数据结构与算法第四次作业

Shuxin Chen, 2015013229

2017 年 3 月 18 日

1. 最小平均完成时间调度问题

   题目中要求最小化的是平均完成时间，即 $\frac{1}{n}\sum_{i=1}^{n} c_i$，因为 n 是给定的，因此最小化平均完成时间等价于最小化总完成时间，即 $\sum_{i=1}^{n} c_i$。

   (a) 设 $\{k_n\} = k_1, k_2, \ldots k_n$ 为 $1 \sim n$ 的一个排列，依次执行任务 $a_{k_1}, a_{k_2}, \ldots, a_{k_n}$，此时我们可以算出总完成时间为

$$\begin{aligned} T &= \sum_{i=1}^{n} c_{k_i} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{i} p_{k_j} \\ &= \sum_{i=1}^{n} i * p_{k_{n-i+1}} \end{aligned}$$

   根据排序不等式（切比雪夫不等式），当 $\{p_{k_n}\}$ 单调不降时，上述 T 取到最小值。(假设 $\{p_{k_n}\}$ 非单调不降，则 $\exists 1 \le i, j \le n$，且 $i \ne j$，有 $p_{k_i} > p_{k_j}$，设此时的总完成时间为 T'，通过作差发现 $T' - T > 0$，故 $\{p_{k_n}\}$ 单调不降时总完成时间最短)

   我们得到了结论，当 $\{p_{k_n}\}$ 单调不降时，总完成时间最短，我们设计如下算法：

   - 将 $\{a_n\}$ 根据 p 的值从小到大排序，得到 $\{a_{k_n}\}$，时间复杂度为 $\Theta(n \log n)$；
   - 直接通过 T 的定义式计算总完成时间，时间复杂度为 $\Theta(n)$。

   因此，总的时间复杂度为 $\Theta(n \log n)$。

   (b) 因为任务执行时可抢占的，因此我们定义 v_i 为第 i 个任务的**剩余时间**。最初所有的 $v_i = p_i$，任务 i 每执行 1 个单位时间，v_i 减少 1，当任务 i 被任务 j 抢占之后，在接下来的 1 个单位时间，v_i 不变，v_j 减少 1。

我们直接说明算法，并在下面给出算法能够得到最优的总完成时间的证明。

设集合 S' 为所有当前能执行的任务集合（前面已经给出定义），并把任务按照释放时间 r_i 从小到大排序。然后我们依次模拟每一个单位时间，假设当前模拟第 t 个单位时间，首先把所有 $r_i = t$ 的任务加入 S'，然后取出 S' 中 v_i 最小的那个任务，把 v_i 的值减 1，如果 $v_i = 0$ 就删去这个任务。集合 S' 用到了 INSERT，EXTRACT-MIN，DECREASE-KEY 和 DELETE 操作，我们可以用小根堆（优先队列）进行维护。

我们使用反证法来证明上述算法能够得到最优的总完成时间。设上述算法所表示的调度为 DI，再假设存在某一种调度 DI'，它最早在第 t_0 个单位时间时，就没有选择当前 v_i 最小的任务 a_i 执行，而是执行了任务 a_j，且有 $v_j > v_i$。我们设从 t_0 时刻开始直到结束，任务 a_i 在 $t_{i_1}, t_{i_2}, \ldots, t_{i_x}$ 时刻执行，任务 a_j 在 $t_{j_1}, t_{j_2}, \ldots, t_{j_y}$ 时刻执行，且有 $v_i = x < y = v_j$ 和 $t_0 = t_{j_1} < t_{i_1}$。根据 T 的定义式，我们有

$$\begin{aligned} T &= \sum_{k=1}^{n} c_i \\ &= \sum_{\substack{1 \le k \le n \\ k \ne i, k \ne j}} c_i + t_{i_x} + t_{j_y} \end{aligned}$$

我们将 a_i 的 i_x 个时刻和 a_j 的 i_y 个时刻合并在一起，并按照时刻从小到大排序，得到一个新的时刻序列 $t_{n_1}, t_{n_2}, \ldots, t_{n_{x+y}}$。然后我们让 a_i 在时刻序列中的前 x 个时刻执行，a_j 在时刻序列中的前 y 个时刻执行，此时，只有 a_i 和 a_j 两个任务的完成时间发生了变化，其余任务的完成时间均不变，设总完成时间为 T'，我们有

$$\begin{aligned} T' &= \sum_{k=1}^{n} c_i \\ &= \sum_{\substack{1 \le k \le n \\ k \ne i, k \ne j}} c_i + t_{n_x} + t_{n_{x+y}} \end{aligned}$$

显然，$t_{n_{x+y}} = \max\{t_{i_x}, t_{j_y}\}$。而 t_{n_x} 为新序列中第 x 小的项，由于 $t_{j_1} < t_{i_1} < t_{i_2} < \cdots < t_{i_x}$，所以 t_{i_x} 对应到新序列中至少为第 $x+1$ 小的项，因此有 $t_{n_x} < t_{i_x}$。又因为 $t_{j_1} < t_{j_2} < \cdots < t_{j_y}$，$t_{j_y}$ 对应到新序列中至少为第 y 小的项，因此有 $t_{n_x} < t_{j_y}$。于是，我们有 $t_{n_x} < \min\{t_{i_x}, t_{j_y}\}$。

我们通过作差来判定 T 和 T' 的大小:

$$
\begin{aligned}
T - T' &= (\sum_{\substack{1\le k\le n\\ k\ne i, k\ne j}} c_i + t_{i_x} + t_{j_y}) - (\sum_{\substack{1\le k\le n\\ k\ne i, k\ne j}} c_i + t_{n_x} + t_{n_{x+y}}) \\
&= (t_{i_x} + t_{j_y}) - (t_{n_x} + t_{n_{x+y}}) \\
&> (t_{i_x} + t_{j_y}) - (\min\{t_{i_x}, t_{j_y}\} + \max\{t_{i_x}, t_{j_y}\}) \\
&= 0
\end{aligned}
$$

那么，调度 DI' 并没有我们新构造出来的一种调度优。当把所有 DI' 中没有选择最小的时刻修正之后，就变成了 DI，因此我们给出的算法能够得到最优的总完成时间。

回到之前提到的算法，这个算法的时间复杂度分为两部分。第一部分是排序，为 $\Theta(n\log n)$；第二部分是堆的操作，单次 $\Theta(\log n)$，由于我们是模拟了每一个单位时间，因此需要估算最后一个任务的完成时间，大约为 $O(\max\{r_i\}+\sum p_i)$，这个值大于 n。因此总的时间复杂度为 $O((\max\{r_i\}+\sum p_i)\log n)$。

这个时间复杂度是很恐怖的，如何进行优化呢？我们发现，EXTRACT-MIN 操作只会减少堆顶元素的值，而这样是不会改变堆的结构的，这个操作的实际时间复杂度为 $\Theta(1)$。因此，我们可以等到有新的元素加入堆中时，再一次性的减少堆顶元素的 v_i 值。那么新的算法如下所示。

首先将 r_i 排好序，假设当前待加入的任务为 a_j，堆顶的任务为 a_k，当前的时间为 t。若 $v_k \le a_j - t$，即堆顶任务执行完之后新任务也不会加入，那么将 t 加上 v_k，删去堆顶元素。若 $v_k > a_j - t$，即堆顶任务执行完之前有新任务加入，那么将 v_k 的值减去 $a_j - t$，t 加上 $a_j - t$，将 a_j 加入堆中。这个算法的时间复杂度不能对 t 进行分析，因为它是跳跃的。我们发现，每一个任务被加入堆和从堆中删除各一次，DECREASE-KEY 操作的次数依赖于不等式的比较，由于每一次比较会导致一个任务的加入或者一个任务的删除，因此操作的次数为 $\Theta(n)$，同时这也是修改 t 的总次数。这些加起来就是总的时间复杂度，即为 $\Theta(n\log n)$。

2. 离线缓存

(a) 首先我们将所有的 n 个元素映射到 $1\sim n$ 的正整数，使用某种哈希算法，然后存放到数组 r 中，时间复杂度为 $\Theta(n)$。接下来的伪代码如下所示。

```
Offline-Cache(r, n, k)
let pos[1..n], succ[1..n] be new tables
for i = 1 to n
    pos[i] = 0
    succ[i] = ∞
for i = 1 to n
    if pos[r[i]] ≠ 0
        succ[pos[r[i]]] = i
    pos[r[i]] = i
ans = 0
Cache = a cache
for i = 1 to n
    if Cache.CONTAINS(r[i])
        Cache.UPDATE(r[i], succ[i])
    else
        Cache.INSERT(r[i], succ[i])
        ans = ans + 1
return ans
```

$Cache$ 是一个复杂的数据结构，可以由一个数组 + 优先队列/线段树组成。它包含三个函数，CONTAINS，UPDATE 和 INSERT。CONTAINS(x) 是对缓存中是否有元素 x 的一次询问，通过在数组中打标记可以做到时间复杂度 $\Theta(1)$。UPDATE(x, y) 是将元素 x 的下一个位置修改成 y，时间复杂度为 $\Theta(\log k)$。INSERT(x, y) 是将一个新的元素 x 以及它的下一个位置 y 放入缓存中，并弹出下一个位置最远的那个元素，时间复杂度为 $\Theta(\log k)$。因此伪代码的总时间复杂度为 $\Theta(n \log k)$。

(b) 设 $f[i]$ 为前 i 次请求中缓存未命中的最少次数，S_i 为 $f[i]$ 对应的缓存选择方案的集合，则有 $f[i-1] \le f[i] \le f[i-1]+1$。

我们使用反证法，假设离线缓存问题不具有最优子结构性质，那么 $\exists s'_{i-1} \notin S_{i-1}$，它对应的 $f'[i-1] > f[i-1]$，但 $f'[i] < f[i]$。但 $f'[i] \ge f'[i-1] > f[i-1] \ge f[i]-1$，即 $f'[i] \ge f[i]$，矛盾！因此离线缓存问题具有最优子结构性质。

(c) 我们假设用将来最远的贪心策略得到的操作序列为 S_{FF}，S 为某操作序列，它的前 i 个元素和 S_{FF} 相同而第 $i+1$ 个元素不同。我们可以构造一个 S'，使得 S' 的前 $i+1$ 个元素和 S_{FF} 相同，且 S' 的缓存未命中次数不多于 S，即 S' 不劣于

S，这样就能说明 S_{FF} 是最优的。

假设在第 $i+1$ 步时，S_{FF} 选择扔掉元素 e 而 S 选择扔掉元素 f，且 e 下一次出现的位置远于 f，此时 S_{FF} 和 S 的缓存情况从相同变成不同。我们让 S' 也扔掉元素 e，并且从第 $i+2$ 步开始，它的选择均与 S 相同，直到发生下一次缓存未命中，且未命中的元素 g：

i. 若 $g \neq f$，在 S 直接扔掉 g 或是扔掉非 e 的元素放入 g 的情况下，S' 与 S 操作仍然保持一致；若 S 选择扔掉 e 放入 g，则 S' 可以扔掉 f 放入 g，此时它们的缓存情况相同，且目前 S' 和 S 的缓存未命中次数一样多，接下来让 S' 和 S 的策略保持一致，这样 S' 不劣于 S。

ii. 若 $g = f$，在 S 扔掉 e 放入 f 的情况下，S' 直接扔掉 g，这样它们的缓存情况相同，且 S' 比 S 的缓存未命中次数少一次，接下来让 S' 和 S 的策略保持一致，这样 S' 优于 S；若 S 扔掉 $h \neq e$ 放入 f，且 h 出现的位置远于 e（若近于 e，则 S' 不进行任何操作），则 S' 仍然不进行任何操作。等到下一次 S 的缓存未命中，若发生在 e 到来之前，且 S 扔掉 e 放入新元素，则 S' 可以扔掉 g 放入新元素，这样 S' 不优于 S；若发生在 e 到来之后，则当 e 到来时，S' 扔掉 g 放入 e，这样 S' 不劣于 S。

我们从 $1 \sim n$ 依次枚举 i，那么就逐步生成了一个最优的操作序列 S'，且 $S' = S_{FF}$。因此，将来最远策略可以保证最小缓存未命中次数。

3. 动态二分查找

(a) 因为每个数组都是有序的，但是任意两个数组之间没有任何关系，因此 SEARCH 操作需要对所有的数组进行二分查找。

数组 A_i 的长度为 2^i，最坏的情况为查找失败，需要进行 $i+1$ 次查找。每一个 n 通过二进制表示成 $k = \lceil \log(n+1) \rceil$ 位，且包含小于等于 k 个满的 A 数组。最坏的情况为 $n = 2^k - 1$，此时所有的 k 个数组均满，要在每一个数组上进行二分查找，因此最坏的时间复杂度为

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{k-1} (i+1) \\
&= \frac{1}{2} k(k+1) \\
&= \frac{1}{2} \log n(\log n + 1) \\
&= \Theta(\log^2 n)
\end{aligned}
$$

因此最坏情况运行时间为 $\Theta(\log^2 n)$。

(b) 设 A_{i_0} 为下标最小的空数组, 即 $\forall i \geq 0$, 若 A_i 为空, 则有 $i \geq i_0$。当执行 INSERT 操作时，所有下标小于 i_0 的数组（如果有的话）中的元素之和恰好比 A_{i_0} 的长度少 1，将这些元素与待加入的元素一起用单次归并排序的方式（因为这些数组原本都是有序的）放入 A_{i_0} 中。由于单次归并排序的时间复杂度为 $\Theta(L)$，其中 L 为待排序的元素个数，因此每一次 INSERT 操作的时间复杂度为 $\Theta(2^{i_0})$。最坏情况下，$i_0 = k$，此时 $2^{i_0} = \Theta(n)$，故最坏情况运行时间为 $\Theta(n)$。

摊还时间可以用聚合分析的方式计算。对于数组 A_i，它参与归并排序的次数为 $\lfloor \frac{n}{2^i} \rfloor$，每次贡献的操作次数为 2^i，因此前 n 次 INSERT 操作的时间复杂度为

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor * 2^i \\
&= \sum_{i=0}^{k-1} \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$

总时间为 $\Theta(n \log n)$，因此摊还时间为 $\Theta(\log n)$。

(c) 我们设计如下的 DELETE 算法：

- 对待删除的元素执行 SEARCH 操作, 时间复杂度为 $\Theta(\log^2 n)$。若没有找到, 直接结束该算法。
- 设带删除的元素在 A_i 中，下标最小的一个满数组为 A_j，若 $i \neq j$，则将 A_j 中任意一个元素放入 A_i 中，执行插入排序，时间复杂度为 $\Theta(2^i)$。再将 A_i 中剩余的所有元素按顺序放入 $A_0, A_1, \ldots, A_{i-1}$ 中, 时间复杂度为 $\Theta(2^j)$。因为 $i \leq j$，因此总时间复杂度为 $\Theta(2^i)$。

在最坏情况下，$i = k - 1$，因此 DELETE 算法的最坏复杂度为 $\Theta(n)$。