CSE 304
Design & Analysis of Algorithms

Greedy Algorithms (Part 1)

Greedy Algorithm

- Greedy algorithms make the choice that looks best at the moment.
- This locally optimal choice may lead to a globally optimal solution (i.e. an optimal solution to the entire problem).

When can we use Greedy algorithms?

We can use a greedy algorithm when the following are true:

1) **The greedy choice property:** A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

2) **The optimal substructure property:** The optimal solution contains within its optimal solutions to subproblems.

Designing Greedy Algorithms

1. Cast the optimization problem as one for which:
    - we make a choice and are left with only one subproblem to solve
2. Prove the GREEDY CHOICE
    - that there is always an optimal solution to the original problem that makes the greedy choice
3. Prove the OPTIMAL SUBSTRUCTURE:
    - the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

Example: Making Change

- Instance: amount (in cents) to return to customer
- Problem: do this using fewest number of coins
- Example:
  - Assume that we have an unlimited number of coins of various denominations:
    - 1c (pennies), 5c (nickels), 10c (dimes), 25c (quarters), 1$ (loonies)
  - Objective: Pay out a given sum $5.64 with the smallest number of coins possible.

The Coin Changing Problem

- Assume that we have an unlimited number of coins of various denominations:
  - 1c (pennies), 5c (nickels), 10c (dimes), 25c (quarters), 1$ (loonies)
- Objective: Pay out a given sum S with the smallest number of coins possible.

- The greedy coin changing algorithm:
  - This is a $\Theta(m)$ algorithm where m = number of denominations.

```
while S > 0 do
   c := value of the largest coin no larger than S;
   num := S / c;
   pay out num coins of value c;
   S := S - num*c;
```

Example: Making Change

- E.g.:

$5.64 = $2 +$2 + $1 +
.25 + .25 + .10 +
.01 + .01 + .01 +.01

Making Change – A big problem

- Example 2: Coins are valued \$.30, \$.20, \$.05, \$.01
  - Does not have greedy-choice property, since \$.40 is best made with two \$.20's, but the greedy solution will pick three coins (which ones?)

The Fractional Knapsack Problem

- **Given:** A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- **Goal:** Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize

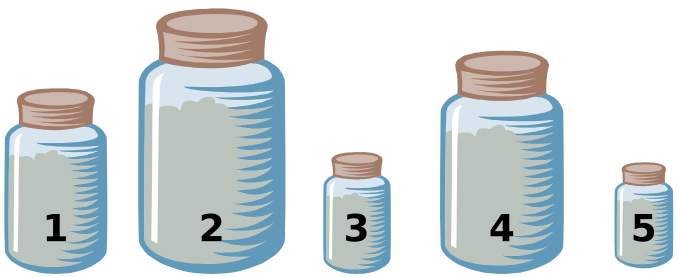
$$\sum_{i \in S} b_i (x_i / w_i)$$

  - Constraint:

$$\sum_{i \in S} x_i \leq W, 0 \leq x_i \leq w_i$$

Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with total weight at most W.



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: ($ per ml)	3	4	20	5	50

"knapsack"

10 ml

Solution: P

- 1 ml of 5 50$
- 2 ml of 3 40$
- 6 ml of 4 30$
- 1 ml of 2 4$

The Fractional Knapsack Algorithm

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$

Algorithm *fractionalKnapsack*(*S*, *W*)
Input: set S of items w/ benefit b_i and weight w_i; max. weight W
Output: amount x_i of each item i to maximize benefit w/ weight at most W

for *each item i in S*
    $x_i \leftarrow 0$
    $v_i \leftarrow b_i / w_i$ {value}
$w \leftarrow 0$ {total weight}
while $w < W$
    remove item i with highest v_i
    $x_i \leftarrow \min\{w_i, W - w\}$
    $w \leftarrow w + \min\{w_i, W - w\}$

The Fractional Knapsack Algorithm

- Running time: Given a collection S of n items, such that each item i has a benefit b_i and weight w_i, we can construct a maximum-benefit subset of S, allowing for fractional amounts, that has a total weight W in O(nlogn) time.
  - Use heap-based priority queue to store S
  - Removing the item with the highest value takes O(logn) time
  - In the worst case, need to remove all items

Huffman Codes

- Widely used technique for data compression
- Assume the data to be a sequence of characters
- Looking for an effective way of storing the data
- ***Binary character code***
  - Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

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

- 3 bits needed
- a = 000, b = 001, c = 010, d = 011, e = 100, f = 101
- Requires: $100{,}000 \cdot 3 = 300{,}000$ bits

Huffman Codes

- Idea:
  - Use the frequencies of occurrence of characters to build a optimal way of representing each character

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

Variable-Length Codes

E.g.: Data file containing 100,000 characters

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

- Assign short codewords to frequent characters and long codewords to infrequent characters
- a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
- $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000$

  $= 224{,}000$ bits

Prefix Codes

- Prefix codes:
  - Codes for which no codeword is also a prefix of some other codeword
  - Better name would be “prefix-free codes”
- We can achieve optimal data compression using prefix codes
  - We will restrict our attention to prefix codes

Encoding with Binary Character Codes

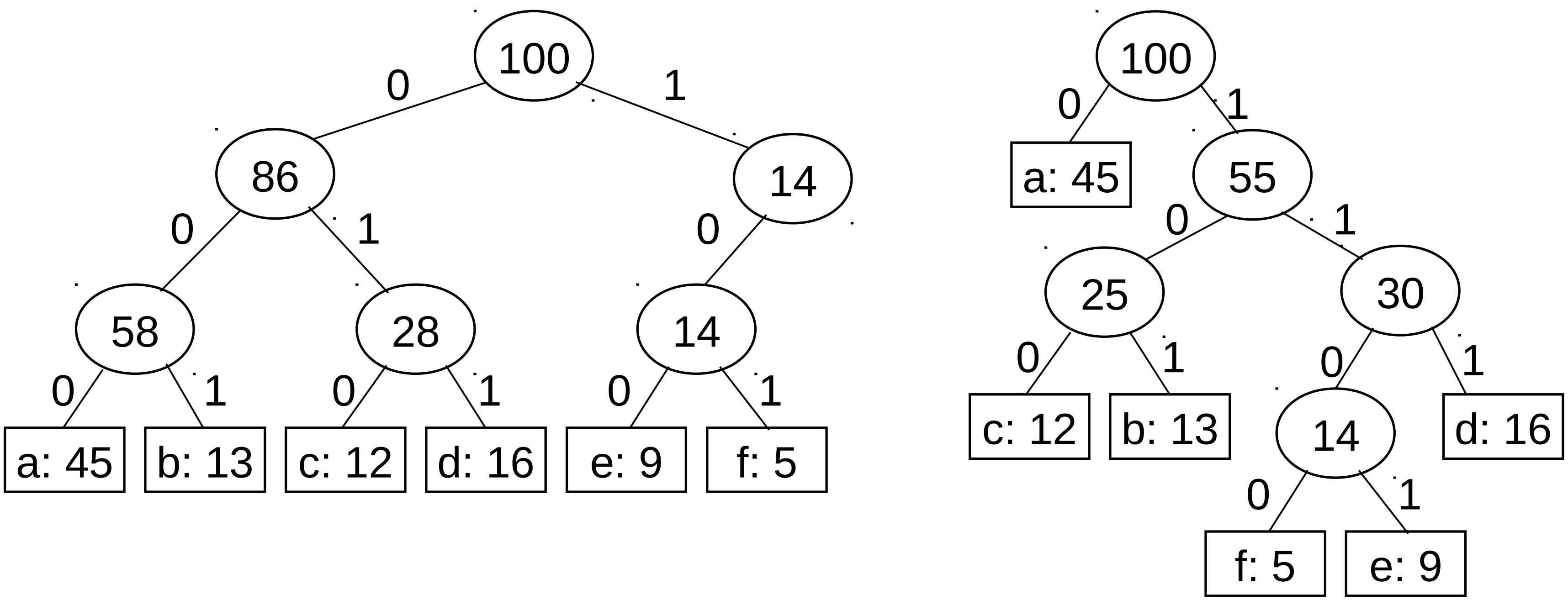
- Encoding
  - Concatenate the codewords representing each character in the file
- E.g.:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - abc = 0 · 101 · 100 = 0101100

Decoding with Binary Character Codes

- Prefix codes simplify decoding
  - No codeword is a prefix of another ⇒ the codeword that begins an encoded file is unambiguous
- Approach
  - Identify the initial codeword
  - Translate it back to the original character
  - Repeat the process on the remainder of the file
- E.g.:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - 001011101 = $0 \cdot 0 \cdot 101 \cdot 1101$ = aabe

Prefix Code Representation

- Binary tree whose leaves are the given characters
- Binary codeword
  - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”
- Length of the codeword
  - Length of the path from root to the character leaf (depth of node)



Optimal Codes

- An optimal code is always represented by a **full binary tree**
  - Every non-leaf has two children
  - Fixed-length code is not optimal, variable-length is
- How many bits are required to encode a file?
  - Let C be the alphabet of characters
  - Let f(c) be the frequency of character c
  - Let $d_T(c)$ be the depth of c's leaf in the tree T corresponding to a prefix code

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$ the cost of tree T

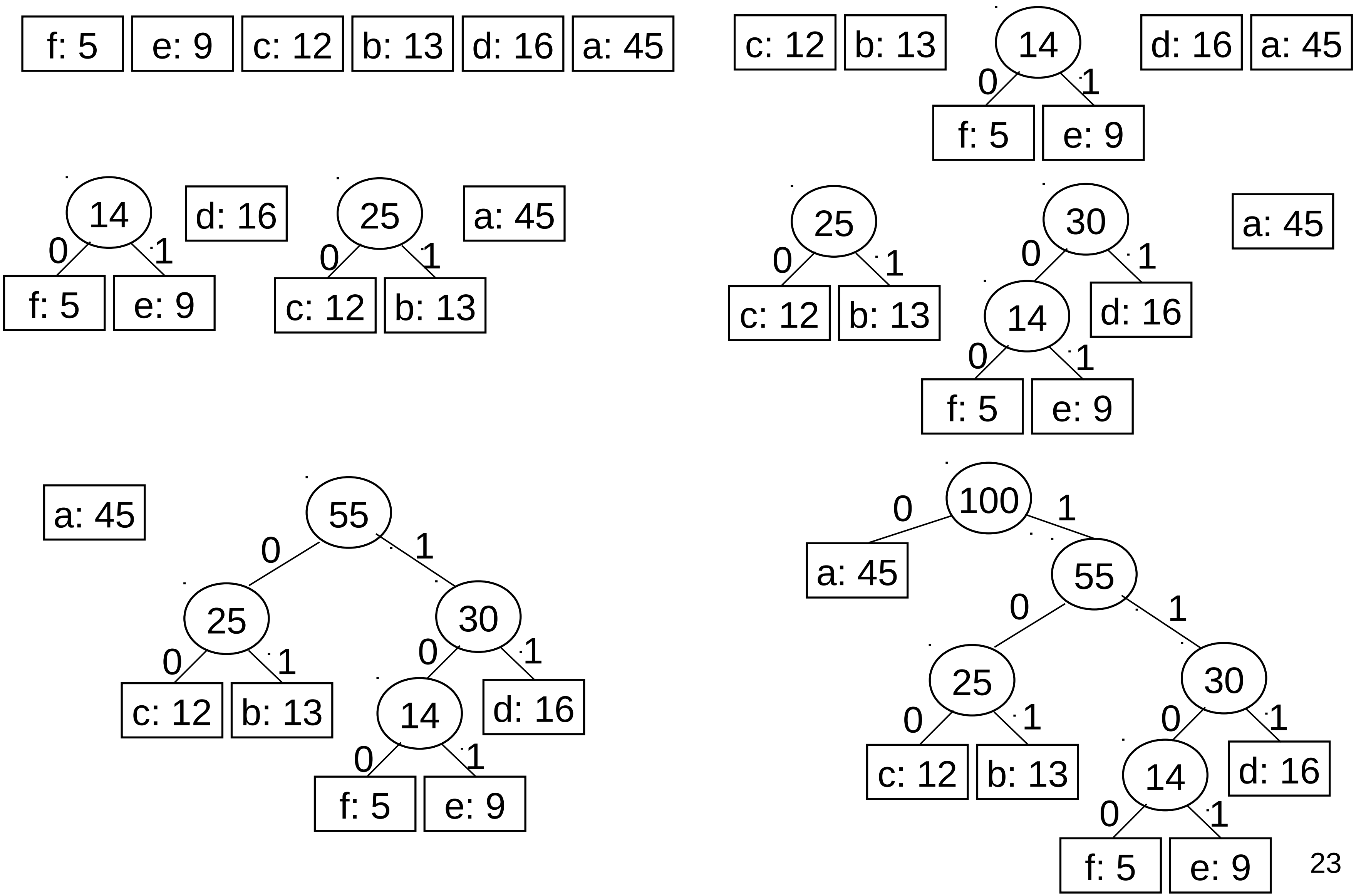
Constructing a Huffman Code

- A greedy algorithm that constructs an optimal prefix code called a **Huffman code**
- Assume that:
  - C is a set of n characters
  - Each character has a frequency f(c)
  - The tree T is built in a bottom up manner
- Idea:

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

  - Start with a set of |C| leaves
  - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
  - Use a min-priority queue Q, keyed on f to identify the two least frequent objects

Example



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

Building a Huffman Code

Alg.: HUFFMAN(C)

Running time: O(nlgn)

$n \leftarrow |C|$

$Q \leftarrow C$ ← O(n)

for $i \leftarrow 1$ **to** $n - 1$

    do allocate a new node z

        $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN(Q)

        $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN(Q)

        $f[z] \leftarrow f[x] + f[y]$

        INSERT (Q, z)

O(nlgn)

return EXTRACT-MIN(Q)