算法设计与分析

主讲人：吴庭芳

Email: tfwu@suda.edu.cn

苏州大学

计算机科学与技术学院







第十二讲 贪心算法

内容提要：

- 贪心算法思想
- 活动选择问题
- 贪心算法原理

贪心算法思想

- **求解最优化问题的算法通常需要经过一系列的步骤，在每个步骤面临多种选择。使用动态规划算法是将这些选择都计算一遍，从而得到最优选择**
- **实际上，对于有些问题可以使用更简单、更高效的算法：贪心算法**
  - 贪心算法是这样一种策略：它在每一步都做出**当时看起来的最佳选择**，也就是做出**局部最优选择**，并寄希望这样的选择最终能获得**全局最优解**
  - 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
  - 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，比如最小生成树算法、单源最短路径的 Dijkstra 算法等







第十二讲 贪心算法

内容提要：

- 贪心算法思想
- 活动选择问题
- 贪心算法原理

活动选择问题

- **调度竞争共享资源的多个活动选择问题：**
  - 假定有一个包含 n 个活动的集合 $S=\{a_1, a_2, \cdots, a_n\}$，这些活动使用同一个资源（例如同一个教室），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 和结束时间 f_i，其中 $0 \le s_i < f_i$。如果被选中，任务 a_i 发生在半开区间 $[s_i, f_i)$ 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，即 $s_i \ge f_j$ 或 $s_j \ge f_i$
  - 在活动选择问题中，希望选出一个最大兼容活动集。不失一般性，设活动已经按照结束时间单调递增排序：

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

活动选择问题

- 例如：设有 11 个待安排的活动，它们的开始和结束时间如下，并假设活动已经按结束时间的非减序排序：

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

按结束时间的非减序排序

- 则 $\{a_3, a_9, a_{11}\}$、$\{a_1, a_4, a_8, a_{11}\}$、$\{a_2, a_4, a_9, a_{11}\}$ 都是兼容活动集合
- 其中 $\{a_1, a_4, a_8, a_{11}\}$、$\{a_2, a_4, a_9, a_{11}\}$ 是最大兼容活动集合。显然最大兼容活动集合不一定是唯一的

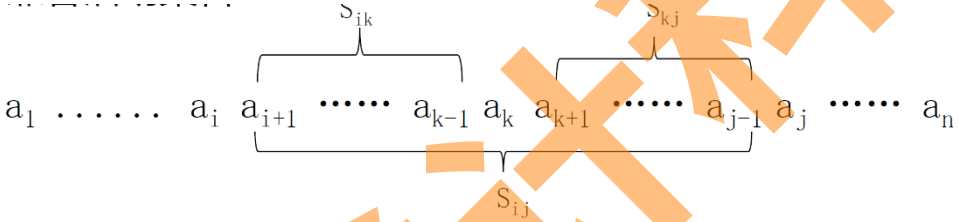
活动选择问题

- **活动选择问题分析：**
  - 可以通过动态规划方法将活动选择问题分为两个子问题，然后将两个子问题的最优解组合成原问题的一个最优解。在确定将哪些子问题用于组合成最优解时，需要考虑几种选择
  - 贪心算法更简单一些，只需要考虑一个选择（即贪心选择）

活动选择问题

活动选择问题的最优子结构

- 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



- 设 A_{ij} 是 S_{ij} 的一个最大兼容活动集合，并设 A_{ij} 包含活动 a_k（做出一个选择 k），则得到两个子问题：寻找 S_{ik} 的最大兼容活动集合（在 a_i 结束之后开始且 a_k 开始之前结束的那些活动）和寻找 S_{kj} 的最大兼容活动集合（在 a_k 结束之后开始且 a_j 开始之前结束的那些活动）
- 设 $A_{ik} = A_{ij} \cap S_{ik}$，$A_{kj} = A_{ij} \cap S_{kj}$，则 A_{ik} 包含 A_{ij} 中 a_k 开始之前结束的活动子集，A_{kj} 包含 A_{ij} 中 a_k 结束之后开始的活动子集，因此有 $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$（原问题 S_{ij} 的最优解 A_{ij} 由两个子问题的解所构成）

活动选择问题

- **活动选择问题具有最优子结构性，即：**
  - **证明**：用剪切-粘贴法证明最优解 $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ 必然包含两个子问题 S_{ik} 和 S_{kj} 的最优解，即 A_{ik} 必然是 S_{ik} 一个最大兼容活动子集，A_{kj} 必然是 S_{kj} 一个最大兼容活动子集
    - 设 S_{kj} 存在另一个最大兼容活动集 A'_{kj}，满足 $|A'_{kj}| > |A_{kj}|$，则可以将 A'_{kj} 作为 S_{ij} 最优解的一部分。这样就构造出一个兼容活动集合，其大小：

$$|A_{ik}|+|A'_{kj}|+ 1 > |A_{ik}|+ |A_{kj}|+1 = |A_{ij}|,$$

与 A_{ij} 是最大兼容活动集合相矛盾

活动选择问题

- **动态规划方法**
  - 活动选择问题具有最优子结构性，所以可用态规划方法求解
  - 令 $c[i, j]$ 表示集合 S_{ij} 的最优解大小，即兼容活动的个数，可以得到递归式如下：

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

为了选择 k，有：

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

  - 可以设计带备忘机制的或自底向上的动态规划算法进行求解

活动选择问题

活动选择问题的贪心算法

- 假如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中，那么可以省去上述递归式中考查所有选择的过程（从活动集合 S_{ij} 遍历 a_k）。实际上，对于活动选择问题，只需考虑一个选择：**贪心选择**
- **贪心选择**：在贪心算法的每一步所做的当前最优选择（局部最优选择）
- 比如，活动选择问题的贪心选择：每次总选择具有最早结束时间的兼容活动加入到集合 A 中
- **为什么？**直观上，按这种方法选择兼容活动可以为未安排的活动留下尽可能多的时间。也就是说，该算法的贪心选择意义是使剩余的可安排时间段最大化，以便安排尽可能多的兼容活动

活动选择问题

活动选择问题的贪心算法

- 注意：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，其他贪心选择有：选择持续时间最短者、选择与其他剩余活动重叠最少者、以及选择时间最早开始者，但均不能得到最优解

活动选择问题

- 例：

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

结束时间递增

- **贪心选择：**由于活动已按结束时间单调递增的顺序排序，选择结束时间最早的活动 a_1。当做出贪心选择后，只剩下一个子问题需要求解：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？因为 $s_1 < f_1$，且 f_1 是最早结束时间，所以不会有活动的结束时间早于 s_1。因此所有与 a_1 兼容的活动都是在 a_1 结束之后开始
- **子问题定义：**令 $S_k = \{a_i \in S \mid s_i \geq f_k\}$，即在 a_k 结束之后开始的所有活动集合。当做出贪心选择 a_1 后，剩下的 S_1 是唯一需要求解的子问题

活动选择问题

- 例：

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

结束时间递增

- 活动选择问题的贪心策略下的最优子结构：如果 A 是原问题 S 的一个最优解，并且已经做出了贪心选择，选择了结束时间最早的活动 a_1，那么 A 可以由 a_1 和子问题 S_1 的一个最优解 A_1 合并而成，即原问题 S 的一个最优解 A = {贪心选择 a_1} ∪ 子问题 S_1 的最优解 A_1
- **注意：**这里说“S 的一个最优解”，因为可能有很多个最优解，但是只关心包含 a_1 的那个最优解。即只关心“是否存在”一个最优解，而不关心这个最优解是否唯一

活动选择问题

- **例：**

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

结束时间递增

- **贪心策略下的最优子结构特点：**
  1. 做出贪心选择后，将原问题简化为唯一的一个子问题（在 a_1 结束后开始的活动集合 S_1 中选择最优解）
  2. 这个子问题与原问题具有相同的结构，可以递归应用贪心策略
  3. 不需要考虑不包含 a_1 的情况，因为**贪心选择性质**保证了 a_1 一定在某个最优解中

活动选择问题

- **例：**

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

结束时间递增

- **动态规划策略下的最优子结构特点：**

1. 原问题的最优解需要遍历所有可能的选择（选择哪个活动 $a_k \in S_{ij}$ 作为第一个），然后合并两个子问题的最优解
2. 需要解决多个子问题（所有可能的 S_{ik} 和 S_{kj}），并比较这些子问题的最优组合
3. 动态规划通常使用表格来存储所有子问题的解，避免重复计算

活动选择问题

- **直觉正确吗？即按照上述贪心选择（最早结束的活动）方法选择的活动集合是问题最优解吗？即证明贪心选择性质**
- **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 必在 S_k 的某个最大兼容活动子集中
- **证明：** 令 A_k 是 S_k 中的一个最大兼容活动子集，且 a_j 是 A_k 中结束最早的活动。1）若 $a_j = a_m$，则得证

  2）否则，令 $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$，即将 A_k 中的 a_j 替换为 a_m。因为 A_k 中的活动都不相交，a_j 是 A_k 中结束时间最早的活动，而 a_m 是 S_k 中结束时间最早的活动，所以 $f_m \leq f_j$，即 A'_k 中的活动也是不相交的

  由于 $|A'_k| = |A_k|$，所以 A'_k 也是 S_k 的一个最大兼容活动子集，且包含 a_m，得证

活动选择问题

- **定理 16.1 的含义：选结束时间最早的 a_m 不会错！**
  - 对于活动选择问题，虽然可以用动态规划方法进行求解，但是并不需要这么麻烦
  - 贪心选择性质保证了当前选择是安全的，并且会导向全局最优解。因此，从 S_0 开始，可以反复选择结束时间最早的活动，重复这一过程直至不再有剩余的兼容活动。所得子集就是最大兼容活动集合
  - 由于结束时间严格递增，故只需按照结束时间的单调递增顺序处理所有活动，每个活动只需考查一次
  - 贪心选择算法通常是自顶向下设计：作出一个贪心选择之后，只需要求解唯一的一个子问题

活动选择问题

活动选择问题的贪心算法

- 采用自顶向下的设计：首先做出一个选择，然后求解剩下的子问题。每次选择将问题转化成一个规模更小的问题

```
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 ∅
```

查找 S_k 中最早结束的活动，直至找到第一个与 a_k 兼容的活动 a_m，即开始时间 s_m 大于活动 a_k 的结束时间 f_k，$s_m > f_k$

$m > n$，意味着在 S_k 中未找到与 a_k 兼容的活动

- 数组 s、f 分别表示 n 个活动的开始时间和结束时间，下标 k 指出要求解的子问题 S_k。并假定 n 个活动已经按照结束时间单调递增排列好，算法返回 S_k 的一个最大兼容活动集
- 初次调用：RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)

活动选择问题

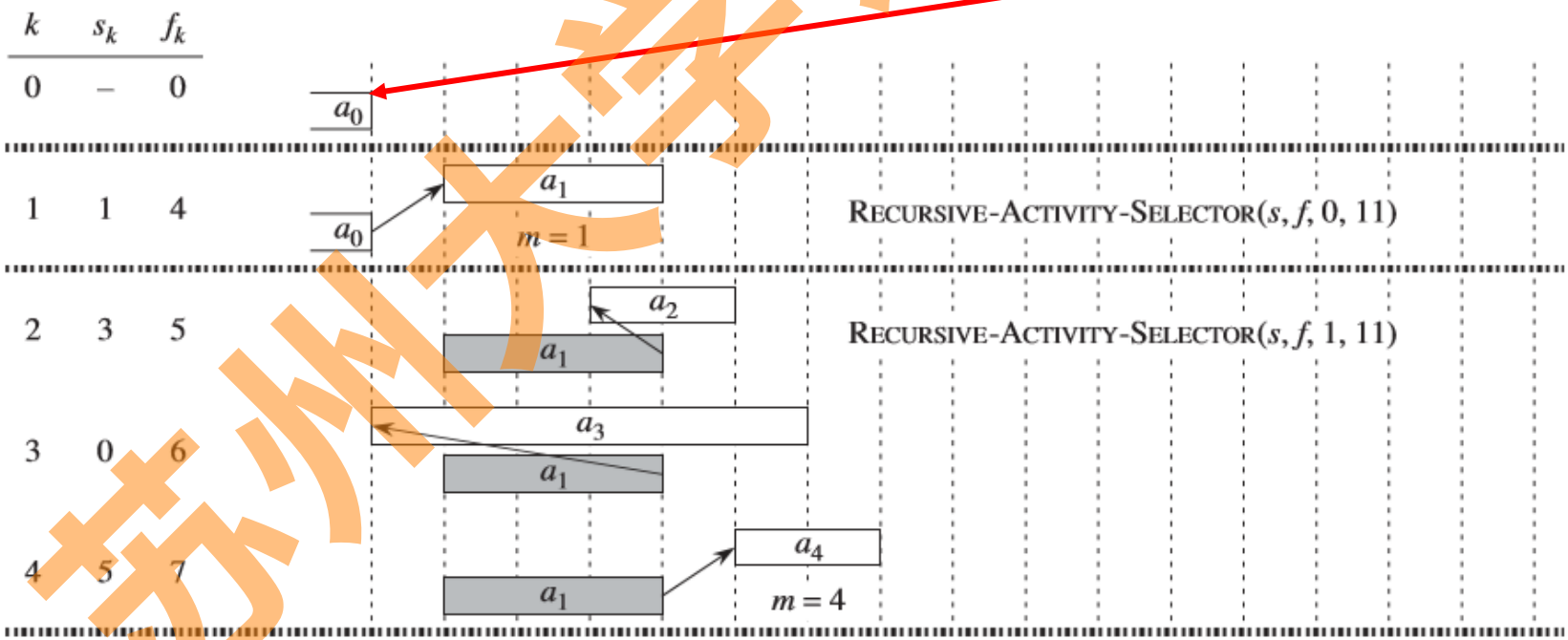
例：

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

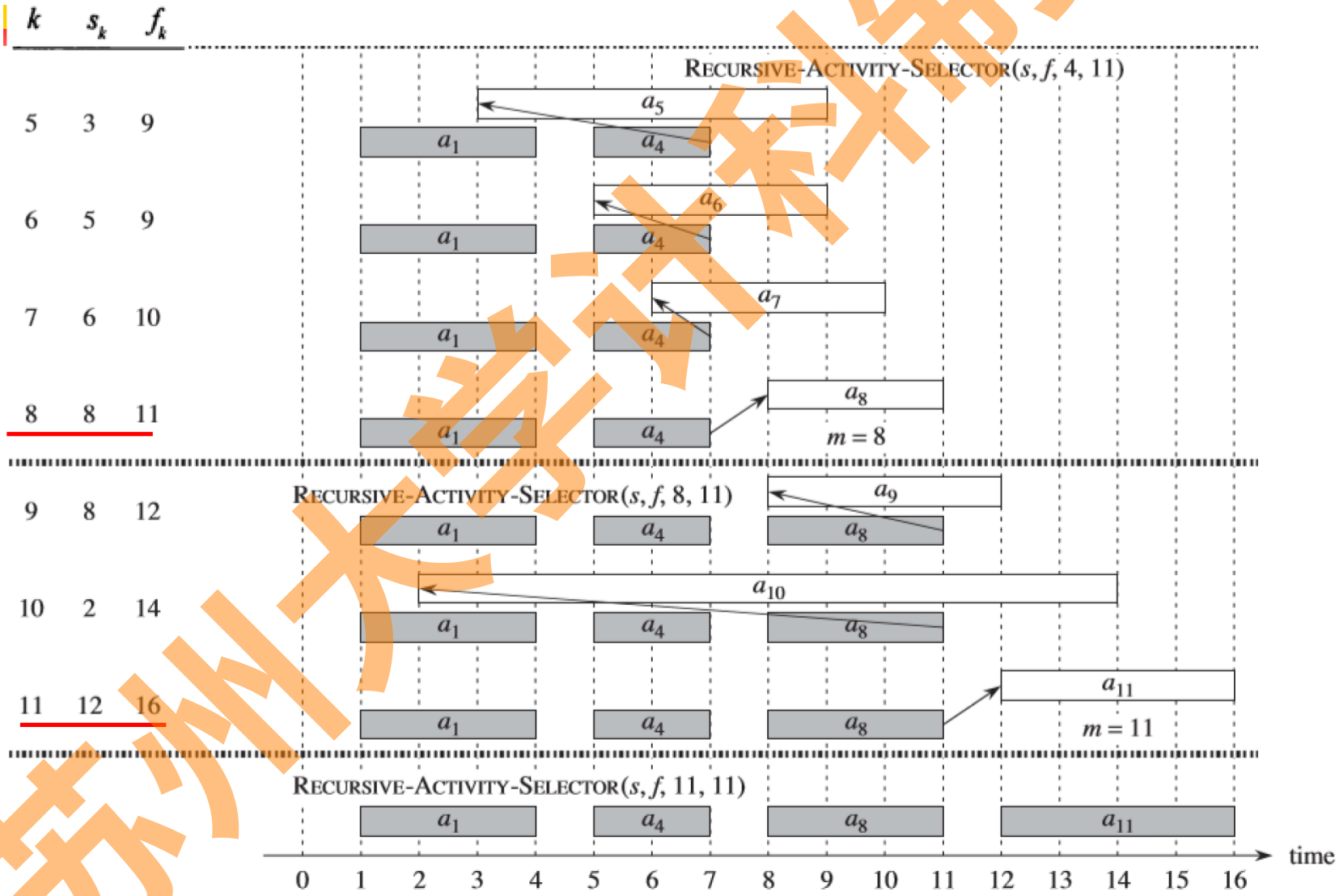
结束时间递增

执行过程如图所示：

注：为了处理的方便，这里引入一个虚拟活动 a_0，其结束时间 $f_0 = 0$



活动选择问题



活动选择问题

- 例：

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

结束时间递增

- 假定输入的活动已按结束时间的递增顺序排列，贪心算法只需 $O(n)$ 的时间即可选择出来 n 个活动的最大兼容活动集合。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
- 如果所给出的活动未按非减序排列，可以用 $O(n\lg n)$ 的时间进行排序

活动选择问题

- **迭代实现的贪心算法**
  - 上述 RECURSIVE-ACTIVITY-SELECTOR 是一个“尾递归”过程：以一个对自身的递归调用再接一次并集操作结尾，可以很容易地转化为迭代形式
  - 假定活动已经按照结束时间单调递增的顺序排列好

```
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
```

集合 A 用于保存选出的活动

变量 k 对应最后一个加入 A 的活动的下标

f_k 是 A 中活动的最大结束时间

for 循环查找 S_k 中最早结束的活动，若 a_m 与之前选出的活动兼容，即开始时间 s_m 大于 f_k，则将 a_m 加入 A

算法的运行时间是 $O(n)$







第十二讲 贪心算法

内容提要：

- 贪心算法思想
- 活动选择问题
- 贪心算法原理

贪心算法原理

- **贪心算法通过做出一系列选择来求问题的最优解**

——即**贪心选择**：在每个决策点，它做出在当前看来是最佳的选择

  - 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，相比动态规划算法，贪心算法简单和直接得多

- **贪心算法通常采用自顶向下的设计：做出一个贪心选择，并且这个选择将原问题简化为唯一的一个规模更小的子问题**

贪心算法原理

- 贪心算法设计的一般步骤：

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 证明贪心选择总是安全的：需要证明贪心选择（结束时间最早的活动）一定在某个最优解中

③ 证明作出贪心选择后，剩余的子问题满足：其最优子解与前面的贪心选择组合即可得到原问题最优解 (具有最优子结构)

贪心算法原理

- 对应每个贪心算法，都有一个动态规划算法，但动态规划算法要繁琐的多
- 如何证明一个最优化问题适合用贪心算法求解?
  - 没有适合所有情况的方法
  - **贪心选择性质**和**最优子结构**是两个关键要素

贪心算法原理

- **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
  - 贪心选择性质使得我们进行选择时，只需做出当前看起来最优的选择，然后递归地求解剩下的那个子问题，而不需要去考虑其他选择对应的子问题，也不需要比较多种选择

贪心算法原理

- **最优子结构：一个问题的最优解包含其子问题的最优解**
  - 最优子结构是能否应用动态规划和贪心方法的关键要素
- **贪心算法更为直接地使用最优子结构：**
  - 通过对原问题应用贪心选择后即可得到唯一子问题：将子问题的最优解与贪心选择组合在一起就能组合成原问题的最优解
  - 贪心策略的最优子结构表现为**线性结构**：原问题的最优解 = 贪心选择 + 唯一子问题的最优解
  - 而动态规划的最优子结构表现为**树状或网状结构**：原问题的最优解需要从多个子问题的最优解中组合得到
  - 贪心算法之所以高效，正是因为它利用问题特性将复杂的子问题结构简化为线性结构

贪心算法原理

- **贪心策略 VS 动态规划策略：**
  - 在动态规划方法中，每个步骤都需要进行一次选择，但这种选择通常依赖于子问题的解，反过来决定当前选择。因此，通常以自底向上地方式求解动态规划问题，先求解较小的子问题，然后才能求解较大的子问题
  - 在贪心算法中，我们总是做出当前看来最佳的选择，然后求解剩下唯一的一个子问题。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子问题的解
  - 动态规划要先求解子问题才能进行第一次选择，贪心算法在进行第一次选择之前不需要求解任何子问题
  - 动态规划算法通常采用自底向上的方式完成计算，而贪心算法通常是自顶向下的，每一次选择，将给定问题实例转换成更小的问题

贪心算法原理

- **如何证明每次贪心选择能生成全局最优解?**
  - 必须证明每个步骤做出贪心选择能生成全局最优解
  - 通常首先考查某个子问题的最优解，然后**用贪心选择替换某个其它选择**来修改此解，从而得到一个相似的子问题
  - 通过预处理或使用合适的数据结构，使得每一步的贪心选择可以高效地完成，从而整体上提高算法的效率。具体来说，贪心算法通常需要反复做出当前最优选择，如果每次选择都需要遍历所有候选元素或者进行复杂的计算，那么算法可能会比较低效。通过预处理（如排序）或者使用高效的数据结构（如优先队列），可以快速地找到当前最优选择，使得贪心算法的整体复杂度降低

0-1 背包问题

- **0-1 背包问题问题描述：**
  - 一个正在抢劫商店的小偷发现了 n 个商品，第 i 个商品的重量是 w_i 磅，其价值为 v_i 美元， w_i 和 v_i 都是整数。小偷的背包最多容纳 W 磅重的商品， W 是一个整数
  - 小偷应如何选择装入背包的商品，使得装入背包中商品的总价值最大?
  - 小偷在选择装入背包的商品时，对每种商品 i 只有 2 种选择，要么完整拿走，要么把它留下；不能将商品 i 装入背包多次，也不能只装入部分的商品 i

0-1 背包问题

- **0-1 背包问题 Knap(1, n, W) 的形式化定义：**
  - 给定 $W>0$，$w_i>0$，$v_i>0$，$1\le i\le n$，求 n 元 0-1 向量 $(x_1, x_2, \ldots, x_n)$，使得：

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \leqslant W \\ x_i \in \{0,1\},\ 1 \leqslant i \leqslant n \\ w_i>0,\ v_i>0,\ W>0,\ \text{且均为整数},\ 1 \leqslant i \leqslant n \end{cases}$$

  - 例如：$w = (w_1, w_2, w_3) = (2, 3, 4)$，$v = (v_1, v_2, v_3) = (1, 2, 5)$，求 Knap(1, 3, 6)
  - 取 $x = (1, 0, 1)$，Knap(1, 3, 6) = $(v_1x_1+v_2x_2+v_3x_3)$ = 1*1 + 2*0 + 5*1 = 6 最大

0-1 背包问题

- **穷举法求解 0-1 背包问题 Knap(1, n, W)：**
  - 遍历所有可能的 2^n 种物品选择方案，在每种方案下计算其总价值和重量，最后返回不超过背包承重 W 的最大价值

```
function Knapsack(n, W, w[], v[]):
    max_value = 0
    for each subset S of {1, 2, ..., n}:  // 穷举所有可能的物品组合
        weight = 0
        value = 0
        for each item i in subset S:
            weight = weight + w[i]
            value = value + v[i]
        if weight <= W:
            max_value = max(max_value, value)
    return max_value
```

0-1 背包问题

- **穷举法求解 0-1 背包问题的时间和空间复杂度：**
  - 穷举算法会遍历所有的可能组合，物品数为 n，每个物品有两种选择（选或不选），因此共有 2^n 种组合
  - 对于每种组合，检查它是否符合背包的最大承重要求以及计算该组合的总价值的时间复杂度为 $O(n)$
  - 因此，整体时间复杂度为 $O(n \cdot 2^n)$
  - 空间复杂度为 $O(1)$

0-1 背包问题

- **0-1 背包问题子问题定义：**
  - 设所给 0-1 背包问题的子问题记为 Knap(1，i，w)，$w \le W$ （假设 W，w_i 取整数），其定义为：

$$\max \sum_{k=1}^{i} v_k x_k$$

$$\begin{cases} \sum_{k=1}^{i} w_k x_k \le w \\ x_k \in \{0,1\},\ 1 \le k \le i \\ w_k > 0,\ v_k > 0,\ j > 0,\ 1 \le k \le i \end{cases}$$

  - 设最优值（最大价值）为 $f(i, w)$，即 $f(i, w)$ 是背包最大容量为 w，可选物品为 $\{1, 2, \ldots, i\}$ 的 0-1 背包问题的最优值

0-1 背包问题

- **0-1 背包问题 Knap(1, n, W) 具有最优子结构：**
  - **剪切-粘贴证明：**设 $(x_1, x_2, \ldots, x_n)$ 是 Knap(1, n, W) 的一个最优解，下面证明 $(x_1, \ldots, x_{n-1})$ 是 Knap(1, n-1, W-$w_n x_n$) 子问题的一个最优解

    若不然，设 $(y_1, \ldots, y_{n-1})$ 是 Knap(1, n-1, W-$w_n x_n$) 的最优解，因此有：

$$\sum_{i=1}^{n-1} v_i y_i > \sum_{i=1}^{n-1} v_i x_i \text{ 且 } \sum_{i=1}^{n-1} w_i y_i \leqslant W - w_n x_n$$

$$\Longrightarrow v_n x_n + \sum_{i=1}^{n-1} v_i y_i > \sum_{i=1}^{n} v_i x_i \text{ 又有 } w_n x_n + \sum_{i=1}^{n-1} w_i y_i \leqslant W$$

则说明 $(y_1, y_2, \ldots, y_{n-1}, x_n)$ 是 Knap(1, n, W) 的一个更优解，矛盾

0-1 背包问题

- **最优值的递归式如下：**
  - 由最优子结构性质，可以计算出 $f(i, w)$ 的递归式如下：

$$f(i,\ w)=\begin{cases}\max\{f(i-1,\ w),\ f(i-1,\ w-w_i)+v_i\} & w\geqslant w_i\\ f(i-1,\ w) & 0\leqslant w<w_i\end{cases}$$

  - 面对每个物品，只有拿或者不拿两种选择。$f(i,\ w)$ 的上述递归关系解释如下：
    - **不能选择第 i 个物品**：如果放入第 i 个物品，背包的容量 $w<w_i$，相当于放入物品 i 是不合法的，此时问题转化为前 i-1 个物品和背包最大承重为 w 的问题，$f(i, w)$ 得到的最大价值与 $f(i\text{-}1, w)$ 相等

0-1 背包问题

- **最优值的递归式如下：**
  - 由最优子结构性质，可以计算出 $f(i, j)$ 的递归式如下：

$$f(i,\ w)=\begin{cases}\max\{f(i-1,\ w),\ f(i-1,\ w-w_i)+v_i\} & w\geqslant w_i\\ f(i-1,\ w) & 0\leqslant w<w_i\end{cases}$$

    - **可以放入第 i 个物品**：如果选择第 i 个物品，背包的容量仍能满足要求。此时，第 i 个物品有两种可能：选或者不选。如果选，背包的容量减小为 w-w_i，变为子问题 $f(i$-$1,\ w$-$w_i)$，则 $f(i,\ w)$ 的值为 $f(i$-$1,\ w$-$w_i)$+v_i。如果不选，背包的容量不变，值仍然为 $f(i$-$1, w)$
  - **边界条件：**
    - $f(0, w)=0$，对于所有 w，表示没有物品时，背包的最大价值为 0
    - $f(i, 0)=0$，对于所有 i，表示背包容量为 0 时，背包无法容纳任何物品

0-1 背包问题

- 0-1 背包问题的动态规划算法 Knapsack(*n*, *W*, *weights*[], *values*[]):

```
function Knapsack(n, W, weights, values):
    // 初始化dp数组
    dp = 2D array of size (n+1) x (W+1)
    for i from 0 to n:
        for w from 0 to W:
            if i == 0 or w == 0:
                dp[i][w] = 0    // 边界情况：没有物品或背包容量为0时最大价值为0
            else if weights[i] <= w:
                // 可以选择物品i
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i]] + values[i])
            else:
                // 不能选择物品i
                dp[i][w] = dp[i-1][w]
    return dp[n][W]    // 返回最大价值
```

0-1 背包问题

- **算法描述：**
  - 首先，声明一个大小为 $(n+1)\times(W+1)$ 的二维数组 dp 来存储子问题的解，其中 dp[i][w] 表示前 i 个物品，背包容量为 w 时的最大价值
  - 初始化边界条件，所有 dp[i][0] = 0 和 dp[0][w] = 0，即当没有物品或者背包容量为 0 时，价值为 0
  - 对于每个物品 i 和每个背包容量 w，有两种选择：
    - 不可以选择第 i 个物品，此时最大价值为 dp[i-1][w]
    - 可以选择第 i 个物品，此时最大价值为 $\max\{dp[i-1][w-w_i] + v_i, dp[i-1][w]\}$
  - 该背包问题的最大价值保存在 dp[n][W] 中

0-1 背包问题

- **动态规划算法求解 0-1 背包问题的时间和空间复杂度：**
  - 动态规划算法是一个双重嵌套循环，因此，算法的时间复杂度为：$O(n \cdot W)$，其中 n 是物品的数量，W 是背包的容量
  - 动态规划算法需要存储一个二维数组 dp[i][w]，因此空间复杂度为 $O(n \cdot W)$

0-1 背包问题

- **0-1 背包问题的穷举算法 vs 动态规划算法：**
  - **穷举算法**：
    - 优点：简单直观，可以用于较小的输入规模。不需要额外的空间，除了存储输入数据外，几乎不占用额外内存
    - 缺点：时间复杂度为 $O(n \cdot 2^n)$，随着物品数量的增加，计算时间急剧增加，因此在大规模问题中效率极低
  - **动态规划算法**：
    - 优点：动态规划的时间复杂度是 $O(n \times W)$，尤其是当背包的容量 W 不太大时，远低于穷举算法。对于较大规模的背包问题，动态规划无疑是更好的选择
    - 缺点：空间复杂度较高，尤其当背包容量 W 较大时，可能需要大量的内存

分数背包问题

- **分数背包问题问题描述：**
  - 设定与 0-1 背包问题类似，但对每个商品，小偷可以只拿走商品 i 的一部分，而不是只能做出二元（0 或 1）选择
  - 比如，可以将 0-1 背包问题中的商品想象为金锭，而分数背包问题中的商品更像金砂

分数背包问题

- **分数背包问题的形式化定义：**
  - 给定 $c>0$，$w_i>0$，$v_i>0$，$1\le i\le n$，求 n 元 $[0,1]$ 向量 $(x_1, x_2, \ldots, x_n)$，使得：

$$\max\sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le c & w_i>0 \\ 0\le x_i\le 1 & i=1,2,\ldots,n \\ v_i>0, w_i>0, c>0 & i=1,2,\ldots,n \end{cases}$$

x_i为装入物品i的比例

w_i为重量，c为背包容量

v_i为价值

v_i/w_i为价值率(单位重量价值)

  - **例子：n=3，c=20，v=(25, 24, 15)，w= (18, 15, 10)，价值率=(25/18, 8/5, 3/2)，列举 4 个可行解：**

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum v_i x_i$
①	(1/2, 1/3, $\frac{1}{4}$)	16.5	24.5
②	(1, 2/15, 0)	20	28.2
③	(0, 2/3, 1)	20	31
④	(0, 1, $\frac{1}{2}$)	20	31.5 (最优解)

分数背包问题

- **贪心策略设计：**
  - 策略1：按价值最大贪心，使目标函数增长最快

    按价值排序从高到低选择物品→②解（次最优解）
  - 策略2：按重量最小贪心，使背包重量增长最慢

    按重量排序从小到大选择物品→ ③解（次最优解）
  - 策略3：按价值率最大贪心，使单位重量价值增长最快

    按价值率排序从大到小选择物品→ ④解（最优）

分数背包问题

- **用贪心算法解分数背包问题的基本步骤：**
  - 首先计算每种商品价值率 v_i/w_i。然后，遵循贪心策略，小偷首先尽可能多地拿走价值率最高的商品。如果该商品已全部拿走而背包尚未满，继续尽可能多地拿走价值率第二高的商品，依此类推，直至达到背包重量上限 W
  - 因此，通过将商品按每磅价值排序，贪心算法的运行时间为 $O(n\lg n)$

分数背包问题

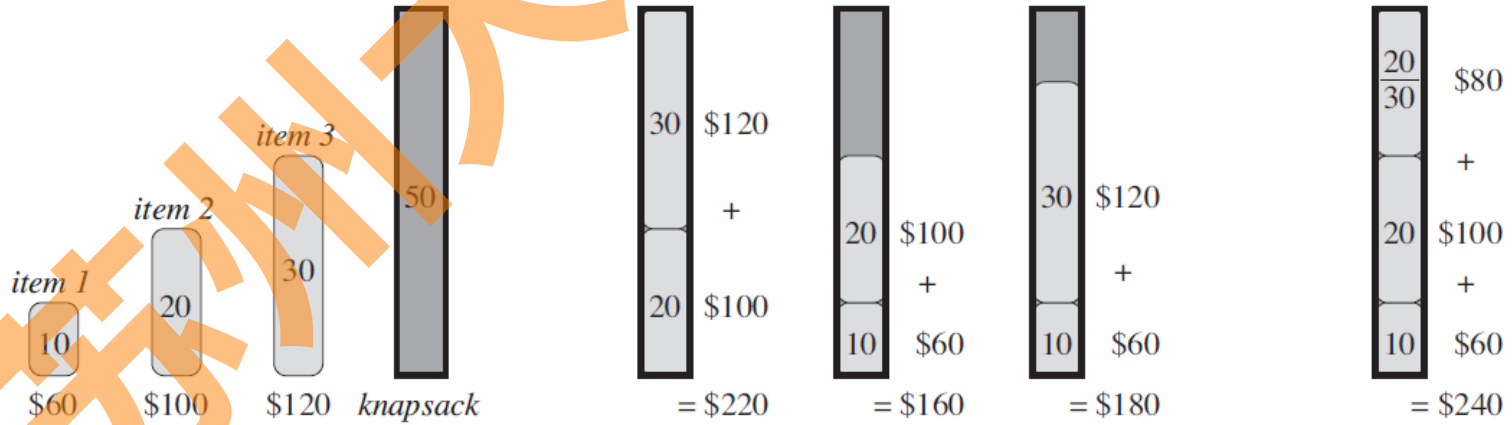
- 分数背包问题的贪心算法：

```
GreedyKnapsack(n, W, v[], w[], x[])
{   //按价值率最大贪心选择
    Sort(n, v, w);  //使得 v1/w1 ≥ v2/w2 ≥ … ≥ vn/wn
    for i = 1 to n do x[i]=0;
    for i = 1 to n do
    {
        if (w[i] > W) break;
        x[i]=1;
        W-=w[i];
    }
    if (i ≤ n) x[i] = W/w[i];   //使物品i是选择的最后一项
}
```

时间复杂度：$T(n) = O(n\lg n)$

0-1 背包问题

- 对于 0-1 背包问题，贪心策略是无效的
- 下图所给出的问题实例：商品 1 的每磅价值为 6 美元，商品 2 的每磅价值为 5 美元，商品 3 的每磅价值为 4 美元。对于 0-1 背包问题，按照上述贪心策略，首先拿走商品 1；而最优解为拿走商品 2 和商品 3，而留下商品 1
- 因此，贪心策略对于 0-1 背包问题之所以无效是因为在这种情况下，它无法保证最终能将背包装满，部分闲置背包空间使得每磅背包空间的价值降低了









谢 谢！