数据结构与算法 II 作业 (11.3)

中国人民大学　信息学院　崔冠宇　2018202147

T16-2　（最小平均完成时间调度问题）假定给定任务集合 $S = \{a_1, a_2, \cdots, a_n\}$，其中任务 a_i 在启动后需要 p_i 个时间单位完成。你有一台计算机来运行这些任务，每个时刻只能运行一个任务。令 c_i 表示任务 a_i 的**完成时间**，即任务 a_i 被执行完的时间。你的目标是最小化平均完成时间，即最小化 $(1/n)\sum_{i=1}^{n} c_i$。例如，假定有两个任务 a_1 和 a_2，$p_1 = 3$，$p_2 = 5$，如果 a_2 首先运行，然后运行 a_1，则 $c_2 = 5$，$c_1 = 8$，平均完成时间为 $(5 + 8)/2 = 6.5$。如果 a_1 先于 a_2 执行，则 $c_1 = 3$，$c_2 = 8$，平均完成时间为 $(3 + 8)/2 = 5.5$。

a. 设计算法，求平均完成时间最小的调度方案。任务的执行都是非抢占的，即一旦 a_i 开始运行，它就持续运行 p_i 个时间单位。证明你的算法能最小化平均完成时间，并分析算法的运行时间。

b. 现在假定任务并不是在任意时刻都可以开始执行，每个任务都有一个**释放时间** r_i，在此时间之后才可以开始。此外假定任务执行是可以抢占的 (preemption)，这样任务可以被挂起，稍后再重新开始。例如，一个任务 a_i 的运行时间为 $p_i = 6$，释放时间为 $r_i = 1$，它可能在时刻 1 开始运行，在时刻 4 被抢占。然后在时刻 10 恢复运行，在时刻 11 再次被抢占，最后在时刻 13 恢复运行，在时刻 15 运行完毕。任务 a_i 共运行了 6 个时间单位，但运行时间被分割成三部分。在此情况下，a_i 的完成时间为 15。设计算法，对此问题求解平均运行时间最小的调度方案。证明你的算法确实能最小化完成时间，分析算法的运行时间。

解: 由于任务个数 n 在某一情境下是固定的，不随各任务的排列顺序而改变，所以最小化平均完成时间只需要最小化总完成时间即可。

a. 根据日常经验，应该按照 p_i 递增调度任务，可以使总完成时间最短。下面给出证明：

证明: 依题意，我们要找寻 $< 1, 2, \cdots, n >$ 的一个排列 $< i_1, i_2, \cdots, i_n >$，最小化总完成时间

$$\sum_{i=1}^{n} c_i = \sum_{l=1}^{n} \sum_{m=1}^{l} p_{i_m}$$

用反证法。任取一个最优序列（即总完成时间最小）$< p_{i_1}, p_{i_2}, \cdots, p_{i_s}, \cdots, p_{i_t}, \cdots, p_{i_n} >$，若它不是递增排列，则可取其中任意一对逆序，设为 (p_{i_s}, p_{i_t})（其中 $s < t$ 且 $p_{i_s} > p_{i_t}$）。交换二者的调度次序以消除逆序，此时调度顺序为 $< p_{i_1}, p_{i_2}, \cdots, p_{i_t}, \cdots, p_{i_s}, \cdots, p_{i_n} >$。为了分析交换二者引起的总调度时间的变化，我们可以分两步变换以完成交换：

1. 第一步：将第 s 个任务的执行时间减少到 p_{i_t}，于是第 s 个被执行的任务之后（含）的所有任务的完成时间都减少了 $(p_{i_s} - p_{i_t})$，一共有 $n - s + 1$ 个任务受到影响，所以总时间减少了 $(p_{i_s} - p_{i_t})(n - s + 1)$；
2. 第二步：将第 t 个任务的执行时间增加到 (p_{i_s})，于是第 t 个被执行的任务之后（含）的所有任务的完成时间都增加了 $(p_{i_s} - p_{i_t})$，一共有 $n - t + 1$ 个任务受到影响，所以总时间增加了 $(p_{i_s} - p_{i_t})(n - t + 1)$。

综合上述两步，可得交换二者引起的总时间变化为

$$-(p_{i_s} - p_{i_t})(n - s + 1) + (p_{i_s} - p_{i_t})(n - t + 1) = (p_{i_s} - p_{i_t})(s - t) < 0$$

这与原来的调度顺序是最优的矛盾。所以最优序列必然没有逆序对，也就是最优序列一定是递增序列。

因此，设计非抢占的最小平均完成时间调度算法，只需要按各任务的执行时间 p_i 从小到大排序，就可以得到最优序列。在给定各任务所需时间 p_i 的条件下，可以写出一个“静态算法”，提前给操作系统决定好各任务的执行顺序：

```
// 非抢占最优任务调度
// 参数：
//      a：任务序列
//      p：各任务执行时间
// 返回值：
//      a_optimal：最优任务序列
NON-PREEMPTIVE-OPTIMAL-DISPATCH(a, p):
    // 将 a[i] 和 p[i] 组成的数对构成新数组
    arr[] = {(a[i], p[i])}
    // 根据执行时间升序排序任务数组 a
    sort(arr) by p[i] ascending
    // 把排好序的任务数组的任务构成新数组
    a_optimal = {arr[i].a}
    return a_optimal
```

上述算法中，两次构成新数组的时间复杂度都是 $\Theta(n)$ 的，显然这个算法的时间复杂度就是排序的时间复杂度 $\Theta(n \log n)$（假定选择堆排序）。排序的总时间平摊至每个任务上是 $\Theta(\log n)$ 的，也即操作系统每次选择任务执行时都需要花费 $\Theta(\log n)$ 的代价。

或者可以从操作系统每次选择任务执行的角度出发，认为各任务在 $t = 0$ 时被同时释放，然后将各任务和它们的用时建立一个优先队列（最小堆实现），每次取出堆顶的任务执行，从而编写一个“动态算法”：

```
// 预处理
// 参数：
//      a：任务序列
//      p：各任务执行时间
// 返回值：
//      q：任务调度优先队列
Prepare(a, p):
    // 将 a[i] 和 p[i] 组成的数对构成新数组
    q[] = {(a[i], p[i])}
    // 按所需执行时间建立最小堆
    BUILD-MINIMUM-HEAP(q) by p[i]
    return q
// 调度
// 参数：
//      q：任务调度优先队列
```

```
// 返回值:
//       task: 选中的调度任务
DYNAMIC-NON-PREEMPTIVE-OPTIMAL-DISPATCH(q):
    // 弹出任务
    task = EXTRACT-MIN(q)
    return task

// 操作系统调度时执行下列代码
q = Prepare(a, p)
// 当还有任务没执行
while not q.empty():
    // 选中任务
    task = DYNAMIC-NON-PREEMPTIVE-OPTIMAL-DISPATCH(q)
    // 给任务分配执行时间（不可抢占）
    EXEC-TASK(task)
```

由于 EXTRACT-MIN 的时间复杂度是 $\Theta(\log n)$ 的，所以 DYNAMIC-NON-PREEMPTIVE-OPTIMAL-DISPATCH 的时间复杂度是 $\Theta(\log n)$。这也验证了选择任务平均需要花费对数级别时间的代价。

综上，**在非抢占条件下，决定一个调度任务的平均时间是 $\Theta(\log n)$ 的，而总时间是 $\Theta(n \log n)$ 的。**

b. 根据操作系统的知识，在每一时刻执行当前可执行的剩余执行时间最短的任务能最小化总运行时间。下面给出证明：

假定在某一时刻 t，某些任务已经释放。假设此时任务队列里有 $a_{i_1}, a_{i_2}, \cdots, a_{i_k}$，类似上面的证法，在新任务到来前，假如不先选择剩余时间最短的任务执行，通过将最短剩余时间的任务交换至前面运行可以使总运行时间减少，由此引出矛盾。所以最优执行序列一定是每次选择当前剩余执行时间最短的任务运行形成的序列。

同样可以从操作系统的角度出发，动态维护一个优先队列，当任务到来时将其剩余用时插入优先队列，然后取队列头部的任务执行：

```
// 抢占最优任务调度
// 参数:
//       a, p: 意义同上
//       q: 任务调度优先队列
// 返回值:
//       被选中调度的任务
DYNAMIC-PREEMPTIVE-OPTIMAL-DISPATCH(a, p, q):
    // 各任务按照释放时间到达 q
    // CHECK-NEW-TASK 会检查 q 中是否有新任务到来

    // 没有新任务，就继续执行优先队列中剩余时间最短的
```

```
    if CHECK-NEW-TASK() == false:
        task = EXTRACT-MIN(q)
        return task
    // 新任务到达时，对比新任务剩余时间和现在执行的任务的剩余时间
    else
        new_task = GET-NEW-TASK()
        now_task = GET-NOW-TASK()
        now_task.p -= its executed time slices
        // 选中剩余时间小的，剩下的放回调度队列
        if now_task.p < new_task.p:
            HEAP-INSERT(q, new_task)
            return now_task
        else:
            HEAP-INSERT(q, now_task)
            return new_task

// 操作系统执行
// 建立一个空调度队列
q = PriorityQueue() by p[i]
// 仍有任务没被执行
while has tasks not executed:
    // 只有当某任务结束或新任务到达时才会调度
    if some task finished right now or new task released:
        task = DYNAMIC-PREEMPTIVE-OPTIMAL-DISPATCH(a, p, q)
```

假定检查是否有新任务 CHECK-NEW-TASK、获取新任务 GET-NEW-TASK 和获取当前任务 GET-NOW-TASK 的时间复杂度是 $\Theta(1)$ 的，由于 EXTRACT-MIN 和的时间复杂度是 $\Theta(\log n)$ 的，所以 DYNAMIC-NON-PREEMPTIVE-OPTIMAL-DISPATCH 的时间复杂度是 $\Theta(\log n)$ 的。

综上，**在可抢占条件下，决定一个调度任务的时间是 $\Theta(\log n)$ 的，而总时间也是 $\Theta(n \log n)$ 的。**

T16.3-3 如下所示，8 个字符对应的出现频率是斐波那契数列的前 8 个数，此频率集合的赫夫曼编码是怎样的?

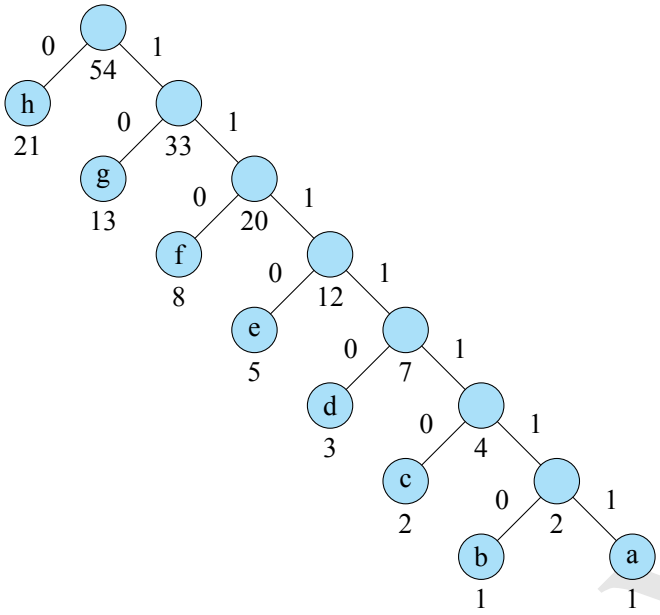
a: 1　　b: 1　　c: 2　　d: 3　　e: 5　　f: 8　　g: 13　　h: 21

你能否推广你的结论，求频率集为前 n 个斐波那契数的最优前缀码?

解:

(1) 运行赫夫曼编码的算法，得到一棵如图所示的树:

所以它们的赫夫曼编码为:



a: 1111111　　b: 1111110　　c: 111110　　d: 111103　　e: 1110　　f: 110　　g: 10　　h: 0

(2) 推广得到频率集为前 $n(n \geq 2)$ 个斐波那契数的最优前缀码为：

$$\texttt{HUFFMAN-CODE}(F(i)) = \underbrace{11\cdots1}_{n-i\text{个}1}0$$

证明：要证明斐波那契数列构成的赫夫曼树是如上图偏斜的，只需证明第 k 次选择权值最小的两节点合并时，都会选择 $\sum_{i=0}^{k} F_i$（设 $F_0 = 0$，即上图右子树）与 F_{k+1}（即上图左子树）即可。首先证明以下性质：

$$F_{n+2} = \sum_{i=0}^{n} F_i + 1$$

用归纳法：

① $n = 0$ 时，$1 = F_2 = F_0 + 1$，成立；$n = 1$ 时，$2 = F_3 = F_0 + F_1 + 1$，也成立；

② 假设对 $n \leq k$ 的自然数该性质都成立，则 $n = k + 1$ 时，

$$\begin{aligned} F_{k+3} &= F_{k+2} + F_{k+1} \\ &= \sum_{i=0}^{k} F_i + 1 + F_{k+1} \\ &= \sum_{i=0}^{k+1} F_i + 1 \end{aligned}$$

所以命题对 $n = k + 1$ 也成立。

由数学归纳法，命题对任意自然数都成立。所以 $\sum_{i=0}^{n} F_i < F_{n+2}$，并且根据斐波那契数列的定义，有 $F_{n+1} \leq F_{n+2}$。

当第一次合并节点时，有 $F_1 = 1, F_2 = 1, \cdots, F_n$ 这些节点，会选择两个权值为 1 的节点合并；可以归纳证明：

第 k 次合并节点时，有 $\sum_{i=0}^{k} F_i, F_{k+1}, F_{k+2}, \cdots, F_n$ 这些节点，由于 $\sum_{i=0}^{k} F_i < F_{k+2}$ 且 $F_{k+1} \leq F_{k+2}$，所以会选择 $\sum_{i=0}^{k} F_i$ 以及 F_{k+1} 合并。于是上图偏斜形式的树是赫夫曼树之一。所以

$$\texttt{HUFFMAN-CODE}(F(i)) = \underbrace{11\cdots1}_{n-i\text{个}1}0$$

是最优前缀码之一。