习题四

202328015926048-丁力

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 $t_i, 1 \leq i \leq n$。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)。

> 让当前等待时间最少的客户先接受服务，这样的策略能够让等待时间达到最小。
>
> 现在开始证明：
>
> 下面，我们利用数学归纳法进行证明：
>
> - 当n=2时：
>
> 假设此时的顾客为A和B（其中A和B任意），他们的服务时间为t_A和t_B，此时，不妨假设$t_A > t_B$，按照上述策略，先安排时间短的B，再安排A，总等待时间为A等待B的时间：
>
> $$t_{wait} = t_B$$
>
> 可知，如果先安排A的话，耗时为：
>
> $$t_{wait} = t_A > t_B$$
>
> 所以此时的安排策略是最优的。
>
> - 假设有$n = k$时安排策略总等待时间最短。
>   - 当$n = k + 1$时，
>   从 $k + 1$个顾客中移除一个服务时间最长的顾客，剩下的 k 个顾客应按服务时间从短到长的顺序进行服务，这是由归纳假设确定的。现在，我们再加入之前移除的那个顾客。为了最小化总的等待时间，这个顾客应当是最后一个得到服务的，因为他的服务时间是最长的。
>
> Q.E.D

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 $n > 1$

> 首先，我们需要确定前8个Fibonacci数。然后，我们可以构建一个Huffman编码树。
>
> Fibonacci数列的前8项为：
>
> $$F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, F_5 = 8, F_6 = 13, F_7 = 21$$
>
> 为了构建Huffman编码树，我们首先按照频率对字符进行排序：
>
> $$(a, 1), (b, 1), (c, 2), (d, 3), (e, 5), (f, 8), (g, 13), (h, 21)$$

我们可以从最小的两个频率开始，合并它们，然后继续合并，直到只剩下一个节点。合并过程中，我们会给左边的子节点分配一个'0'，给右边的子节点分配一个'1'。这样，当我们从根到叶子节点遍历时，就可以得到每个字符的Huffman编码。

现在，我们开始构建Huffman编码树并得到每个字符的编码。对于字符 a 到 h 和它们对应的前8个 Fibonacci 数的频率，我们得到了以下的 Huffman 编码：

$$
\begin{aligned}
h &: 0 \\
g &: 10 \\
f &: 110 \\
e &: 1110 \\
d &: 11110 \\
c &: 111110 \\
a &: 1111110 \\
b &: 1111111
\end{aligned}
$$

为了推广到 n 个字符的情况，我们可以观察上面的结果中的一些模式。特别是，我们可以看到字符的编码长度随着它们的频率的增加而减少。具体来说，最高频率的字符有最短的编码，而最低频率的字符有最长的编码。

另外，我们可以看到最低频率的两个字符的编码只在最后一位上有所不同，而其它部分是相同的。这是因为它们在 Huffman 树中是相邻的叶子节点，它们共享相同的父节点。

考虑到这些观察结果，对于 n 个字符的情况，我们可以预期：

1. 最高频率的字符将有最短的编码。
2. 最低频率的两个字符将有最长的编码，并且只在最后一位上有所不同。
3. 一起编码这n项数列时，其中第n项被编码成n位数，其中第一项数列被编码成0，第n项被编码成全为1的n位数， 其他的项被编码成除了末尾为0，其他位均为1的对应长度位数。

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 $\sum_{p_i \in Q} a_i \le L$ 。

> 证明：不妨设 $a_1 \le a_2 \le \ldots \le a_n$，若该贪心策略构造的子集合 Q 为 $\{a_1, a_2, \cdots, a_5\}$, 则 s 满足 $\sum_{i=1}^{s} \mathrm{a_i} \le \mathrm{L}$、$\sum_{i=1}^{s} \mathrm{a_s} + \mathrm{a}_{s+1} > \mathrm{L}$ 。
>
> 要证明能找到最大子集，只需说明 s 为可包含的最多程序段数即可。即证不存在多于 s 个的程序集合 $\tilde{Q} = \{a_{i_1}, a_{i_2}, \cdots, a_{i_1}\}, (k > s)$, 使得 $\sum_{p_i \in \mathscr{Q}} a_i \le L$ 。
>
> 反证法，假设存在多于 s 个的程序集 $\tilde{Q} = \{\mathrm{a_{i_1}, a_{i_2}, \cdots, a_{i_p}}\}, (\mathrm{k > s})$, 满足 $\sum_{\mathrm{j=1}}^{\mathrm{k}} \mathrm{a_{i_j}} \le \mathrm{L}$ 。
>
> 因为$\mathrm{a_1 \le a_2 \le \ldots \le a_n}$ 非降序排列，则 $\mathrm{a_1 + a_2 + \cdots a_5 + \cdots + a_k \le a_{i_1} + a_{i_2} + \cdots + a_{i_1} \le L}$ 。
>
> 因为 $k > s$ 且为整数，则其前 $s+1$ 项满足 $a_1 + a_2 + \cdots a_s + a_{s+1} \le L$ 。
>
> 这与贪心策略构造的子集和 Q 中 s 满足的 $\sum_{i=1}^{s} \mathrm{a}_s + \mathrm{a}_{s+1} > \mathrm{L}$ 矛盾。故假设不成立，得证。

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?

> 磁带的利用率为 $\sum_{\mathrm{p_i \in Q}} \mathrm{a_i/L}$; (甚至最小可为 0 ，此时任意 $\mathrm{a_i > L}$ 或者 $\sum_{\mathrm{p_i \in Q}} \mathrm{a_i} << \mathrm{L}$)

3. 试说明 1)中提到的设计策略不一定得到使 $\sum a_i / L$ 取最大值的子集合。

> 按照1的策略可以使磁带上的程序数量最多，但程序的总长度不一定是最大的，假设 $\{a_1, a_2, \cdots, a_i\}$ 为 Q 的最大子集，但是若用 a_{i+1} 代替 a_i，仍满足 $\sum_{k=1}^{i-1} a_k + a_{i+1} < L$，则 $\{a_1, a_2, \cdots, a_{i-1}, a_{i+1}\}$ 为总长度更优子集。

4. 写出 Huffman 编码的伪代码，并编程实现。

- 伪代码

```
function Huffman_Encode(characters):
    初始化一个优先队列 Q 和一个计数器 counter
    for 每个字符 c 和其频率 f 在 characters 中:
        将 (f, counter, Leaf(c, f)) 插入到 Q 中
        增加 counter 的值
    while Q 的长度 > 1:
        (weight1, _, left_node) = 弹出 Q 中的最小项
        (weight2, _, right_node) = 弹出 Q 中的下一个最小项
        创建一个新节点 N, N.left = left_node, N.right = right_node
        将 (weight1 + weight2, counter, N) 插入到 Q 中
        增加 counter 的值
    root = 弹出 Q 中的唯一项的第三个元素 (即节点)
    初始化一个空字典 code_dict
    Generate_Code(root, "", code_dict)
    return code_dict

function Generate_Code(node, current_code, code_dict):
    if node 是叶节点:
        code_dict[node.char] = current_code
    else:
        Generate_Code(node.left, current_code + "0", code_dict)
        Generate_Code(node.right, current_code + "1", code_dict)
```

- Python实现

```python
import heapq
from collections import namedtuple

# 定义 Huffman 树的节点结构
class Node(namedtuple("Node", ["left", "right"])):
    def walk(self, code, acc):
        self.left.walk(code, acc + "0")
        self.right.walk(code, acc + "1")

class Leaf(namedtuple("Leaf", ["char", "weight"])):
    def walk(self, code, acc):
        code[self.char] = acc or "0"

def huffman_encode(characters):
    # 初始化优先队列和计数器
    Q = [(weight, i, Leaf(char, weight)) for i, (char, weight) in enumerate(characters)]
    heapq.heapify(Q)

    counter = len(Q)
    while len(Q) > 1:
        weight1, _, left = heapq.heappop(Q)
        weight2, _, right = heapq.heappop(Q)
        new_node = Node(left, right)
        heapq.heappush(Q, (weight1 + weight2, counter, new_node))
        counter += 1

    root = heapq.heappop(Q)[2]
    code_dict = {}
    root.walk(code_dict, "")

    return code_dict

# 解码函数
def huffman_decode(encoded, code_dict):
```

```python
    reverse_dict = {v: k for k, v in code_dict.items()}
    decoded = []
    while encoded:
        for k in reverse_dict:
            if encoded.startswith(k):
                decoded.append(reverse_dict[k])
                encoded = encoded[len(k):]
                break
    return ''.join(decoded)

# 测试
chars = [('a', 5), ('b', 9), ('c', 12), ('d', 13), ('e', 16), ('f', 45)]
code_dict = huffman_encode(chars)
encoded = ''.join([code_dict[char] for char in 'abcdef'])
decoded = huffman_decode(encoded, code_dict)

print("编码字典:", code_dict)
print("编码后的字符串:", encoded)
print("解码后的字符串:", decoded)
```

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:

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

> 该问题可以转换成一个图论的问题， 假设每种货币为一个节点， 然后$R[i, j]$ 代表$i \rightarrow j$ 的有向路径的权重， 对$R\left[i_1, i_2\right] R\left[i_2, i_3\right] \cdots R\left[i_k, i_1\right]>1$ 求 `log`， 可以得到:
>
> $$\log \left(R\left[i_1, i_2\right]\right)+\log \left(R\left[i_2, i_3\right]\right)+\cdots+\log \left(R\left[i_k, i_1\right]\right)>0$$
>
> 也就是求得1,2...,k中一条路径使得其权重值大于0，那么我们可以利用弗洛伊德算法求得最长路径，然后判断最长路径的对应的log值是否大于0， 如果最长的都小于0，那么不存在，反之存在。

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。

> 为了找到并打印所有满足条件的序列，我们可以使用深度优先搜索（DFS）。
>
> 算法步骤如下:
>
> 1. 对于每个节点 `i` ，如果 `D[i][i] > 0`，则从该节点开始执行DFS。
> 2. 在DFS中，从当前节点 `i` 访问其所有邻居 `j`，如果 `D[i][j] + D[j][i] > 0`，则将 `j` 添加到当前路径中并继续DFS。
> 3. 如果在DFS过程中返回到起始节点并且路径权重总和大于0，则打印该路径。
>
> 计算时间分析:
>
> - 弗洛伊德算法的时间复杂度为 $O(n^3)$。
> - DFS的时间复杂度在最坏情况下为 $O(n!)$，因为它可能需要遍历所有可能的路径。
>
> 总的时间复杂度为 $O(n^3+n!)$。但在实际应用中，不太可能遍历所有的 $n!$ 路径，因为很多路径可能不满足条件。

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。

对于最小生成树问题，考虑无向图 $G=(\mathrm{V},\mathrm{E})$，我们可以定义 $\mathrm{M}=(\mathrm{S},\mathrm{L})$：

- S 是边集 E。
- $\mathrm{L}=\{\mathrm{X}\mid \mathrm{X}\subseteq \mathrm{E}$ 且 x 组成的图无环 $\}$

这个 M 显然满足拟阵的前两个条件。
根据 L 的定义，对 $\forall \mathrm{x}\in \mathrm{L}$，$\forall \mathrm{y}\subseteq \mathrm{x}$，假设 y 形成环，则 x 形成环，矛盾，所以 y 不形成环，所以 $\mathrm{y}\in \mathrm{L}$，因此M满足遗传性。

考虑 $\forall A\in L$，$B\in L$，$|A|<|B|$，我们将 A 组成的森林命名为 GA， B 组成的森林命名为 GB 。 GA 有 $|\mathrm{V}|-|\mathrm{A}|$ 个连通分量， GB有 $|\mathrm{V}|-|\mathrm{B}|$ 个连通分量。 $|\mathrm{A}|<|\mathrm{B}|$，所以 $|\mathrm{V}|-|\mathrm{B}|<|\mathrm{V}|-|\mathrm{A}|$，所以 GB 中存在的一个连通分量 T，T 中的点在 GA 中不连通。那么 T 中必然存在一条边 x 连接 GA 中不同的连通分量的边，显然 $x\notin A$ 且 $x\in B$，且 $A\cup\{x\}$ 无环，即 $A\cup\{\mathrm{x}\}\in \mathrm{L}$ 。所以 M 满足交换性。因此 M 是一个拟阵。

因为 M 是一个拟阵，故可以用贪心算法解决它的子集优化问题。

对于最小生成树问题，我们希望找到一个权重最小的边集，使得这些边连接图中的所有顶点而不形成任何环。这是一个典型的优化问题。

为了证明这个问题可以用贪心算法解决，我们需要证明它具有拟阵结构。如上所述，我们已经定义了一个拟阵 $M=(S,L)$，其中 S 是边集 E，而 L 是所有不形成环的边的子集。

接下来，我们需要定义一个赋值函数。对于最小生成树问题，一个自然的赋值函数是边的权重。我们记这个函数为 $w:E\rightarrow\mathbb{R}$，其中$w(e)$ 是边 e 的权重。

现在，我们来看两种常用的贪心算法：Prim 算法和 Kruskal 算法。

1. **Prim 算法**:
   - 从任意一个顶点开始。
   - 在每一步中，选择一条连接已选顶点和未选顶点的权重最小的边。
   - 重复这个过程，直到所有的顶点都被选中。

   Prim 算法在每一步都确保了选择的边是当前可选边中权重最小的，这是一个贪心的选择。由于我们已经证明了最小生成树问题具有拟阵结构，所以 Prim 算法可以得到最优解。

2. **Kruskal 算法**:
   - 将所有的边按权重排序。
   - 从权重最小的边开始，按顺序选择每一条边，但只选择那些不会与已选的边形成环的边。
   - 重复这个过程，直到选择的边的数量等于 (|V| - 1)。

   Kruskal 算法在每一步都确保了选择的边是当前可选边中权重最小的，而且不会形成环。这也是一个贪心的选择。同样，由于最小生成树问题具有拟阵结构，Kruskal 算法也可以得到最优解。