Greedy Algorithms

交通大學 資訊工程系

Introduction

- **Greedy algorithms**
  - **Similar to dynamic programming.**
  - **Used for optimization problems.**
  - **Do not always yield an optimal solution. But sometimes they do.**
- ***Idea:***
  - **When we have a choice to make, make the one that looks best *right now*.**
  - **Make a *locally optimal choice* in hope of getting a *globally optimal solution*.**

Activity-Selection Problem[1]

- **Activity selection**
  - Set of activities $S = \{a_1, \ldots, a_n\}$.
  - *n activities* require ***exclusive*** use of a common resource. For example, scheduling the use of a classroom.
  - a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i = start time and f_i = finish time, where $0 \le s_i < f_i < \infty$.
  - If a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
  - *Goal:* Select the largest possible set of ***mutually compatible*** activities.

Activity-Selection Problem[2]

- **Example:**
  - **Sorted in monotonically increasing order of 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	8	9	10	11	12	13	14

  - **Largest subsets of mutually compatible activities.**

$\{a_1, a_4, a_8, a_{11}\}$

$\{a_2, a_4, a_9, a_{11}\}$

Activity-Selection Problem[3]

- Optimal substructure of activity selection
  - Let $S_{ij} = \{a_k \in S \mid f_i \le s_k < f_k \le s_j\}$ = activities that start after a_i finishes and finish before a_j starts.
  - Activities in S_{ij} are compatible with
    - all activities that finish by f_i, and
    - all activities that start no earlier than s_j.
  - Fictitious activities a_0 and a_{n+1}, where $f_0 = 0$ and $s_{n+1} = \infty$.
  - Then $S = S_{0,n+1}$. Range for S_{ij} is $0 \le i, j \le n + 1$.
  - Assume that activities are sorted by monotonically increasing finish time: $f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$.
  - Then $i \ge j \Rightarrow S_{ij} = \phi$.

Activity-Selection Problem[4]

- So only need to worry about S_{ij} with $0 \le i < j \le n + 1$. All other S_{ij} are ϕ.
- Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:
  - S_{ik} (start after a_i finishes, finish before a_k starts)
  - S_{kj} (start after a_k finishes, finish before a_j starts)
- Solution to S_{ij} is (solution to S_{ik}) $\cup$ $\{a_k\}$ $\cup$ (solution to S_{kj}).
- If an optimal solution A_{ij} to S_{ij} includes a_k , then the solutions A_{ik} to S_{ik} and A_{kj} to S_{kj} used within this solution must be optimal as well. (Cut-and-paste)

Activity-Selection Problem[5]

- **Recursive solution to activity selection**
  - Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.
    - $i \geq j \Rightarrow S_{ij} = \phi \Rightarrow c[i, j] = 0$.
  - If $S_{ij} \neq \phi$, suppose we know that a_k is in the subset. Then $c[i, j] = c[i, k] + c[k, j] + 1$.
  - Full recursive definition of $c[i, j]$ becomes:

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

Activity-Selection Problem[6]

- **Making the greedy choice**
  - **Choose an activity that leaves the resource available for as many other activities as possible.**
    - **Must be the first one to finish.**
  - **If we make the greedy choice, we have only one remaining subproblem to solve:**
    - **Finding activities that start after a_1 finishes.**
  - **Why don't we have to consider activities that finish before a_1 starts?**
    - **Since $s_1 < f_1$, and f_1 is the earliest finish time of any activity.**

Activity-Selection Problem[7]

- **Theorem 16.1**
  - Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.
- **Proof**
  - Let a_j be the activity in A_k with the earliest finish time.
  - If $a_j = a_m$, we are done.
  - If $a_j \neq a_m$, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j.

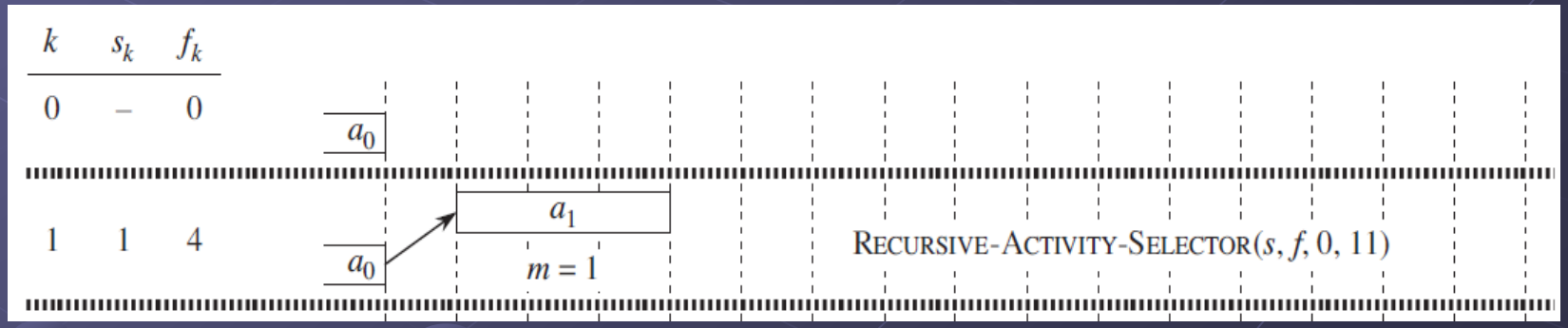
Activity-Selection Problem[8]

- According to Theorem 16.1
  - We can repeatedly choose the activity that finishes first.
  - Keep only the activities compatible with this activity, and
  - Repeat until no activities remain.
- Greedy algorithms typically have the top-down design
  - Make a choice and then solve a subproblem.
- A recursive greedy algorithm
  - Index k that defines the subproblem S_k it is to solve.
  - The size n of the original problem.
  - Add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S.

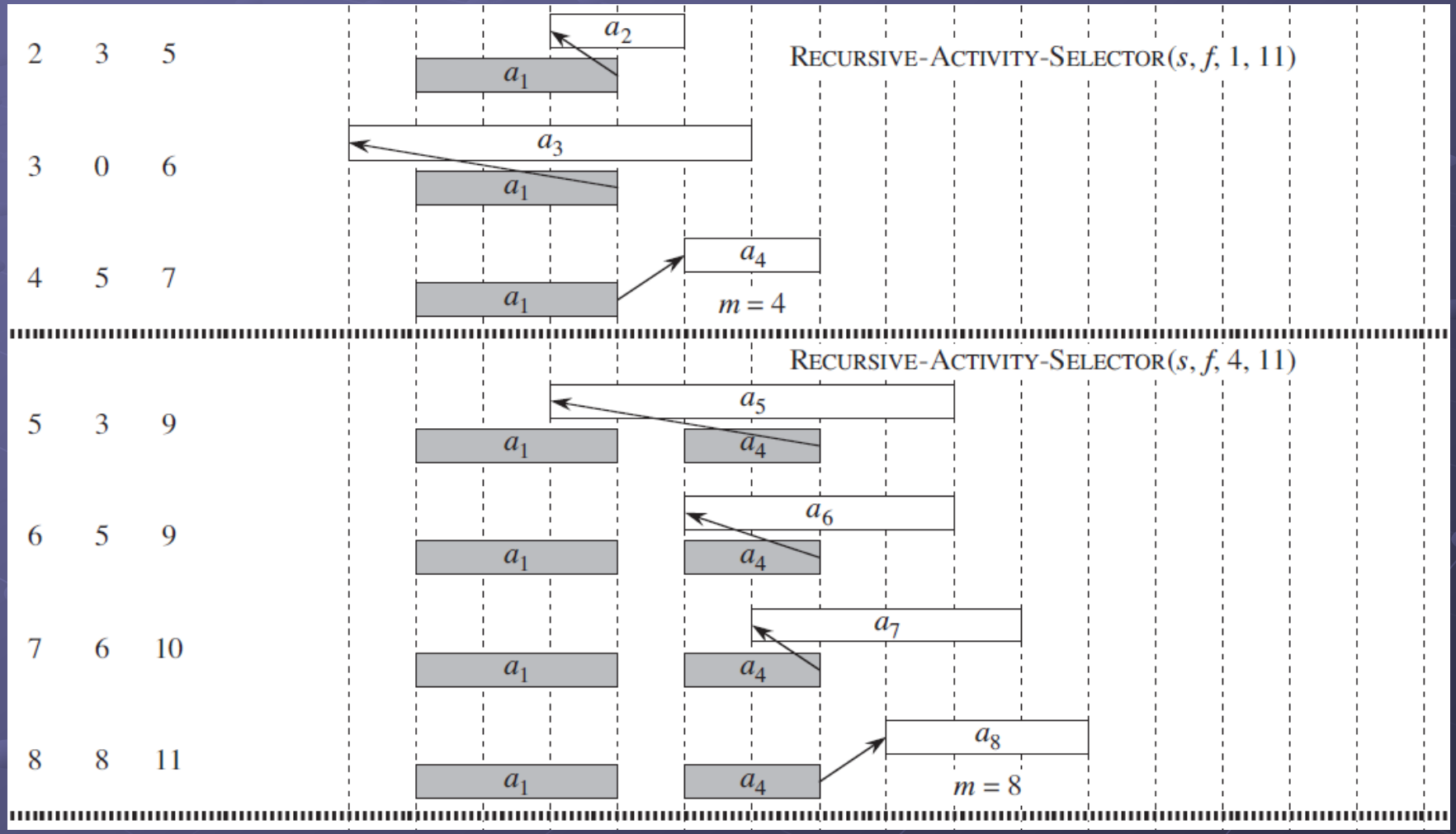
Activity-Selection Problem[9]

- The initial call is RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n).

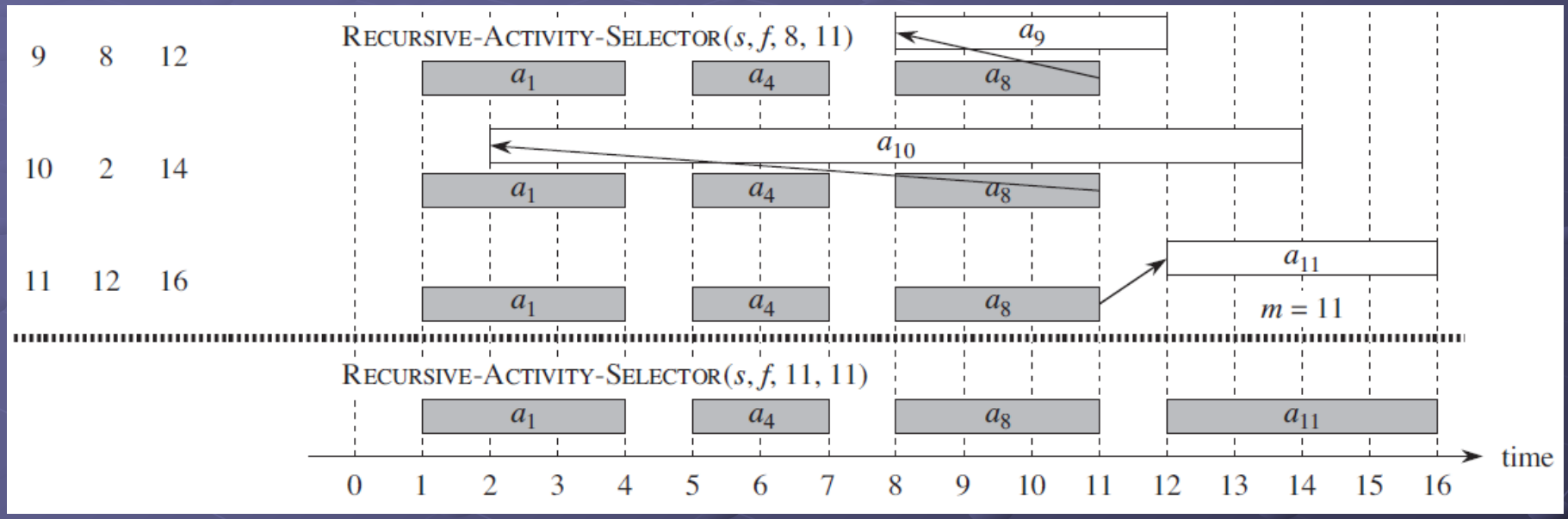
```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```



Activity-Selection Problem[10]



Activity-Selection Problem[11]



☞ **Running time: $\Theta(n)$**

Activity-Selection Problem[12]

- An iterative greedy algorithm
  - ***Tail recursive***: It ends with a recursive call to itself followed by a union operation.

```
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
```

  - Running time: $\Theta(n)$.

Elements of the Greedy Strategy[1]

- **Greedy strategy**
  - **The choice that seems best at the moment is the one we go with.**
- **What did we do for activity selection?**
  1. Determine the optimal substructure.
  2. Develop a recursive solution.
  3. Show that if we make the greedy choice, then only one subproblem remains.
  4. Prove that it is always safe to make the greedy choice.
  5. Develop a recursive greedy algorithm.
  6. Convert it to an iterative algorithm.

Elements of the Greedy Strategy^2

- **Develop the substructure with an eye toward**
  - **Making the greedy choice,**
  - **Leaving just one subproblem.**
- **Three steps to design greedy algorithms:**

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Show that greedy choice and optimal solution to subproblem ⇒ optimal solution to the problem.

Elements of the Greedy Strategy[3]

- No general way to tell if a greedy algorithm is optimal, but two key ingredients are
  - *Greedy-choice property*
  - *Optimal substructure*
- Greedy-choice property
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
  - In *Dynamic programming:*
    - Make a choice at each step.
    - Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
    - Solve *bottom-up*.

Elements of the Greedy Strategy[4]

- In *Greedy:*
  - Make a choice at each step.
  - Make the choice ***before*** solving the subproblems.
  - Solve ***top-down***.
- Typically show the greedy-choice property by what we did for activity selection:
  - Look at a globally optimal solution.
  - If it includes the greedy choice, done.
  - Else, modify it to include the greedy choice, yielding another solution that's just as good.

Elements of the Greedy Strategy[5]

- We can usually make the greedy choice more efficiently than consider a wide set of choices.
  - Preprocess input to put it into greedy order.
  - Or, if dynamic data, use a priority queue.

☞ Optimal substructure

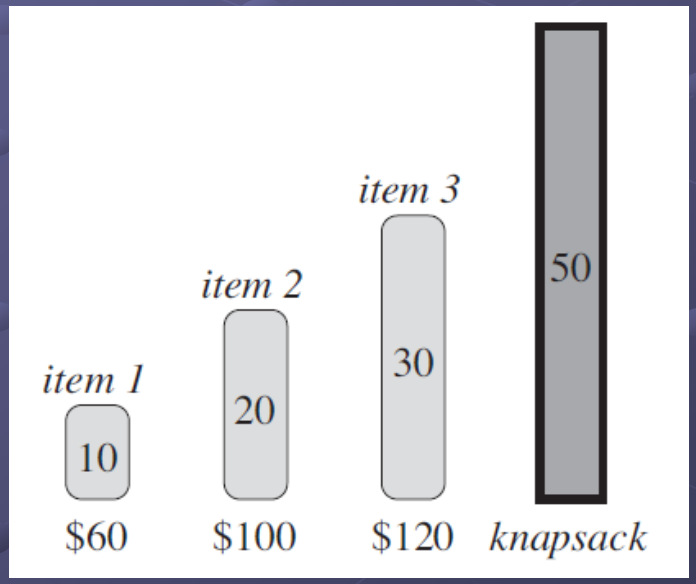
- A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Just show that optimal solution to subproblem and greedy choice ⇒ optimal solution to problem.

Elements of the Greedy Strategy[6]

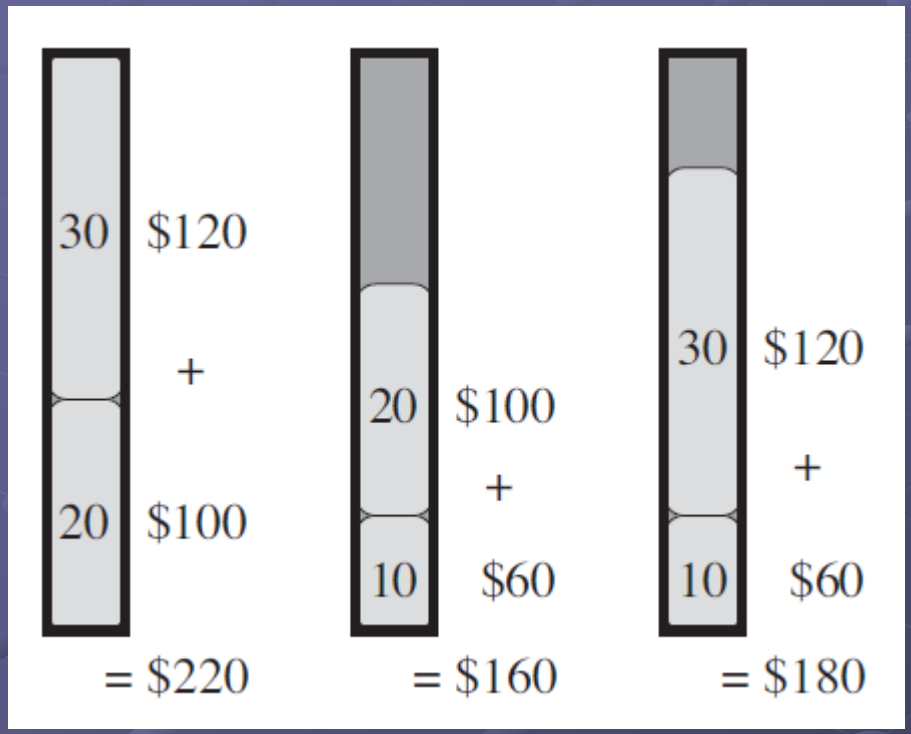
- ***0-1 knapsack problem*:**
  - ***n* items.**
  - **Item *i* is worth v_i dollars, weighs w_i pounds, where v_i and w_i are integers.**
  - **Find a most valuable subset of items with total weight $\leq W$.**
  - **Have to either take an item or not take it — can't take part of it.**
- ***Fractional knapsack problem*:**
  - **Like the 0-1 knapsack problem, but can take fraction of an item.**

Elements of the Greedy Strategy[7]

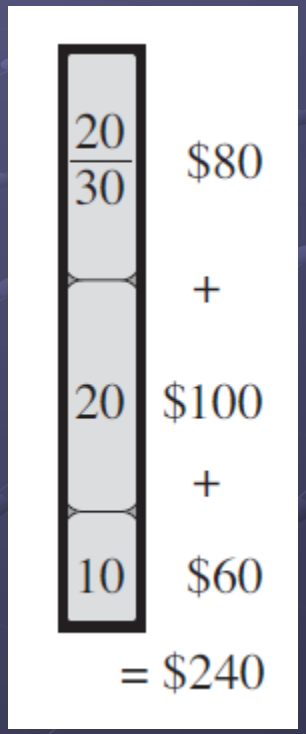
- Both have optimal substructure.
  - But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.
  - To solve the fractional problem, rank items by value/weight: v_i/w_i. Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i .



Elements of the Greedy Strategy[8]



0-1 knapsack



Fractional knapsack

Elements of the Greedy Strategy[9]

- **Taking item 1 doesn't work in the 0-1 problem**
  - **The thief is unable to fill his knapsack to capacity.**
  - **The empty space lowers the effective value per pound of his load.**
  - **We must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the choice.**
- **The hallmark of dynamic programming**
  - **The problem formulated in the way gives rise to *many overlapping subproblems*.**

Huffman Codes[1]

- **Huffman codes**
  - **Very effective technique for compressing data.**
  - **Saving 20% to 90% are typical.**
  - **Consider the data to be a sequence of characters.**
- **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.**
  - **Fixed-length code**
  - **Variable-length code**

Huffman Codes[2]

- **Example:**
  - **100,000-character data file.**
  - **Only six different characters appear.**
  - **Frequent characters short codewords.**
  - **Infrequent characters long codewords.**

	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

(45×3 + 13 ×3 + 12 ×3 + 16 ×3 + 9 ×3 + 5 ×3) ×1000 = 300,000 bits
(45×1 + 13 ×3 + 12 ×3 + 16 ×3 + 9 ×4 + 5 ×4) ×1000 = 224,000 bits

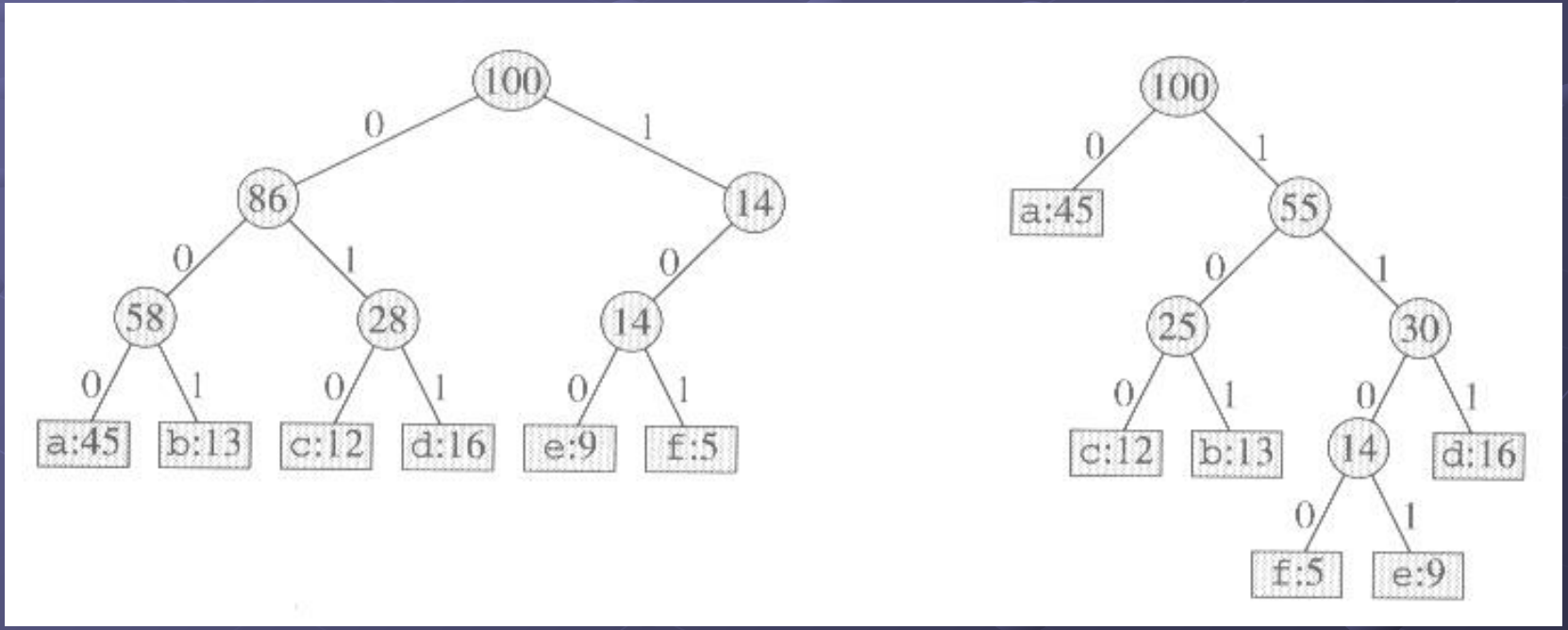
Huffman Codes[3]

☞ ***Prefix codes*** (***Prefix-free codes***)

- **No codeword is also a prefix of some other codeword.**
- **The optimal data compression achievable by a character code can always be achieved with a prefix code.**
- **Prefix code are desirable because they simplify decoding.**
  - **For example: 001011101 → 0 · 0 ·101 ·1101 (aabe)**
- **Representation for prefix code: *Binary tree*.**
  - **Whose leaves are the given characters.**
  - **Interpret the binary codeword for a character as the path from the root to that character.**

Huffman Codes[4]

- 0 means “go to the left child”.
- 1 means “go to the right child”.



Huffman Codes[5]

- An optimal code for a file is always represented by a ***full binary tree***.
  - Every nonleaf node has two children.

☞ The number of bits required to encode a file (The cost of the tree T):

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

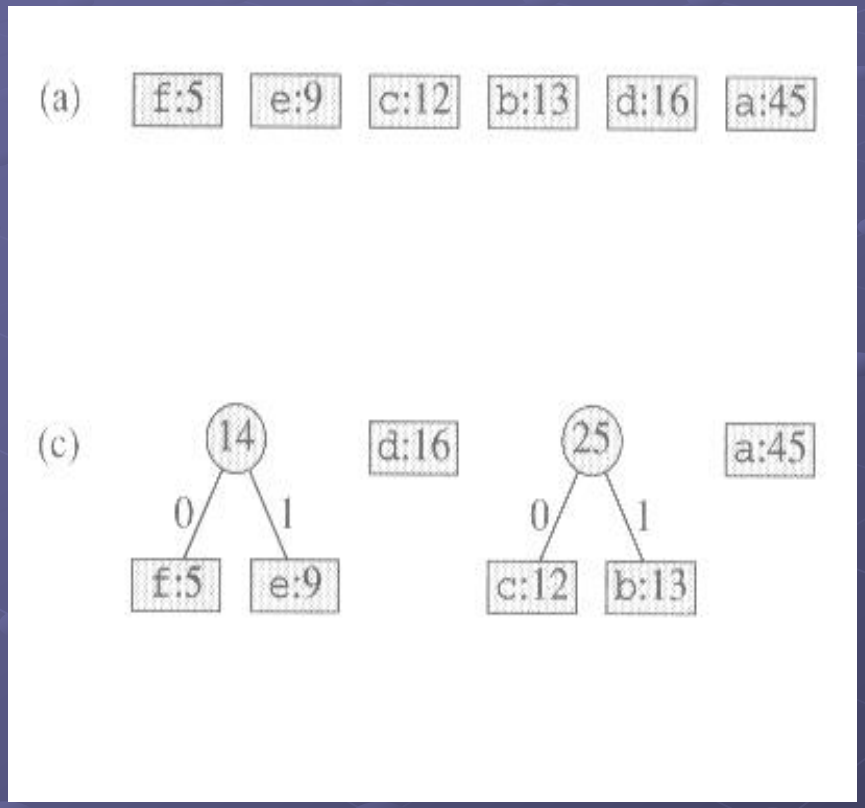
- $c.freq$ denote the frequency of c in the file.
- $d_T(c)$ denote the depth of c's leaf in the tree.

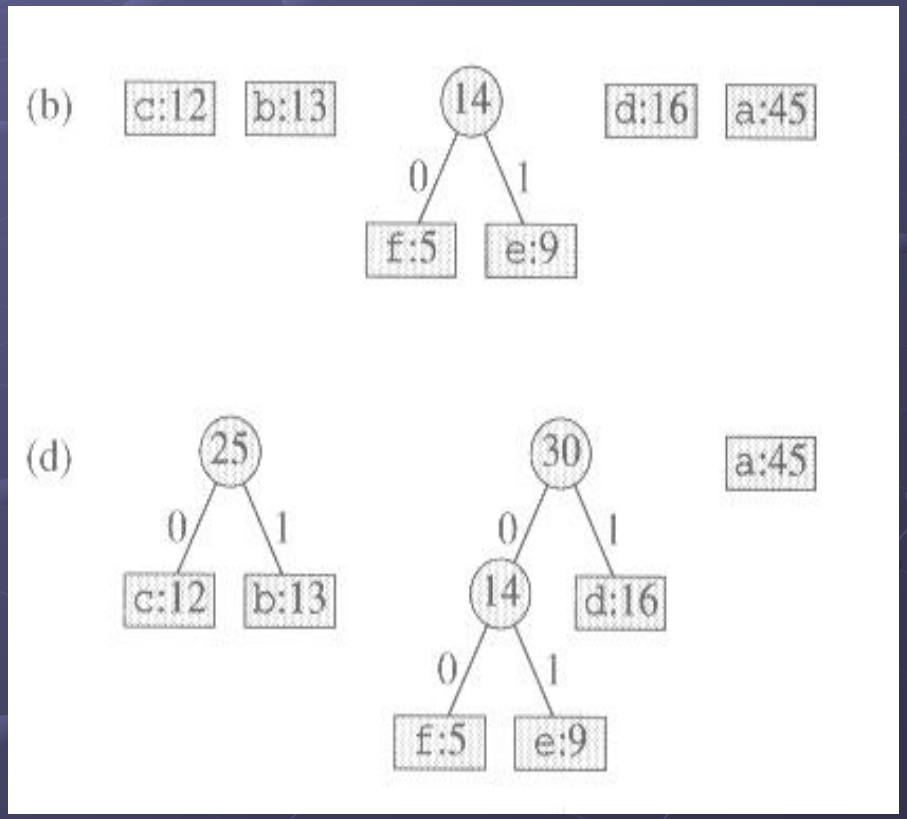
Huffman Codes[6]

- **Constructing a Huffman code: O(*n lg n*).**
  - **A min-priority queue *Q*, keyed on *freq*, is used to identify the two least-frequent objects to merge together.**

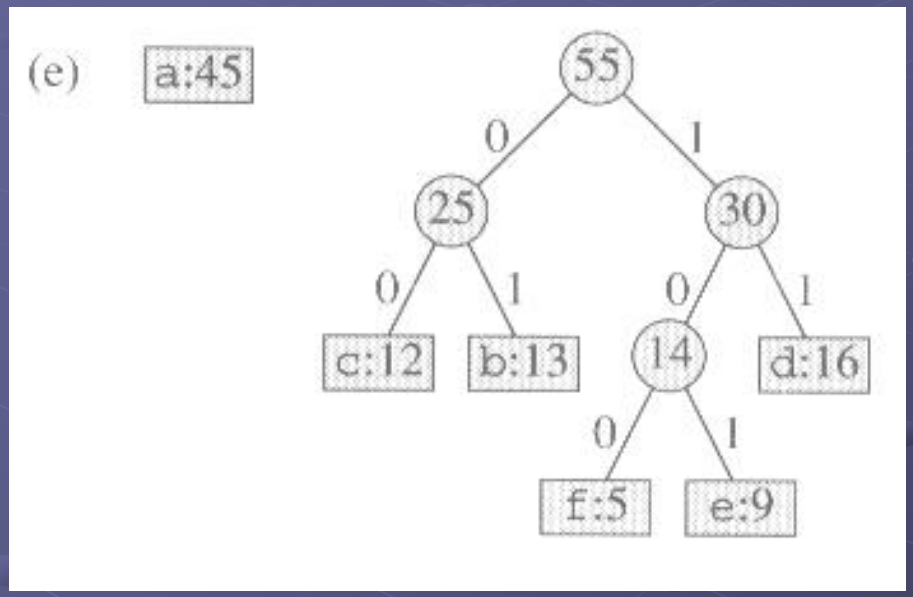
```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

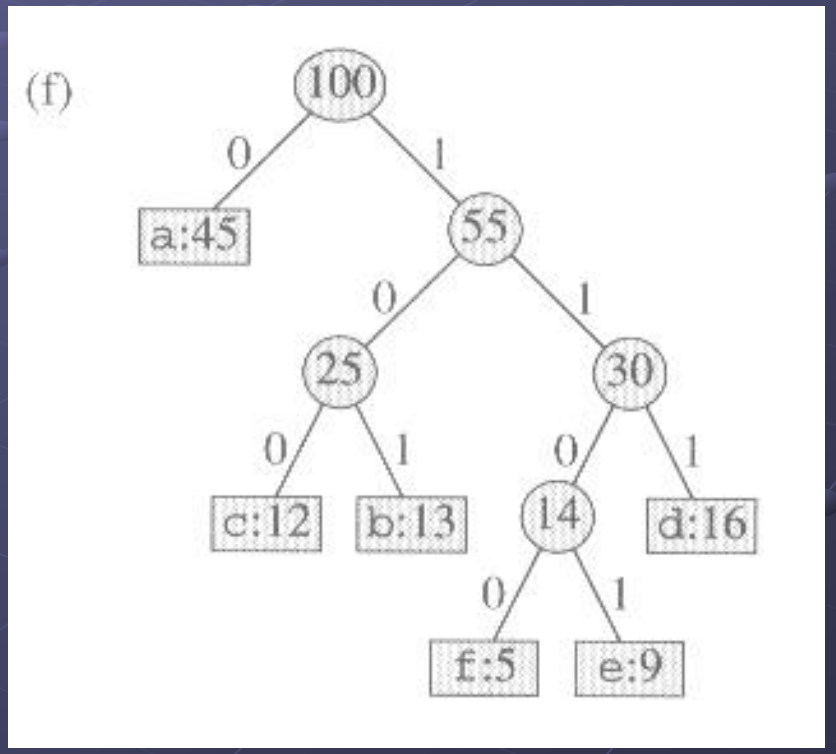
Huffman Codes[7]





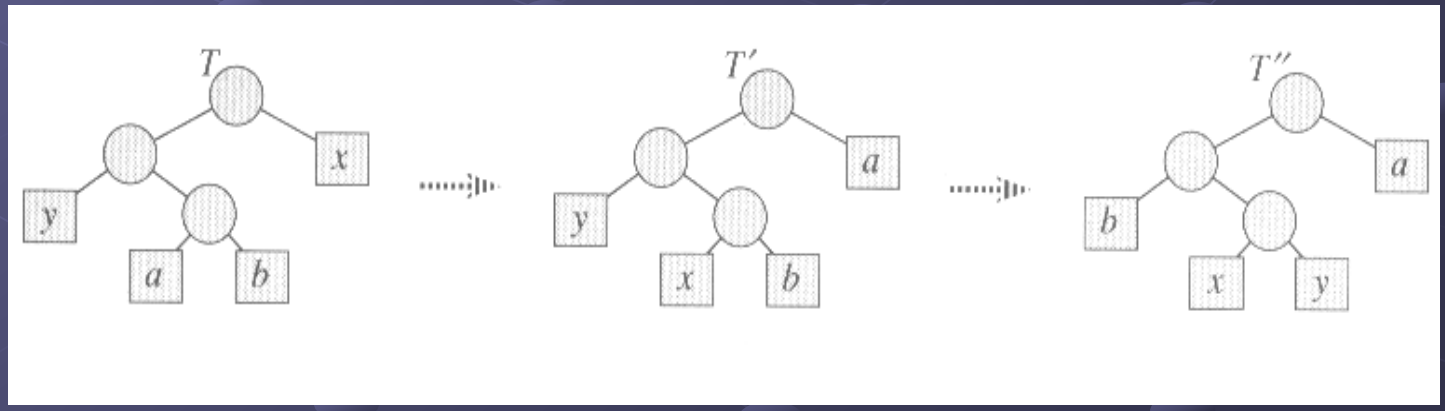
Huffman Codes[8]





Correctness of Huffman's Algorithm[1]

- ***Lemma 16.2***
  - **Let C be an alphabet in which $c \in C$ has frequency $c.freq$. Let x and y be the two characters in C having the lowest frequencies. Then there exists an optimal prefix code in C in which the codeword for x and y having the same length and differ only in the last bit.**
  - **Proof:**



Correctness of Huffman's Algorithm[2]

- The difference in cost between T and T' is

$$
\begin{aligned}
& B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0 ,
\end{aligned}
$$

- Similarly, $B(T') - B(T'')$ is also nonnegative.
- Therefore, $B(T'') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$ which implies $B(T'') = B(T)$.
- Thus T'' is an optimal tree.

Correctness of Huffman's Algorithm[3]

☞ *Lemma 16.3*

- Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with characters x, y removed and character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $z.freq = x.freq + y.freq$. 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 the alphabet C.

Correctness of Huffman's Algorithm[4]

- **Proof:**

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

$$B(T) = B(T') + x.freq + y.freq$$

or, equivalently,

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

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

a contradiction to the assumption that T' represents an optimal prefix code for C'.

Correctness of Huffman's Algorithm[5]

- ***Theorem 16.4***
  - **Procedure HUFFMAN produces an optimal prefix code.**
  - **Proof: Immediate form Lemmas 16.2 and 16.3.**