CS2045M: 高级数据结构与算法分析 **2025-2026 学年秋冬学期**

Lecture 9: 贪心算法

编写人：吴一航 yhwu_is@zju.edu.cn

说到贪心算法，想必诸位也不陌生。实际上我们已经学过很多贪心算法了，例如 Dijkstra 算法以及最小生成树的两种算法。在这些算法中，我们的任务是要得到一个最优的结果，于是每一步都选取满足某些性质的最好结果（例如 Dijkstra 每一步都找目前能找到的最近点）。如果能证明这些局部最优就是全局最优，那么贪心算法就是正确的。当然很可能贪心算法不一定能带来最优结果，那么这时可能要更换策略，或者本身问题就很难得到最优解，那么贪心作为一种高效率的启发式算法，可以尝试计算一个贪心算法能和最优解之间的差的最大比例（将在近似算法一节讨论）。接下来将讨论几个经典的贪心算法的应用，讨论设计贪心算法的关键以及证明其正确性的思路。

9.1 活动选择问题

给定一个活动集合 $S = \{a_1, a_2, \ldots, a_n\}$，其中每个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i，且 $0 \leq s_i < f_i < \infty$。如果活动 a_i 和 a_j 满足 $f_i \leqslant s_j$ 或者 $f_j \leqslant s_i$，则称活动 a_i 和 a_j 是兼容的（即二者时间不会重合）。活动选择问题就是要找到一个最大的兼容活动子集。

假设输入中是按照 n 个活动结束时间从小到大排列的。在学完动态规划后，相信读者应该对这一问题并不陌生——这里又遇到了类似于加权独立集合中有冲突的情况，所以同样可以设计出两种动态规划的递推式来解决这一问题：

1. 设 S_{ij} 表示活动 a_i 和 a_j 之间的最大兼容活动集合（开始时间在 a_i 结束之后，结束时间在 a_j 开始之前），其大小记为 c_{ij}，那么有

$$c_{ij} = \max\{c_{ik} + c_{kj} + 1 \mid f_i \leqslant s_k < f_k \leqslant s_j\} \tag{9.1}$$

   这一解法的思想更接近矩阵乘法顺序问题，会选择中间的最优解然后分为左右子问题递归。

2. 设 S_i 表示活动 $a_1, a_2, \ldots, a_i$ 的最大兼容活动集合，其大小记为 c_i，那么有

$$c_i = \max\{c_{i-1}, c_{k(i)} + 1\} \tag{9.2}$$

   其中 $k(i)$ 表示在 $1 \leqslant k \leqslant i$ 中，$f_k \leqslant s_i$ 且 c_k 最大的 k，即不与 a_i 冲突的最晚结束的活动。这一思想更接近背包问题的思路，即考察最后一个是否在解中，分成两种情况考虑。

看起来第二种办法更加便捷（也更加符合常人思维，特别是和加权独立集合问题比较），但也需要 $O(n^2)$ 的时间来解决这一问题（第二种递推式中 $k(i)$ 的计算也是需要时间的）。第一种方法这里也要提一下，

看起来时间复杂度是 $O(n^3)$ 的，但实际上上一讲最优二叉搜索树的平方优化也可以用在这里，所以复杂度也可以降到 $O(n^2)$。但对速度的追求不止于此，我们希望有一个更高效的算法，因此还需要新的思路，即使用贪心的思路，因为很多时候贪心只需要一个遍历即可确定出答案，所以时间上有很大的优势。

PPT 上给出了几种错误的贪心想法，这也表明了贪心并不是那么容易就能找到正确的贪心策略，需要不断地举反例否定自己，如果觉得自己逻辑足够严密就可以开始尝试证明。在这里可以找到一个可能正确的贪心策略为从前到后每次选择不冲突的最早结束的活动——这其实是一个很自然的想法，因为结束越早越能给后面的活动安排留出余地，因此也顺理成章能让更多活动都安排进来。下面来证明这一策略的正确性：令 $S_k = \{a_i \in S \mid s_i \geqslant f_k\}$，即为在 a_k 结束后开始的任务集合。当做出贪心选择，选择了 a_1 后，剩下只需要求解 S_1 这一子问题即可。如果 a_1 确实在最优解中，那么原问题的最优解显然就由活动 a_1 及子问题 S_1 的最优解构成（这一点称为贪心算法的最优子结构，根据前面讨论的动态规划的最优子结构性质可知是显然的，当然后面也马上会给严谨证明）。然后 S_1 内又可以按照贪心策略选择新的结束时间最早的活动，以此类推得到全部的解。现在还剩下一个大问题：贪心选择最早结束的活动真的是最优解的一部分吗？下面的定理证明了这一点。

定理 9.1 (活动选择问题贪心选择性质) *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。很显然的，A'_k 中的活动是不相交的，因为 A_k 中的活动是不相交的，而 a_m 是 A_k 中结束时间最早的活动，即 $f_m \leqslant f_j$，所以 a_m 不可能和 A_k 后面的活动冲突。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。 ■

这一证明思想称为 “交换参数法”，即可以假设存在一个最优选择，其中某个元素可能不在贪心选择中，然后通过交换贪心选择和最优选择的元素来构造一个不可能变差的解。这一思想在很多贪心算法的证明中都有应用，在后面的讨论中也会看到。在说明了这一性质（称为贪心选择性质）后，需要将之前挖的最优子结构的坑填上，只有结合这两点才能保证不断选择贪心选择的确是能得到最优解的：

定理 9.2 (活动选择问题最优子结构) *在活动选择问题中，用贪心策略选择 a_1 之后得到子问题 S_1，那么 a_1 和子问题 S_1 的最优解合并一定可以得到原问题的一个最优解。*

证明: 反证法：假设 a_1 和子问题 S_1 的最优解 C_1 合并得到的解 C，不是原问题的一个最优解。假设 C' 是原问题的一个最优解，则 $|C'| > |C|$。根据贪心选择性质可知，将 a_1 替换掉 C' 中的第一个元素得到 C'' 不会使得结果变差，即 $|C''| \geqslant |C'| > |C|$，那么将 C'' 除去 a_1 后，剩余的部分其实也是子问题 S_1 的一个解 C''_1，由于 $|C''| > |C|$，所以 $|C''_1| > |C_1|$，这和 C_1 是子问题 S_1 的最优解矛盾，因此原问题的最优解一定是 a_1 和子问题 S_1 的最优解合并得到的解。 ■

事实上思路非常简单，就是反证法，如果综合起来不是最优解，那 C_1 也不可能是子问题最优解，因为能找到一个更优的解。事实上以前的动态规划问题的最优子结构也可以采用这样的证明方法，只是动态规划中都认为最优子结构性质太显然所以没有特别强调。

综合以上贪心选择性质和最优子结构性质，最优解的构造方式就是：首先根据贪心选择性质找到最早结束的活动 a_1，根据最优子结构性质可知，它和 S_1 的最优解一起可以形成全问题的最优解，然后任务就变成了找 S_1 的最优解。重复贪心选择性质，要找到 S_1 中最早结束的 a_{i_1} 作为贪心选择，这样根据最优子结构，S_1 的最优解又可以表达为 a_{i_1} 和剩余子问题 S_{i_1} 的最优解的组合，然后找 S_{i_1} 的最优解，用贪心选择然后得到剩余子问题..... 以此类推，利用归纳法即可得到，直到子问题为空时可以得到整个问题的最优解。

当然，对称的也可以从后往前选择最晚开始的活动，看起来非常合理，事实上是否正确呢？答案是肯定的，证明方法也和上面差不多，留作讨论题：

【讨论】证明：从后往前依次选择不冲突的最晚开始的活动是一个正确的贪心策略。

最后来看时间复杂度，很显然的，在输入活动已按结束时间排序的前提下，只需要按结束时间升序遍历一遍所有活动即可，因此只需 $O(n)$ 的时间就能找到最优解。这一算法的时间复杂度是线性的，因此是非常高效的。如果未排序，可以在 $O(n \log n)$ 的时间内完成排序，因此总时间复杂度是 $O(n \log n)$。

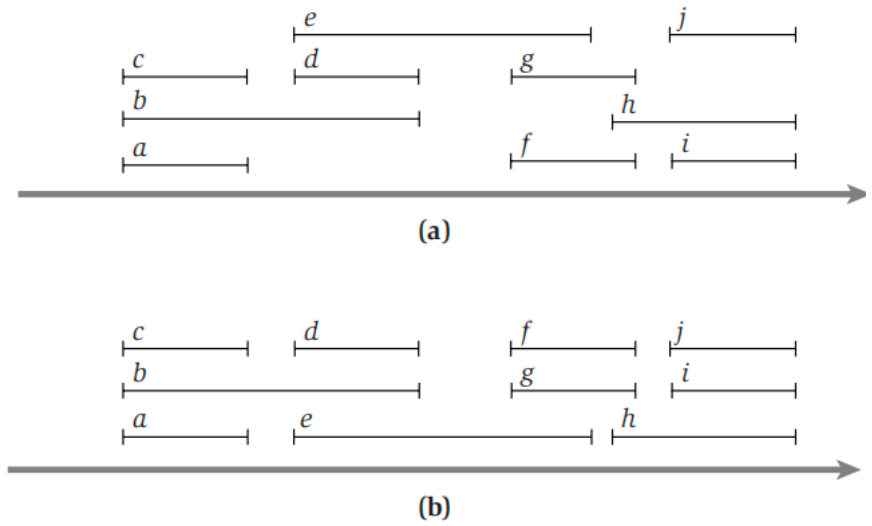
当然这一问题有很多有趣且非常常见的变体，可以在这里简单说明一下：

【变体 1】（加权活动选择问题）现在的问题不再是希望能包容进来的活动越多越好，而是每个活动都有一个权重 w_i，希望找到一个最大权重的兼容活动集合（前面讨论的问题实际上只是所有活动权重想等的特例）。事实上，很容易举出反例说明一般的贪心算法失效，所以现在可以转向最开头给出的动态规划算法，只是所有的加 1 应当改为加权重。

【变体 2】（区间调度问题）现在的问题不再是最大化兼容集合的大小或者权重，而是所有活动都必须举办，考虑将所有活动分配到最少的教室中，使得每个教室内的活动不冲突。

事实上这一问题只需要最简单的贪心算法就可以了。首先给出算法，然后给出简要的正确性说明。首先将所有活动按照开始时间排序，设置初始教室数量为 1，然后从前往后遍历。每次选择一个活动时，都看当前的教室中的活动有没有不冲突的，如果有就直接放进对应的教室，如果全部冲突则新开一个教室。

这样的算法为什么正确？可以看下页图中的例子。图 (a) 展示了 10 个输入的活动，事实上可以立刻断言：至少也需要三个教室才能完成活动。原因很简单，因为最多出现了三个活动同时进行的情况。而根据刚刚的方法，不可能分配出超过 3 个教室的情况，因为是出现冲突时才新开教室的，如果开出了四个教室那就说明在某个时刻有四个活动同时进行，这是不可能的。因此算法在这一场景下一定是正确的，显然更进一步的推广到任意冲突的情况也是正确的。图 (b) 给出了一个可行的解。这里贪心算法正确性的证明没有像前面那样用贪心选择和最优子结构，因为有更显然的教室数量上下界相等的性质可以更方便地说明。



【拓展】给你一个非负整数数组 nums，你最初位于数组的第一个下标。数组中的每个元素的值代表你在该位置可以跳跃的最大长度。

1. 设计一个线性时间算法判断你是否能够到达最后一个下标，如果可以，返回 true；否则，返回 false；
2. 假设题目给定的都是可以到达最后一个下标的数组，那么如何设计一个算法，使得你到达最后一个下标的步数最少？一个平方级别的算法可以用动态规划实现，但可以用贪心算法实现一个线性级别的算法。

举个例子，nums = [2,3,1,1,4]，那么显然第一个问题答案为 true，第二个问题答案为 2，实现方法为从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

本题来源于 leetcode，感兴趣的同学可以利用互联网找到问题的解答。这类题目往往思路比较巧妙，因为在线算法的设计往往需要一些观察和技巧。

9.2 调度问题

假设现在有 n 个任务，每个任务 i 都有一个正的完成需要的时间 l_i 和一个权重 w_i。假定只能按一定顺序依次执行这些任务，不能并行。

很显然的，有 $n!$ 种调度方法，记 σ 为某一种调度，那么在调度 σ 中，任务 i 的完成时间 $C_i(\sigma)$ 是 σ 中在 i 之前的任务长度之和加上 i 本身的长度。换句话说，在一种调度中，一个任务的完成时间就是这个任务从整个流程开始到它自己被执行完毕总共执行的时间。目标是最小化加权完成时间之和：

$$T = \min_{\sigma} \sum_{i=1}^{n} w_i C_i(\sigma).$$

举个例子，有三个任务，$l_1 = 1$，$l_2 = 2$，$l_3 = 3$，$w_1 = 3$，$w_2 = 2$，$w_3 = 1$。如果把第一个任务放在最前，第二个放在其次，第三个放在最后，那么三个任务的完成时间就是 1、3、6，因此加权时间和为 $3 \times 1 + 2 \times 3 + 1 \times 6 = 15$。如果读者检查全部的六种排序，这的确是最小的那一个。

接下来希望设计一个贪心算法高效地对所有情况都返回一个正确的结果，因为贪心算法看着就非常适合这一问题：可以按某个标准不断确定下一个任务是哪个，直至最后选出完整的最优解。思考贪心策略并不是一件很简单的事情，在后面会总结一些技巧例如考虑极端情况或者控制变量，这里就尝试控制变量：如果所有任务长度相同，应该把权重大还是小的放在前面？如果每个任务权重相同，应该先放时间短的还是长的？答案太显然了，就是先放权重大、时间短的。所以自然而然得到了如下两种可能的贪心策略：

1. 计算每个任务 i 的 $w_i - l_i$，按从大到小降序调度任务；
2. 计算每个任务 i 的 w_i/l_i，按从大到小降序调度任务。

显然二者都符合前面的直观，但哪个是正确的结果呢？如果回顾加权完成时间之和的公式，会发现权重和时间是会相乘的，所以第二种直观上看起来合理一些，事实上也的确如此，考虑一个例子：有两个任务，$l_1 = 5, l_2 = 2, w_1 = 3, w_2 = 1$，两种方法会返回不同的结果，而第二种才能返回最优解。

当然这只能说明第一种肯定是错误的，第二种的正确性还需要严谨证明。类似于前面的活动选择问题，这里仍然分为贪心选择性质和最优子结构两个部分进行证明：

定理 9.3（调度问题的贪心选择性质） *令 i 是当前 w_i/l_i 最大的任务，则在当前问题下，则一定存在将 i 排在首位的最优调度方式。*

证明： 仍然使用 “交换参数法”：假设有一个最优解 C，如果它的第一个任务是 i，结束证明。如果不是，考虑将 i 不断与前一个任务交换，直到换到第一个位置的过程。假设现在排在 i 前面一位的是 j，则一定有

$$\frac{w_i}{l_i} \geqslant \frac{w_j}{l_j},$$

又所有数都是正数，故移项有 $w_i l_j - w_j l_i \geqslant 0$。不难看出，$i$ 和 j 交换前后其它的任务加权时间完全没有变化，变化只在 i 和 j，设在 i 和 j 前面的总时间为 t，则交换前 i 和 j 的加权时间和为

$$t_1 = w_j(t + l_j) + w_i(t + l_j + l_i),$$

交换后为

$$t_2 = w_i(t + l_i) + w_j(t + l_i + l_j),$$

二者作差化简有

$$t_1 - t_2 = w_i l_j - w_j l_i \geqslant 0,$$

故把 i 往前换，加权时间和一定不会变大，故仍然保证最优解。那么就可以不断把 i 往前换，直到它在第一个位置，这样就证明了贪心选择性质。 ■

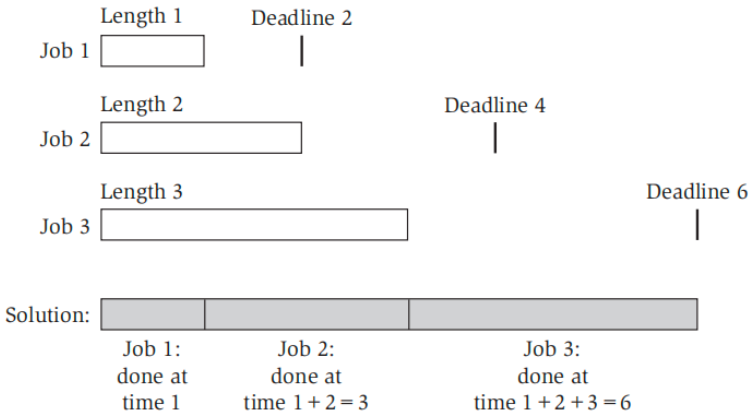
定理 9.4 (调度问题的最优子结构) *在调度问题 S 中，用贪心策略首先选择了最大的 w_i/l_i 对应的任务 i 后，剩下的子问题 S_1（即在除 i 外的任务中寻找一个最小化加权完成时间之和的解）的最优解 C_1 和 i 一起一定构成了原问题的一个最优解 C。*

证明: 和活动选择问题完全一致，非常直观的结论就用反证法秒杀：如果 C 不是最优解，那么一定有一个最优解 C'，它对应的加权完成时间之和 $T' < T$，其中 T 是 C 对应的加权完成时间之和。

根据贪心选择性质，如果把 C' 中的 i 不断通过相邻交换换到第一个位置，情况一定不会变差，因此还是最优解，将这一新的最优解记为 C''。于是 C'' 在选择了 i 之后，剩下的选择实际上也是 S_1 的一个解，由于 $T' < T$，这表明 C'' 中对应的 S_1 的解必定比 C_1 更好，但 C_1 是最优解，因此得到矛盾。所以 C_1 和 i 一起一定构成了原问题的一个最优解 C。 ■

综合以上两点，就很容易递推得到整个问题的最优解可以通过按 w_i/l_i 降序进行选择的方式得到。这一问题的时间复杂度显然最耗时的部分也在排序上，不再重复。

【变体 1】（最小化最大延时）设有 n 个任务，每个任务 j 具有一个完成需要的时间 t_j，以及一个截止日期 d_j。只能依次次完成这些任务，不能并行，如下图所示：



三个任务长度分别为 1、2、3，截止日期分别为 2、4、6，按如图方式排序，所有任务都能在截止前完成。当然有的时候可能没这么幸运，如果有任务超时，定义延迟时间 l_j 为其完成的时间减去截止日期 d_j，如果没有超时则延迟定义为 0，目标是最小化所有任务的最大延迟。

对这一问题已经有前面基础调度问题的经验，可以很容易想到如下几种贪心策略：

1. 按完成时间 t_j 从小到大排序，然后依次完成任务；这很符合小学的时候学习的接水的角度问题，但这一策略在这一问题下是不正确的，因为完成时间短的可能截止日期很长，例如考虑两个任务，一个需要 1 天，截止日期 100 天后；另一个需要 10 天，截止日期 10 天后，那么按照这里的调度方式显然不对；
2. 也可以根据前面基础调度问题的思想设计，看起来按照 $d_j - t_j$ 从小到大排序是个很好的选择：这表征了某个任务的冗余时间量，冗余时间越短应该越早安排你完成任务。这看起来很完美，但事

实上也是错的！例如考虑两个任务，一个需要 1 天，截止日期 2 天后；另一个需要 10 天，截止日期 10 天后，那么按照这里的调度方式也不对。

事实上上面两种思路出现了一个矛盾：第一个表明完成时间 t_j 越短越应该先完成，看起来很合理；第二个 t_j 越短反而冗余时间变大，应该晚完成——我们陷入了一个尴尬的局面，这是前面的基础调度问题没有的：这个 t_j 究竟应该在贪心中占据什么地位呢？答案是，没有地位就行。正确的贪心策略是直接按截止日期 d_j 从小到大排序进行调度：这看起来有些不可思议，因为完全忽略了 t_j，但事实证明如果利用 “交换参数” 法是可以证明这一贪心选择的合理性的，过程和上面基础调度问题类似，可以找到一组相邻逆序对然后交换，很容易发现交换后情况不会变差，即交换后二者延时最大值不会上升，那么整体最大的延时也不会上升（如果还是不太会可以参考《Algorithm Design》）。说明了这一贪心选择性质后，就可以利用显然的最优子结构性质导出得到最优解的方式：首先选择截止时间最早的任务 a_{i_1} 完成，然后剩下的问题调度是一个子问题 S_{i_1}，即除去 a_{i_1} 后剩下的任务的最小化最大延迟的问题，显然 a_{i_1} 和 S_{i_1} 的最优解放在一起能构成全问题的最优解（如果不相信，可以仿照前面问题的证明方式自己尝试写证明），所以继续取 S_{i_1} 中截止时间最早的任务即可，以此类推，每次都选择剩下的任务中截止最早的任务就能得到全问题最优解。

【变体 2】（最小化延时惩罚）设有 n 个任务，每个任务 j 都只需要单位时间即可完成，但是每个任务 j 都有一个截止日期 d_j，如果超出截止日期则会受到惩罚 w_j，希望找到一个调度方案，使得总的惩罚最小。这一问题见《算法导论》16.5 节，可以在初步了解后续介绍的拟阵后给出一个非常通用有效的贪心解法。

【变体 3】（完成工时最小化）假设有 n 项作业，每个作业 j 具有已知的长度 l_i，不同于前面的问题都只有一台机器顺序执行所有工作，这里有 m 台相同的机器可以同时处理作业。希望找到一个调度方案，使得所有作业的完成时间最小。

举个简单的例子，假如有两台一样的机器，作业长度分别为 $[1,2,2,3]$，那么可以将作业分配为 $[1,2],[2,3]$，这样完成时间最小为 5。如果将作业分配为 $[1,3],[2,2]$，那么完成时间最小为 4。这一问题直接使用贪心算法无法保证最优解，将在近似算法一讲中讨论这一问题。

【拓展】（离线最优缓存问题）如果本学期各位同学同时在学习计算机组成/计算机系统 II，那么这一问题是一个令人兴奋的联动。简而言之就是 cache 容量有限，在 cache miss 且 cache 已被装满的时候如何调整 cache 内的元素使得 cache miss 最小化。感兴趣的读者可以在《算法导论》的思考题 16-5 找到本题的详细版本，那里叙述了一种所谓 “将来最远” 的贪心策略，要求诸位证明其具有最优性质。

9.3 贪心算法的讨论

9.3.1 贪心算法的范式

贪心算法通过做出一系列选择来求出问题的最优解。在每个决策点，它做出在当时看来最佳的选择。这种启发式策略并不保证总能找到最优解，但对有些问题确实有效，如活动选择问题。一般地，可以按如下步骤设计贪心算法：

1. 将最优化问题转化为这样的形式：对其做出一次选择后，只剩下一个子问题需要求解；
2. 证明做出贪心选择后，原问题总是存在最优解，即贪心选择总是安全的；
3. 证明做出贪心选择后，剩余的子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解，这样就得到了最优子结构。

可以代入活动选择问题来看这一范式的应用，回忆对子问题 S_k 的定义以及正确性的证明，上述思路是非常清晰的。下面的问题是如何证明一个贪心算法是否能求解一个最优化问题呢？并没有适合所有情况的方法，例如之前涉及的 “交换参数法” 并不是唯一的证明贪心选择正确性的方法，在活动选择变体 2 中已经见到了对于特定问题更简单的证明方式。如果读者回顾 Dijkstra 算法，就会发现在那里是利用了数学归纳法来证明算法的正确性的。而最小生成树的两种算法正确性证明则涉及了图问题的一些更复杂的特性。当然这里学到的 “交换参数法” 是对于很大一部分问题比较直观的一种证明手段，但贪心这一思想实在是应用太广泛，所以无法保证这种证明方法对所有问题都适用。

但对于大部分问题贪心算法的证明而言，贪心选择性质和最优子结构是两个关键要素（读者可以代入前面已经讲到的问题进行理解）：

1. 贪心选择性质：可以通过做出局部最优（贪心）选择来构造全局最优解。换句话说，当进行选择时，直接做出在当前问题中看来最优的选择，而不必考虑子问题的解。

   这也是贪心算法与动态规划的不同之处。在动态规划方法中，每个步骤都要进行一次选择，但选择通常依赖于子问题的解。但在贪心算法中，总是做出当时看来最佳的选择，然后求解剩下的唯一的子问题。贪心算法进行选择时可能依赖之前做出的选择，但不依赖任何将来的选择或是子问题的解。

   如果进行贪心选择时不得不考虑众多选择，通常意味着可以改进贪心选择，使其更为高效。例如，在活动选择问题中，假定已经将活动按结束时间单调递增顺序排好序，则对每个活动能够只需处理一次。通过对输入进行预处理或者使用适合的数据结构（通常是优先队列），通常可以使贪心选择更快速，从而得到更高效的算法。

2. 最优子结构：一个问题的最优解包含其子问题的最优解。事实上在动态规划中就提到这一名词，事实上二者的含义是完全类似的。如前活动选择问题所述，通过对原问题应用一次贪心选择即可得

到子问题，那么最优子结构的工作就是论证：将子问题的最优解与贪心选择组合在一起的确能生成原问题的最优解。这种方法隐含地对子问题使用了数学归纳法，证明了在每个步骤对子问题进行贪心选择，一步一步推进就会生成原问题的最优解。

当然还有一个很困难的问题就是如何找到一个合理的贪心思路：这的确有些困难，有时候你可以去考虑一些极端情况，从中会发现具有哪些性质的解是最想要的（例如调度问题用这一思想就很容易想到可能的解法）。在多参数（例如调度问题有权重、完成时间、截止时间等参数）的情况下，也可以固定某些参数看另一些参数的变化等，这些小技巧都能帮助对问题的基本结构有一个观察。最后在证明时也是不应完全被套路束缚，有时候也应当从一些直观中找到灵感，特别是有的时候最优子结构其实非常显然，例如前面的两个问题就是如此，所以直观的感受结合将直观转化为严谨的叙述的过程是很重要的。

【讨论】（分数背包问题）给定 n 个物品和一个容量为 C 的背包，物品 i 的重量为 w_i，价值为 v_i。希望找到一个最优的方案，使得背包中物品的总价值最大。注意和 0-1 背包问题的区别在于，可以取物品的一部分，而不一定要取全部。

1. 设计一个解决这一问题的贪心算法，并计算其时间复杂度；
2. 证明你设计的贪心算法能返回问题的全局最优解；
3. 你设计的贪心算法对 0-1 背包问题是否适用？如果适用请给出证明，不适用请举出反例。

9.3.2 拟阵

在上一讲中已经简单探讨了动态规划与运筹学的关联，这两讲的优化问题很多都可以转变为数学规划问题求解，但因为有些问题有更特殊的结构，所以可以用更高效的算法实现。动态规划的结构已经非常熟悉，并且已经形成了某种“套路”，但贪心算法看起来似乎无章可循。事实上，对于一类贪心算法（并非全部，因为贪心实在是太常用），其背后也有一个比较复杂但美丽的统一结构，称其为拟阵。相关内容可以在《算法导论》中找到对应，当然也可以阅读组合优化相关教材学习，这里不展开描述。

9.4 哈夫曼编码

哈夫曼编码希望找到一个字母表的期望长度最小（依据字母出现频率）的前缀编码，在之前最优二叉搜索树的讨论中似乎有一个类似的目标，但那里是希望最小化搜索的期望时间，并没有前缀编码的需求。事实上，哈夫曼编码具有非常强的信息论背景。相信读者应该对于香农定义的“信息熵”并不陌生：

定义 9.5 (信息熵) *对于一个离散随机变量 X，其信息熵定义为*

$$H(X) = -\sum_x p(x) \log_2 p(x) \tag{9.3}$$

其中 $p(x)$ 是 X 取值为 x 的概率。

使用以 2 为底的对数是为了保证信息熵的单位是比特，在平均意义下，信息熵可以理解为对于一个随机变量 X，需要多少比特来表示它。举几个简单的例子：

例 9.6 考虑一个服从均匀分布且有 32 种可能取值的随机变量 X。为了确定一个结果，需要一个能容纳 32 个不同值的标识，因此用 5 个比特足矣。而其信息熵为

$$H(X) = -\sum_{i=1}^{32} \frac{1}{32} \log_2 \frac{1}{32} = 5 \tag{9.4}$$

这也是预期的结果。

例 9.7 假定有 8 匹马参加的一场赛马比赛，它们的获胜概率分别为 $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$。那么这场比赛的信息熵为

$$H(X) = -\sum_{i=1}^{8} p_i \log_2 p_i = 2 \tag{9.5}$$

假定我们要把哪匹马会赢的信息告诉给别人，其中一个策略是发送胜出的马的编号，这样对于任何一匹马都需要 3 个比特。但由于概率不是均等的，明智的方法是对概率大的马用更短的编码，对概率小的马用更长的编码。例如使用以下编码：$0, 10, 110, 1110, 111100, 111101, 111110, 111111$，这样平均每匹马需要 2 个比特，比等长的比特数更短。

事实上，在信息论中，可以证明随机变量的任一前缀 $0-1$ 编码的期望长度必定大于等于其信息熵。而哈夫曼编码就是一种前缀编码，其期望长度等于信息熵。因此有很多人都尝试从信息论角度研究最优的前缀编码。1951 年，哈夫曼和他在 MIT 信息论的同学需要选择是完成学期报告还是期末考试。导师 Robert M. Fano 给他们的学期报告的题目是，寻找最有效的二进制编码。由于无法证明哪个已有编码是最有效的，哈夫曼放弃对已有编码的研究，转向新的探索，最终发现了基于有序频率二叉树编码的想法，并很快证明了这个方法是最有效的。由于这个算法，学生终于青出于蓝，超过了他那曾经和信息论创立者香农共同研究过类似编码的导师。事实上哈夫曼的构造思想在编码理论中算得上完独树一帜的，因为这一贪心算法的构造的确与当时主流的想法完全不同。当然算法过程过于简单不再赘述。在此直接跳跃到正确性证明，对 PPT 中两个看起来不明所以的证明正确性引理给出一些解释。事实上这两个证明恰好也对应了之前讨论的贪心算法的两个关键要素。

引理 9.8 (贪心选择性质) C 为一个字母表，其中每个字符 $c \in C$ 都有一个频率 $c.freq$。令 x 和 y 是 C 中频率最低的两个字符。那么存在 C 的一个最优前缀码，x 和 y 的码字长度相同，且只有最后一个二进制位不同。

这一引理说明在哈夫曼编码中使用的贪心选择（即把两个频率最小的结合成一个新结点）是可以保证最优解仍然存在的。证明比较简单，在这里也就只叙述思想了。思路仍然是交换参数：令 T 表示任意一个最优前缀码所对应的编码树，对其进行修改，得到表示另外一个最优前缀码的编码树，使得在新树中，x 和 y 是深度最大的叶结点，且它们为兄弟结点。如果可以构造这样一棵树，那么 x 和 y 的码字

将有相同长度，且只有最后一位不同。幸运的是 PPT 第 14 页的构造就满足这一点，也很容易验证新的树的代价只可能小于等于原来的树，而原来的树是最优的，因此新的树必定也是最优的。

接下来，很自然地需要证明最优子结构性质，即贪心选择加上剩下的子问题的最优解构成整体最优解：

引理 9.9 (最优子结构性质) *C 为一个字母表，其中每个字符 $c \in C$ 都有一个频率 $c.freq$。令 x 和 y 是 C 中频率最低的两个字符。令 C' 为 C 去掉字符 x 和 y，加入一个新字符 z 后的字母表。给 C' 也定义频率集合，不同之处只是 $z.freq = x.freq + y.freq$。令 T' 为 C' 的任意一个最优前缀码树，那么可以将 T' 中叶结点 z 替换为一个以 x 和 y 为孩子的内部结点得到一个 C 的一个最优前缀码树 T。*

如果上面这一命题正确，那么每次合并 x 和 y 得到 z 之后，按照没有 x 和 y，只有 z 的子问题继续推进贪心算法可以得到 T' 这一子问题的最优解，它和合并 x 和 y 得到 z 这一前面已经验证正确性的贪心选择一起，就构成了整体的最优解。下面证明这一引理，关键在于反证法，即如果最后不能得到最优解，那么子问题也不可能是最优解：

证明: 记 $B(T)$ 树 T 的代价，即所有字母的期望编码长度（即 PPT 上的 cost）。事实上很容易验证 $B(T') = B(T) - x.freq - y.freq$，因为 T' 就相当于把 T 中 x 和 y 的频率代价上移一层。然后可以用反证法证明这一引理。假设 T 不是 C 的最优前缀编码树，即存在树 T'' 使得 $B(T'') < B(T)$。根据前面的引理，将 T'' 中的 x 和 y 和它们的父结点替换成 z，得到一个新树 T'''，其中 $z.freq = x.freq + y.freq$。那么有

$$B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T'),$$

这与 T' 是 C' 的最优前缀编码树矛盾。因此假设是错误的，T 必定是 C 的最优前缀编码树。 ■

所以这一证明其实非常简单，就是很符合直觉的如果 T 不是最优那么 T' 必定不可能是从最优解来的。综上两个引理就证明了哈夫曼编码的正确性。这一证明也是贪心算法证明的一个典型例子，可以看到贪心选择性质和最优子结构性质是贪心算法证明的两个关键要素。

9.5 一些拓展问题

9.5.1 田忌赛马问题

田忌赛马是中国历史上有名的扬长避短而在竞技中获胜的故事，下面是这一故事的改编：田忌和齐王赛马，他们各有 n 匹马，每次双方各派出一匹马进行赛跑，获胜一方记 1 分，失败一方记 -1 分，平局不计分。假设每匹马只能出场一次，每匹马都有一个速度值，比赛中速度快的马一定会获胜。田忌知道所有马的速度值，且田忌可以安排每轮赛跑双方出场的马，问田忌如何安排马的出场顺序，使得最后获胜的比分最大?

【提示】贪心法求解田忌赛马的贪心策略是保证每一场赛跑都是最优方案，分别考虑如下情况：

1. 田忌最快的马比齐王最快的马快，则拿两匹最快的马进行赛跑，因为田忌最快的马一定能赢一场，此时选齐王最快的马是最优的；

2. 田忌最快的马比齐王最快的马慢，则拿田忌最慢的马和齐王最快的马进行赛跑，因为齐王最快的马一定能赢一场，此时选田忌最慢的马是最优的；

3. 田忌最快的马与齐王最快的马速度相等，考虑以下两种情况：

   (a) 田忌最慢的马比齐王最慢的马要快，则拿两匹最慢的马进行赛跑，因为齐王最的马一定会输一场，此时田忌选最慢的马一定是最优的；

   (b) 否则用田忌最慢的马与齐王最快的马赛跑，因为田忌最慢的马一定不能赢一场，而齐王最快的马一定不会输一场，此时选田忌最慢的马一定是最优的。

显然，这一时间复杂度是线性的，因此要做 n 次选择，每次都是常数时间就能决定。当然目前只是直观上认为上述策略是合理的，如果要严谨说明仍然需要按照两个要素来证明。

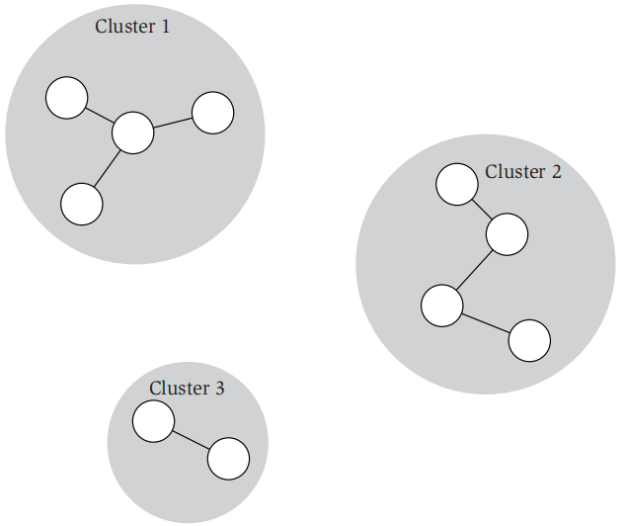
9.5.2 聚类问题

聚类问题是一个非常经典的无监督机器学习问题，它来源于一个非常自然的场景，即希望将一组数据分成若干个类别，使得每个类别内的数据尽可能相似，而不同类别之间的数据尽可能不同。在聚类问题中，通常会使用一些距离度量来衡量数据之间的相似性，具体的距离函数可以忽略，只需要满足度量的基本性质即可。

然后可以将聚类问题形式化：给定 n 个点，希望将这 n 个点分成 k 个类别 $C_1, \ldots, C_k$（其中 k 是给定的，称为 $k-$聚类）。能选择的优化目标很多，这里首先定义 $k-$聚类 的 “间隙值” 为处于不同类别的两个点的最近距离，然后希望最大化聚类结果的间隙值（还有另一种对称的优化目标，即最小化每个点到聚类中心的距离，这一问题将在近似算法中讨论）。

有一个比较直观的做法，就是按照点之间的距离从小到大排序，然后将依次将最短距离对应的边加入图中，最后连上边的就认为在同一个聚类中，实际上知道每个聚类就是一个联通分支，于是直到最后只剩下 k 个联通分支时就停止。当然要注意的是如果下一个最短边将要连接的是已经在同一个联通分支中的两个点，就不用再连接这条边了。有趣的是，这一算法事实上就是用 Kruskal 算法得到的最小生成树的过程，只不过最后还剩下 $k-1$ 条边没有连接罢了。

算法正确性的证明并不困难，记根据上述贪心算法得到的结果为 C，如果有另一种聚类方式 C' 的话（还是考虑交换参数），那么 C' 中一定存在 C 中处于同一聚类的两个点到了不同聚类中，又这两个点的距离按照 Kruskal 算法的计算顺序一定小于等于那些在 C 中没有连接的边（因为 C 的连接顺序是按照 Kruskal 算法的计算顺序来的），因此 C' 的间隙值首先至少小于等于这两点的距离，然后它们的距离又小于等于 C 的间隙值，因此算法是最优的。



9.5.3 匹配问题

【房屋交换问题】有 n 个人，每个人初始时都有一个房子，但每个人不一定最喜欢他自己当前住的房子。每个人对所有房子都有个偏好顺序，即他最喜欢的房子排在第一位，次喜欢的房子排在第二位，以此类推。现在要求找到一个房子分配新方案，使得每个人能得到比现在更喜欢的房子（或者至少保持原样）。如何设计一个贪心算法来解决这个问题?

【稳定匹配问题】有 n 个男生和 n 个女生，每个人对异性都有一个偏好顺序，即他最喜欢的异性排在第一位，次喜欢的异性排在第二位，以此类推。现在要求找到一个匹配方案，使得没有一对人会同时对这个匹配方案不满意（即男生 v 和女生 w 没有匹配成功，但相比于他们当前的匹配，他们更喜欢对方）。如何设计一个贪心算法来解决这个问题?

以上两个有趣的问题属于经典的算法博弈论领域的基本问题，感兴趣的读者可以阅读《斯坦福算法博弈论二十讲》的 9、10 两章，或者利用互联网搜索相关资料。

除此之外，如果读者还对贪心算法感兴趣的话，可以参考《算法导论》、《Algorithm Design》等教材上的例题与习题等，作业中的找零钱问题就源于《算法导论》，多阶段的贪心算法（最小成本树状图）等可见《Algorithm Design》，限于篇幅和时间这里不能详细展开描述。