Chapter 16

Greedy Algorithms

贪心（贪婪）算法

songyou@buaa.edu.cn

16 Greedy Algorithms

- Similar to dynamic programming. Used for optimization problems.
- Optimization problems typically go through a sequence of steps, with a set of choices at each step.
  最优化问题：在求解过程每一步中，通常有多种选择，如何选择使得解最优?
- For many optimization problems, using dynamic programming to determine the best choices is overkill.
  对许多最优化问题，使用DP方法显得过犹不及（大材小用，杀鸡用牛刀）
- Greedy Algorithm: Simpler, more efficient.

16 Greedy Algorithms

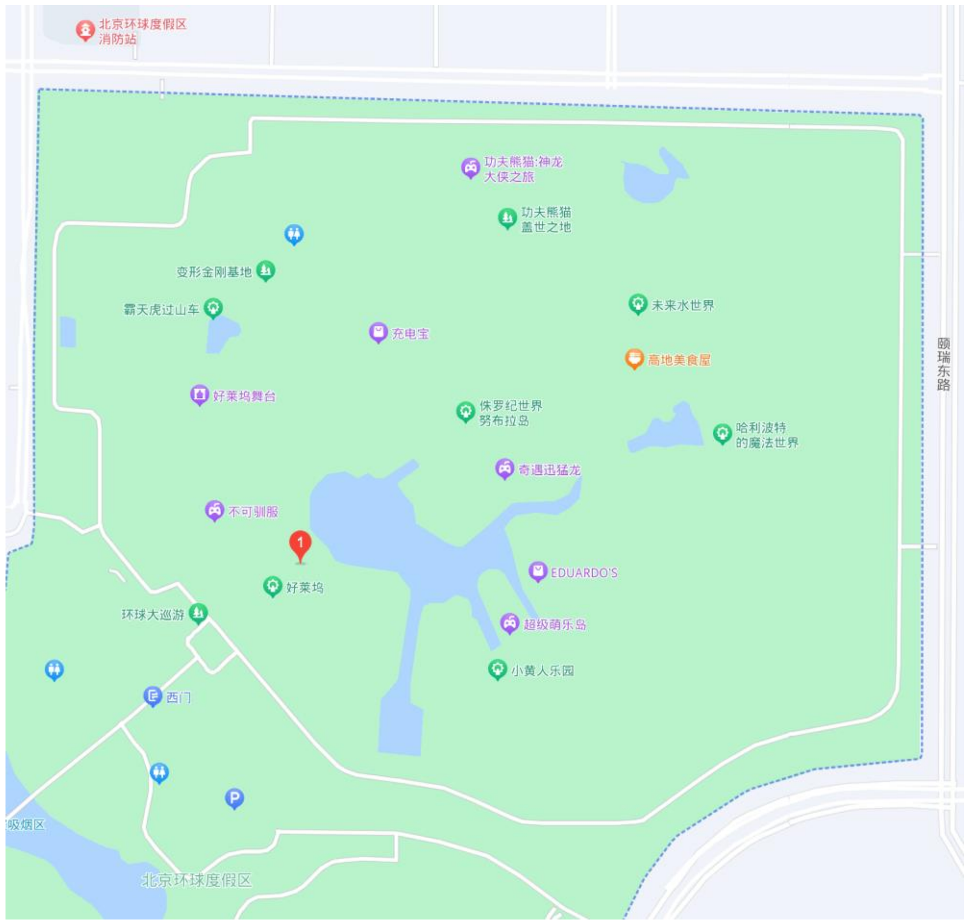
Greedy algorithms (GA) do not always yield optimal solutions, but for many problems they do.　贪心算法对一些最优化问题不一定能求出最优解，但对许多问题很实用。

- 16.1, the activity-selection problem （活动安排）
- 16.2, basic elements of the GA; knapsack prob （贪婪算法的基本特征；背包问题）
- 16.3, an important application: the design of data compression (Huffman) codes （哈夫曼编码）
- *16.4 Matroids and greedy methods
- *16.5, A task-scheduling problem as matroid (unit-time tasks scheduling, 有限期作业调度)

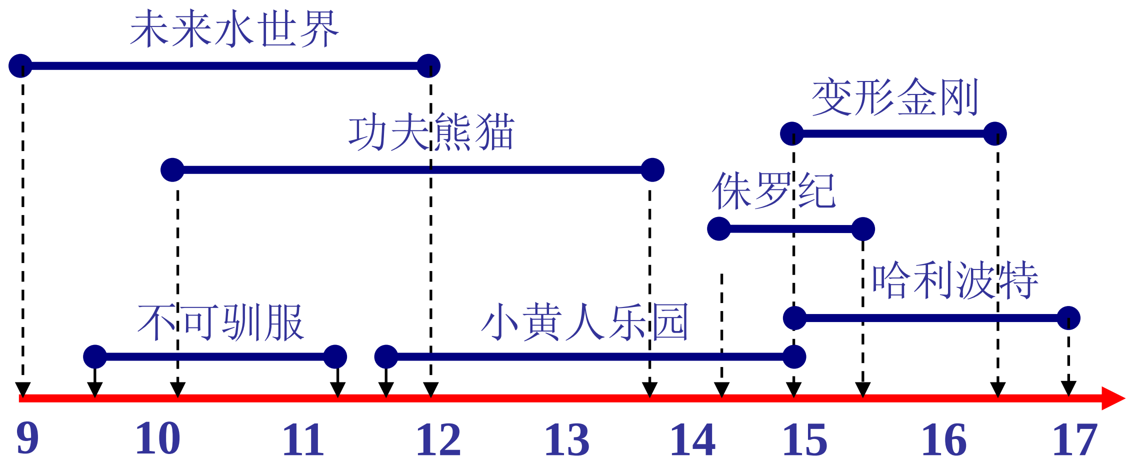
16 Greedy Algorithms

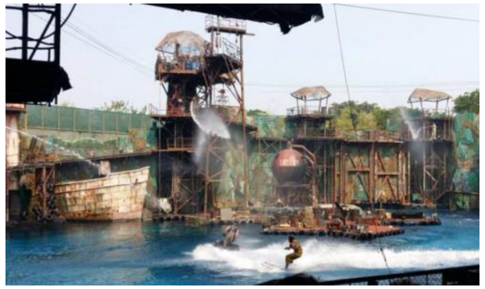
- minimum-spanning-tree algorithms (Chap 23)　（最小生成树）
- shortest paths from a single source (Chap 24)　（最短路径）
- set-covering heuristic (Chap 35)　（集合覆盖）
- …

Example: 环球度假区游玩的活动安排

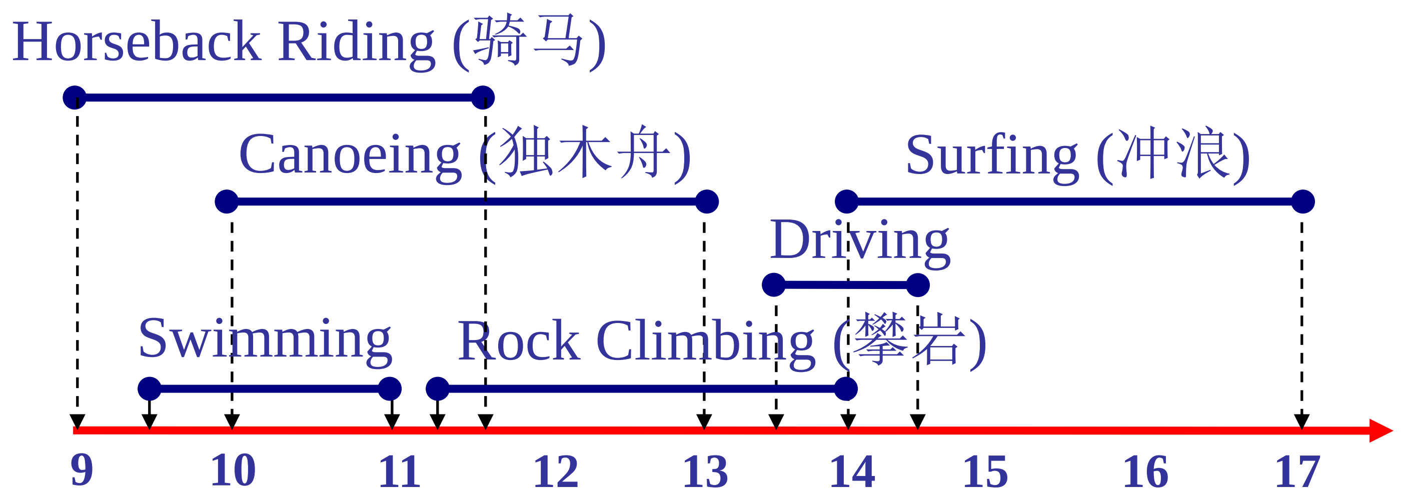


Universal Studio, Disneyland, 欢乐谷, …





Example: 活动安排 (Activity Selection)



How to make an arrangement to have the more activities? 如何安排活动，使得参与的活动数最多？

- S1. Shortest activity first （最短活动优先原则）

  Driving , Swimming
- S2. First starting activity first （最早开始活动优先原则）

  Horseback Riding , Driving
- S3. First finishing activity first （最早结束活动优先原则）

  Swimming , Rock Climbing , Surfing
- …

16.1 An activity-selection problem

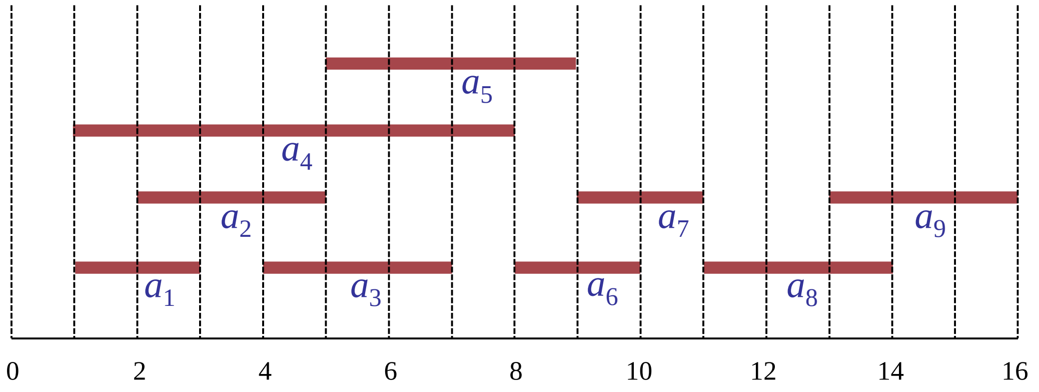
更多应用场景：借体育馆、借会议室、借教室、…

n activities require *exclusive* use of a common resource.

n 项活动，1个资源，任一活动进行时需唯一占用该资源

playground

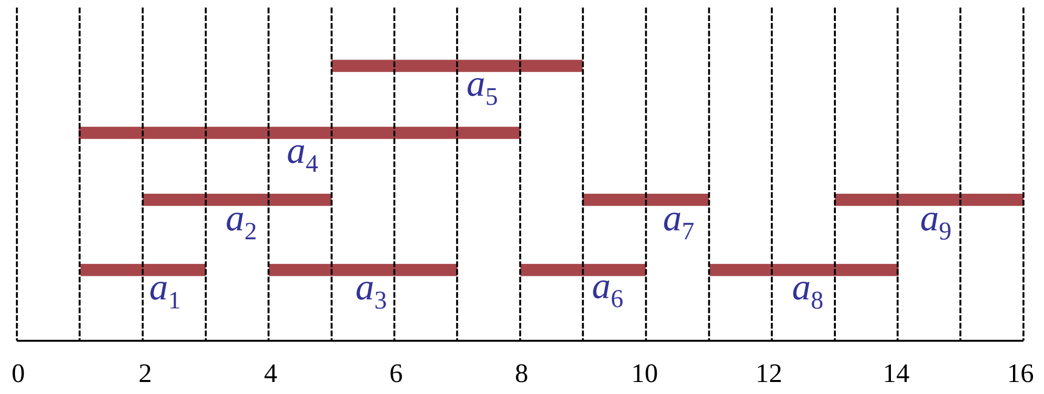
- Set of activities $S = \{a_1, a_2, \ldots, a_n\}$.
- a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i is start time and f_i is finish time.
- *Goal:* Select the largest possible set of nonoverlapping (*mutually compatible*) activities.
  安排一个活动计划，使得相容的活动数目最多
- Other objectives:
  Maximum duration time, 占用时间最长（利用率高）
  Maximize income rental fees, 租金收益最高
  …



16.1 An activity-selection problem

- *n activities* require *exclusive* use of a common resource.
  - Set of activities $S = \{a_1, a_2, \ldots, a_n\}$
  - a_i needs resource during period $[s_i, f_i)$
- *Example: S* sorted by finish time:

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



Maximum-size mutually compatible set:

$\{a_1, a_3, a_6, a_8\}$

Not unique: also

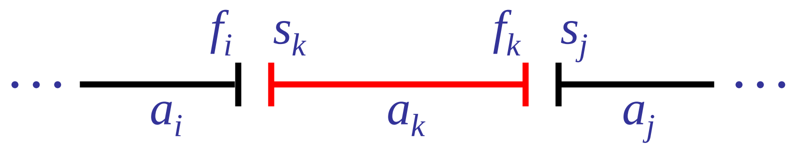
$\{a_2, a_5, a_7, a_9\}$

16.1.1 Optimal substructure of activity selection

Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ = activities that start after a_i finishes & finish before a_j starts

子问题空间S_{ij}：a_i 结束之后才开始，且 a_j 开始之前就已经结束的所有活动 a_k

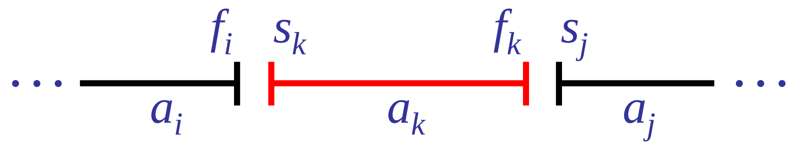


- To represent the entire problem, add fictitious activities:
  - $a_0 = [-\infty, 0)$, $\quad a_{n+1} = [\infty, \text{“}\infty+1\text{”})$
    第一个活动可能开始时间为0，最后一个活动可能结束时间很大，因此引入两虚拟活动 a_0 和 a_{n+1}
  - We don’t care about $-\infty$ in a_0 or “$\infty+1$” in a_{n+1}
- Then $S = S_{0,n+1}$. Range for S_{ij} is $0 \leq i, j \leq n+1$

16.1.1 Optimal substructure of activity selection

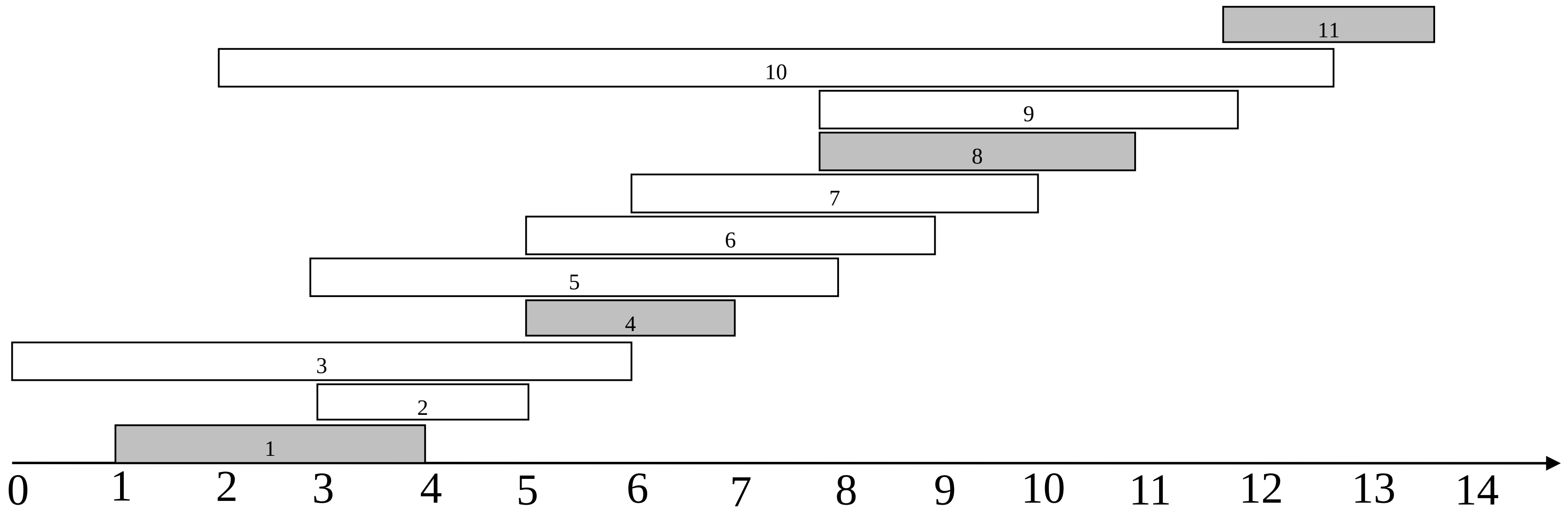
Space of subproblems

- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$



- Assume that activities are sorted by monotonically increasing finish time
  以结束时间单调增的方式对活动进行排序

$$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1} \text{ (if } i \le j \text{, then } f_i \le f_j) \qquad (16.1)$$



16.1.1 Optimal substructure of activity selection

以结束时间单调增的方式对活动进行排序

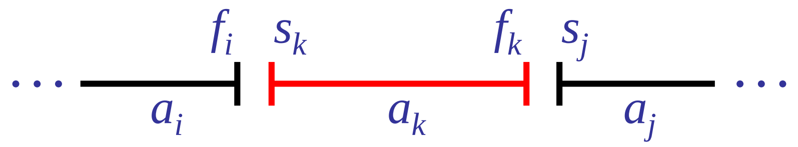
$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$ (if $i \le j$, then $f_i \le f_j$) (16.1)

- Then $i \ge j \Rightarrow S_{ij} = \varnothing$

  Proof If there exists $a_k \in S_{ij}$, then

  $f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j$.

  But $i \ge j \Rightarrow f_i \ge f_j$. Contradiction.

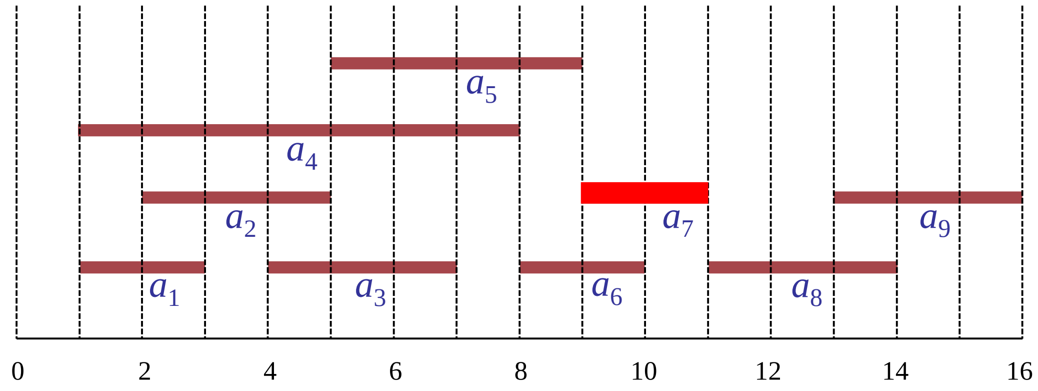


- So only need to worry about S_{ij} with $0 \le i < j \le n+1$. All other S_{ij} are $\varnothing$.

  因此，仅考虑 $0 \le i < j \le n+1$ 时的子问题空间 S_{ij} 的活动安排问题

16.1.1 Optimal substructure of activity selection

子问题空间S_{ij}：
a_i 结束之后才开始，且 a_j 开始之前就已经结束的所有活动 a_k



- 设问题 S_{ij} 的一个解中包括活动 a_k，则有两个子问题：S_{ik}和 S_{kj}
- 问题与子问题解的关系： (Solution to S_{ij}) = (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj}),
- 显然，解的数量的关系： | solution to S | = | solution to S_{ik} | + 1 + | solution to S_{kj} | .
- Optimal substructure, : If an optimal solution to S_{ij} includes a_k , then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well. (use usual cut-and-paste argument).
- Let A_{ij} = optimal solution to S_{ij} ， 令 A_{ij} 为问题 S_{ij} 的一个最优解（一种最优安排活动集合）

  so $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, (16.2)

16.1.2 A recursive solution

- 令 $c[i, j]$ 表示问题 S_{ij} 的一种最优安排的活动数量

$$i \geq j \Rightarrow S_{ij} = \varnothing \Rightarrow c[i, j\,] = 0.$$

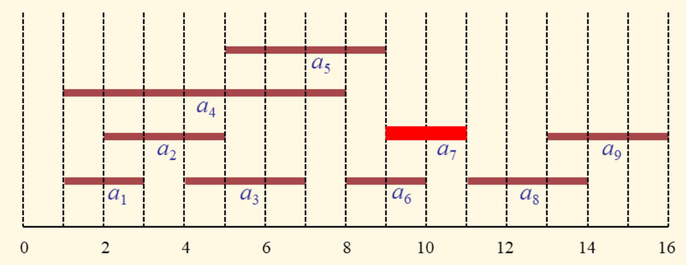
- 设 a_k 是问题 S_{ij} 的一种最优安排的某个活动，则

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

- k 未知，需要在 i 和 j 之间遍历

S_{ij} : 以结束时间单调增的方式对活动进行排序

$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}$ (if $i \leq j$, then $f_i \leq f_j$)



$$c[i, j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k, j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

根据定义 $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\} \Rightarrow a_k$ 不能为 a_i 或 a_j （若 $k = i$, 得 $c[i, j] = c[i, j] + 1$）.

核心要素：

1. 递归计算 $c[i, j]$，或表示为 $c(i, j)$
2. 遍历 $O(n)$ 次，找出 k

16.1.3 Converting a DP solution to a greedy solution

$$c[i,j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

- 基于递归方程（转移函数）(16.3)，可以设计算法.

  (1) Direct recursion algorithm (pseudo-code)? (2) Complexity?

  (3) Dynamic programming algorithm (pseudo-code)? (4) Complexity?

- For (16.3):

  (5) How many choices?

  (6) How many subproblems for a choice?

  (7) How many subproblems totally?

Exercise ?

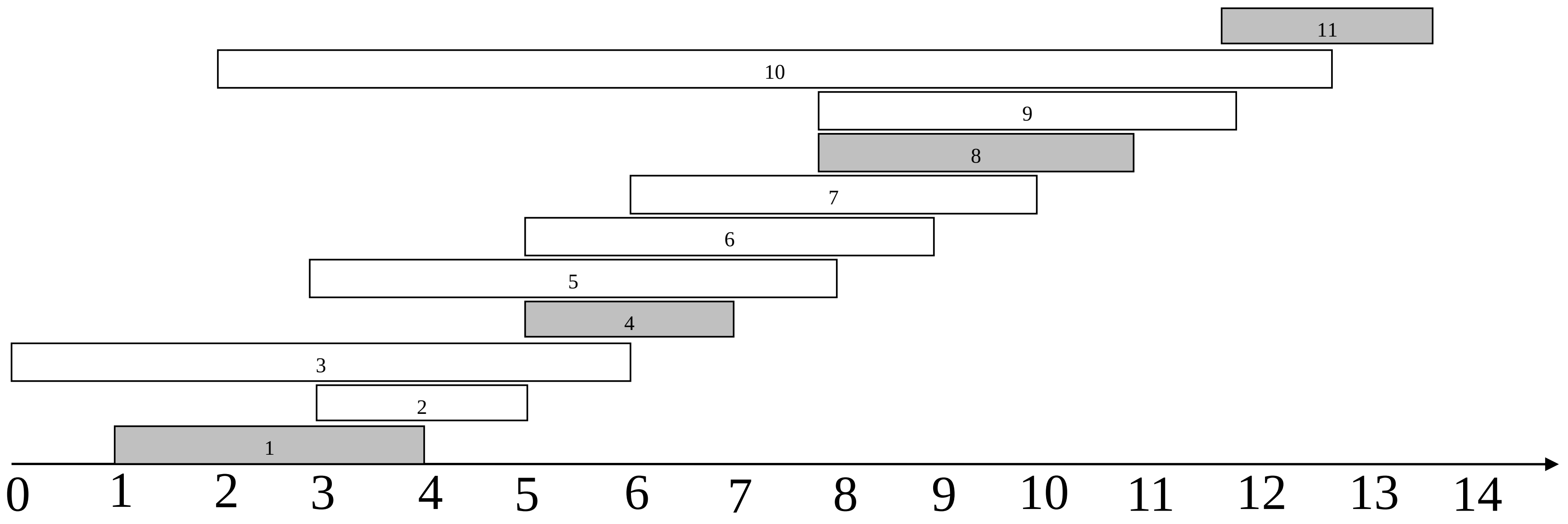
- Can we simplify our solution? 是否有更简单的求解方法?

16.1.3 Converting a DP solution to a greedy solution

Theorem 16.1

设 $S_{ij} \neq \varnothing$, 且 a_m 是 S_{ij} 的中具有最早结束时间的活动，即 $f_m = \min\{f_k : a_k \in S_{ij}\}$，则

1. a_m 包含在某个最大相容活动子集中（即是某个最优解的一个活动），
2. $S_{im} = \varnothing$，即，选择 a_m 后，仅剩下一个非空子问题 S_{mj} .



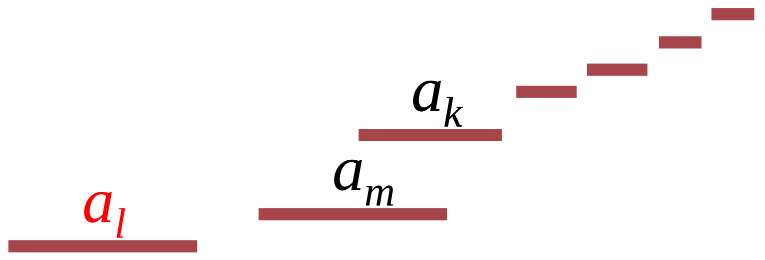
16.1.3 Converting a DP solution to a greedy solution

Theorem 16.1

设 $S_{ij} \neq \varnothing$, 且 a_m 是 S_{ij} 的中具有最早结束时间的活动，即 $f_m = \min\{f_k : a_k \in S_{ij}\}$，则

1. a_m 包含在某个最大相容活动子集中（即是某个最优解的一个活动），

2. $S_{im} = \varnothing$，即，选择 a_m 后，仅剩下一个非空子问题 S_{mj} .



Proof

2. Suppose there is some $a_l \in S_{im}$. Then $f_i \leq s_l < f_l \leq s_m < f_m$
$\Rightarrow f_l < f_m$. Then $a_l \in S_{ij}$ and it has an earlier finish time than f_m , which contradicts our choice of a_m . Therefore, there is no $a_l \in S_{im} \Rightarrow S_{im} = \varnothing$.

16.1.3 Converting a DP solution to a greedy solution

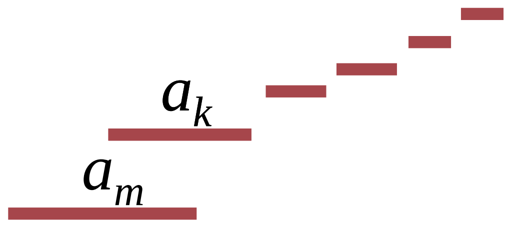
Theorem 16.1

设 $S_{ij} \neq \varnothing$, 且 a_m 是 S_{ij} 的中具有最早结束时间的活动，即 $f_m = \min\{f_k : a_k \in S_{ij}\}$，则

1. a_m 包含在某个最大相容活动子集中（即是某个最优解的一个活动），
2. $S_{im} = \varnothing$，即，选择 a_m 后，仅剩下一个非空子问题 S_{mj} .

Proof 1. Let A_{ij} be a maximum-size subset of mutually Compatible activities in S_{ij} . Order activities in A_{ij} in monotonically increasing order of finish time. Let a_k be the first activity in A_{ij} .

- If $a_k = a_m$, done (a_m is used in a maximum-size subset).



- Otherwise, construct $B_{ij}=A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m).

  Activities in B_{ij} are disjoint. (Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish. $f_m \leq f_k \Rightarrow a_m$ doesn’t overlap anything else in B_{ij}). Since $|B_{ij}| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is B_{ij} .

16.1.3 Converting a DP solution to a greedy solution

$$c[i,j] = \begin{cases} 0 & , \quad \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & , \quad \text{if } S_{ij} \neq \varnothing. \end{cases} \qquad (16.3)$$

Theorem 16.1

设 $S_{ij} \neq \varnothing$, 且 a_m 是 S_{ij} 的中具有最早结束时间的活动，即 $f_m = \min\{f_k : a_k \in S_{ij}\}$，则

1. a_m 包含在某个最大相容活动子集中（即是某个最优解的一个活动），
2. $S_{im} = \varnothing$，即，选择 a_m 后，仅剩下一个非空子问题 S_{mj} .

This theorem is great:

	before theorem	after theorem
# of sub-prob in optimal solution	2	1
# of choices to consider	$O(j-i-1)$	1

And, the choice is safe (a_m belongs to a solution)!

16.1.3 Converting a DP solution to a greedy solution

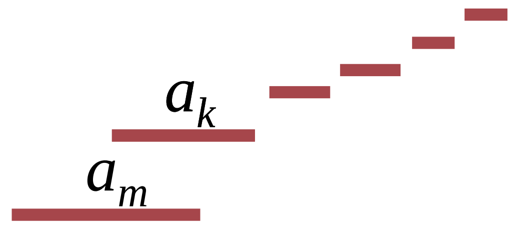
Theorem 16.1

设 $S_{ij} \neq \emptyset$, 且 a_m 是 S_{ij} 的中具有最早结束时间的活动，即 $f_m = \min\{f_k : a_k \in S_{ij}\}$，则

1. a_m 包含在某个最大相容活动子集中（即是某个最优解的一个活动），
2. $S_{im} = \emptyset$，即，选择 a_m 后，仅剩下一个非空子问题 S_{mj} .

按 top-down 的方式求解子问题 S_{ij} (DP的方法是 bottom-up)

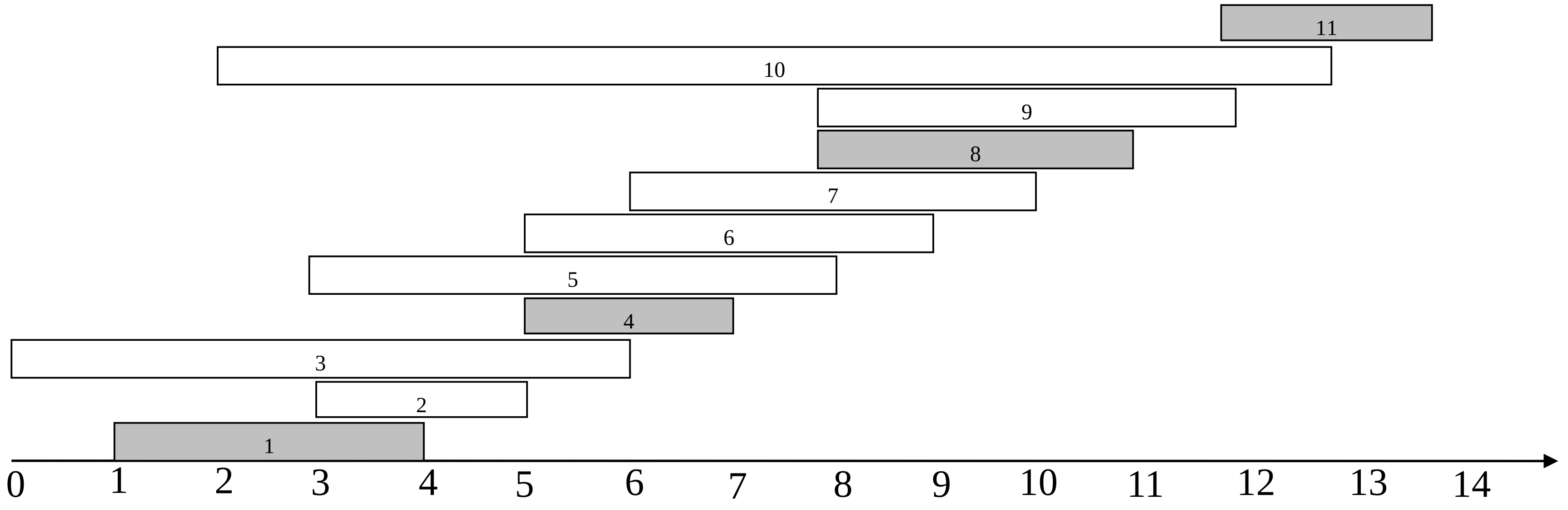
- 选择最早完成时间的活动 a_m，剩下的时间最长，自然可以安排更多的活动。这样的选择，称为贪心选择
- 求解剩下的子子问题 S_{mj}



16.1.3 Converting a DP solution to a greedy solution

所有的问题分析：

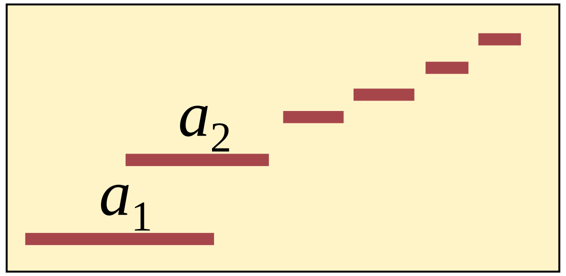
- 原问题 $S_{0,\,n+1}$ 〔 $a_0 = [\,-\infty, 0\,);\ a_{n+1} = [\,\infty, \text{“}\infty+1\text{”}\,)$ 〕
- 设第一个选择的活动是 a_{m1} （其实是a_1）
- 留下的子问题是 $S_{m1,\,n+1}$
- 下一个选择的活动是 a_{m2} （一定是 a_2 吗）
- 留下的下一个子问题 $S_{m2,\,n+1}$
- …



16.1.3 Converting a DP solution to a greedy solution

所有的问题分析：

- 原问题 $S_{0,\ n+1}$ 〔 $a_0 = [\ -\infty, 0\ )$; $a_{n+1} = [\ \infty, \text{“}\infty{+}1\text{”}\ )$ 〕
- 设第一个选择的活动是 a_{m1} （其实是a_1）
- 留下的子问题是 $S_{m1,\ n+1}$
- 下一个选择的活动是 a_{m2} （一定是 a_2 吗）
- 留下的下一个子问题 $S_{m2,\ n+1}$
- …

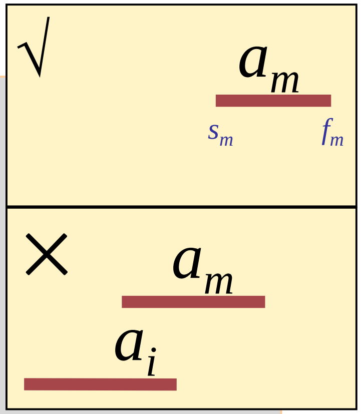


- 每一个子问题是 $S_{mi\ ,\ n+1}$
- 所选的子问题，其活动的完成时间是增序排列
- 因此，考虑每一个活动仅一次（选或不选该活动）

16.1.4 A recursive greedy algorithm

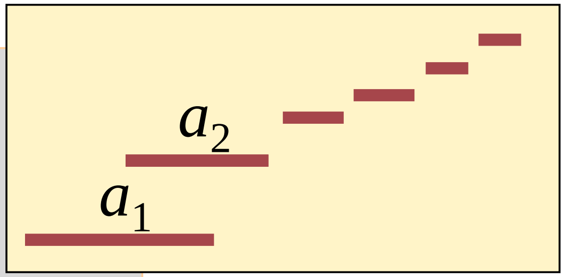
- 原问题 $S_{0, n+1}$
- 每一个子问题是 $S_{mi\,, n+1}$
- 所选的子问题，其活动的完成时间是增序排列（预处理时间$O(n\lg n)$）
- REC-ACTIVITY-SELECTOR(s, f, i, n) 求得 $S_{i, n+1}$ 最优解

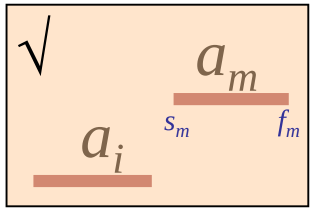
```
REC-ACTIVITY-SELECTOR($s, f, i, n$)
1  $m \leftarrow i+1$ // initially $i = 0$, $m = 1$
2  while $m \le n$ and $s_m < f_i$    // Find next activity in $S_{i, n+1}$.
3        $m \leftarrow m+1$
4  if $m \le n$
5        return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR($s, f, m, n$)
6  else  return $\varnothing$
```



16.1.4 A recursive greedy algorithm

```
REC-ACTIVITY-SELECTOR($s, f, i, n$)
1  $m \leftarrow i+1$ // initially $i = 0$, $m = 1$
2  while $m \leq n$ and $s_m < f_i$    // Find next activity in $S_{i, n+1}$.
3          $m \leftarrow m+1$
4  if $m \leq n$
5          return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR($s, f, m, n$)
6  else  return $\emptyset$
```





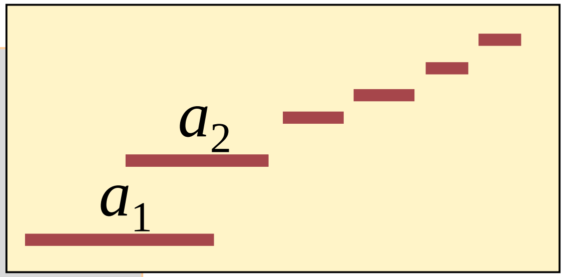
***Initial call*:** REC-ACTIVITY-SELECTOR($s, f, 0, n$).

Idea: The **while** loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m, n+1}$, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

16.1.4 A recursive greedy algorithm

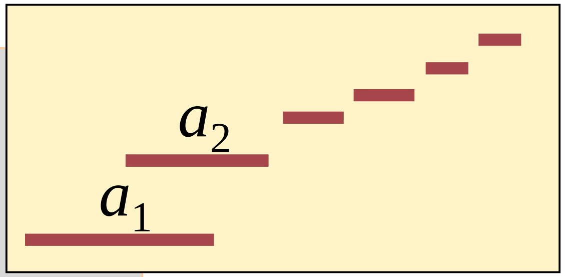
```
REC-ACTIVITY-SELECTOR($s$, $f$, $i$, $n$)
1  $m \leftarrow i+1$ // initially $i = 0$, $m = 1$
2  while $m \leq n$ and $s_m < f_i$    // Find next activity in $S_{i, n+1}$.
3          $m \leftarrow m+1$
4  if $m \leq n$
5          return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR($s$, $f$, $m$, $n$)
6  else  return $\varnothing$
```



Running Time?

16.1.4 A recursive greedy algorithm

```
REC-ACTIVITY-SELECTOR($s, f, i, n$)
1 $m \leftarrow i+1$ // initially $i = 0$, $m = 1$
2 while $m \le n$ and $s_m < f_i$    // Find next activity in $S_{i,\, n+1}$.
3         $m \leftarrow m+1$
4 if $m \le n$
5         return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR($s, f, m, n$)
6 else  return $\varnothing$
```



Running Time? $\Theta(n)$ — each activity examined exactly once.

$$T(n) = m_1 + T(n - m_1) = m_1 + m_2 + T(n - m_1 - m_2)$$
$$= m_1 + m_2 + m_3 + T(n - m_1 - m_2 - m_3) = \cdots$$
$$= \sum m_k + T(n - \sum m_k)$$

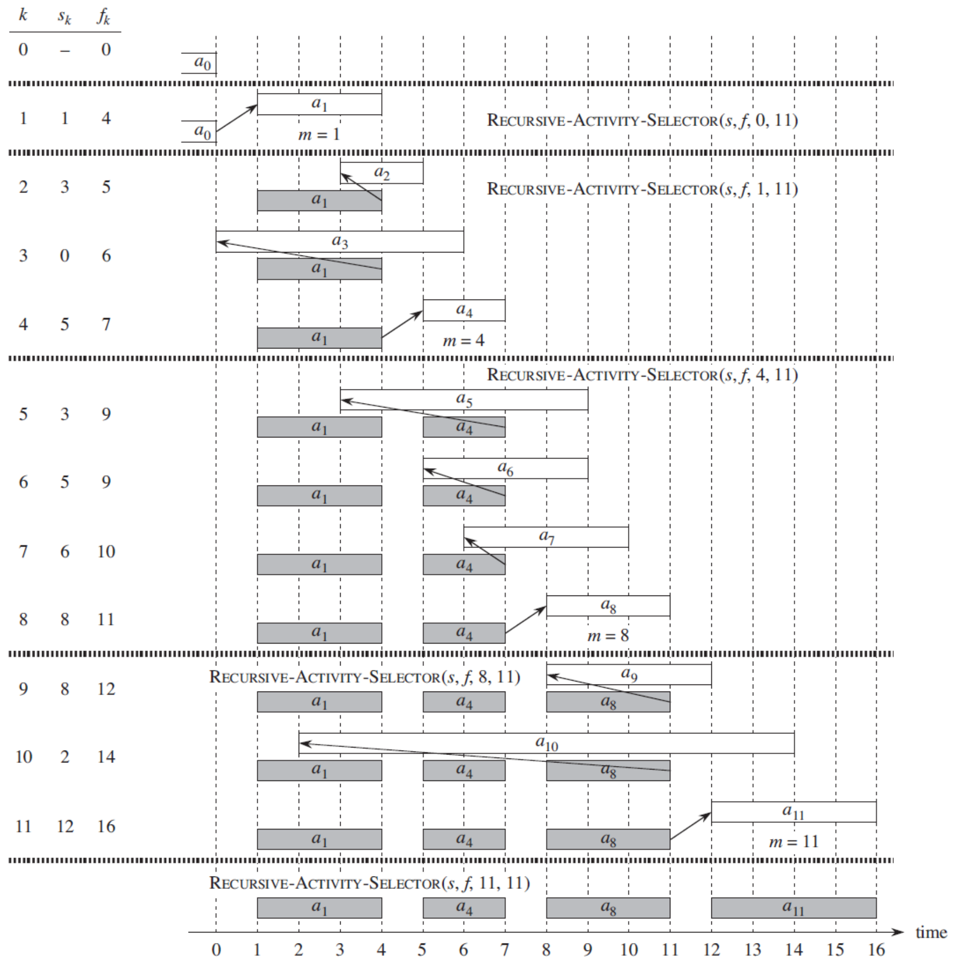
basecase: $n - \sum m_k = 1$, then $\sum m_k = n - 1$, $\sum m_k + T(1) = \Theta(n)$

16.1.4 A recursive greedy algorithm

***Initial call*:** REC-ACTIVITY-SELECTOR(s, f, 0, n).

Idea: The **while** loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

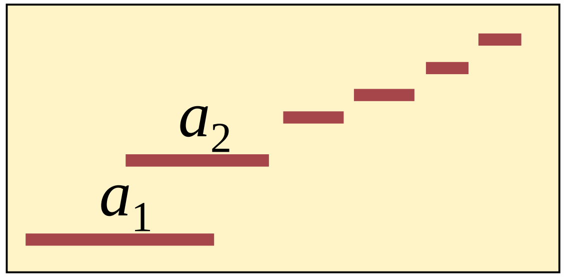


k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16

RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
$m = 1$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
$m = 4$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
$m = 8$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
$m = 11$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11}
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

16.1.5 An iterative greedy algorithm

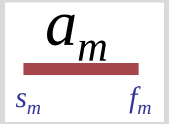
- REC-ACTIVITY-SELECTOR 几乎是一种尾递归 $T(n)=m_1+T(n-m_1)$
- 很方便转化为迭代实现（有的编译器能自动优化）

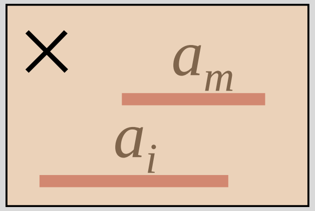


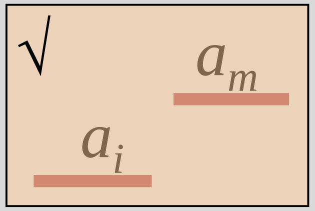
```
REC-ACTIVITY-SELECTOR($s, f, i, n$)
1  $m \leftarrow i+1$ // initially $i = 0$, $m = 1$
2  while $m \le n$ and $s_m < f_i$   // Find next activity in $S_{i,\,n+1}$.
3      $m \leftarrow m+1$
4  if $m \le n$
5      return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR($s, f, m, n$)
6  else return $\varnothing$
```

尾递归：在递归调用中，递归调用是函数中的最后一个操作，即递归调用的结果直接作为函数的返回值，且在递归调用之后没有额外的操作需要执行。

```
GREEDY-ACTIVITY-SELECTOR($s, f, n$)
1  $A \leftarrow \{a_1\}$
2  $i \leftarrow 1$
3  for $m \leftarrow 2$ to $n$
4      if $s_m \ge f_i$
5          $A \leftarrow A \cup \{a_m\}$
6          $i \leftarrow m$   // $a_i$ is most recent addition to $A$
7  return $A$
```







Review

- 贪心算法思想：当前选择看起来是最好的，就选他；当前的局部最优选择，能产生全局最优选择。

- 贪心算法优点：简单，高效。
- DP: 每个子问题都必须求解。
  $T(n) \approx$ 所有的子问题个数 * 求解每个子问题时所做的选择数。
- GA: 选择次数大量减少，需要求解的子问题也可能减少。

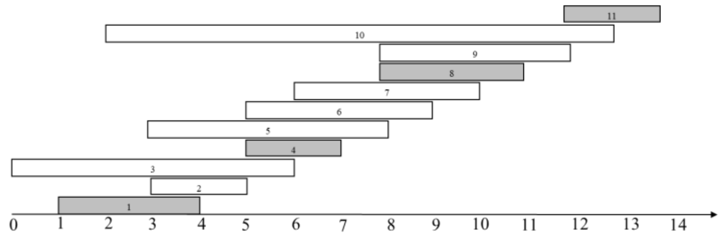
16 Greedy Algorithms

- 16.2,
  basic elements of the GA （贪婪算法的基本特征）
  knapsack prob （背包问题）

- 16.3, an important application: the design of data compression (Huffman) codes （哈夫曼编码）

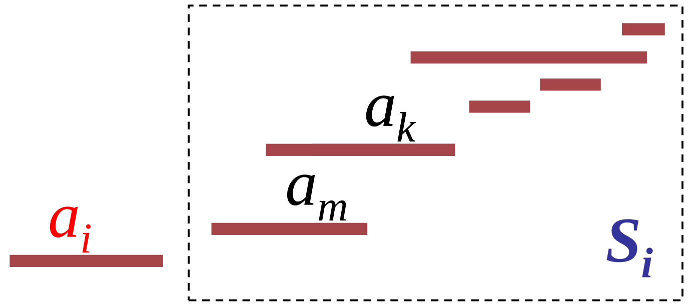
16.2 Elements of the greedy strategy

- 每次决策时，当前所做的选择看起来是“最好”的
- 活动安排问题的要素：
  1. 分析最优子结构特征
  2. 设计递归解（转移函数）
  3. 证明在每个低估时所做的选择是贪心
  4. 通过贪婪选择，只有一个子问题非空
  5. 设计递归算法
  6. 设计迭代算法



(Solution to S_{ij}) = (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj})

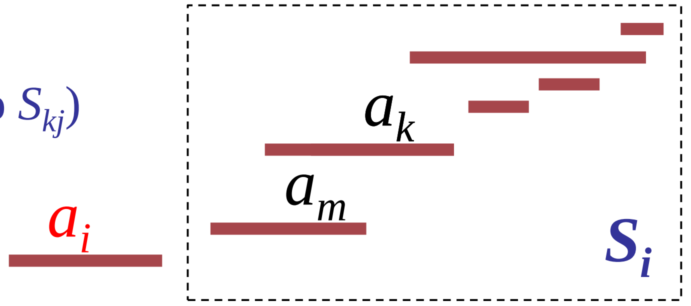
$$c[i,j]=\begin{cases}0 & \text{, if } S_{ij}=\emptyset,\\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & \text{, if } S_{ij}\neq\emptyset.\end{cases} \quad (16.3)$$



16.2 Elements of the greedy strategy

- 与DP方法的步骤相似

(Solution to S_{ij}) = (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj})

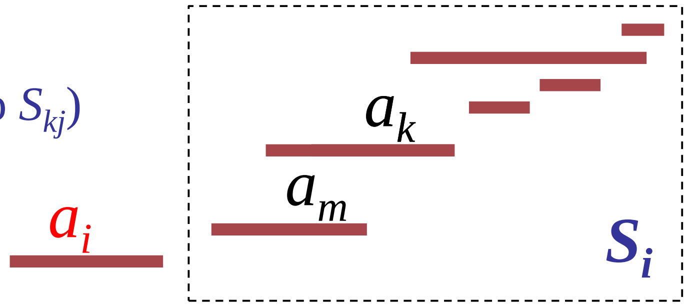


- 针对活动安排问题，做出贪心后，活动 S_{ij} 只有 i 变化，j 固定在 n+1
- 因此，子问题可以定义为
  - $S_i = \{a_k \in S : f_i \leq s_k\}$, 所有在 a_i 结束之后开始的活动
  - 贪心选择，选 S_i 中的活动 a_m
    求解子问题 S_m

$\left.\begin{array}{l}\text{贪心选择，选 } S_i \text{ 中的活动 } a_m \\ \text{求解子问题 } S_m\end{array}\right\} \Leftarrow$ 最优解 S_i

16.2 Elements of the greedy strategy

(Solution to S_{ij}) = (solution to S_{ik}) $\cup$ $\{a_k\}$ $\cup$ (solution to S_{kj})



几个关键步骤：

1. 做选择，留下一个待求的子问题
2. 贪心选择出来的部分是解的一部分，因此贪心选择是安全的
3. 贪婪选择 + 子问题的最优解 $\Leftarrow$ 原问题的最优解

16.2 Elements of the greedy strategy

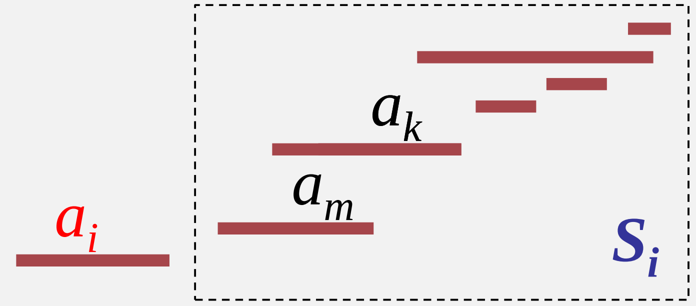
No general way to tell if a greedy algorithm is optimal, but two key ingredients.

没有一般化的规则来说明贪心算法是否最优，但有两个基本要点

1. greedy-choice property 贪心算法属性
2. optimal substructure 最优子结构

(Solution to S_{ij})

= (solution to S_{ik}) ∪ {a_k} ∪ (solution to S_{kj})

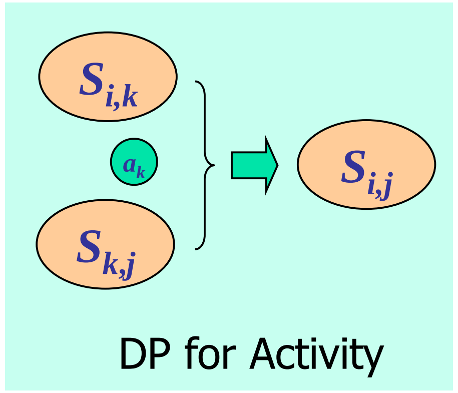


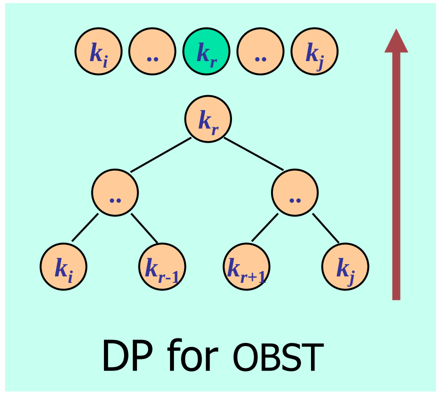
子问题的最优解 + 贪心选择 ⟸ 原问题的最优解

Greedy-choice property

Dynamic programming

- 依赖于已知子问题的最优解再作出选择（先解子问题）
- 求解步骤：bottom-up

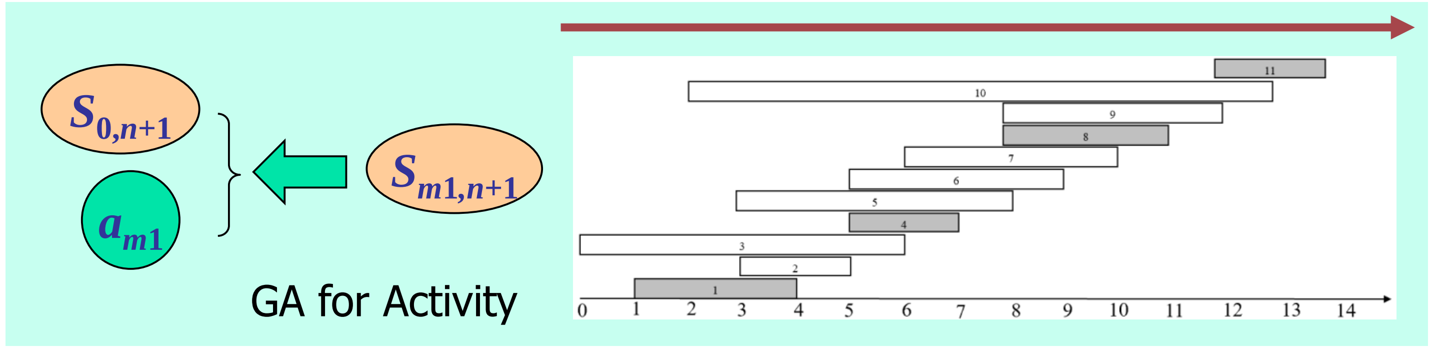




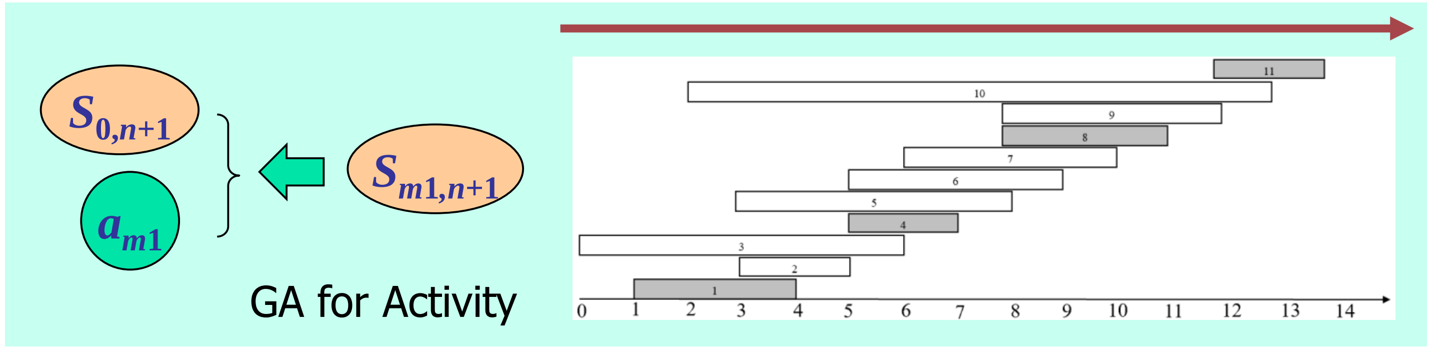
Greedy

- 直接选择，留更小的子问题进行求解
- 求解步骤：top-down

证明子问题的最优解和贪婪选择 ⇒ 原问题的最优解

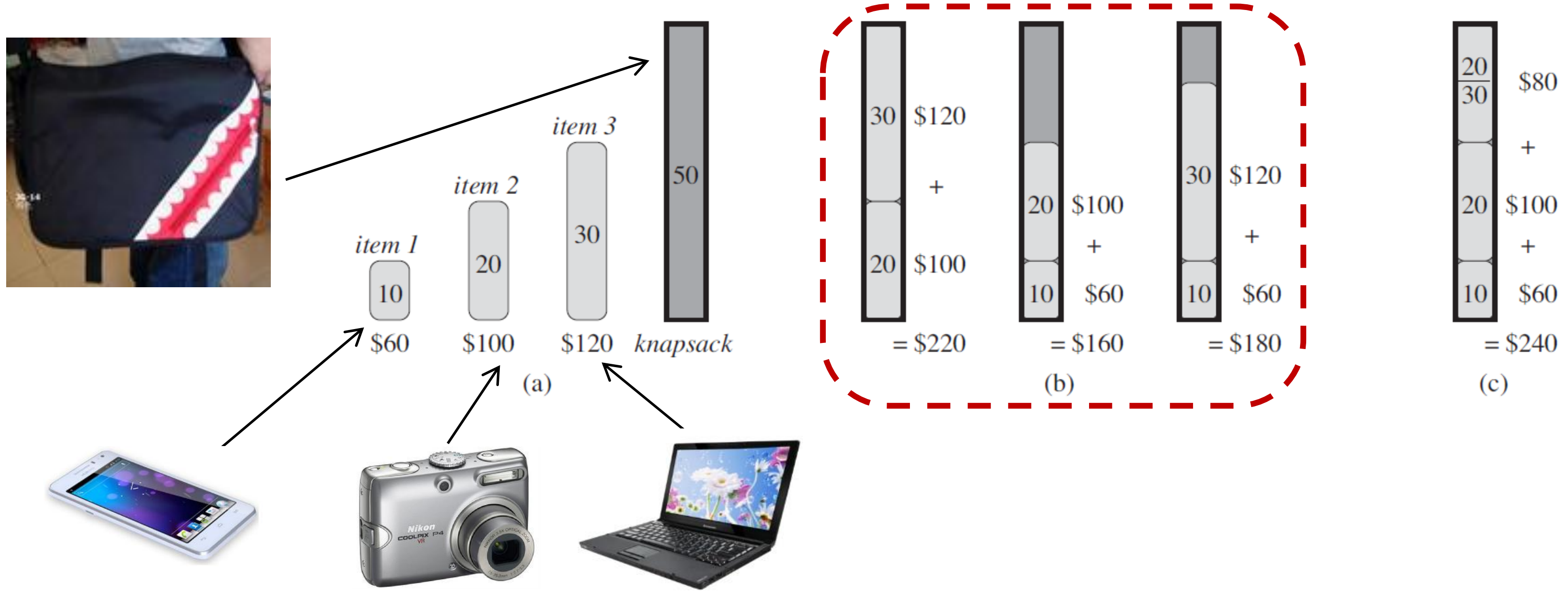


Greedy-choice property



- We must prove that a greedy choice at each step yields a globally optimal solution. Sometimes it’s Difficult! Cleverness may be required!
- Typically, Theorem 16.1, shows that the solution (A_{ij}) can be modified to use the greedy choice (a_m), resulting in one similar but smaller subproblem (A_{mj}).
- We can get efficiency gains from greedy-choice property. (For example, in activity-selection, examine each activity just once.)

knapsack: Greedy vs. dynamic programming

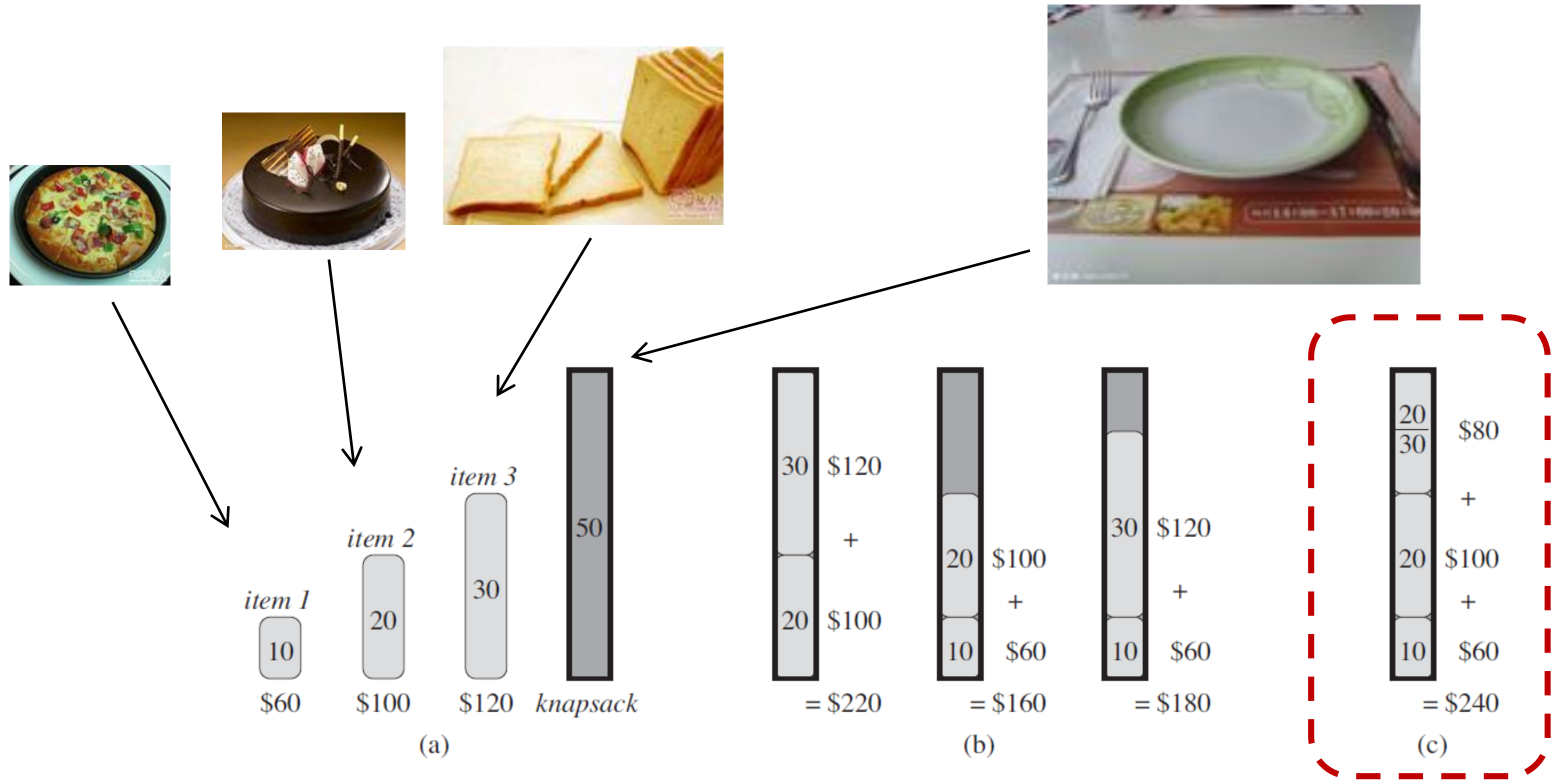


0-1 knapsack problem （0-1背包问题）

- n items, item i is worth \$$v_i$, weighs w_i P（物品 i 价值 v_i， 重 w_i）
- Find a most valuable subset of items with total weight $\leq W$. （不超过背包总负荷 W 的情况下，如何装物，使得总价值最大）
- Have to either take an item or not take it—can’t take part of it. （物品选取或不选，不能选一个物品的部分）

1 0

knapsack: Greedy vs. dynamic programming



Fractional knapsack problem **（分数背包问题，可以选择一个物品的部分）**

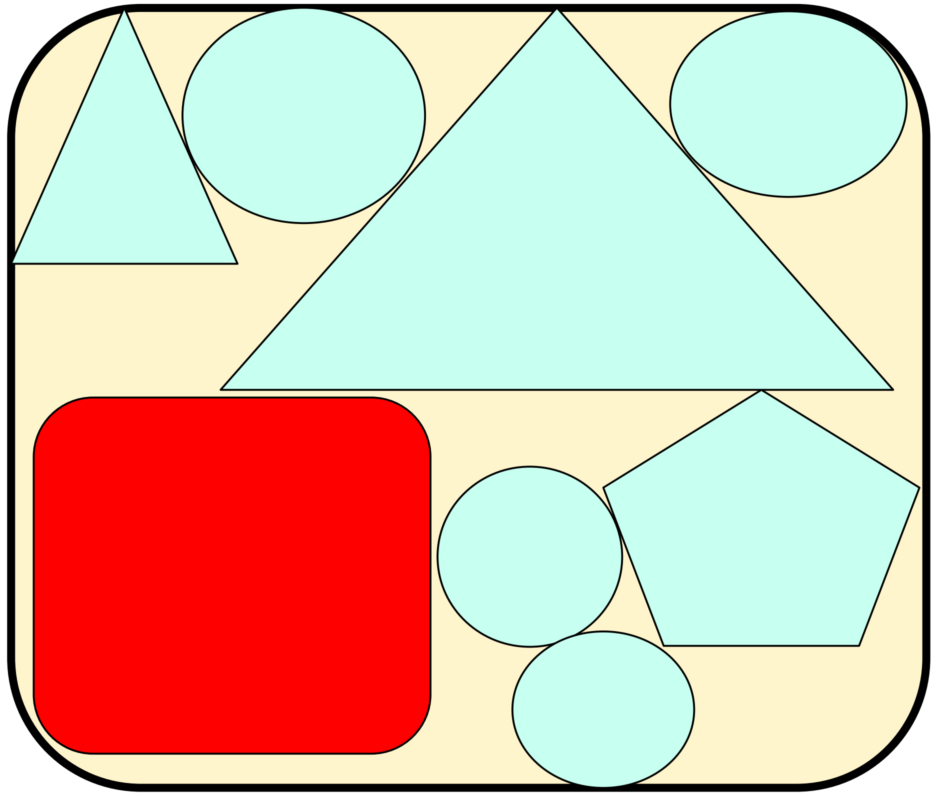
- Like the 0-1 knapsack problem, but can take fraction of an item.

knapsack: Greedy vs. dynamic programming

- 0-1 knapsack problem （0-1背包问题）
- Fractional knapsack problem （分数背包问题）

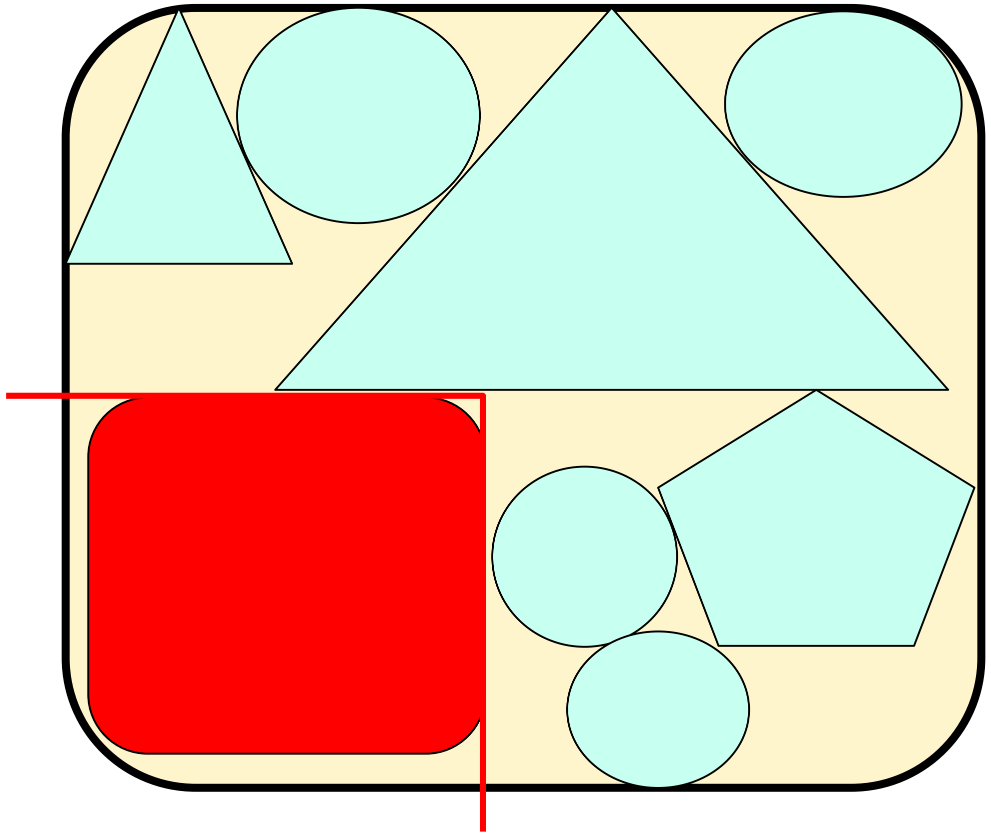
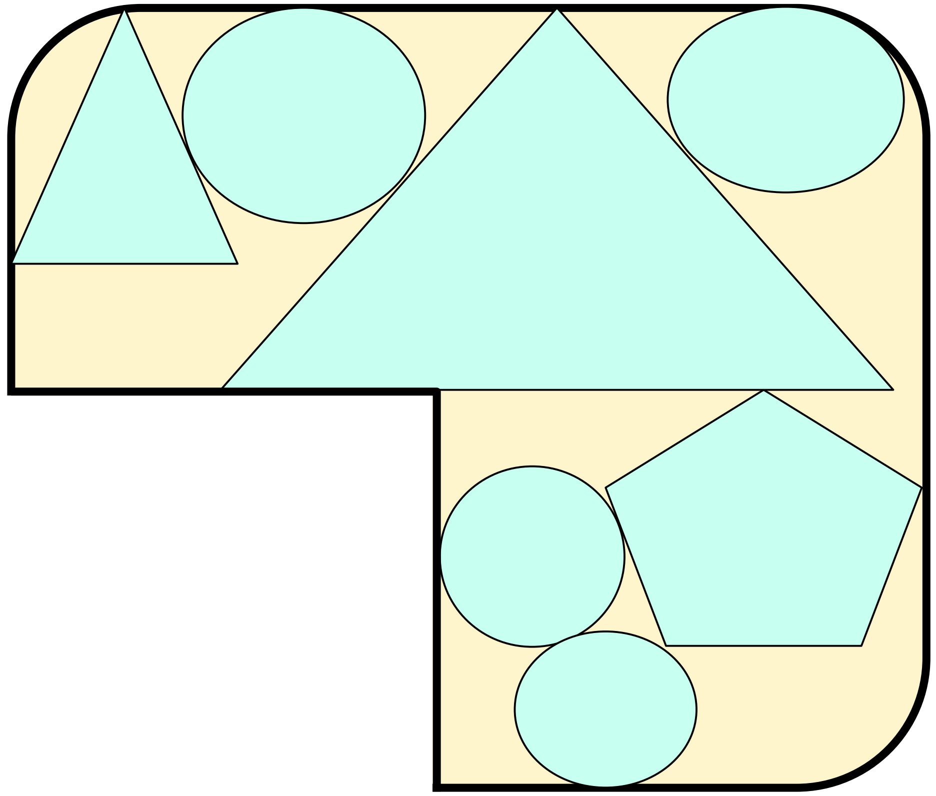
- Both have optimal substructure property
  - 0-1 : ?
  - fractional: ?

都有最优子结构性质，如何体现？



背包问题的最优子结构性质：

完整的圆角矩形框（这种装法）是一个最优背包。

背包问题的最优子结构性质：

完整的圆角矩形框（这种装法）是一个最优背包。

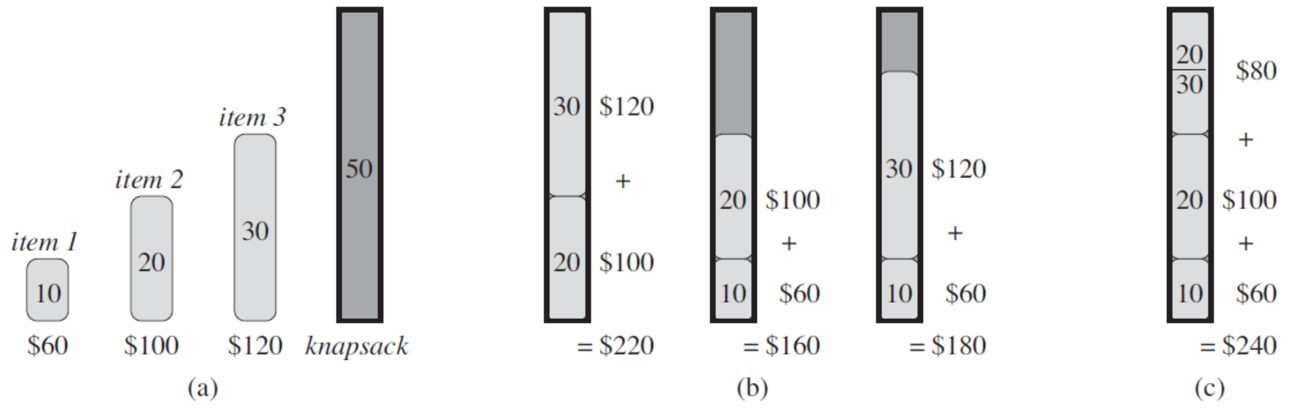
去掉左图的左下角的红色部分剩下的部分是一个子背包，则该子背包（这种装法）也是一个最优背包。

knapsack: Greedy vs. dynamic programming

- 经典问题： 0-1背包问题，分数背包问题
- 更多问题：背包九讲

- 分数背包问题有贪心选择属性，0-1背包问题则没有该性质

knapsack: Greedy vs. dynamic programming



- 分数背包问题有贪心选择属性
- 0-1背包问题 无贪心选择属性
- 求解分数背包问题，先按物品的单位价值 v_i/w_i 降序排序 $(v_i/w_i \geq v_{i+1}/w_{i+1}$ for all $i)$
- Time: 排序 $O(n\lg n)$, 贪心选择 $O(n)$

```
FRACTIONAL-KNAPSACK(v, w, W)
1  load ← 0
2  i ← 1
3  while load < W and i ≤ n
4      if w_i ≤ W - load
5          take all of item i
6          add what was taken to load
7          i ← i + 1
8      else
9          take W-load of w_i from item i
10         break
```

knapsack: Greedy vs. dynamic programming

0-1背包实例

- let W = 50 for the following example.
  Greedy solution:
  - take items 1 and 2
  - value = 160, weight = 30

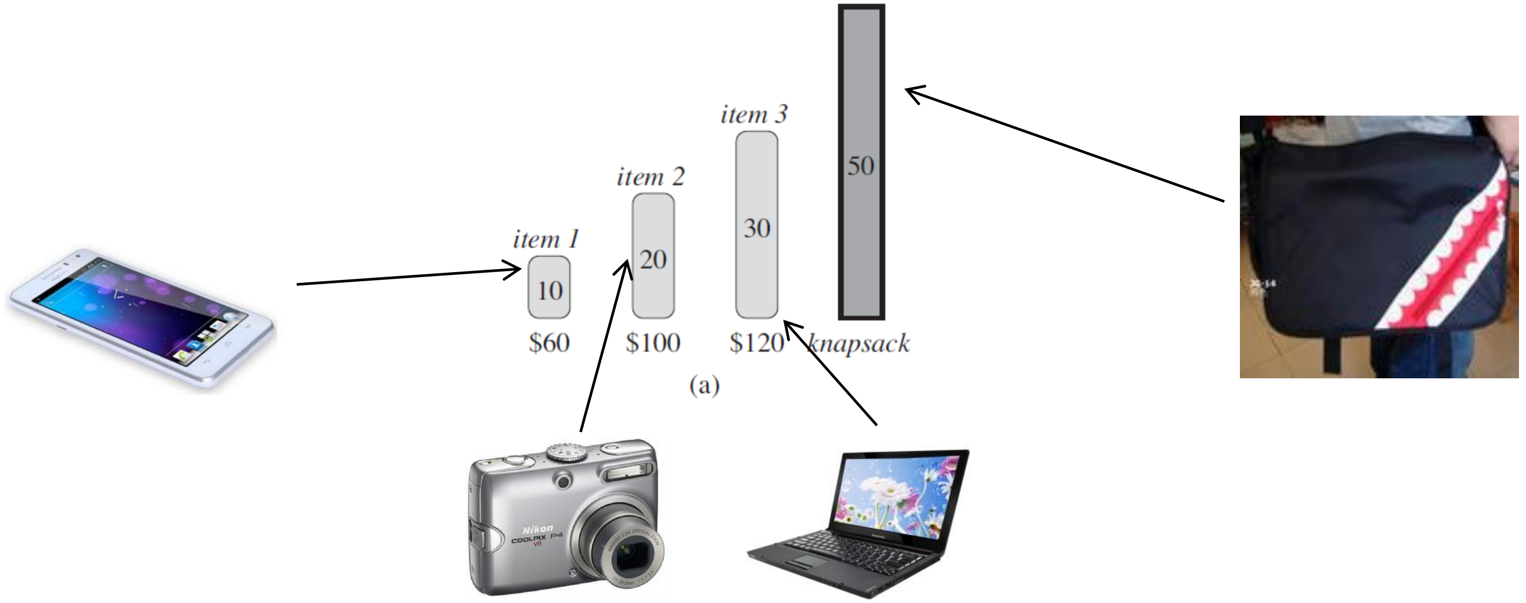
  20 pounds of capacity leftover.
  背包未装满，但不能再装了。

- Optimal solution:
  - Take items 2 and 3
  - value=220, weight=50

  No leftover capacity. 背包装满

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

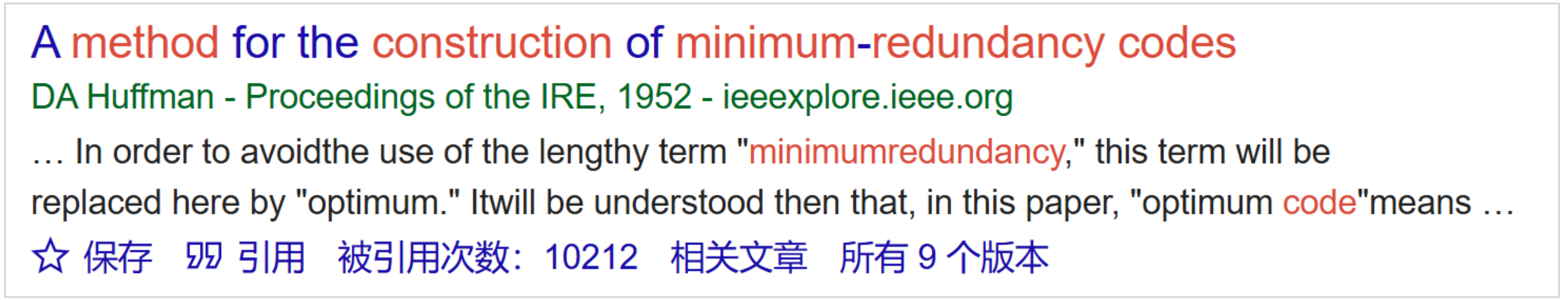
每单位重量的价值为6 元钱

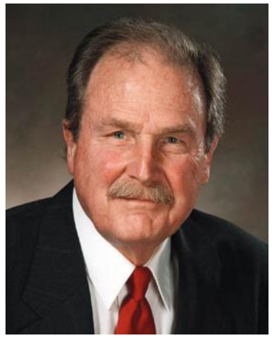


16.3 Huffman codes

An important application: the design of data compression.
Huffman codes, **哈夫曼编码**

1951年，哈夫曼在MIT读博士，导师Robert M. Fano给出一个学期报告的题目“寻找最有效的二进制编码”。哈夫曼发现了基于有序频率二叉树编码的想法，并证明了这个方法是最有效的。哈夫曼使用自底向上的方法构建二叉树，避免了次优算法Shannon-Fano编码的弊端——自顶向下构建树。1952年，哈夫曼发表了《一种构建极小多余编码的方法》，后人称之为Huffman编码。


A method for the construction of minimum-redundancy codes
DA Huffman - Proceedings of the IRE, 1952 - ieeexplore.ieee.org
… In order to avoidthe use of the lengthy term "minimumredundancy," this term will be replaced here by "optimum." Itwill be understood then that, in this paper, "optimum code"means …
☆ 保存 ﾂﾂ 引用 被引用次数：10212 相关文章 所有 9 个版本



戴维·霍夫曼（David Albert Huffman, 1925-1999)，1999年10月17日因癌症去世，享年74岁。但他作为信息论的先驱，对计算机科学、通信等学科所作出的巨大贡献将永远为世人所铭记。
霍夫曼在MIT一直工作到1967年。之后他转入加州大学Santa Cruz分校，是该校计算机科学系的创始人，1970—1973年任系主任。1994年霍夫曼退休。
除了霍夫曼编码外，霍夫曼在其他方面也还有不少创造，比如他设计的二叉最优搜索树算法就被认为是同类算法中效率最高的，因而被命名为霍夫曼算法。

16.3 Huffman codes

- Huffman codes: widely used and very effective technique for encoding file or compressing data. 哈夫曼编码：广泛使用且非常有效的编码文件、压缩数据的一种方法
  - savings of 20% to 90%

- Consider the data to be a sequence of characters，

  abaaaabbbdcffeaaaeaeec…abaadefe

- Huffman's greedy algorithm:

  uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

  哈夫曼贪心算法：依据字符出现的频率表，使用二进串来表示字符的一种最佳方法

作业：每人写一个压缩软件

16.3 Huffman codes

2002，无间道

不是摩尔斯电码 这是二进制
It's not Morse, Murph. It's binary.

2014，星际穿越

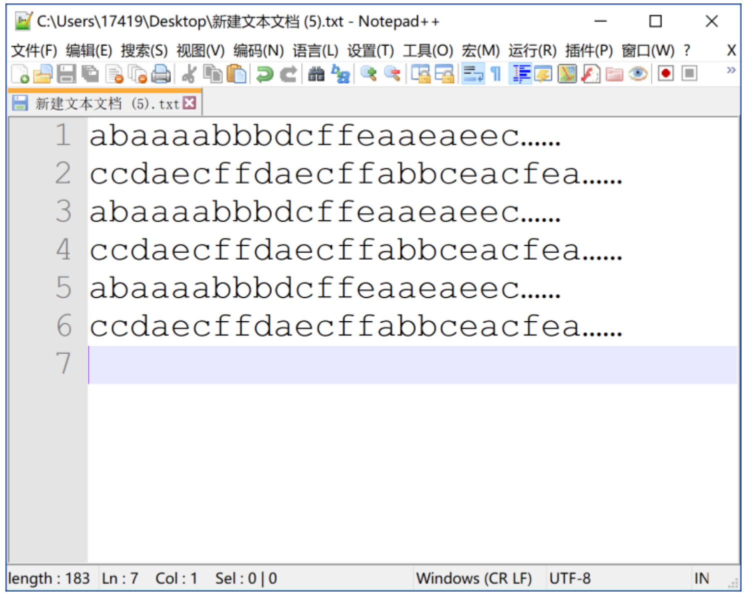
16.3 Huffman codes

- Wish to store compactly 100,000-character data file. Only six different characters appear.

  abaaaabbbdcffeaaeaeec … ccdaecffdaecffabbceacfea …

  Frequency table:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100


C:\Users\17419\Desktop\新建文本文档 (5).txt - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
新建文本文档 (5).txt
length : 183 Ln : 7 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 IN

```
abaaaabbbdcffeaaeaeec......
ccdaecffdaecffabbceacfea......
abaaaabbbdcffeaaeaeec......
ccdaecffdaecffabbceacfea......
abaaaabbbdcffeaaeaeec......
ccdaecffdaecffabbceacfea......
```

<u>000</u><u>001</u><u>000</u>..

<u>0</u><u>101</u><u>0</u>..

6个不同的基本字符，构成一个数据文件，文件大小为10万个字符，如何对字符进行编码，使得文件存储所占的空间最小？

- Many ways (encodes) to represent such a file of information
- *binary character code* (or *code* for short): each character is represented by a unique binary string.
  - *fixed-length code*: if use 3-bit codeword, the file can be encoded in 300,000 bits. Can we do better?

  定长码：文本文件（一个字节编码一个字符），文件占80万位；用3位二进制表示一个字符，能用30万位二进制来编码该数据。能否进一步压缩？

16.3 Huffman codes

- 100,000-character data file

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- *binary character code* (or *code* for short)
  - *variable-length code*: by giving frequent characters short codewords and infrequent characters long codewords, here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f.

    $(45\cdot1 + 13\cdot3 + 12\cdot3 + 16\cdot3 + 9\cdot4 + 5\cdot4)\cdot 1{,}000 = 224{,}000$ bits

    变长码：高频出现的字符以短字码表示，低频字符用长字码表示。

16.3 Huffman codes

- 100,000-character data file

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

$3 \cdot 100{,}000$

$= 300{,}000$ bits

$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000$

$= 224{,}000$ bits

- *binary character code* (or *code* for short)
  - *fixed-length code*: 300,000 bits.
  - *variable-length code*: 224,000 bits, a savings of 25.3%. In fact, this is an optimal character code for this file.

定长码：30万位 vs 变长码：22.4万位

节省空间 (30-22.4)/30 = 25.3%

16.3.1 Prefix codes

- *prefix codes* (prefix-free codes): no codeword is a prefix of some other codeword.
  前缀码〔前缀无关码〕：没有字码是其他字码的前缀

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Encoding is always simple for any binary character code
  - Concatenate the codewords representing each character. For example, “abc”, with the variable-length prefix code as 0·101·100 = 0101100, where we use ‘·’ to denote concatenation.
- Prefix codes simplify decoding.

编码很简单，把每个字符的“二进制编码（codeword, 字码, 编码）”用连接符‘·’连接在一起即可，连接符通常也可以省略（节省存储）。用前缀码使得解码同样简单。

16.3.1 Prefix codes

- *prefix codes* (prefix-free codes): no codeword is a prefix of some other codeword.
  前缀码〔前缀无关码〕：没有字码是其他字码的前缀

	a	b	c	d	e	f
Variable-length codeword	0	101	100	111	1101	1100

- Prefix codes simplify decoding.
  - Since no codeword is a prefix of any other.
  - We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file.
  - Exam: 001011101 uniquely as 0·0·101·1101, which decodes to “aabe”.

根据前缀码的性质，可以很方便把编码翻译回原始的字符。

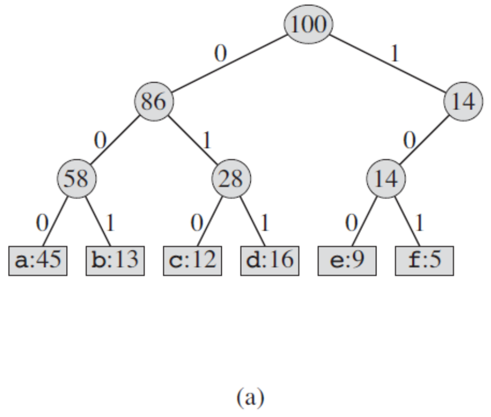
16.3.1 Prefix codes

编码表

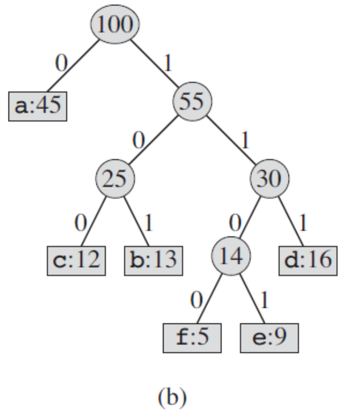
a	b	c	d	e	f
0	101	100	111	1101	1100

编码文件 aabe $\rightarrow$ 001011101

解码文件 uniquely as 0·0·101·1101, which decodes to “aabe”.



定长码



变长码

Decoding: A binary tree whose leaves are the given characters provides one such representation.

二叉树是一种方便的表示方法，树叶为给定字符，从树根到树叶的过程就是相应字符的解码过程。

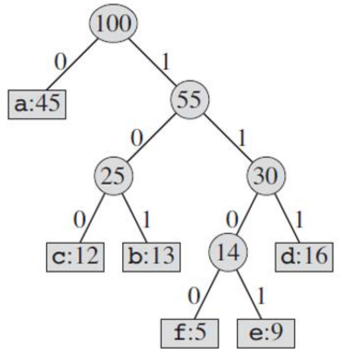
16.3.1 Prefix codes

编码表

a	b	c	d	e	f
0	101	100	111	1101	1100

编码文件 aabe $\rightarrow$ 001011101

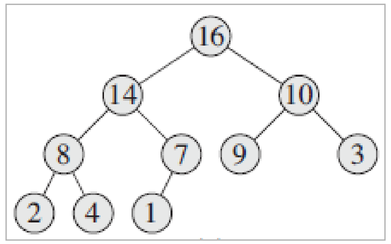
解码文件 uniquely as 0·0·101·1101, which decodes to “aabe”.



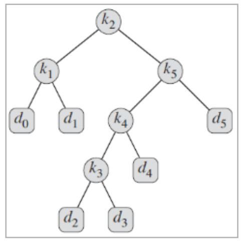
Decoding: We interpret the binary codeword for a character as the path from the root to that character.

字符的二进制编码（二进制编码）为一条从树根到树叶的路径上的二进位序列。

编码是构建二叉编码树，解码是遍历二叉编码树。



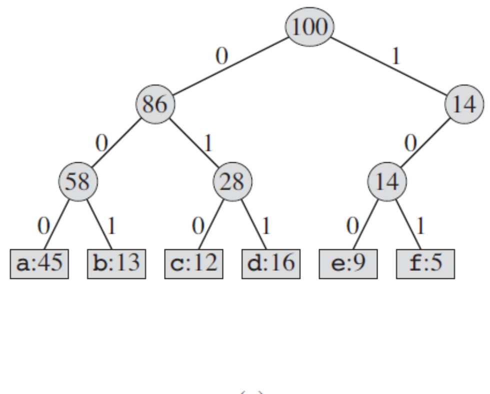
这是最大堆



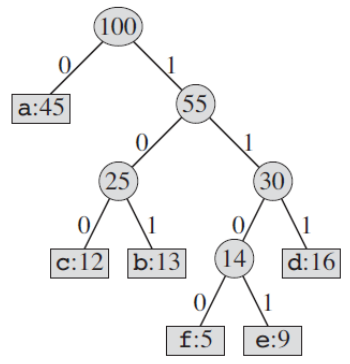
这是二叉搜索树

注意：不要混淆各种二叉树，如，最大（小）堆、二叉搜索树，哈夫曼树

16.3.1 Prefix codes



(a)



(b)

full binary tree 满二叉树

- 国际定义：除叶子节点外，所有节点都有两个孩子。
- 国内定义：除了满足如上定义，所有叶子节点还需要在同一层上。

- An optimal code for a file is always represented by a *full* binary tree, every nonleaf node has two children (Ex16.3-1). The fixed-length code in our example is not optimal.
- We can restrict our attention to full binary trees
  - C is the alphabet,
  - all character frequencies > 0
  - the tree for an optimal prefix code has $|C|$ leaves, one for each letter of C , and exactly $|C|$-1 internal nodes.

一个文件的最优编码通常表示为一颗满二叉树（编码树），其性质：字母表 C 有 n 字符，每个字符在文件中出现的次数大于0，字母表的一种最优前缀码对应这颗编码树，该树有 n 个叶子，有 n-1 个内部节点。

16.3.1 Prefix codes

Compute # of bits required to encode a file: Given a tree T corresponding to a prefix code, for each character c in the alphabet C, $f(c)$: frequency of c in the file

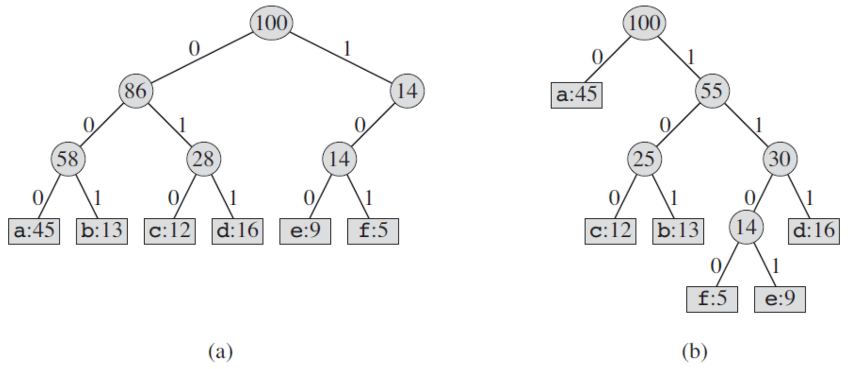
- $d_T(c)$: depth of c's leaf in the tree (length of the codeword for character c), then, # of bits required to encode a file

$$B(T) = \sum_{c \in C} f(c) d_T(c) \qquad (16.5)$$

which we define as the *cost* of the tree T.

编码一个文件所需的二进制位数定义为：一颗编码树 T（对应一种前缀码），字母表 C 的某个字符 c，其在文件中出现的次数为 $f(c)$，在树 T 中的深度为 $d_T(c)$（树根的深度为0），则用该二叉树编码该文件所需要的位数为 $B(T) = \sum_{c \in C} f(c) d_T(c)$，称为该文件的编码树 T 的消费(cost)。

不同的文件，其编码树 T 不一样。



$B(T)$ 最小的编码树，称为最优编码树，哈夫曼是一种最优编码树。

决定编码树 T 的消费(cost)的两个因素：

- 每个树叶(字符c)的次数 $f(c)$；
- 每个字符(字符c)的深度 $d_T(c)$，即每个字符的编码，对应树的结构。

16.3.2 Constructing a Huffman code

Huffman code: a greedy algorithm that constructs an optimal prefix code.

哈夫曼编码：构造最优前缀码的一种贪心算法。

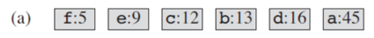
```
HUFFMAN($C$)
1  $n \leftarrow |C|$ ,  $Q \leftarrow C$
2  for $i \leftarrow 1$ to $n$ - 1
3        allocate(分配) a new node $z$
4        $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN ($Q$)
5        $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN ($Q$)
6        $f[z] \leftarrow f[x] + f[y]$
7        INSERT($Q$, $z$)
8  return EXTRACT-MIN($Q$)   //return the root of the tree.
```

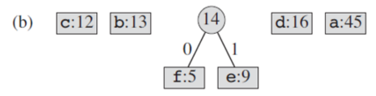
C: set of n characters, $c \in C$: an object with frequency $f[c]$.

- Build the tree T corresponding to the optimal code.
- Begin with $|C|$ leaves, perform $|C|$-1 "merging" operations.
- A min-priority queue Q, keyed on f , is used to identify the two least-frequent objects to merge together. Result of the merger is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

字母表 C，有 n 个字符，字符 $c \in C$ ，在文件中出现的次数为 $f(c)$：

- 构建一颗编码树 T，表示最优前缀码。
- 从 n 个树叶开始，执行 n-1 次合并操作。
- 使用最小优先队列 Q 存储 $f(c)$, 从队列 Q 中取出最小的两个数（出现最少的两个字符的次数）进行合并（相加），合并后的数放入队列 Q。本步骤反复进行，直到队列只有一个对象（树根节点） 。

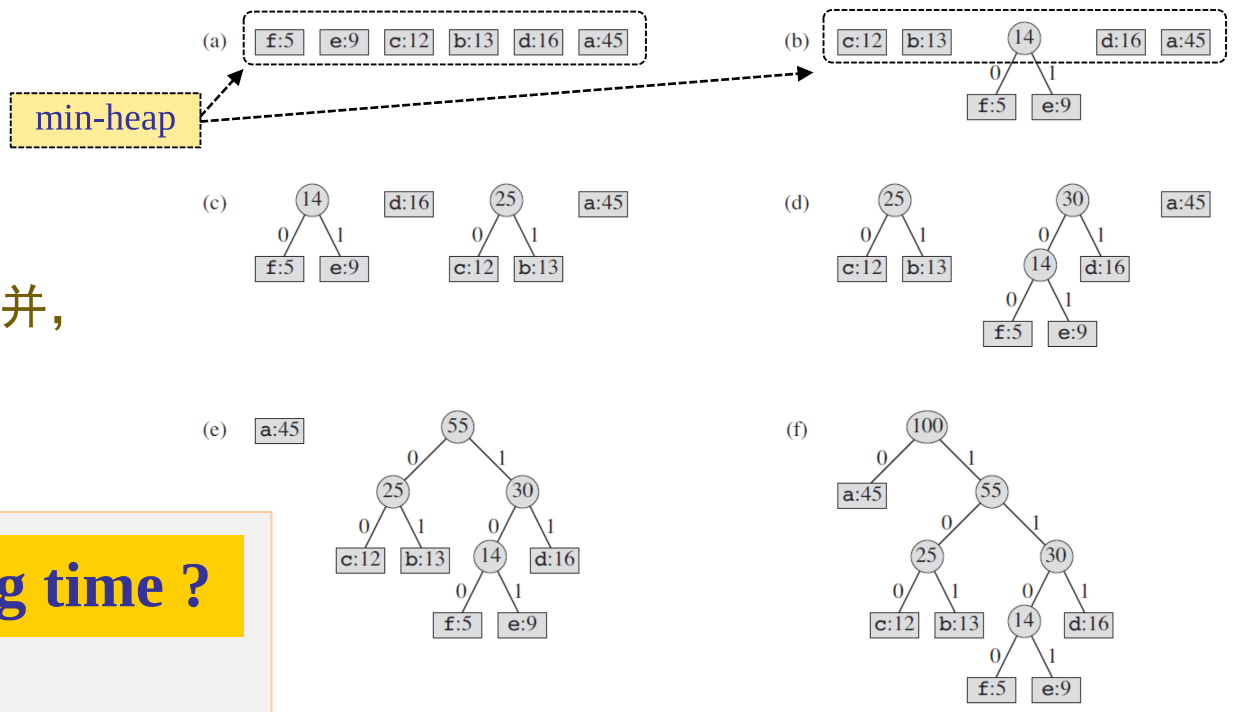




16.3.2 Constructing a Huffman code

Example: Huffman's algorithm proceeds. 6 letters, 5 merge steps. The final tree represents the optimal prefix code.

哈夫曼算法执行的一个实例：6个字符，5次合并，最后构成的二叉编码树就是最优前缀码。


min-heap
(a) f:5 e:9 c:12 b:13 d:16 a:45
(b) c:12 b:13 14 d:16 a:45 f:5 e:9
(c) 14 d:16 25 a:45 f:5 e:9 c:12 b:13
(d) 25 30 a:45 c:12 b:13 14 d:16 f:5 e:9
(e) a:45 55 25 30 c:12 b:13 14 d:16 f:5 e:9
(f) 100 a:45 55 25 30 c:12 b:13 14 d:16 f:5 e:9
0
1

```
HUFFMAN($C$)
1  $n \leftarrow |C|$ ,  $Q \leftarrow C$
2  for $i \leftarrow 1$ to $n - 1$
3        allocate(分配) a new node $z$
4        $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN ($Q$)
5        $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN ($Q$)
6        $f[z] \leftarrow f[x] + f[y]$
7        INSERT($Q$, $z$)
8  return EXTRACT-MIN($Q$)   //return the root of the tree.
```

Running time ?

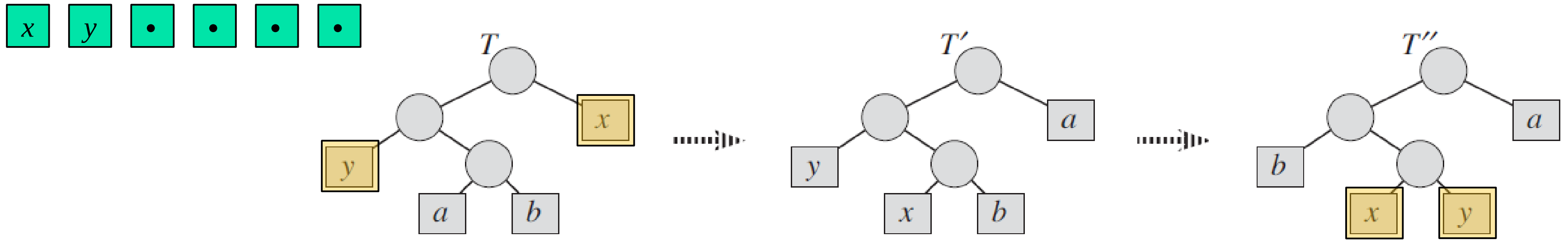
最小优先队列就是一个最小堆，抽取并删除最小元素、插入新元素的堆操作的时间都是 $O(\lg n)$, 循环的总消费是 $O(n\lg n)$.
采用van Emde Boas tree进行最小堆操作，计算时间可以改进到 $O(n\lg\lg n)$.

*16.3.3 Correctness of Huffman's algorithm

Problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

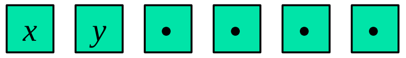
Lemma 16.2 (greedy-choice property): Let C be an alphabet, each character $c \in C$ has frequency $f[c]$. x and $y \in C$, and having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

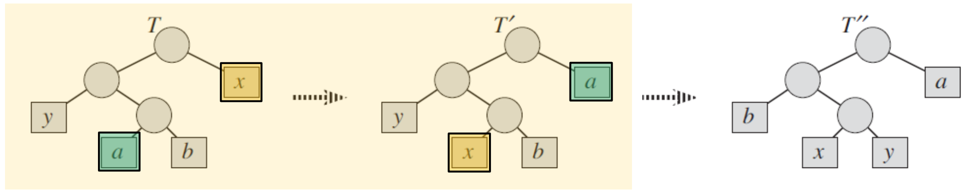
Proof idea: take the tree T representing an arbitrary optimal prefix code, and modify it to make a tree representing another optimal prefix code such that x and y appear as sibling leaves（姐妹叶） of maximum depth in the new tree.



*16.3.3 Correctness of Huffman's algorithm

Lemma 16.2: $c \in C$ has frequency $f[c]$. $x, y \in C$, having the lowest frequencies. Then, exist an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.





$$B(T) = \sum_{c \in C} f(c) d_T(c) \qquad (16.5)$$

Proof : Let a and b are sibling leaves of max depth in optimal T. Assume that $f[a] \le f[b]$, $f[x] \le f[y]$. $f[x]$ and $f[y]$ are the two lowest leaf frequencies, $f[a]$, $f[b]$ are two arbitrary frequencies, in order,
$\Rightarrow f[x] \le f[a]$, $f[y] \le f[b]$. Exchange the pos in T of a and x to produce a tree T' . By (16.5), we have

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\
&= (f[x] - f[a])d_T(x) + (f[a] - f[x])d_T(a) \\
&= (f[a] - f[x])(d_T(a) - d_T(x)) \ \ge \ 0
\end{aligned}$$

$$\longleftrightarrow$$

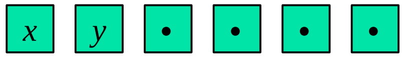
字符的频率：$f \le F$,(x 为 f , a 为 F)
字符在编码树中的高度：$d \le D$

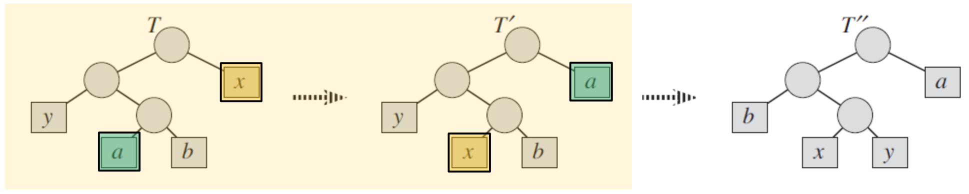
	x	**a**
in T	$f \cdot d$	$F \cdot D$
in T'	$f \cdot D$	$F \cdot d$

$B(T) - B(T') = fd + FD - fD - Fd$
$= f(D - d) + F(D - d) = (F - f)(D - d) \ge 0$

*16.3.3 Correctness of Huffman's algorithm

Lemma 16.2: $c \in C$ has frequency $f[c]$. $x, y \in C$, having the lowest frequencies. Then, exist an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.





$$B(T) = \sum_{c \in C} f(c) d_T(c) \qquad (16.5)$$

Proof : Let a and b are sibling leaves of maximum depth in optimal T. Assume that $f[a] \le f[b]$, $f[x] \le f[y]$. $f[x]$ and $f[y]$ are the two lowest leaf frequencies, $f[a]$, $f[b]$ are two arbitrary frequencies, in order, $\Rightarrow$ $f[x] \le f[a]$, $f[y] \le f[b]$. Exchange the positions in T of a and x to produce a tree T', and then exchange the positions in T' of b and y to produce a tree T''. By (16.5), we have,

similarly, $B(T') - B(T'') \ge 0$,

therefore, $B(T'') \le B(T)$.

Since T is optimal, $B(T) \le B(T'')$.

Then $B(T'') = B(T)$.

Thus, T'' is an optimal tree.

引理16.2的重要意义：

贪心选择 x 和 y 是安全的（x 和 y 具有最长编码，且这种编码在最优解中）。

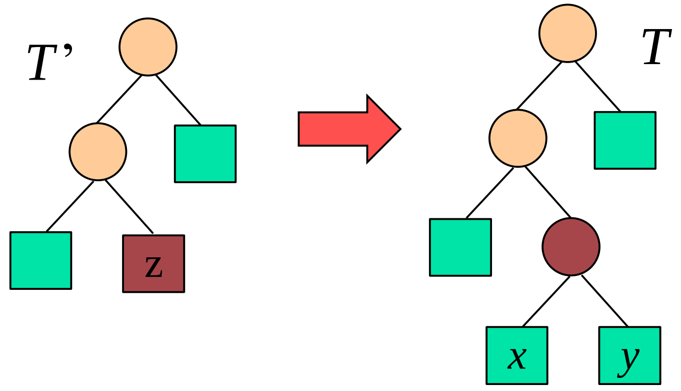
*16.3.3 Correctness of Huffman's algorithm

Lemma 16.3 (optimal-substructure property): Alphabet C , each character $c \in C$ has frequency $f[c]$. x and $y \in C$, and having the lowest frequencies. $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $f[z]=f[x]+f[y]$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for C.

$C' = C - \{x, y\} \cup \{z\}$, 除 $f[z]=f[x]+f[y]$ 以外，f 在 C' 中的定义与在 C 中相同。若 T' 为 C' 的最优编码，则编码 T' 加上贪心选择 ($z = x \cup y$) 得到的 T 为关于 C 的最优编码。

$C : \{c_1, \ldots, c_m, x, y\}$, $C' : \{c_1, \ldots, c_m, z\}$,

T' is optimal $\Rightarrow$ T is optimal



*16.3.3 Correctness of Huffman's algorithm

Lemma 16.3 (optimal-substructure property)

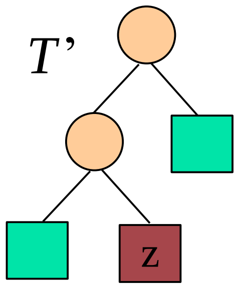
Proof : For each $c \in C - \{x, y\}$, we have $d_T(c) = d_{T'}(c)$, then $f[c]d_T(c) = f[c]d_{T'}(c)$.

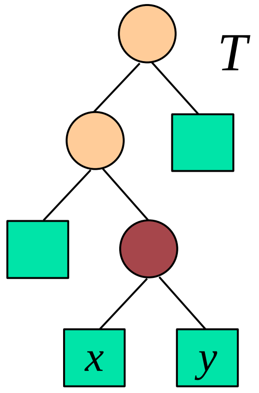
Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y]),$$

from which we conclude that $B(T) = B(T') + f[x] + f[y]$.

$(B(T) = f[x]d_T(x) + f[y]d_T(y) + f[c]d_T(c),\ B(T') = f[z]d_{T'}(z) + f[c]d_T(c))$



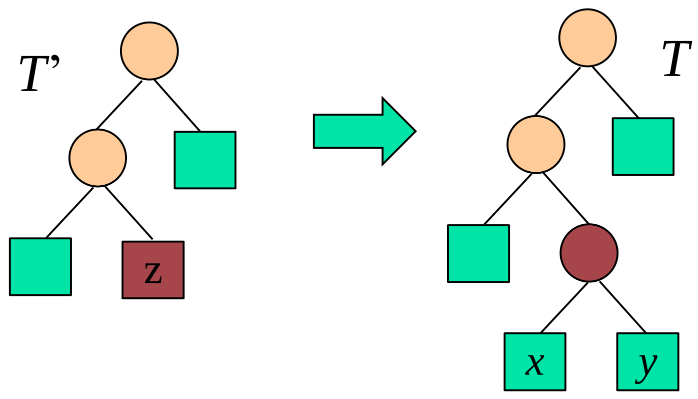


*16.3.3 Correctness of Huffman's algorithm

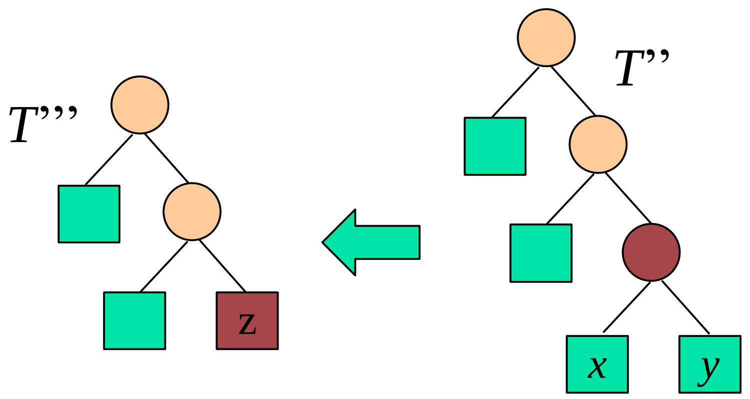
Lemma 16.3 (optimal-substructure property)

$C : \{c_1, \ldots, c_m, x, y\}$, $C' : \{c_1, \ldots, c_m, z\}$,

T' is optimal $\Rightarrow$ T is optimal



Here, $B(T) = B(T') + f[x] + f[y]$



Suppose that T is not optimal, T'' is. Then $B(T'') < B(T)$. Without loss of generality (by Lemma 16.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $f[z]=f[x]+f[y]$. Then

$$B(T''') = B(T'')-f[x]-f[y]$$
$$< B(T)-f[x]-f[y] = B(T'),$$

yielding a contradiction to the assumption that T' represents an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C.

*16.3.3 Correctness of Huffman’s algorithm

Theorem 16.4: Procedure HUFFMAN produces an optimal prefix code.

Proof

Immediate from Lemmas 16.2（每一次选择是贪婪的、是正确的）

and Lemmas 16.3（确保由子问题的最优解能构造原问题的最优解）.

Exercises

$$c[i,j]=\begin{cases}0 & , \quad \text{if } S_{ij}=\varnothing, \\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & , \quad \text{if } S_{ij}\neq\varnothing.\end{cases} \qquad (16.3)$$

- 基于递归方程（转移函数）(16.3)，可以设计算法.

  Direct recursion algorithm (pseudo-code), RE-AS(i, j)? (1) Complexity?

  Dynamic programming algorithm (pseudo-code) , DP-AS(i, j)? (2) Complexity?

- For (16.3):

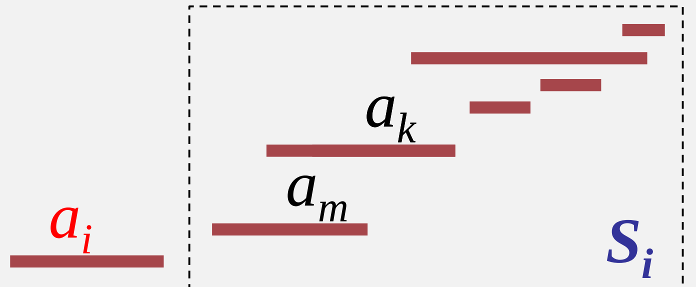
  (3) 总共有多少个子问题?

  (4) 求解一个子问题时，需要做多少次选择?

  (5) 做出一个选择后留下几个待解的子问题?

(Solution to S_{ij})

= (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj})



```
DP-AS(0, n+1)
for i ← 0 to n
    c[i, i+1] ← 0
for x ← 2 to n+1
    for i ← 0 to n+1-x
        j ← i+x
        c[i, j] ← 0
        for k ← i+1 to j – 1
            if a_k is ok // 这个的选择也麻烦
                q ← c[i, k]+c[k, j]+1
                if q > c[i, j]
                    c[i, j] ← q
                    A[i, j] ← k
return c and A
```

```
RE-AS(i, j)
if j - i <= 1  // i 和 j 之间至少有一个活动才能进行安排，
                  此时 j - i > 1，比如4-2>1，此时可能安排的活动是3
    return 0
c[i, j] ← 0
for k ← i+1  to  j-1
    if  a_k is ok  // 这个的选择也是略为麻烦
        q ← RE-AS(i, k) + RE-AS(k, j) + 1
    if q > c[i,  j]
    c[i,  j] ← q
    A[i,  j] ← k
return c[i,  j] and A[i,  j]
```

$S_{ij} = \{a_k \in S: f_i \le s_k < f_k \le s_j\} \Rightarrow a_k$ 不能为 a_i 或 a_j
（若 $k = i$, 得 $c[i, j] = c[i, j] + 1$）.

https://ks.wjx.top/vm/OrgZpKx.aspx#

$$c[i,j]=\begin{cases}0 & , \quad \text{if } S_{ij}=\varnothing,\\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & , \quad \text{if } S_{ij}\neq\varnothing.\end{cases} \qquad (16.3)$$

- 基于递归方程（转移函数）(16.3)，可以设计算法.

  Direct recursion algorithm (pseudo-code), RE-AS(i, j)? (1) Complexity?

  Dynamic programming algorithm (pseudo-code) , DP-AS(i, j)? (2) Complexity?

- For (16.3):

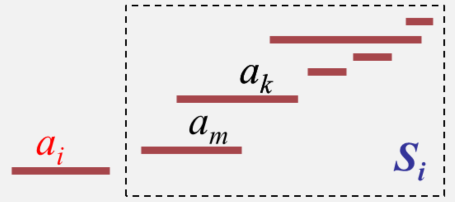
  (3) 总共有多少个子问题?

  (4) 求解一个子问题时，需要做多少次选择?

  (5) 做出一个选择后留下几个待解的子问题?

(Solution to S_{ij})

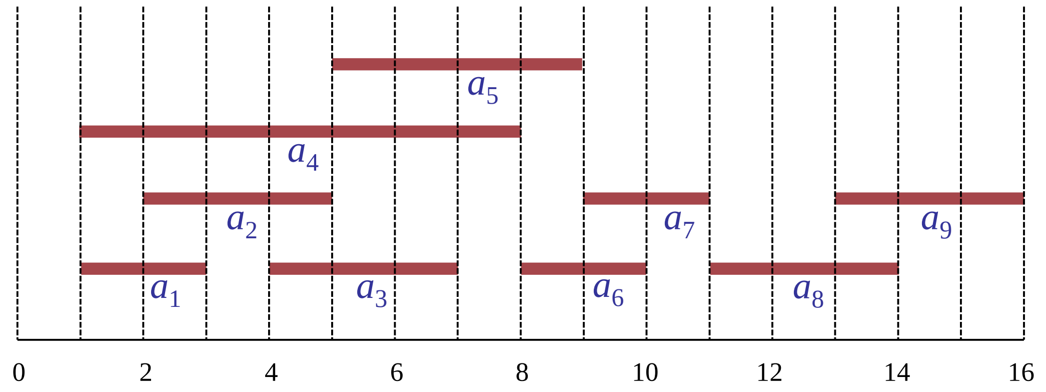
= (solution to S_{ik}) ∪ $\{a_k\}$ ∪ (solution to S_{kj})



Exercises

16.1-1,

16.1-2，最晚开始优先原则



16.3-2, 基于fibonacci数列的前8个数作为数据文件相应字符及其频次的集合，这些字符的最优编码是什么？

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Answer

a, 0000? or 111? or ..

b, …

c, …

…

又一些大作业或讨论题提示：

- 活动安排、Huffman code等等，是否都能描述为背包问题?
- 本书有多少算法是用的贪心策略（小论文：贪心算法十个经典问题？）
- 哈夫曼编码（用哈夫曼压缩方法，设计一个压缩软件：测试一下，算法导论这本书的压缩率能到多少？）
- OBST（最优二叉搜索树构建（以某本书里的词汇为基础？））

- 活动安排、分数背包
- 0-1背包、钢管切割、ALS、MCM、LCS、最短路径
- 雇佣（雇佣多少人）、取帽子、相同生日
- RSA加密解密、FFT、串匹配、计算几何、最大流

- 算法实验室（问题求解工具、算法效果展示平台、多种算法时间复杂度对比分析、多种算法空间复杂度对比分析、IO导入、…… 。支持穷举、递归、回溯、分治、DP、贪心；排序、查找；随机；图、树；等等若干方法（算法）的仿真演示。）