算法分析第四章习题

1. 设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 $t_i, 1 \le i \le n$ 。应该如何安排 n 个顾客的服务次序才能使总的等待时间达到最小？总的等待时间是各顾客等待服务的时间的总和。试给出你的做法的理由（证明）。

n 个顾客按照需要的服务时间 t 从小到大（非降次序）依次服务

证明：

证明：设 $t_{i1} \le t_{i2} \le t_{i3} \le \cdots \le t_{in}$ 为服务的顺序

则等待时间为

$$T = (n-1)t_{i1} + (n-2)t_{i2} + \cdots + 2t_{in-2} + t_{in-1}$$

假设互换 i, j $(i<j)$ 两者的次序，则等待时间由

$$T = (n-1)t_{i1} + \cdots + (n-i)t_{ii} + \cdots + (n-j)t_{ij} + \cdots t_{in}$$

变为

$$T_{change} = (n-1)t_{i1} + \cdots + (n-i)t_{ij} + \cdots + (n-j)t_{ii} + \cdots + t_{in-1}$$

$$T_{change} - T = (j-i)(t_{ij} - t_{ii}) \ge 0$$

即不存在比所给服务顺序更优的序列，使得总服务时间减少。

2. 字符 $a \sim h$ 出现的频率分布恰好是前 8 个 Fibonacci 数，它们的 Huffman 编码是什么？将结果推广到 n 个字符的频率分布恰好是前 n 个 Fibonacci 数的情形。Fibonacci 数的定义为：$F_0 = 1, F_1 = 1, F_n = F_{n-2} + F_{n-1}\ if\ n > 1$

	a	b	c	d	e	f	g	h
频率	1	1	2	3	5	8	13	21
编码	1111111	1111110	111110	11110	1110	110	10	0

推广：

可以发现规律：

对于给出的 n 个字符，

第 1 个字符：n-1 个 1，0 个 0 组成，长度为 n-1

第 2 个字符：n-2 个 1，1 个 0 组成，长度为 n-1

第 3 个字符：n-3 个 1，1 个 0 组成，长度为 n-2（以下依次递减一个“1”，直到 0 个 1 结束）

....
第 n 个字符：0 个 1，1 个 0 组成，长度为 1

3. 设 $p_1, p_2, \cdots, p_n$ 是准备存放到长为 L 的磁带上的 n 个程序，程序 p_i 需要的带长为 a_i。设 $\sum_{i=1}^{n} a_i > L$，要求选取一个能放在带上的程序的最大子集合（即其中含有最多个数的程序）Q。构造 Q 的一种贪心策略是按 a_i 的非降次序将程序计入集合。

1) 证明这一策略总能找到最大子集 Q，使得 $\sum_{p_i \in Q} a_i \le L$。

2) 设 Q 是使用上述贪心算法得到的子集合，磁带的利用率可以小到何种程度？

3) 试说明 1)中提到的设计策略不一定得到使 $\sum_{p_i \in Q} a_i / L$ 取最大值的子集合。

证明：

按照贪心策略构造的子集Q为 $\{a_1, a_2, \cdots a_s\}$，其中 $a_1 \le a_2 \le \cdots \le a_n$，且 $\sum_{i=1}^{s} a_i \le L$，$\sum_{i=1}^{s} a_i + a_{s+1} > L$

假设存在多于s个的集合 $\tilde{Q} = \{\tilde{a}_1, \tilde{a}_2, \cdots \tilde{a}_k\}$ $(k>s)$，满足

$$\sum_{i=1}^{k} \tilde{a}_i \le L \quad ①$$

由于Q中是按照从小到大选的s个，而k>s，那么

$$\sum_{i=1}^{k} \tilde{a}_i > \sum_{i=1}^{s} \tilde{a}_i \ge \sum_{i=1}^{s} a_i$$

由于 $\tilde{Q}$ 满足条件①，也就是说Q中选出的s个元素在加入k-s个元素后仍然小于L，与按照贪心策略构造出的Q的性质矛盾，故假设不成立

磁带利用率最小为 0，所有的程序所需要的磁带长度都大于 L
例如磁带长度为 5，程序所需长度分别为 1，1，2，3，按照 1 中贪心算法，应当选取 1，1，2，但是另外一种数量相同的选取 1，1，3，能够让利用率变大，所以这种贪心算法无法取得利用率最大的子集合

4. 写出 Huffman 编码的伪代码，并编程实现。

伪代码：编程代码见 python 代码文件

```
frequency[char]; //  字符的频率表
Huffman[char]; //  字符的 Huffman 编码表，初始为空，也可以将这个融入 node 的属性中

define: node //堆中元素为 node 类,排序规则使用 node.value 的值
                   //node 属性：value left_node right_node
define: Smallest_Value_Heap //最小堆  Smallest_Value_Heap
define: map<node_address, char> //定义一个 map 将创建 node 后的地址与 char 对应

for char in char_set: //初始化堆
      create “node” : node.value = frequency[char]
      Smallest_Value_Heap.insert(“node”)
      map[“node”] = char

while Smallest_Value_Heap.length > 1: //堆的大小大于 1
      smallest_value_node_0 = Smallest_Value_Heap.delSmallestValue //堆删除后会自动调整
      smallest_value_node_1 = Smallest_Value_Heap.delSmallestValue
      value_0 = smallest_value_node_0.value
      value_1 = smallest_value_node_1.value
      create “newnode” : //创建新的节点
            newnode.value = value_0 + value_1
            newnode.left = smallest_value_node_0
            newnode.right = smallest_value_node_1
      insert “newnode” into Smallest_Value_Heap //将这个新节点插入

node HuffmanTreeHead = Smallest_Value_Heap.getSmallestValue //拿到哈夫曼树的顶点
dfsHuffmanTree(“”, HuffmanTreeHead )
                         //遍历哈夫曼树得到最终的哈夫曼编码，初始编码为空字符串
                         //走左子树的时候编码+”0”
                         //走右子树的时候编码+”1”

dfsHuffmanTree(String current_string, Node current_node):
      left_node = current_node.left
      right_node = current_node.right
      if (left_node == NULL and right_node == NULL)://到达根节点
            char current_char = map[current_node] //通过 map 找到当前根节点对应的 char
            Huffman[char] = current_string //给出编码
            return //  返回
      if (left_node != NULL)
            dfsHuffmanTree(current_string + “0”, left_node)
      if (right_node != NULL)
            dfsHuffmanTree(current_string + “1”, right_node)
```

运行结果示例，完整代码见附加文件：

Huffman.py > ...

```python
            dfsHuffmanTree(current_string + "1", current_node.right, Huffman)

def buildHuffmanTree(frequency):
    char_set = list(frequency.keys())
    Smallest_Value_Heap = []
    # 初始化堆
    for char in char_set:
        node = Node(char, frequency[char])
        heapq.heappush(Smallest_Value_Heap, node)

    # 构建哈夫曼树
    while len(Smallest_Value_Heap) > 1:
        smallest_value_node_0 = heapq.heappop(Smallest_Value_Heap)
        smallest_value_node_1 = heapq.heappop(Smallest_Value_Heap)
        newnode = Node(None, smallest_value_node_0.frequency + smallest_value_node_1.frequency)
        newnode.left = smallest_value_node_0
        newnode.right = smallest_value_node_1
        heapq.heappush(Smallest_Value_Heap, newnode)

    # 获取哈夫曼树的根节点
    HuffmanTreeHead = heapq.heappop(Smallest_Value_Heap)
    Huffman = {}
    dfsHuffmanTree("", HuffmanTreeHead, Huffman)
    return Huffman

# 示例
frequency = {'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45}
# a~h的频率遵循斐波那契数列
# frequency = {'a':1, 'b':1, 'c':2, 'd':3, 'e':5, 'f':8, 'g':13, 'h':21}
Huffman = buildHuffmanTree(frequency)
print(Huffman)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
{'f': '0', 'c': '100', 'd': '101', 'a': '1100', 'b': '1101', 'e': '111'}
```

5. 已知 n 种货币 $c_1, c_2, \cdots, c_n$ 和有关兑换率的 $n \times n$ 表 R，其中 $R[i, j]$ 是一个单位的货币 c_i 可以买到的货币 c_j 的单位数。

1） 试设计一个算法，用以确定是否存在一货币序列 $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ 使得：

$$R[i_1, i_2]R[i_2, i_3]\cdots R[i_k, i_1] > 1$$

2） 设计一个算法打印出满足 1）中条件的所有序列，并分析算法的计算时间。

按照要求不做

6.说明最优生成树问题具有拟阵结构，并给出赋值函数，解释 Prim 算法和 Kruskal 算法都能求得最优解。

见下页

设图为 $G=(V,E)$，则该图诱导出的数集合具有拟阵结构 $M=(E,\mathcal{F})$，其中 $\mathcal{F}$ 为所有独立边子集构成的族。易知 E 有限非空，且设赋权函数 $w(x),x\in E$ 为边 x 的权值，$w(A)=\sum_{x\in A}w(x)$。下证交换性质：
设 A 和 B 分别诱导出 G 的两个森林 T_1,T_2，设它们的分支数分别为 k_1,k_2，则 $|A|=n-k_1,|B|=n-k_2$。
由于 $|A|<|B|$，所以 $k_1>k_2$，这说明 B 中至少有一条边 e 使得其两个端点在 T_1 的不同分支上，这条边不属于 A。于是 $|A|\cup\{e\}\in\mathcal{J}$。
对于 Prim 算法，先找到不属于子树的边集合，使得 $A\cup\{x\}\in\mathcal{J}(M)$，再从中找最小的边纳入到原先的子树中；对于 Kruskal 算法，先按照赋权函数非减排列不属于子树的边集合，再从中从大到小找到满足 $A\cup\{x\}\in\mathcal{F}(M)$ 的边。
这两种算法增边的准则都要满足两个条件：a. 权值最小 b. 满足 $A\cup\{x\}\in\mathcal{F}(M)$，因此在执行条件的顺序上不影响算法。
又由 GreedyMatroid 算法可知这两种算法都满足 GreedyMatroid，因此由定理：设 $M=(S,\mathcal{J})$ 是赋权 w 的拟阵，则算法 GreedyMatroid(M,w) 返回的子集 A 是 M 的最优独立集，以及最优生成树问题具有拟阵结构可知，Prim 算法和 Kruskal 算法均具有最优性。