数据结构与算法贪心算法

陈宇琪

2020 年 4 月 19 日

摘要

主要内容贪心的基本例题。
提交要求：除了 EOJ 上的题目在 EOJ 上提交之外，其余 7 道题目到超星上提交。
请大家尽快用自己的语言回答问题，有一些瑕疵没有问题的！
作业 DDL：2020-04-19

目录

1 简答题 **2**

2 基础编程题 **2**

3 参考答案 **2**

3.1 简答题 2
3.2 编程题 4

1 简答题

1、在 PPT 中讲到了任务调度问题和哈夫曼编码，请对任务调度问题和哈夫曼编码问题简要写出问题描述，算法过程描述。

2、对于字符和词频：a-1,b-2,c-3,d-5,e-8,f-13,g-21,h-34，画出对应的哈夫曼树，并给出每个字符哈夫曼编码。

3、在 PPT 中指出任务调度算法不能按照开始时间排序，也不能按照任务长度排序，请各举一个反例。

4、认真阅读文档最后中关于任务调度问题的算法证明，并结合自己理解简述一下证明过程（不允许直接抄写）。

5、简述贪心算法关于局部最优解和全局最优解的关系（可以参考 CSDN，但是请用自己的语言回答）。

6、请**结合生活场景**，再举一个贪心算法在现实生活中的简单例子（不允许举 PPT 上有过的例子），请先描述一下背景，再具体解释其中蕴含的贪心思想。

2 基础编程题

1、（完整代码）给定 n 个数 $a_1, a_2, ...a_n$，输出 $2n$ 个数 $b_1, b_2, ..., b_{2n}$，满足 $a_i = \min(b_{2i-1}, b_{2i})$ 并且 $b_1, b_2, ..., b_{2n}$ 是 1 到 $2n$ 的一个排列，如果有多解，请输出字典序最小的解，如果无解，请输出-1。

例如：输入：3 1 5 7，输出：3 4 1 2 5 6 7 8；输入 5 6 7 8，输出：-1。

复杂度要求：$O(n \times log(n))$

提示：b_{2i-1} 和 b_{2i} 的大小关系能否确认？使用 vector 还是 set 实现？

2、完成 EOJ 上相关习题。

图 1: EOJ 相关习题

3 参考答案

3.1 简答题

1(1)、任务调度问题

问题：有 n 个任务，给出各自的开始和结束时间，求最多可以完成多少项任务。

算法过程：先将所有任务按照结束时间从小到大进行排序，然后再第一个任务开始向后搜索，并记录下第一个任务的结束时间。如果当前任务的开始时间晚于之前记录的任务的结束时间，则该任务可以被执行，可执行任务数加 1，并且将其结束时间记录下来，继续向后搜索。循环上述操作，直到任务被遍历一遍。

1(2)、哈夫曼编码：

问题：对于给定的字符串，如何编码使得编码后的字符串长度最短，使用的空间最小？

算法过程：先将每个字符按照频次从小到大进行排序，从中选取频次最小的两个左小右大作为左右节点构成一颗二叉树，并从字符集合中去除。生成根节点的频次等于左右子树的频次相加，再将根节点的频次放入字符集合中，重复上述过程，直到字符集合为空。最后按照左枝 0 右枝 1 对字符进行编码。

2、哈夫曼编码：a-1111111,b-1111110,c-111110,d-11110,e-1110,f-110,g-10,h-0，答案不唯一，但是哈夫曼编码要求和哈夫曼树对应。

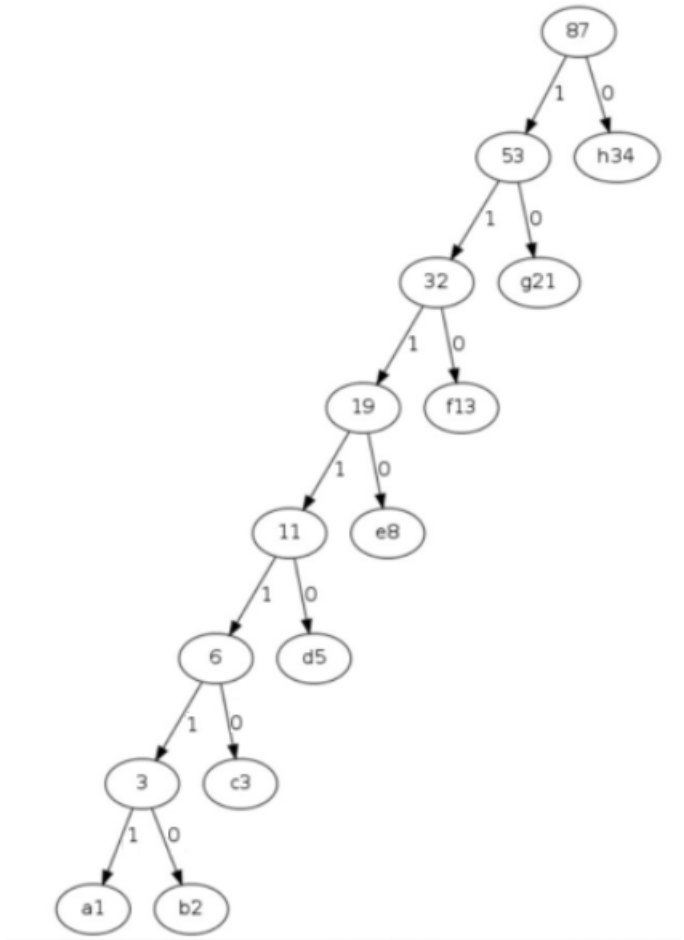

图 2: 哈夫曼树

3、如果按照开始时间排序，当存在例如 (1,2)、(3,9)、(4,5)、(6,9) 此类情况时，执行完 (1,2) 任务后，程序可能会优先选择 (3,9) 而不是 (4,5)，这样本来可以完成 3 项任务最终只完成了两项；如果按照任务长度排序，当存在例如 (1,5)、(4,7)、(6,10) 这样的情况时，程序会优先选择 (4,7) 任务开始执行，而后造成 (6,10) 任务本来可以执行却不能被执行，只执行了 1 项任务。

4、对于活动安排贪心算法的证明，首先假设 a_m 是任意非空子问题 S_k 中结束时间最早的活动，再令 A_k 为 S_k 的一个最大兼容活动子集，其中 a_j 为 A_k 中最早结束的活动。然后考虑两种情况：如果 $a_m = a_j$，那么显然 a_m 在子问题 S_k 的一个最大兼容活动子集当中；如果 $a_m \neq a_j$，则将 A_k 中的 a_j 用 a_m 来替换，得到 A_k，又因为 A_k 为一个最大兼容活动集，其中的活动都是互相兼容没有时间交叉的，并且 a_m 是子问题 S_k 中最早结束的活动，所以有 $f_m < f_j$，A_k 集合中的所有活动也是互相兼容的，而去掉一个 a_j 又补充一个 a_m，所以集合的大小不变，仍为一个最大兼容活动集合，因此也可以得出 a_m 在子问题 S_k 的一个最大兼容活动子集当中。所以每次对于子问题，只需要选择该子问题中结束时间最早的活动。贪心算法成立。

5、贪心算法期望用局部最优得到全局最优，但是从局部最优有时候不一定能得到全局最优。局部最优只关注当前的选择是否能带来最好的收益，而全局最优有时会需要考虑解决问题的每种可能的选择与情况。因此通过局部最优不一定能得到全局最优，例如大部分动态规划问题。但当问题具有贪心选择性质和最优子结构（最优解包含子结构的最优解）的时候，通过贪心算法获得的局部最优解就可以成为全局最优解。

6、结合实际问题应该给定一些合理的假设，一般问题只有在特定的假设条件下才能有贪心的成立。

（参考）幼儿园的时候中午老师会给小朋友吃小点心。胖胖的小朋友要吃大饼干才会开心，瘦瘦的小朋友给小饼干就开心了，只有给了小朋友和小朋友胖瘦等级相当的饼干小朋友才会开心，现在老师有 n 块饼干，老师也看的出小朋友的胖瘦等级，在老师尽量使较多小朋友开心的实践中，老师使用了贪心算法。每次分饼干，

老师总是会挑出目前剩下的最大的饼干先去满足目前还没有饼干的最胖的小朋友，如果饼干还是太小，这个小朋友就无法满足，如果饼干可以满足则给这个小朋友饼干吃。这样一来给饼干留了余地，所以小朋友得到的是最贪心的饼干，开心的小朋友增加了！

3.2 编程题

1、考虑如果 $b_{2i-1} > b_{2i}$ 则交换 b_{2i-1} 和 b_{2i} 条件仍然满足，而且字典序变小。所以 $b_{2i-1} < b_{2i}$，又因为 $a_i = \min\{b_{2i-1}, b_{2i}\}$，所以 $b_{2i-1} = a_i$。其次为了使得 b_{2i} 尽可能小，所以 b_{2i} 应该选择剩下的可以填的数中第一个大于 a_i 的数。所以我们需要一个能够求出第一个大于 x 的数，并且支持删除一个数，所以我们想到了使用 set 来完成。

注意：输入的数可能会有重复的数。

Listing 1: **ans1.cpp**

```
#include <bits/stdc++.h>
using namespace std;
const int maxn=1e6+5;
int n,a[maxn],b[maxn];
int main()
{
	cin>>n;
	set<int> s;
	for (int i=1;i<=2*n;i++)
		s.insert(i);
	for (int i=0;i<n;i++)
	{
		cin>>a[i];
		s.erase(a[i]);
	}
	if (s.size()!=n)
	{
		cout<<-1<<endl;
		return 0;
	}
	for (int i=0;i<n;i++)
	{
		if (s.upper_bound(a[i])==s.end())
		{
			cout<<-1<<endl;
			return 0;
		}
		b[i]=*s.upper_bound(a[i]);
		s.erase(b[i]);
	}
	for (int i=0;i<n;i++)
	{
		cout<<a[i]<<' '<<b[i]<<' ';
	}
	return 0;
}
```

第 16 章
Introduction to Algorithms, Third Edition

贪心算法

求解最优化问题的算法通常需要经过一系列的步骤，在每个步骤都面临多种选择。对于许多最优化问题，使用动态规划算法来求最优解有些杀鸡用牛刀了，可以使用更简单、更高效的算法。**贪心算法**(greedy algorithm)就是这样的算法，它在每一步都做出当时看起来最佳的选择。也就是说，它总是做出局部最优的选择，寄希望这样的选择能导致全局最优解。本章介绍一些贪心算法能找到最优解的最优化问题。在学习本章之前，你应该学习第 15 章动态规划，特别是应认真学习 15.3 节。

贪心算法并不保证得到最优解，但对很多问题确实可以求得最优解。我们首先在 16.1 节介绍一个简单但非平凡的问题——活动选择问题，这是一个可以用贪心算法求得最优解的问题。首先考虑用动态规划方法解决这个问题，然后证明一直做出贪心选择就可以得到最优解，从而得到一个贪心算法。16.2 节会回顾贪心方法的基本要素，并给出一个直接的方法，可用来证明贪心算法的正确性。16.3 节提出贪心技术的一个重要应用：设计数据压缩编码(Huffman 编码)。在 16.4 节中，我们讨论一种称为“拟阵”(matroid)的组合结构的理论基础，贪心算法总是能获得这种结构的最优解。最后，16.5 节将拟阵应用于单位时间任务调度问题，每个任务均有截止时间和超时惩罚。

贪心方法是一种强有力的算法设计方法，可以很好地解决很多问题。在后面的章节中，我们会提出很多利用贪心策略设计的算法，包括最小生成树(minimum-spanning-tree)算法(第 23 章)、单源最短路径的 Dijkstra 算法(第 24 章)，以及集合覆盖问题的 Chvátal 贪心启发式算法(第 35 章)。最小生成树算法提供了一个经典的贪心方法的例子。虽然可以独立学习本章和第 23 章，但 414
你会发现两章结合学习，效果更好。

16.1 活动选择问题

我们的第一个例子是一个调度竞争共享资源的多个活动的问题，目标是选出一个最大的互相兼容的活动集合。假定有一个 n 个**活动**(activity)的集合 $S=\{a_1, a_2, \cdots, a_n\}$，这些活动使用同一个资源(例如一个阶梯教室)，而这个资源在某个时刻只能供一个活动使用。每个活动 a_i 都有一个**开始时间** s_i 和一个**结束时间** f_i，其中 $0 \leqslant s_i < f_i < \infty$。如果被选中，任务 a_i 发生在半开时间区间 $[s_i, f_i)$ 期间。如果两个活动 a_i 和 a_j 满足 $[s_i, f_i)$ 和 $[s_j, f_j)$ 不重叠，则称它们是**兼容的**。也就是说，若 $s_i \geqslant f_j$ 或 $s_j \geqslant f_i$，则 a_i 和 a_j 是兼容的。在**活动选择问题**中，我们希望选出一个最大兼容活动集。假定活动已按结束时间的单调递增顺序排序：

$$f_1 \leqslant f_2 \leqslant f_3 \leqslant \cdots \leqslant f_{n-1} \leqslant f_n \tag{16.1}$$

(稍后，我们会看到这一假设的好处)。例如，考虑下面的活动集合 S：

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

对于这个例子，子集 $\{a_3, a_9, a_{11}\}$ 由相互兼容的活动组成。但它不是一个最大集，因为子集 $\{a_1, a_4, a_8, a_{11}\}$ 更大。实际上，$\{a_1, a_4, a_8, a_{11}\}$ 是一个最大兼容活动子集，另一个最大子集是 $\{a_2, a_4, a_9, a_{11}\}$。

下面分几个步骤来解决这个问题。我们可以通过动态规划方法将这个问题分为两个子问题，然后将两个子问题的最优解整合成原问题的一个最优解。在确定该将哪些子问题用于最优解时，要考虑几种选择。读者稍后会发现，贪心算法只需考虑一个选择(即贪心的选择)，在做贪心选择时，子问题之一必是空的，因此，只留下一个非空子问题。基于这些观察，我们将找到一种递归贪心算法来解决活动调度问题，并将递归算法转化为迭代算法，以完成贪心方法的过程。虽然本节介绍的步骤比典型的贪心算法的设计过程更为复杂，但它们说明了贪心算法和动态规划之间
415 的关系。

活动选择问题的最优子结构

我们容易验证活动选择问题具有最优子结构性质。令 S_{ij} 表示在 a_i 结束之后开始，且在 a_j 开始之前结束的那些活动的集合。假定我们希望求 S_{ij} 的一个最大的相互兼容的活动子集，进一步假定 A_{ij} 就是这样一个子集，包含活动 a_k。由于最优解包含活动 a_k，我们得到两个子问题：寻找 S_{ik} 中的兼容活动(在 a_i 结束之后开始且 a_k 开始之前结束的那些活动)以及寻找 S_{kj} 中的兼容活动(在 a_k 结束之后开始且在 a_j 开始之前结束的那些活动)。令 $A_{ik}=A_{ij}\cap S_{ik}$ 和 $A_{kj}=A_{ij}\cap S_{kj}$，这样 A_{ik} 包含 A_{ij} 中那些在 a_k 开始之前结束的活动，A_{kj} 包含 A_{ij} 中那些在 a_k 结束之后开始的活动。因此，我们有 $A_{ij}=A_{ik}\cup\{a_k\}\cup A_{kj}$，而且 S_{ij} 中最大兼容任务子集 A_{ij} 包含 $|A_{ij}|=|A_{ik}|+|A_{kj}|+1$ 个活动。

我们仍然用剪切－粘贴法证明最优解 A_{ij} 必然包含两个子问题 S_{ik} 和 S_{kj} 的最优解。否则，如果可以找到 S_{kj} 的一个兼容活动子集 A'_{kj}，满足 $|A'_{kj}|>|A_{kj}|$，则可以将 A'_{kj} 而不是 A_{kj} 作为 S_{ij} 的最优解的一部分。这样就构造出一个兼容活动集，其大小 $|A_{ik}|+|A'_{kj}|+1>|A_{ik}|+|A_{kj}|+1=|A_{ij}|$，与 A_{ij} 是最优解的假设矛盾。对子问题 S_{ik} 类似可证。

这样刻画活动选择问题的最优子结构，意味着我们可以用动态规划方法求解活动选择问题。如果用 $c[i,j]$ 表示集合 S_{ij} 的最优解的大小，则可得递归式

$$c[i,j]=c[i,k]+c[k,j]+1$$

当然，如果不知道 S_{ij} 的最优解包含活动 a_k，就需要考查 S_{ij} 中所有活动，寻找哪个活动可获得最优解，于是

$$c[i,j]=\begin{cases}0 & \text{若 } S_{ij}=\varnothing\\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{若 } S_{ij}\neq\varnothing\end{cases}\qquad(16.2)$$

于是接下来可以设计一个带备忘机制的递归算法，或者使用自底向上法填写表项。但我们可能
416 忽略了活动选择问题的另一个重要性质，而这一性质可以极大地提高问题求解速度。

贪心选择

假如我们无需求解所有子问题就可以选择出一个活动加入到最优解，将会怎样？这将使我们省去递归式(16.2)中固有的考查所有选择的过程。实际上，对于活动选择问题，我们只需考虑一个选择：贪心选择。

对于活动选择问题，什么是贪心选择？直观上，我们应该选择这样一个活动，选出它后剩下的资源应能被尽量多的其他任务所用。现在考虑可选的活动，其中必然有一个最先结束。因此，直觉告诉我们，应该选择 S 中最早结束的活动，因为它剩下的资源可供它之后尽量多的活动使用。(如果 S 中最早结束的活动有多个，我们可以选择其中任意一个)。换句话说，由于活动已按结束时间单调递增的顺序排序，贪心选择就是活动 a_1。选择最早结束的活动并不是本问题唯一的贪心选择方法，练习 16.1-3 要求设计其他贪心选择方法。

当做出贪心选择后，只剩下一个子问题需要我们求解：寻找在 a_1 结束后开始的活动。为什么不需要考虑在 a_1 开始前结束的活动呢？因为 $s_1<f_1$ 且 f_1 是最早结束的活动，所以不会有活动的结束时间早于 s_1。因此，所有与 a_1 兼容的活动都必须在 a_1 结束之后开始。

而且，我们已经证明活动选择问题具有最优子结构性质。令 $S_k=\{a_i\in S:\ s_i\geqslant f_k\}$为在 a_k 结束后开始的任务集合。当我们做出贪心选择，选择了 a_1 后，剩下的 S_1 是唯一需要求解的子问题[⊖]。最优子结构性质告诉我们，如果 a_1 在最优解中，那么原问题的最优解由活动 a_1 及子问题 S_1 中所有活动组成。

现在还剩下一个大问题：我们的直觉是正确的吗？贪心选择——最早结束的活动——总是最优解的一部分吗？下面的定理证明了这一点。 417

定理 16.1　*考虑任意非空子问题 S_k，令 a_m 是 S_k 中结束时间最早的活动，则 a_m 在 S_k 的某个最大兼容活动子集中。*

证明　令 A_k 是 S_k 的一个最大兼容活动子集，且 a_j 是 A_k 中结束时间最早的活动。若 $a_j=a_m$，则已经证明 a_m 在 S_k 的某个最大兼容活动子集中。若 $a_j\neq a_m$，令集合 $A_k'=A_k-\{a_j\}\cup\{a_m\}$，即将 A_k 中的 a_j 替换为 a_m。A_k'中的活动都是不相交的，因为 A_k 中的活动都是不相交的，a_j 是 A_k 中结束时间最早的活动，而 $f_m\leqslant f_j$。由于 $|A_k'|=|A_k|$，因此得出结论 A_k'也是 S_k 的一个最大兼容活动子集，且它包含 a_m。 ■

因此，我们看到虽然可以用动态规划方法求解活动选择问题，但并不需要这样做(此外，我们并未检查活动选择问题是否具有重叠子问题性质)。相反，我们可以反复选择最早结束的活动，保留与此活动兼容的活动，重复这一过程，直至不再有剩余活动。而且，因为我们总是选择最早结束的活动，所以选择的活动的结束时间必然是严格递增的。我们只需按结束时间的单调递增顺序处理所有活动，每个活动只考查一次。

求解活动选择问题的算法不必像基于表格的动态规划算法那样自底向上进行计算。相反，可以自顶向下进行计算，选择一个活动放入最优解，然后，对剩余的子问题(包含与已选择的活动兼容的活动)进行求解。贪心算法通常都是这种自顶向下的设计：做出一个选择，然后求解剩下的那个子问题，而不是自底向上地求解出很多子问题，然后再做出选择。

递归贪心算法

我们已经看到如何绕过动态规划方法而使用自顶向下的贪心算法来求解活动选择问题，现在我们可以设计一个直接的递归过程来实现贪心算法。过程 RECURSIVE-ACTIVITY-SELECTOR 的输入为两个数组 s 和 f[⊜]，表示活动的开始和结束时间，下标 k 指出要求解的子问题 S_k，以及问题规模 n。它返回 S_k 的一个最大兼容活动集。我们假定输入的 n 个活动已经按结束时间的单调递增顺序排列好(公式(16.1))。如果未排好序，我们可以在 $O(n\lg n)$时间内对它们进行排序，结束时间相同的活动可以任意排列。为了方便算法初始化，我们添加一个虚拟活动 a_0，其结束时间 $f_0=0$，这样子问题 S_0 就是完整的活动集 S。求解原问题即可调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)。 418

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1   m=k+1
2   while m≤n and s[m]<f[k]                    // find the first activity in S_k to finish
3       m=m+1
4   if m ≤ n
5       return {a_m}∪RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6   else return ∅
```

图 16-1 显示了算法的执行过程。在一次递归调用 RECURSIVE-ACTIVITY-SELECTOR(s,

⊖ 我们有时用 S_k 表示子问题而不是活动集合。根据上下文，可以很清楚地判定 S_k 表示一个活动集还是以该活动集为输入的子问题。

⊜ 因为伪代码把 s 和 f 作为数组，所以用方括号而不是下标来指向它们。

f，k，n)的过程中，第 2～3 行 **while** 循环查找 S_k 中最早结束的活动。循环检查 a_{k+1}，a_{k+2}，…，a_n，直至找到第一个与 a_k 兼容的活动 a_m，此活动满足 $s_m \geqslant f_k$。如果循环因为查找成功而结束，第 5 行返回$\{a_m\}$与 RECURSIVE-ACTIVITY-SELECTOR(s，f，m，n)返回的 S_m 的最大子集的并集。循环也可能因为 $m>n$ 而终止，这意味着我们已经检查了 S_k 中所有活动，未找到与 a_k 兼容者。在此情况下，$S_k=\varnothing$，因此第 6 行返回$\varnothing$。

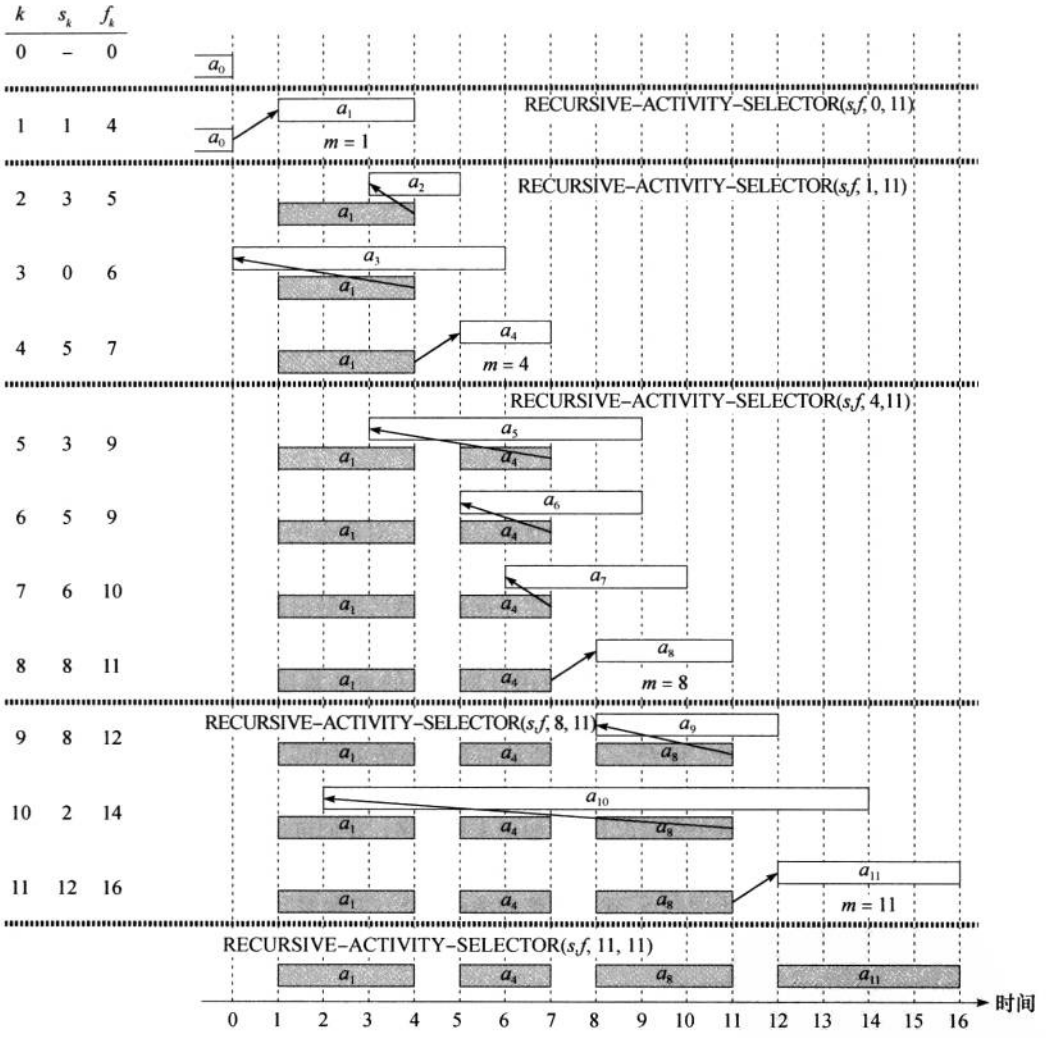

图 16-1 对前文给出的 11 个活动执行 RECURSIVE-ACTIVITY-SELECTOR 的过程。每次递归调用中处理的活动位于水平线之间。虚拟活动 a_0 于时刻 0 结束，因此第一次调用 RECURSIVE-ACTIVITY-SELECTOR(s，f，0，11)会选择活动 a_1。在每次递归调用中，被选择的活动用阴影表示，而白底方框表示正在处理的活动。如果一个活动的开始时间早于最近选中的活动的结束时间(两者间的箭头是指向左侧的)，它将被丢弃。否则(箭头指向右侧)，将选择该活动。最后一次递归调用 RECURSIVE-ACTIVITY-SELECTOR(s，f，11，11)返回$\varnothing$。选择的活动的最终结果集为$\{a_1, a_4, a_8, a_{11}\}$

假定活动已经按结束时间排好序，则递归调用 RECURSIVE-ACTIVITY-SELECTOR(s，f，0，n)的运行时间为 $\Theta(n)$，我们稍后证明这个结论。在整个递归调用过程中，每个活动被且只被

第 2 行的 **while** 循环检查一次。特别地，活动 a_i 在 $k<i$ 的最后一次调用中被检查。

迭代贪心算法

我们可以很容易地将算法转换为迭代形式。过程 RECURSIVE-ACTIVITY-SELECTOR 几乎就是“尾递归”(参见思考题 7-4)：它以一个对自身的递归调用再接一次并集操作结尾。将一个尾递归过程改为迭代形式通常是很直接的，实际上，某些特定语言的编译器可以自动完成这一工作。如前所述，RECURSIVE-ACTIVITY-SELECTOR 用来求解子问题 S_k，即由最后完成的任务组成的子问题。

419 ~ 420

过程 GREEDY-ACTIVITY-SELECTOR 是过程 RECURSIVE-ACTIVITY-SELECTOR 的一个迭代版本。它也假定输入活动已按结束时间单调递增顺序排好序。它将选出的活动存入集合 A 中，并将 A 返回调用者。

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n=s.length
2  A={a_1}
3  k=1
4  for m=2 to n
5      if s[m]≥f[k]
6          A=A∪{a_m}
7          k=m
8  return A
```

过程执行如下。变量 k 记录了最近加入集合 A 的活动的下标，它对应递归算法中的活动 a_k。由于我们按结束时间的单调递增顺序处理活动，f_k 总是 A 中活动的最大结束时间。也就是说，

$$f_k = \max\{f_i : a_i \in A\} \tag{16.3}$$

第 2～3 行选择活动 a_1，将 A 的初值设置为只包含此活动，并将 k 的初值设为此活动的下标。第 4～7 行的 **for** 循环查找 S_k 中最早结束的活动。循环依次处理每个活动 a_m，a_m 若与之前选出的活动兼容，则将其加入 A，这样选出的 a_m 必然是 S_k 中最早结束的活动。为了检查活动 a_m 是否与 A 中所有活动都兼容，过程检查公式(16.3)是否成立，即检查活动的开始时间 s_m 是否不早于最近加入到 A 中的活动的结束时间 f_k。如果活动 a_m 是兼容的，第 6～7 行将其加入 A 中，并将 k 设置为 m。GREEDY-ACTIVITY-SELECTOR(s, f)返回的集合 A 与 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)返回的集合完全相同。

与递归版本类似，在输入活动已按结束时间排序的前提下，GREEDY-ACTIVITY-SELECTOR 的运行时间为 $\Theta(n)$。