Ontario Tech Science

Kourosh Davoudi
kourosh@ontariotechu.ca

Lecture 7: Greedy Algorithms

CSCI 3070U: Design and Analysis of Algorithms

Learning Outcomes

- Greedy Algorithm Strategy
- Case Studies:
  - Counting Coins
  - Fractional Knapsack
  - Huffman Code

Greedy Algorithm Foundations

- What are the ideas of greedy approach?
  - 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*.
- Do greedy algorithms always result in optimal solutions?
  - Answer: NO !

Greedy Algorithm Foundations

- Why greedy Approaches?
  - Sometimes they are optimum
  - If they do not, the solutions are usually near-optimal (approximation)
- So, what is their advantage?
  - They are often tractable/doable solution

Greedy Algorithm Example

- Let’s say I have an amount of change to give a customer (e.g. $3.79)
  - How do I figure out the optimal coin arrangement?
    - i.e. least number of coins

- **One Solution:** Starting with the largest coin and working toward the smallest
  - Use as many (including zero) of that coin is possible
  - Continue until the remainder is $0.00

Example: Counting Coins: $3.79

- How many $2 coins? One
  - Remainder: $1.79
- How many $1 coins? One
  - Remainder: $0.79
- How many $0.25 coins? Three
  - Remainder: $0.04
- How many $0.01 coins? Four
  - Remainder: $0.00

When greedy algorithms are globally optimal?

- Greedy algorithms find optimal solutions for problems that have optimal substructure
  - When globally optimal solutions can be created by combining locally optimal solutions
  - The greedy selection is a part of the optimal solution

- Example:
  - Counting coins
    - For example if you have $8, the greedy algorithm chooses $2 coin as the solution and resulting the problem of $6. The optimal problem says that we can the solution is a $2 coin + solution of problem of $6.

Case Study: Knapsack Algorithm (recap)

- Problem:
  - We have a knapsack with capacity W, and a number of item, where each item has a weight, and a value
  - Objective is to select the items with maximum total value and putting them in knapsack

- Variations:
  - 0/1: At most one of each item weight/value can be included
  - Fractional: We can divide items and take a part of them, for part of the value

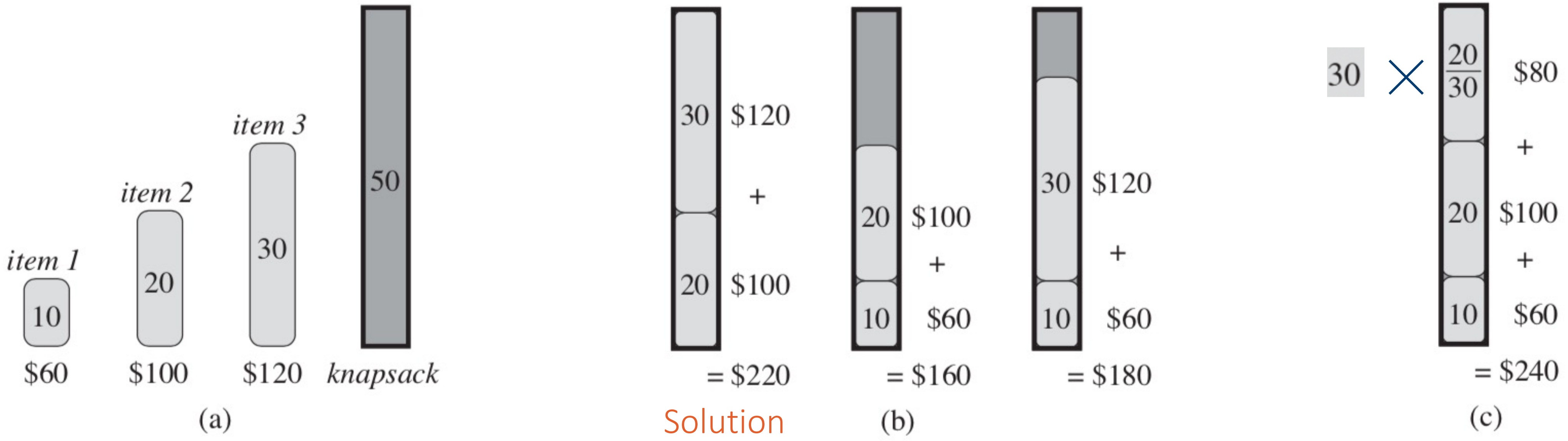
0/1 - Knapsack problem (recap)

- Brute Force Algorithm:
  - Try all combinations of the n items
  - Find the maximum value of the combinations
    - Number of combinations?

$$\Theta(2^n)$$

- Obviously, this is no good !
- Solution:
  - Dynamic Programming (optimal)
  - Branch and Bound (optimal)
  - Greedy (not optimal)

Fractional vs. 0/1 Knapsack



item 1: $60/10 = 6$

item 2: $100/20 = 5$

item 3: $120/30 = 4$

For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

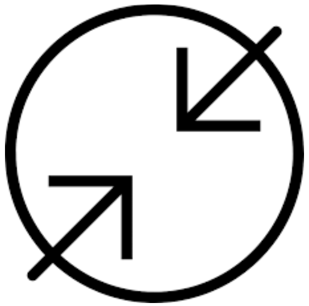
Fractional Knapsack

- Solution:
  - Greedy algorithm (optimal)

- General scheme:
  - Calculate the value per unit of weight
    - We'll call this the unit value
  - Choose the items in order of their unit value, if we have room for them

Case Study: Huffman Code

- Compression has a goal of reducing the required number of bits to store/transmit a sequence of symbols
- Huffman codes compress data very effectively.



Case Study: Huffman Code

- 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:

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

- How to assign a unique binary string to each character to minimize the total file length?

Case Study: Huffman Code

- How to assign a unique binary string to each character to minimize the total file length?
  - Fixed-length code: we need 3 bits to represent 6 characters

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

This method requires 300,000 bits to code the entire file

Case Study: Huffman Code

- How to assign a unique binary string to each character to minimize the total file length?

  - variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long code- words.

	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

$$(45 \cdot 1 \ + \ 13 \cdot 3 \ + \ 12 \cdot 3 \ + \ 16 \cdot 3 \ + \ 9 \cdot 4 \ + \ 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

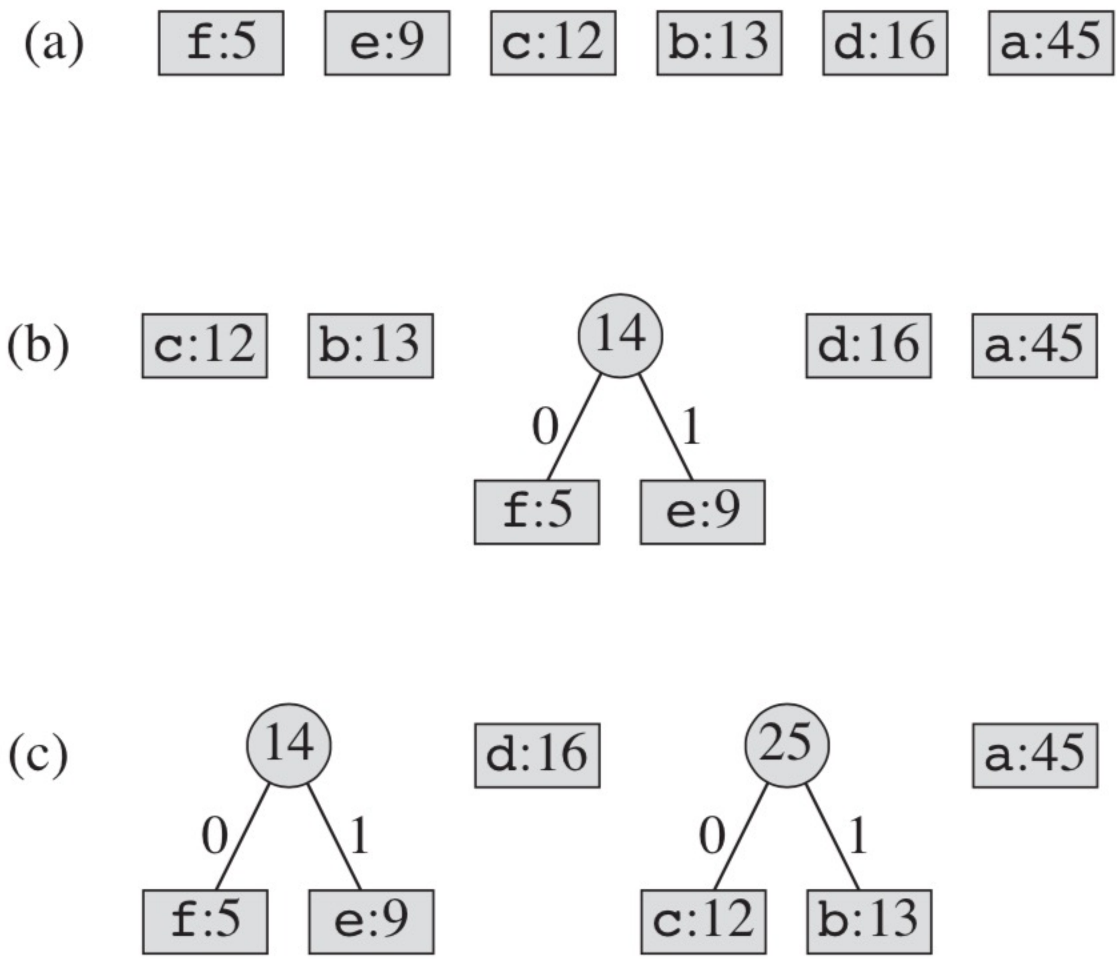
Case Study: Huffman Code

- What property such variable length codes should have?

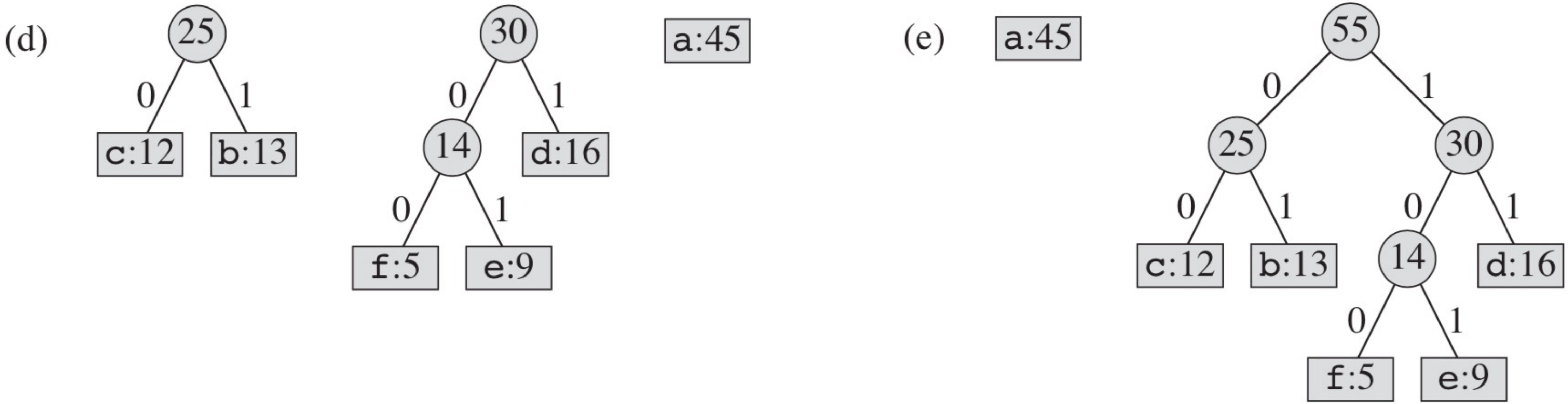
No codeword is a prefix of some other codeword

- How to produce such variable codes (optimum codes)?
  - Huffman's algorithm is an efficient algorithm for finding prefix codes
  - Huffman's algorithm makes use of a priority queue

Huffman Code: Example

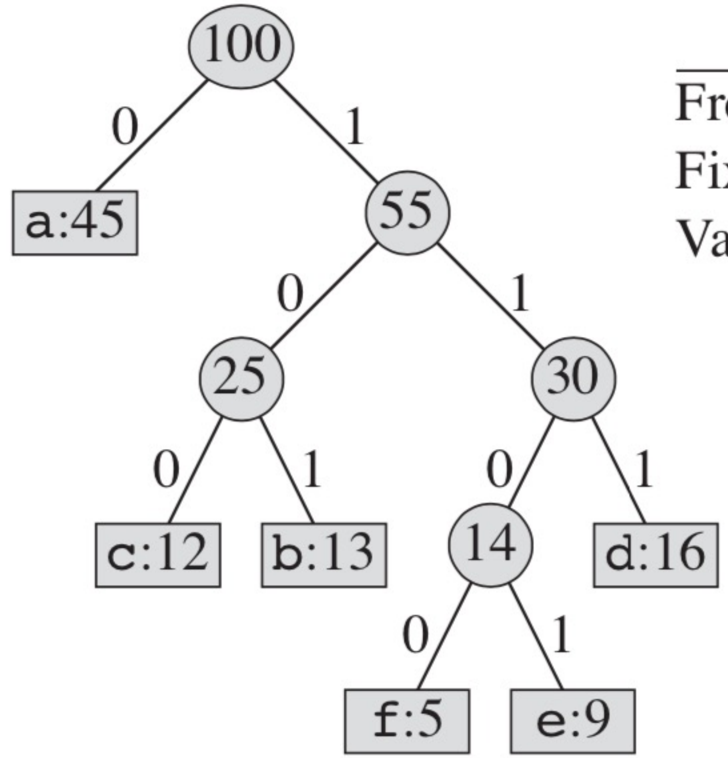


Huffman Code: Example



Huffman Code: Example

(f)



	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

Huffman's Algorithm

Time Complexity: $O(n \log n)$

HUFFMAN(C)

$n = |C|$

$Q = C$ $\quad O(n)$

for $i = 1$ **to** $n - 1$

  allocate a new node z

  $z.left = x =$ EXTRACT-MIN(Q) $\quad O(\log n)$

  $z.right = y =$ EXTRACT-MIN(Q) $\quad O(\log n)$

  $z.freq = x.freq + y.freq$

  INSERT(Q, z) $\quad O(\log n)$

return EXTRACT-MIN(Q) // return the root of the tree $\quad O(\log n)$

Wrap-up

- We learned
  - Foundation of greedy algorithm
  - How greedy approach can provide optimum solution in certain cases:
    - Counting Coins
    - Fractional Knapsack
    - Huffman Codes