Making the Greedy Choice (3/4)

- **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_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

Making the Greedy Choice (4/4)

- Although we might be able to solve the activity-selection problem with dynamic programming, we don't need to.
- ➡ 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 top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A Recursive Greedy Algorithm (1/2)

- The procedure RECURSIVE-ACTIVITY-SELECTOR takes the start and finish times of the activities, represented as arrays s and f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem.
- We assume that the n input activities are already ordered by monotonically increasing finish time.

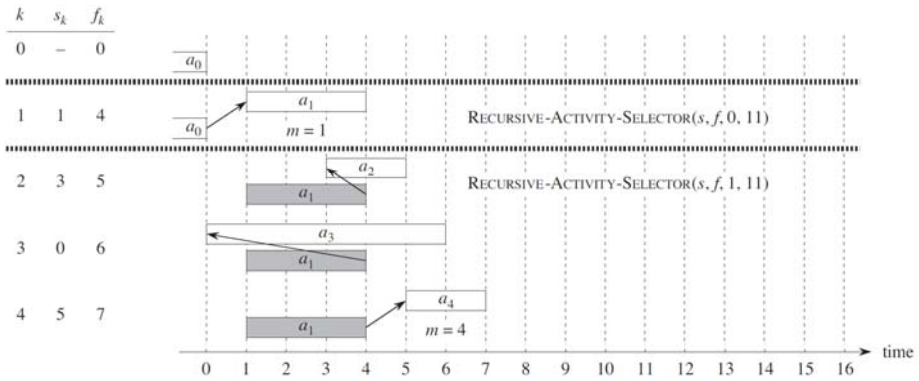
```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]   // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```


A Recursive Greedy Algorithm (2/2)

- In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S.
- The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n).
- **Example:** Operation of the Algorithm
- Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n) is $\Theta(n)$.

Operation of the Algorithm (1/3)

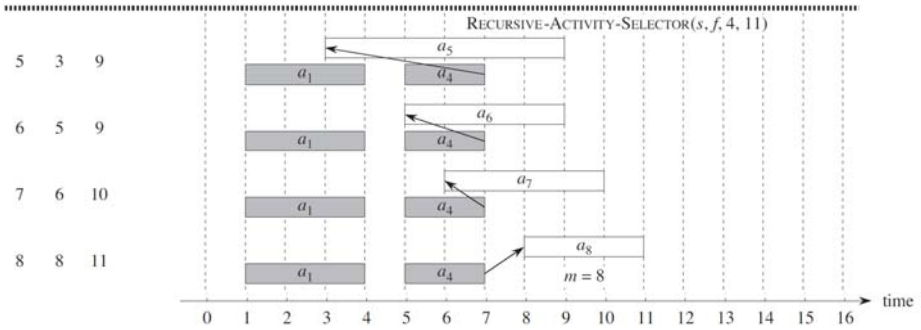
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Next

Operation of the Algorithm (2/3)

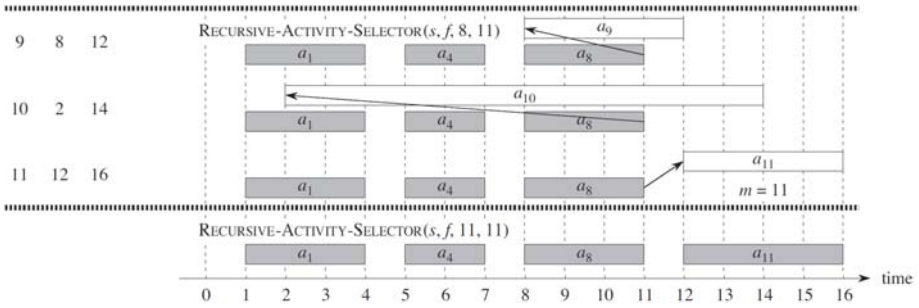
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Next

Operation of the Algorithm (3/3)

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



Back

An Iterative Greedy Algorithm

```
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
```

- Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Outline

- An Activity-Selection Problem
- ***Elements of the Greedy Strategy***
- Huffman Codes

Elements of the Greedy Strategy (1/3)

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
  - At each decision point, the algorithm makes choice that seems best at the moment.
- This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does.

➡ This section discusses some of the general properties of greedy methods.

Elements of the Greedy Strategy (2/3)

- We design greedy algorithms according to the following sequence of steps:

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 is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Elements of the Greedy Strategy (3/3)

- How can we tell whether a greedy algorithm will solve a particular optimization problem?

➡ No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients.

Greedy-Choice Property

- Greedy-choice property: when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.
- Dynamic Programming
  - solves the subproblems before making the first choice
  - in a bottom-up manner (or top down + memoizing)
- Greedy Algorithm
  - makes its first choice before solving any subproblems
  - in a top-down fashion
  - Of course, we must prove that a greedy choice at each step yields a globally optimal solution.

Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
- As an example, recall how we demonstrated in Activity-Selection Problem that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}.
- Based on this observation, we were able to devise the recurrence that described the value of an optimal solution.

Greedy vs. Dynamic Programming (1/3)

- To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem:
- The 0-1 knapsack problem: A thief robbing a store finds n items. The i th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take?

Greedy vs. Dynamic Programming (2/3)

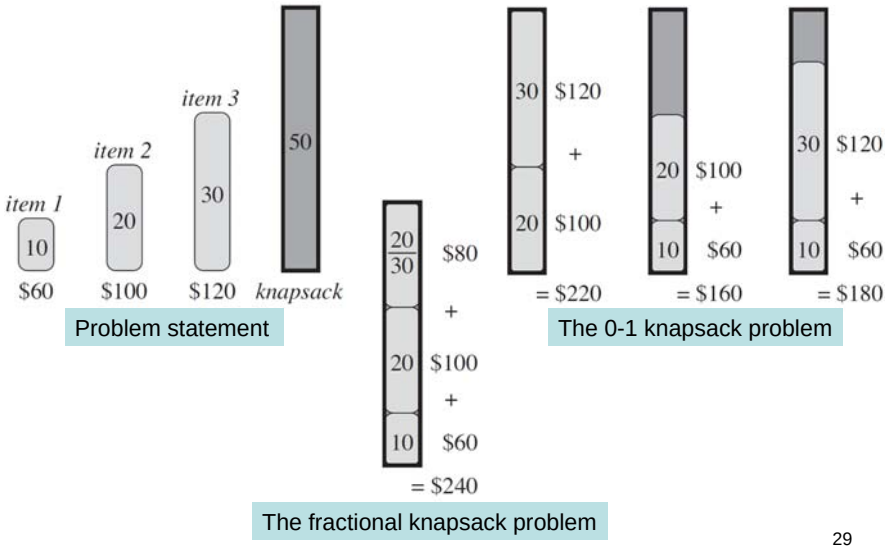
- The fractional knapsack problem: The setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.
- Both knapsack problems exhibit the optimal-substructure property. How?
- What kind of strategy should we use to solve the two problems?

Greedy vs. Dynamic Programming (3/3)

- Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy.
- For the fractional problem:
  1. Compute the value per pound v_i / w_i for each item.
  2. Obey a greedy strategy.

- Why this greedy strategy does not work for the 0-1 knapsack problem?

Example



Outline

- An Activity-Selection Problem
- Elements of the Greedy Strategy
- ***Huffman Codes***

Character-Coding Problem (1/2)

- Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given as follows:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

- How would you encode these characters?
  - Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string (codeword).

Character-Coding Problem (2/2)

	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

- If we use a fixed-length code, we need 3 bits to represent 6 characters!
  - It requires 300,000 bits to code the entire file.

Can we do better?

- A variable-length code can do considerably better by giving frequent characters short codewords and infrequent characters long codewords.
  - This code requires 224,000 bits to represent the file, a savings of approximately 25%.

Huffman Codes

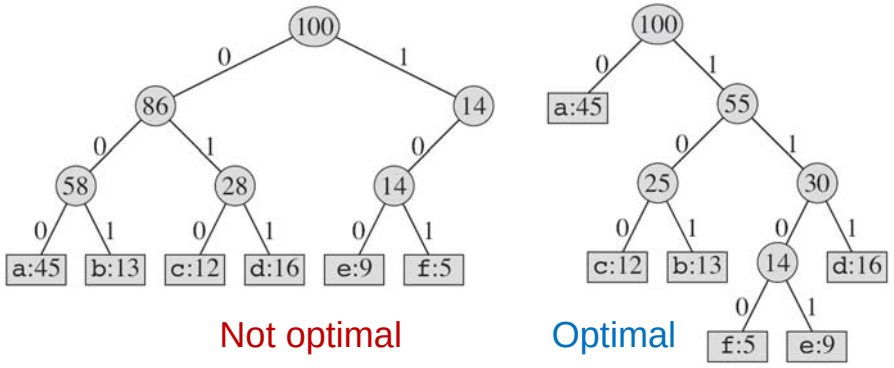
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.
- Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

Prefix Codes

- Prefix codes: no codeword is also a prefix of some other codeword.
- A prefix code can always achieve the optimal data compression among any character code, so we restrict our attention to prefix codes.
- Prefix codes are desirable because they simplify decoding (due to the characteristic of unambiguity).
- An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children

Trees Corresponding to the Coding Schemes (1/3)

	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



Not optimal

Optimal

Trees Corresponding to the Coding Schemes (2/3)

- If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet.
- How many internal nodes would be?

Trees Corresponding to the Coding Schemes (3/3)

- For each character c in the alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree (i.e., the length of the codeword for character c).
- The number of bits required to encode a file is thus

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

which we define as the cost of the tree T.

Constructing a Huffman Code

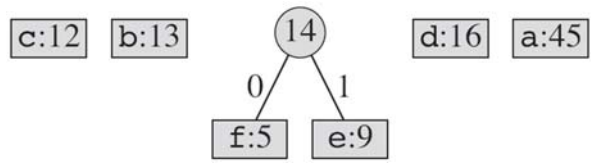
```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```


The Steps of Huffman's Algorithm (1/6)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

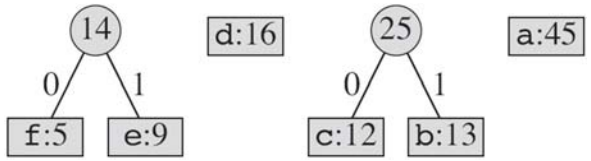
The Steps of Huffman's Algorithm (2/6)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



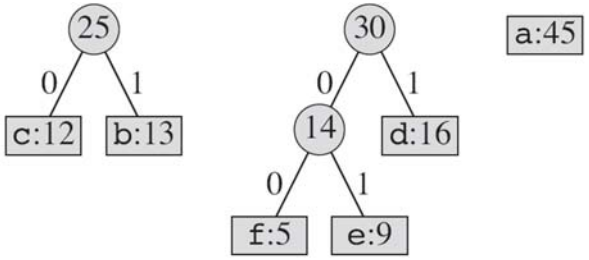
The Steps of Huffman's Algorithm (3/6)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



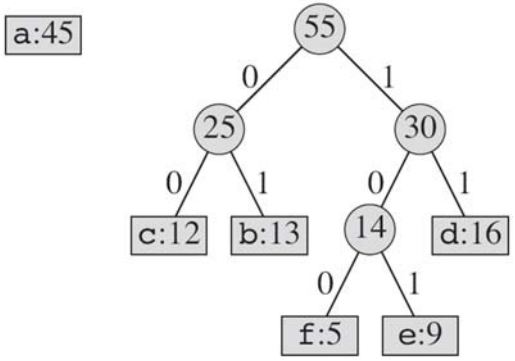
The Steps of Huffman's Algorithm (4/6)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



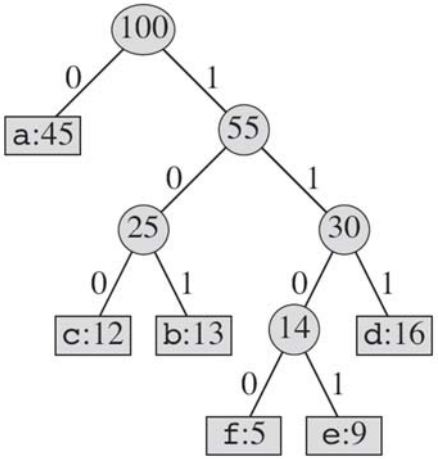
The Steps of Huffman's Algorithm (5/6)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



The Steps of Huffman's Algorithm (6/6)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

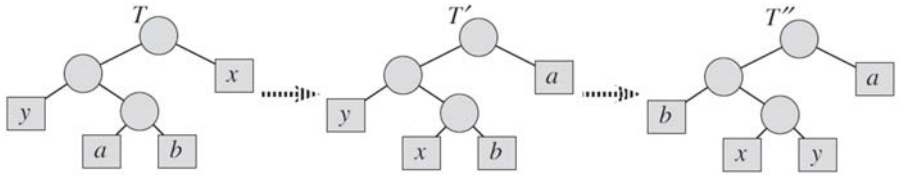


Correctness of Huffman's Algorithm (1/2)

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C 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 The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.



Correctness of Huffman's Algorithm (2/2)

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 the characters x and y removed and a new 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.

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

Proof Immediate from Lemmas 16.2 and 16.3.

Homework Assignment #3

Exercise 16.1-5

- TAs will announce the detailed Input/Output format in Moodle.
- Please submit your program to e-Tutor.
- Please submit your README document to Moodle.
- **Due Date: 26 April 2017**.