Week 7

软73 沈冠霖 2017013569

April 16, 2019

1 T1

1.1 1

算法 把每个任务的执行时间由小到大排序，然后对排好序的任务序列从前往后执行

证明 设第i个执行完的任务是第t_i个任务，那么其完成时间是$c_i = \sum_{j=1}^{i} p_{t_i}$，总共的完成时间是$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n}(n+1-i)p_{t_i}$。
而假设使得总完成时间最短的任务执行序列中存在i,j,使得$i < j, p_{t_i} > p_{t_j}$，则将t_i, t_j两个任务交换顺序，总完成时间会减少$(j-i)(p_{t_i} - p_{t_j})$。因此序列p_{t_i}必须单调递增。 因为n是常数，所以总共完成时间最短，则平均完成时间最短。因此算法正确。

复杂度 先对任务的执行时间排序，复杂度O(nlgn)，之后依次求出每个任务的完成时间，并且求和取平均，复杂度$\theta(n)$,因此总共时间复杂度O(nlgn)。

1.2 2

算法 维护一个优先级队列q,里面存储当前时间t时，可以开始进行的任务的剩余完成时间，剩余完成时间越短，优先级越高。
每次操作把时间t加一，然后在新的时间t可以开工的任务加入队列，然后执行队列头的任务。如果执行完毕，这个任务弹出队列，并且记录其结束时间。直到队列为空停止，计算平均完成时间。

证明 1.证明是最优子结构：设每个状态t$[i, j_1, j_2, ...j_n]$记录时间i后每个任务的剩余执行时间，每个时间做出选择，选择哪个可执行任务在这个时间执行。则每个选择的最优解必定是这次选择后完成的总时间+这个选择下一个状态后完成的最优总时间。
2.证明贪心选择正确：设时间t时，可以执行的剩余执行时间最短的任务是j，而最优选择在这一时间却选择了执行时间$r_i > r_j$的任务i。设在最优选择中执行任务i,j的时间分别构成数列t_i, t_j，完成时间分别为e_i, e_j。
先证明最优情况必定有$e_j < e_i$。如果最优情况是$e_j > e_i$，则在不改变其他任

务执行状况前提下，无论怎么分配i，j的任务执行时间，任务i，j全都执行完的时间必定是e_j。而如果选取t_i, t_j合并成的数列的前p_j个时间执行j，剩下时间执行i，这样i，j的任务结束总时间一定是最小的，因此一定有$e_j < e_i$
再证明如果$e_j < e_i$，则如果这个时间选择第j个任务，这个选择一定包含在某个最优解里。设某个最优解在这个时间执行的是第i个任务，那么一定有$e_j < e_i$，那么如果把这个时间换成执行第j个任务，在原先的e_j时间执行第i个任务，那么e_i不变，e_j必定变小，其他任务完成时间不变，总完成时间变小。说明某个最优解在这个时间一定执行的是第j个任务。

复杂度 **假**设在时间i有m_i个任务被推入队列，最多一个任务被推出队列，那么每个时间花费$O((m_i+1)lgn)$，因为$\sum_{t=1}^{max(t)} m_t = n$,则总共时间复杂度为O((n+T)lgn)，T为最后一个任务完成的时间。因为每个任务时长至少为1，而且因为开始时间的限制，可能有的时间并未执行任务，所以n=O(T),总共时间复杂度为O(Tlgn)。

2 T2

证明 首先，缓存不是命中就是未命中，因此最大命中次数等价于最小未命中次数。
1.证明是最优子结构：设每个状态$f[i,j]$,代表读取i个缓存数据后，而且缓存中的数据为大小小于等于k的集合j时的最小缓存不命中次数。每次选择如下：i增加1，如果新的缓存数据命中则无序集合j不变，否则j先加入未命中的数据，再移除至多一个数据使得集合j大小小于等于k。因为这次做的选择并不影响从新的状态到结束的最优解，因此假设$f[i,j]$为从输入完毕第i个数据后开始，初始缓存集合为j，直到输入完毕结束的最小缓存不命中次数，第一个选择使得j变成j'，那么必定有$f[i,j]|choose_1 = g(choose_1) + f[i+1, j']$，其中$g(choose_1)$代表第一个选择中未命中的数据个数，$f[i,j]|choose_1$代表做了这一个选择前提下的最优值。因此有最优子结构。
2.证明贪心可以进行：假设要求任意的$f[i_m, j_m]$，假设第$i_m + 1$次输入未命中而且缓存满了，把未命中的数据加入缓存。
假设使得$f[i_m, j_m]$取得最优解的第一个选择是移除数据i，而实际上可以移除的，最远出现的数据是j。假设i,j下次出现分别在c_i, c_j个元素后,$c_i < c_j$，分四种情况讨论。在每种情况中，我提出的新方法都是第一次操作移除j保留i，最优解都是移除i保留j，而两种方法中任何和元素i，j无关的操作都完全一致。图像解释参见图片1-4
情况1，如果最优解直到输入结束都没让i，j进出缓冲区，假设到结束一共a个i，b个j，到第一个j之前一共有d个i,其余未命中缓冲区c次,$0 \le a, b, c; 1 \le d$。则可以不改变其余的进出缓冲区操作，一开始移除j保留i，到了第一个j之后移除i保留j，其余情况不对i，j做任何进出缓冲区操作，那么原先最优解会未命中$a+1$次，而新的情况会命中$a - d + 2 \le a + 1$次。
而如果遇到特殊情况，再也不会遇到j了，那么假设之后会遇到a个i，其余未命中c次，$0 \le a, c$，那么原最优解未命中$a + c$次，新的情况则为不进行任何关于i，j的缓冲区操作，未命中$c \le a + c$次。
情况2，如果最优解第一次修改缓冲区的i，j操作是让i进入缓冲区，j出缓冲

区，则这个输入的字符必定是i，假设在遇到第一个j前就交换了i与j，那么交换之前有d个i，其余有c次未中,$0 \leq d, c$，那么最优解需要交接$d+c+1$次。而新的情况是不进行i或j的进出缓冲区，这样能命中所有的i，只需要交换$c \leq d+c+1$次，而他们在最优解交换i与j后的状态完全相同。
假设在遇到第一个j后交换了i与j，假设第一个j前有d个i，第一次交换i与j的前面有a个i，b个j，其余交换c次，$1 \leq d \leq a, 1 \leq bfi0 \leq c$，那么最优解需要交换$c+a+1$次。而新的情况是先在第一个j处交换i和j，然后再在最优解第一次交换的位置交换i与j。这样只需要交换$a-d+2+c \leq c+a+1$次，而且交换完毕后与最优解得到的状态完全相同。
情况3，如果最优解第一次修改缓冲区的i，j操作是将j换出缓冲区，换上不是i也不是j的元素t。先假设这次修改在遇到第一个j后进行，修改前遇到了a个i，b个j，第一个j前有d个i，其余修改c次，$1 \leq d \leq a, 1 \leq b, 0 \leq c$，那么最优解未命中了a+1+c次。而新的操作是在第一个j位置换上j换下i，之后在最优解的第一次交换位置换下j换上t。这样只未命中$a-d+2+c \leq a+1+c$次，而且交换完毕后和最优解得到的状态完全相同。
而假设这次修改在遇到第一个j前就进行，修改前遇到了a个i，其余修改了c次，$0 \leq a, c$，那么最优解一共未命中a+1+c次。而新的做法是不动，只在原来最优解第一次修改的时候换下i换上t，那么一共只修改了$1+c \leq a+1+c$次，而且状态和最优解得到的完全相同。
情况4，假设最优解第一次修改是把i加入缓冲区，换下t。假设这次操作在遇到第一个j后进行，那么假设进行这次操作前遇到了a个i，b个j，第一个j前遇到了d个i，其余未命中操作为c，$1 \leq d \leq a, 1 \leq b, 0 \leq c$，那么一共未命中$a+1+c$次。而更新的操作是第一次遇到j的时候先换上j换下i，在和最优解第一次操作同时换上i换下t。这样一共未命中$a-d+2+c \leq a+1+c$次，而且结束的状态和最优解完全相同。
假设最优解第一次操作在遇到第一个j前进行，假设进行前遇到了a个i，进行后到第一个j遇到了b个i，最优解其余未命中c次，$0 \leq a, b, c$，那么等到遇到第一个j时一共未命中a+1+c次。而更新操作是先什么都不做，遇到第一个j时换上j换下t，这样只未命中$1+c'$次，而且因为缓冲区里有t，因此遇到t都能命中，$c' \leq c$，因此$1+c' \leq a+c+1$，而且最后状态和最优解完全相同。
综上，无论是哪种情形，第一次选择移除j总能得到某个最优解，因此贪心正确。

代码和复杂度　为了方便，假设一开始缓存就是满的
情况1：遇到的元素种类较多，而元素个数没那么多

```
MANAGE_BUFFER(L,n,k,B)
for i=1 to n
    success = 0
    for j=1 to k
        if L[i]=B[j]
            success = 1
            break
    if success = 0
        for ii = 1 to k
            next[ii] = n-i+1
```

```
                for jj = 1 to n-i
                    if B[ii] = L[i+jj]
                        next[ii]=jj
                        break
//找到使得next最大的数组下标
            maximum = argmax(next)

//输出处理第几个请求的时候逐出了哪个元素
            print(i,B[maximum])
            B[maximum]=L[i]
    return
```

情况2：遇到的元素种类有限,仅有m种，而元素数量很多

```
    MANAGE_BUFFER(L,n,k,B,m)
    for i=1 to n
        queue[L[i]].push(i)
    for i=1 to m
        queue[i].push(n+1)
    for i=1 to k
        whether_inside[B[i]]=1
    for i=1 to n
        if whether_inside[L[i]]=1
            continue
        else
            max_next = 0
            for j=1 to k
                if queue[B[j]].first ¿ max_next
                    max_next = queue[B[j]].pop()
                    max_place = j
            whether_inside[B[max_place]]=0
            whether_inside[L[i]]=1
            B[max_place] = L[i]
            print(i,B[max_place])
    return
```

复杂度 **第一种情况**：最坏情况：每次缓存都不命中，则每输入一个元素，都需要用k次循环判断是否命中，再用k(n-i)次找每个元素下次在哪里遇到，再用k次找到哪个元素该出去，一共需要$\sum_{i=1}^{n} k(n+2-i) = O(kn^2)$的时间复杂度,并且需要额外空间O(k)来存储每个缓冲区元素的下一个位置。
第二种情况：最坏情况同上，预处理需要O(m+n+k)的复杂度，而循环的时候每次只需要O(nk)的复杂度来找移除哪个元素就可以了，一共需要O(nk+m)的时间复杂度。需要m个总长度n+m的队列来存储每个元素出现的位置，以及m的表来存储每个元素在不在缓冲区。O(m+n)的空间复杂度。

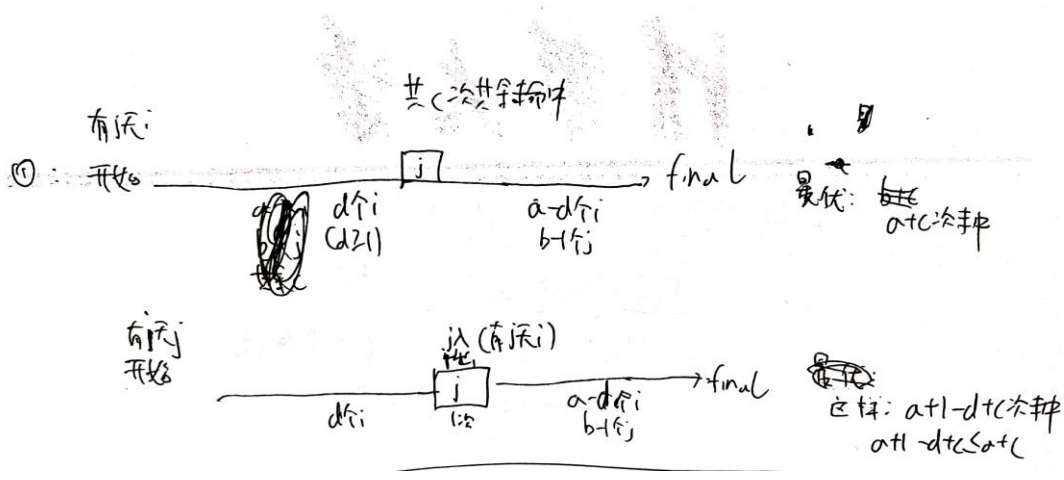

图 1： 第二题贪心正确性证明的情况 1

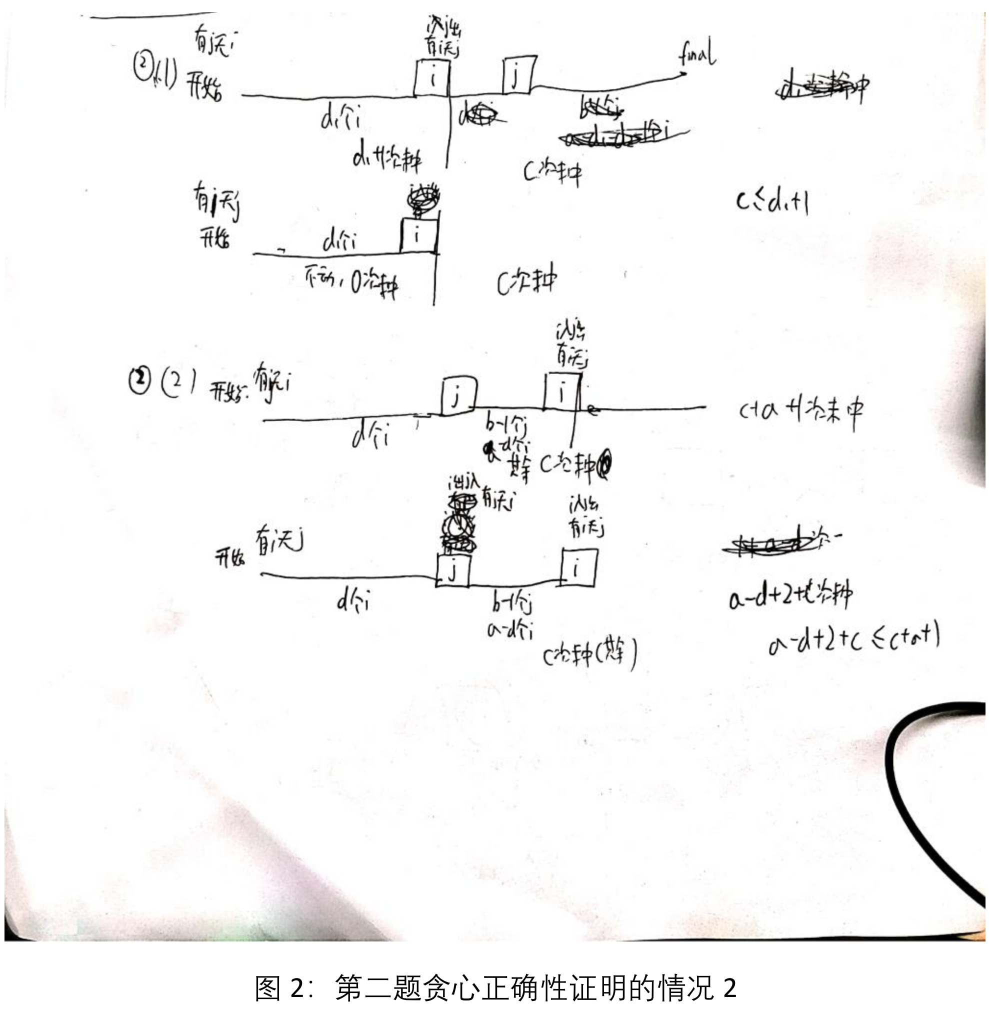

图 2： 第二题贪心正确性证明的情况 2

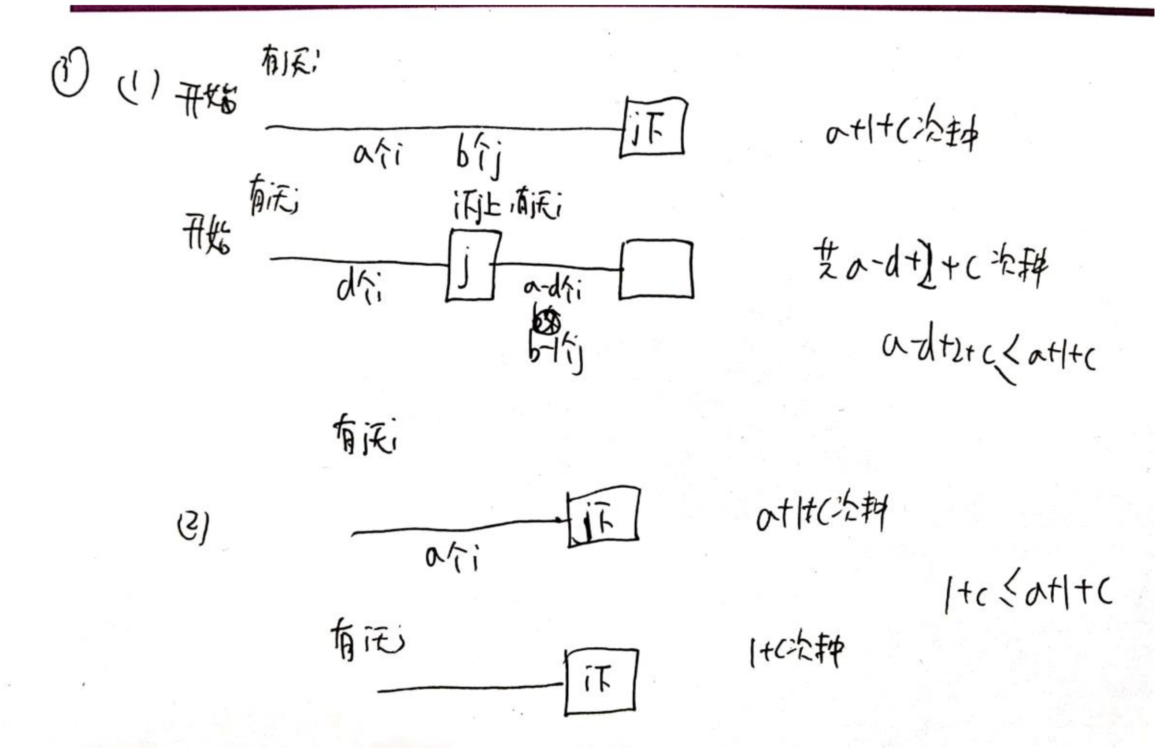

图 1： 第二题贪心正确性证明的情况 3

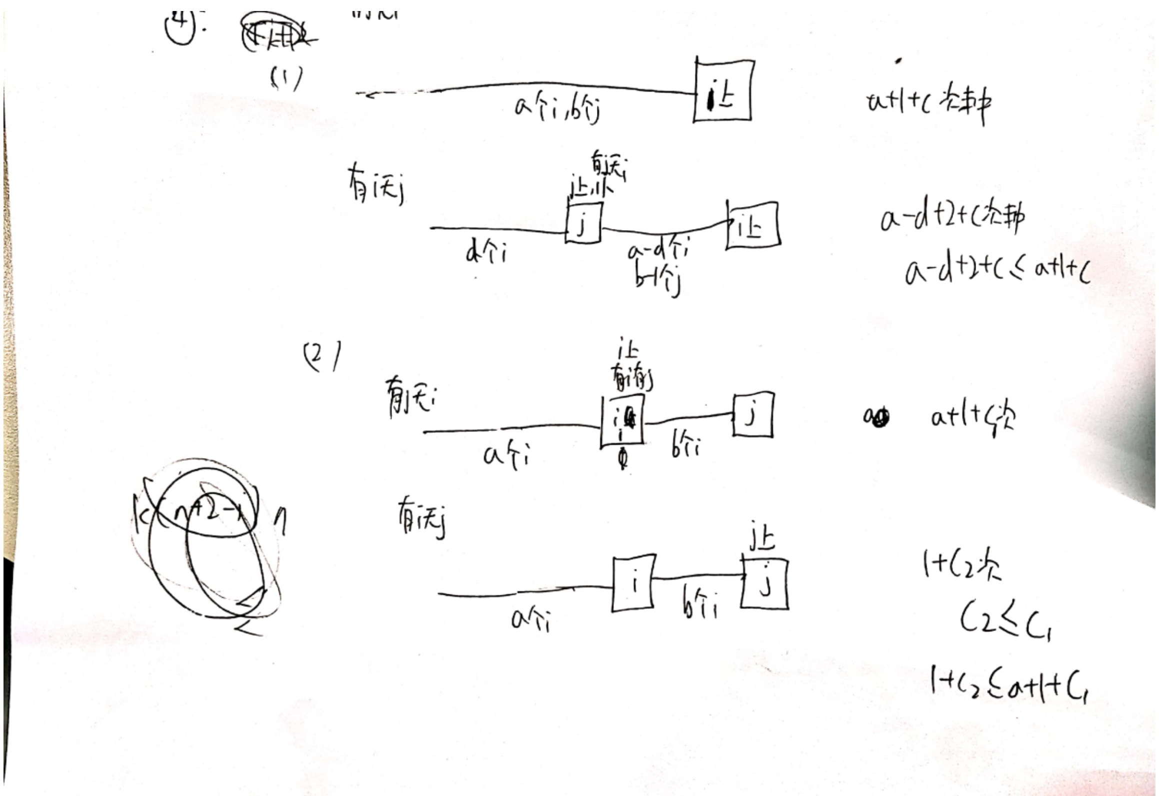

图 4： 第二题贪心正确性证明的情况 4