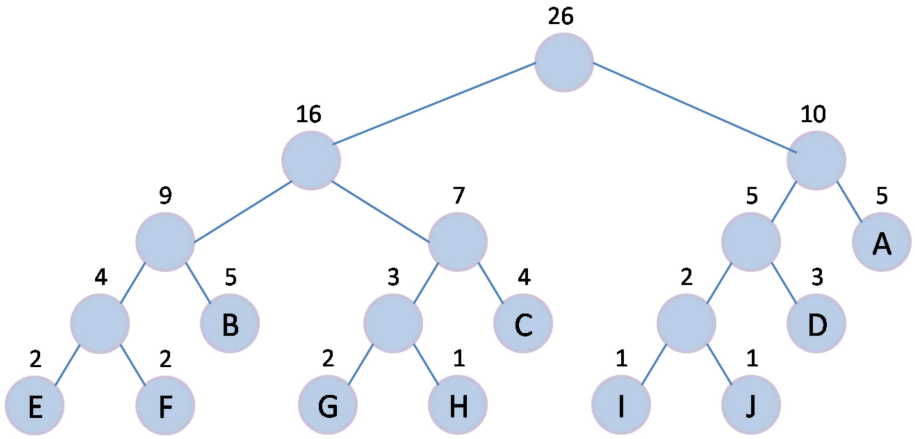
（7）从集合里取出根结点权值最小的两棵树 IJD 和 A 组成新的二叉树 IJDA，根结点权值为 5 + 5 = 10，将二叉树 IJDA 加入集合，把 IJD 和 A 从集合里删除，如下图



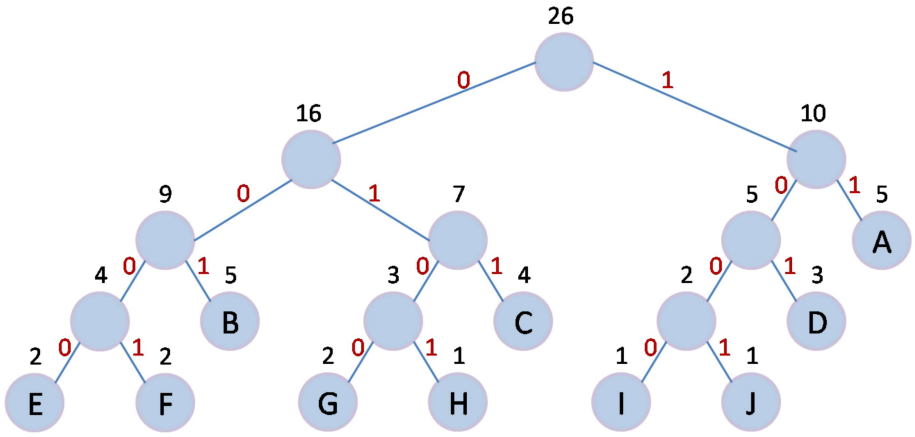
（8）从集合里取出根结点权值最小的两棵树 EFB 和 GHC 组成新的二叉树 EFBGHC，根结点权值为 9 + 7 = 16，将二叉树 EFBGHC 加入集合，把 EFB 和 GHC 从集合里删除，如下图。



（9）从集合里取出根结点权值最小的两棵树 EFBGHC 和 IJDA 组成新的二叉树 EFBGHCIJDA，根结点权值为 16 + 10 = 26，将二叉树 EFBGHCIJDA 加入集合，把 EFBGHC 和 IJDA 从集合里删除，如下图



到这里我们发现集合里就剩一棵二叉树了，那么编码结束，最后这棵二叉树就是我们要得到的哈夫曼树。接下来我们规定非叶子结点的结点，到左子树的路径记为 0，到右子树的路径记为 1，如下图：



根结点到每个叶子结点的路径便是其对应字母的编码了，于是我们可以得到

字符	'd'	' '	'o'	'y'	'g'	'u'	'a'	's'	't'	'p'
编号	A	B	C	D	E	F	G	H	I	J
频率	5	5	4	3	2	2	2	1	1	1
编码	11	001	011	101	0000	0001	0100	0101	1000	1001

$$\text{带权路径长度(WPL)} = 5 \times 2 + 5 \times 3 + 4 \times 3 + 3 \times 3 + 2 \times 4 + 2 \times 4 + 2 \times 4 + 1 \times 4 + 1 + 1 \times 4 = 82$$

现在我们计算如果直接存储字符串需要多少个比特：一个字符占一个字节，一个字节占 8 个比特，所以一共需要 8 * 26 = 208 个比特。 我们再来看看赫夫曼编码需要多少个比特，我们可以发现 WPL 也就是编码后原来字符串所占的比特总长度 82。显然，赫夫曼编码把原数据压缩了好多，而且没有损失。

2. 构造赫夫曼编码

赫夫曼设计了一个贪心算法来构造最有前缀码，被称为赫夫曼编码(Huffman code)。因为最有优缀码的问题具有贪心选择和最优子结构性质，从而，贪心算法HUFFMAN是正确的。

下面给出伪代码，我们假定 C 是一个 n 个字符的集合，而其中每个字符 $c \in C$ 都是一个对象，其属性 $c.freq$ 给出了字符的出现频率。算法自底向上地构造出对应最优编码的二叉树 T。它从 $|C|$ 个叶结点开始，执行 $|C| - 1$ 个“合并”操作创建出最终的二叉树。算法使用一个以属性 $freq$ 为关键字最小优先队列 Q，以识别两个最低频率的对象将其合并。当合并两个对象时，得到的新对象的频率设置为原来两个对象的频率之和。

```
HUFFMAN(C)
1 n=|C|
2 Q=C
3 for i=1 to n-1
4   allcate a new node z
5   z.left=x=EXTRACT-MIN(Q)
6   z.right=y=EXTRACT-MIN(Q)
7   z.freq=x.freq + y.freq
8   INSERT(Q,z)
9 return EXTRACT-MIN(Q)          // return the root of the tree
```

3. 赫夫曼算法的正确性

下面的定理证明了最优前缀码问题具有贪心选择性质：

引理 1 令 C 为一字母表，其中每个字符 $c\inC$ 都有一个频率 $c.freq$。令 x 和 y 是 C 中频率最低的两个字符。那么存在 C 的一个最有前缀码，x 和 y 的码字长都相同，且只有最后一个二进制位不同。

下面的定理证明了构造最优前缀码的问题具有最优子结构性质：

引理 2 令 C 为一字母表，其中每个字符 $c\inC$ 都有一个频率 $c.freq$。令 x 和 y 是 C 中频率最低的两个字符。令 C' 为 C 去掉字符 x 和 y，加入一个新字符 z 后得到的字母表，即 $C' = C - \{x, y\} \cup \{z\}$。类似 C，也为 C' 定义 $freq$，不同之处只是 $z.freq = x.freq + y.freq$。令T' 为字母表 C' 的任意一个最优前缀码对应的编码树。于是我们可以将 T' 中叶节点 z 替换为一个以 x 和 y为孩子的内部节点，得到树 T，而 T 表示字母表 C 的一个最有前缀码。

定理 1 过程 HUFFMAN 会生成一个最优前缀码。

2. 赫夫曼编码Python代码：

In [7]:

```
# 树节点类构建
class TreeNode(object):
  def __init__(self, data):
    self.val = data[0]
    self.priority = data[1]
    self.leftChild = None
    self.rightChild = None
    self.code = ""
# 创建树节点队列函数
def creatnodeQ(codes):
  q = []
  for code in codes:
    q.append(TreeNode(code))
  return q
# 为队列添加节点元素，并保证优先度从大到小排列
def addQ(queue, nodeNew):
  if len(queue) == 0:
    return [nodeNew]
  for i in range(len(queue)):
    if queue[i].priority >= nodeNew.priority:
      return queue[:i] + [nodeNew] + queue[i:]
  return queue + [nodeNew]
# 节点队列类定义
class nodeQeuen(object):

  def __init__(self, code):
    self.que = creatnodeQ(code)
    self.size = len(self.que)

  def addNode(self,node):
    self.que = addQ(self.que, node)
    self.size += 1

  def popNode(self):
    self.size -= 1
    return self.que.pop(0)
# 各个字符在字符串中出现的次数，即计算优先度
def freChar(string):
  d ={}
  for c in string:
    if not c in d:
      d[c] = 1
    else:
      d[c] += 1
  return sorted(d.items(),key=lambda x:x[1])
# 创建哈夫曼树
```

```
def creatHuffmanTree(nodeQ):
  while nodeQ.size != 1:
    node1 = nodeQ.popNode()
    node2 = nodeQ.popNode()
    r = TreeNode([None, node1.priority+node2.priority])
    r.leftChild = node1
    r.rightChild = node2
    nodeQ.addNode(r)
  return nodeQ.popNode()

codeDic1 = {}
codeDic2 = {}
# 由哈夫曼树得到哈夫曼编码表
def HuffmanCodeDic(head, x):
  global codeDic, codeList
  if head:
    HuffmanCodeDic(head.leftChild, x+'0')
    head.code += x
    if head.val:
      codeDic2[head.code] = head.val
      codeDic1[head.val] = head.code
    HuffmanCodeDic(head.rightChild, x+'1')
# 字符串编码
def TransEncode(string):
  global codeDic1
  transcode = ""
  for c in string:
    transcode += codeDic1[c]
  return transcode
# 字符串解码
def TransDecode(StringCode):
  global codeDic2
  code = ""
  ans = ""
  for ch in StringCode:
    code += ch
    if code in codeDic2:
      ans += codeDic2[code]
      code = ""
  return ans
# 举例
string = "AAGGDCCCDDDGFBBBFFGGDDDDGGGEFFDDCCCCDDFGAAA"
t = nodeQeuen(freChar(string))
tree = creatHuffmanTree(t)
HuffmanCodeDic(tree, '')
print(codeDic1,codeDic2)
a = TransEncode(string)
print(a)
```

```
aa = TransDecode(a)
print(aa)
print(string == aa)
```

```
{'E': '0000', 'B': '0001', 'A': '001', 'G': '01', 'D': '10', 'F': '11
0', 'C': '111'} {'0000': 'E', '0001': 'B', '001': 'A', '01': 'G', '10':
'D', '110': 'F', '111': 'C'}
00100101011011111111110101001110000100010001110110010110101010010101000
01101101010111111111111110101100100100 1001
AAGGDCCCDDDGFBBBFFGGDDDDGGGEFFDDCCCCDDFGAAA
True
```

4.12.钱币找零

计算机科学中的许多程序是为了优化一些值而编写的; 例如，找到两个点之间的最短路径，找到最适合一组点的线，或找到满足某些标准的最小对象集。计算机科学家使用许多策略来解决这些问题。本书的目标之一是向你展示几种不同的解决问题的策略。 动态规划 是这些类型的优化问题的一个策略。

优化问题的典型例子包括使用最少的硬币找零。假设你是一个自动售货机制造商的程序员。你的公司希望通过给每个交易最少硬币来简化工作。假设客户放入 1 美元的钞票并购买 37 美分的商品。你可以用来找零的最小数量的硬币是多少？答案是六个硬币：两个 25 美分，一个 10美分 和三个一美分。我们如何得到六个硬币的答案？我们从最大的硬币（25 美分）开始，并尽可能多，然后我们去找下一个小点的硬币，并尽可能多的使用它们。这第一种方法被称为贪婪方法，因为我们试图尽快解决尽可能大的问题。

当我们使用美国货币时，贪婪的方法工作正常，但是假设你的公司决定在埃尔博尼亚部署自动贩卖机，除了通常的 1，5，10 和 25 分硬币，他们还有一个 21 分硬币 。在这种情况下，我们的贪婪的方法找不到 63 美分的最佳解决方案。 随着加入 21分硬币，贪婪的方法仍然会找到解决方案是六个硬币。然而，最佳答案是三个 21 分。

让我们看一个方法，我们可以确定会找到问题的最佳答案。由于这一节是关于递归的，你可能已经猜到我们将使用递归解决方案。让我们从基本情况开始，如果我们可以与我们硬币的价值相同的金额找零，答案很容易，一个硬币。

如果金额不匹配，我们有几个选项。我们想要的是最低一个一分钱加上原始金额减去一分钱所需的硬币数量，或者一个 5 美分加上原始金额减去 5 美分所需的硬币数量，或者一个 10 美分加上原始金额减去 10 美分所需的硬币数量，等等。因此，需要对原始金额找零硬币数量可以根据下式计算:

$$numCoins = min\begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

执行我们刚才描述的算法如 Listing 7 所示。在第 3 行，我们检查基本情况;也就是说，我们正试图支付硬币的确切金额。如果我们没有等于找零数量的硬币，我们递归调用每个小于找零额的不同的硬币值。第 6 行显示了我们如何使用列表推导将硬币列表过滤到小于当前找零的硬币列表。递归调用也减少了由所选硬币的值所需要的总找零量。递归调用在第 7 行。注意在同一行，我们将硬币数 `+1` ，以说明我们正在使用一个硬币的事实。

```
def recMC(coinValueList,change):
    minCoins = change
    if change in coinValueList:
      return 1
    else:
       for i in [c for c in coinValueList if c <= change]:
          numCoins = 1 + recMC(coinValueList,change-i)
          if numCoins < minCoins:
             minCoins = numCoins
    return minCoins

print(recMC([1,5,10,25],63))
```

Listing 7

Listing 7 中的算法是非常低效的。事实上，它需要 `67,716,925` 个递归调用来找到 4 个硬币的最佳解决 63 美分问题的方案。要理解我们方法中的致命缺陷，请参见 Figure 5，其中显示了 377 个函数调用所需的一小部分，找到支付 26 美分的最佳硬币。

图中的每个节点对应于对 `recMC` 的调用。节点上的标签表示硬币数量的变化量。箭头上的标签表示我们刚刚使用的硬币。通过跟随图表，我们可以看到硬币的组合。主要的问题是我们重复做了太多的计算。例如，该图表示该算法重复计算了至少三次支付 15 美分。这些计算找到 15 美分的最佳硬币数量的步骤本身需要52个函数调用。显然，我们浪费了大量的时间和精力重新计算旧的结果。

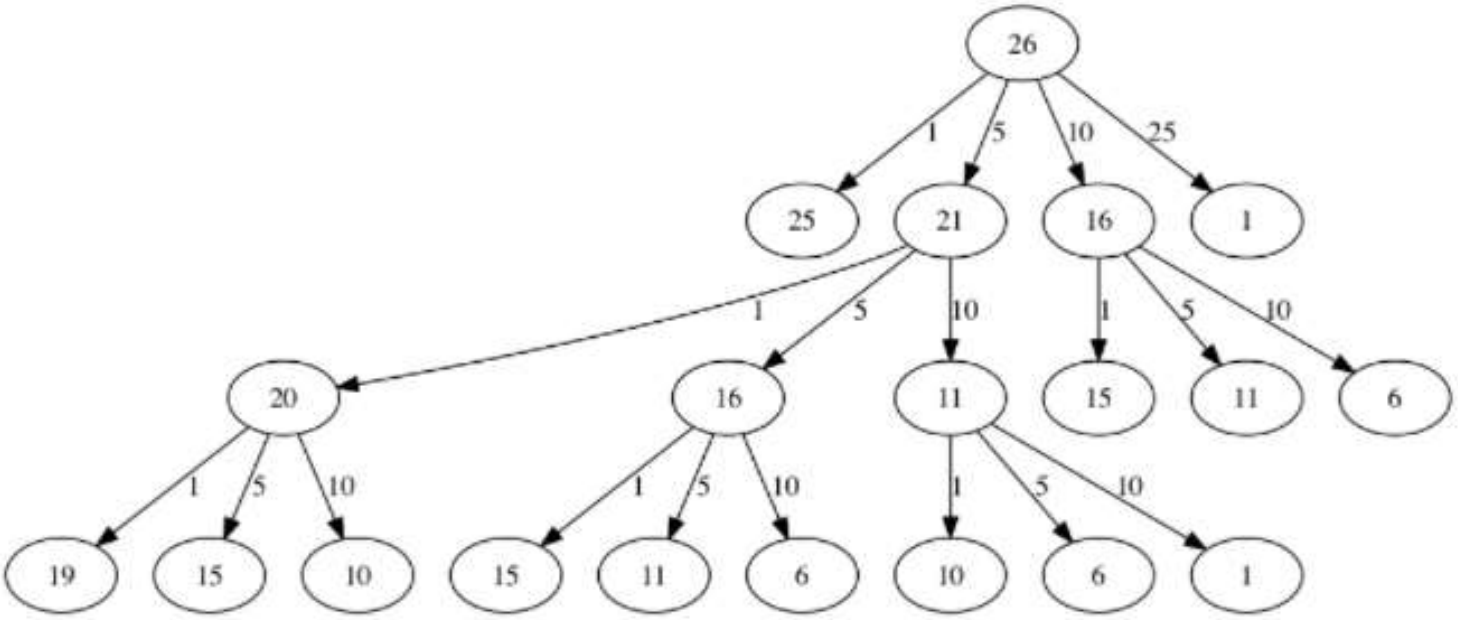


Figure 5

减少我们工作量的关键是记住一些过去的结果，这样我们可以避免重新计算我们已经知道的结果。一个简单的解决方案是将最小数量的硬币的结果存储在表中。然后在计算新的最小值之前，我们首先检查表，看看结果是否已知。如果表中已有结果，我们使用表中的值，而不是重新计算。 ActiveCode 1 显示了一个修改的算法，以合并我们的表查找方案。

```
def recDC(coinValueList,change,knownResults):
   minCoins = change
   if change in coinValueList:
      knownResults[change] = 1
      return 1
   elif knownResults[change] > 0:
      return knownResults[change]
   else:
       for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recDC(coinValueList, change-i,
                              knownResults)
         if numCoins < minCoins:
            minCoins = numCoins
            knownResults[change] = minCoins
   return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

ActiveCode 1

注意，在第 6 行中，我们添加了一个测试，看看我们的列表是否包含此找零的最小硬币数量。如果没有，我们递归计算最小值，并将计算出的最小值存储在列表中。使用这个修改的算法减少了我们需要为四个硬币递归调用的数量，63美分问题只需 221 次调用！

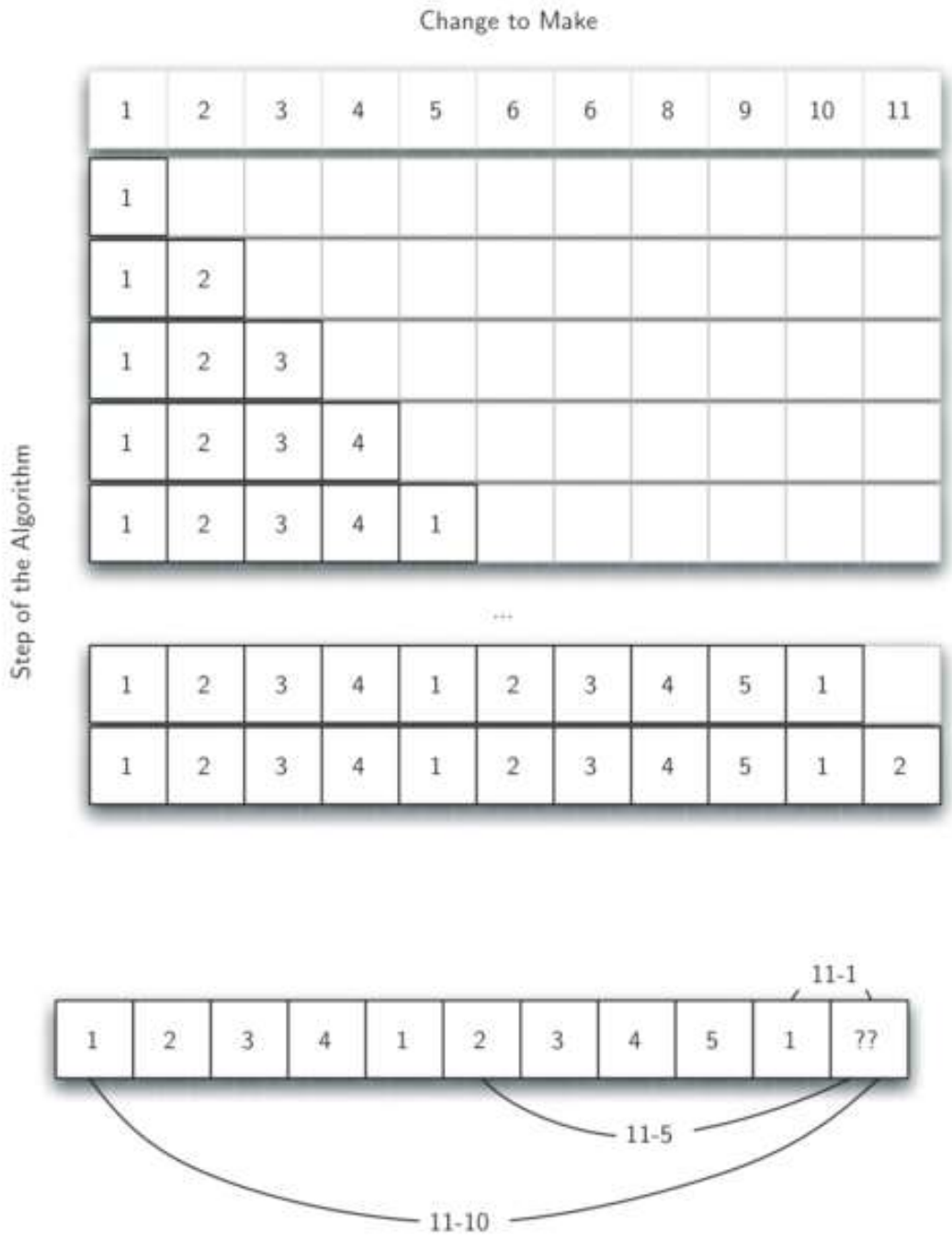
虽然 AcitveCode 1 中的算法是正确的。事实上，我们所做的不是动态规划，而是我们通过使用称为 记忆化 的技术来提高我们的程序的性能，或者更常见的叫做 缓存 。

一个真正的动态编程算法将采取更系统的方法来解决这个问题。我们的动态编程解决方案将从找零一分钱开始，并系统地找到我们需要的找零额。这保证我们在算法的每一步，已经知道为任何更小的数量进行找零所需的最小硬币数量。

让我们看看如何找到 11 美分所需的最小找零数量。Figure 4 说明了该过程。我们从一分钱开始。唯一的解决方案是一个硬币（一分钱）。下一行显示一分和两分的最小值。再次，唯一的解决方案是两分钱。第五行事情变得有趣。现在我们有两个选择，五个一分钱或一个五分钱。我们如何决定哪个是最好的？我们查阅表，看到需要找零四美分的硬币数量是四，再加一个一分钱是五，等于五个硬币。或者我们可以尝试 0 分加一个五分，五分钱等于一个硬币。由于一和五最小的是一，我们在表中存储为一。再次快进到表的末尾，考虑 11 美分。Figure 5 展示了我们要考虑的三个选项：

1. 一个一分钱加上 11-1 = 10分（1） 的最小硬币数
2. 一个五分钱加上 11-5 = 6分（2） 的最小硬币数
3. 一个十分钱加上 11-10 = 1 分（1） 最小硬币数

选项 1 或 3 总共需要两个硬币，这是 11 美分的最小硬币数。



Listing 8 用一个动态规划算法来解决我们的找零问题。 dpMakeChange 有三个参数：一个有效硬币值的列表，我们要求的找零额，以及一个包含每个值所需最小硬币数量的列表。 当函数完成时， minCoins 将包含从 0 到找零值的所有值的解。

```
def dpMakeChange(coinValueList,change,minCoins):
   for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
   return minCoins[change]
```

Listing 8

注意， dpMakeChange 不是递归函数，即使我们开始使用递归解决这个问题。重要的是要意识到，你可以为问题写一个递归解决方案但并不意味着它是最好的或最有效的解决方案。在这个函数中的大部分工作是通过从第 4 行开始的循环来完成的。在这个循环中，我们考虑使用所有可能的硬币对指定的金额进行找零。就像我们上面的 11 分的例子，我们记住最小值，并将其存储在我们的 minCoins 列表。

虽然我们的找零算法很好地找出最小数量的硬币，但它不帮助我们找零，因为我们不跟踪我们使用的硬币。我们可以轻松地扩展 dpMakeChange 来跟踪硬币使用，只需记住我们为每个条目添加的最后一个硬币到 minCoins 表。如果我们知道添加的最后一个硬币值，我们可以简单地减去硬币的值，在表中找到前一个条目，找到该金额的最后一个硬币。我们可以通过表继续跟踪，直到我们开始的位置。

ActiveCode 2 展示了 dpMakeChange 算法修改为跟踪使用的硬币，以及一个函数 printCoins 通过表打印出使用的每个硬币的值。前两行主要设置要找零的金额，并创建使用的硬币列表。 接下来的两行创建了我们需要存储结果的列表。 coinsUsed 是用于找零的硬币的列表，并且 coinCount 是与列表中的位置相对应进行找零的最小硬币数。

注意，我们打印的硬币直接来自 coinsUsed 数组。对于第一次调用，我们从数组位置 63 开始，然后打印 21 。然后我们取 63-21 = 42 ，看看列表的第 42 个元素。我们再次找到 21 存储在那里。 最后，数组第 21 个元素21 也包含 21，得到三个 21。

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
    for cents in range(change+1):
        coinCount = cents
        newCoin = 1
        for j in [c for c in coinValueList if c <= cents]:
              if minCoins[cents-j] + 1 < coinCount:
                 coinCount = minCoins[cents-j]+1
                 newCoin = j
        minCoins[cents] = coinCount
        coinsUsed[cents] = newCoin
    return minCoins[change]

def printCoins(coinsUsed,change):
    coin = change
    while coin > 0:
        thisCoin = coinsUsed[coin]
        print(thisCoin)
        coin = coin - thisCoin

def main():
     amnt = 63
     clist = [1,5,10,21,25]
     coinsUsed = [0]*(amnt+1)
     coinCount = [0]*(amnt+1)

     print("Making change for",amnt,"requires")
     print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
     print("They are:")
     printCoins(coinsUsed,amnt)
     print("The used list is as follows:")
     print(coinsUsed)

main()
```

AcitveCode 2

```
Making change for 63 requires
3 coins
They are:
21
21
21
The used list is as follows:
[1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 21, 1, 1, 1
, 25, 1, 1, 1, 1, 5, 10, 1, 1, 1, 10, 1, 1, 1, 1, 5, 10, 21, 1, 1, 10, 21, 1,
1, 1, 25, 1, 10, 1, 1, 5, 10, 1, 1, 1, 10, 1, 10, 21]
```

引用及参考：

[1]《Python数据结构与算法分析》布拉德利.米勒等著，人民邮电出版社，2019年9月.

[2] https://blog.csdn.net/ddupd/article/details/62328026 (https://blog.csdn.net/ddupd/article/details/62328026)

[3] https://www.jianshu.com/p/ed9b97b73f40 (https://www.jianshu.com/p/ed9b97b73f40)

[4] https://www.jianshu.com/p/25f4a183ede5 (https://www.jianshu.com/p/25f4a183ede5)

[5] https://www.jb51.net/article/166433.htm (https://www.jb51.net/article/166433.htm)

课后练习

1. 写出钢条切割问题的完整Python代码或 C语言代码。

2. 写出矩阵链乘法问题的完整Python代码或 C语言代码，求矩阵规模序列<5,10,3,12,5,50,6>的矩阵链的最优括号化方案。

3. 写出最长公共子序列问题的完整Python代码或 C语言代码，求<1,0,0,1,0,1,0,1>和<0,1,0,1,1,0,1,1,0>的所有LCS。

讨论、思考题、作业：

参考资料（含参考书、文献等）：算法导论. Thomas H. Cormen等，机械工业出版社，2017.

授课类型（请打√）：理论课□　　讨论课□　　实验课□　　练习课□　　其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。