Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 20

The Knapsack Problem

- **The 0-1 knapsack problem**
  - A thief robbing a store finds n items: the i-th item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)
  - The thief can only carry W pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The fractional knapsack problem**
  - Similar to above
  - The thief can take fractions of items

Fractional Knapsack Problem

- Knapsack capacity: W
- There are n items: the i-th item has value v_i and weight w_i
- Goal:
  - Find fractions x_i so that for all $0 \le x_i \le 1$, $i = 1, 2, .., n$

    $\sum w_i x_i \le W$ and

    $\sum x_i v_i$ is maximum

Fractional Knapsack Problem

- Greedy strategy 1:
  - Pick the item with the maximum value
- *E.g.:*
  - $W = 1$
  - $w_1 = 100, \ v_1 = 2$
  - $w_2 = 1, v_2 = 1$
  - Taking from the item with the maximum value:

    Total value (choose item 1) = $v_1 W/w_1 = 2/100$
  - Smaller than what the thief can take if choosing the other item

    Total value (choose item 2) = $v_2 W/w_2 = 1$

Fractional Knapsack Problem

- Greedy strategy 2:
  - Pick the item with the maximum value per pound v_i/w_i
  - If the supply of that element is exhausted and the thief can carry more: take as much as possible from the item with the next greatest value per pound
  - It is good to order items based on their value per pound

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Fractional Knapsack Problem

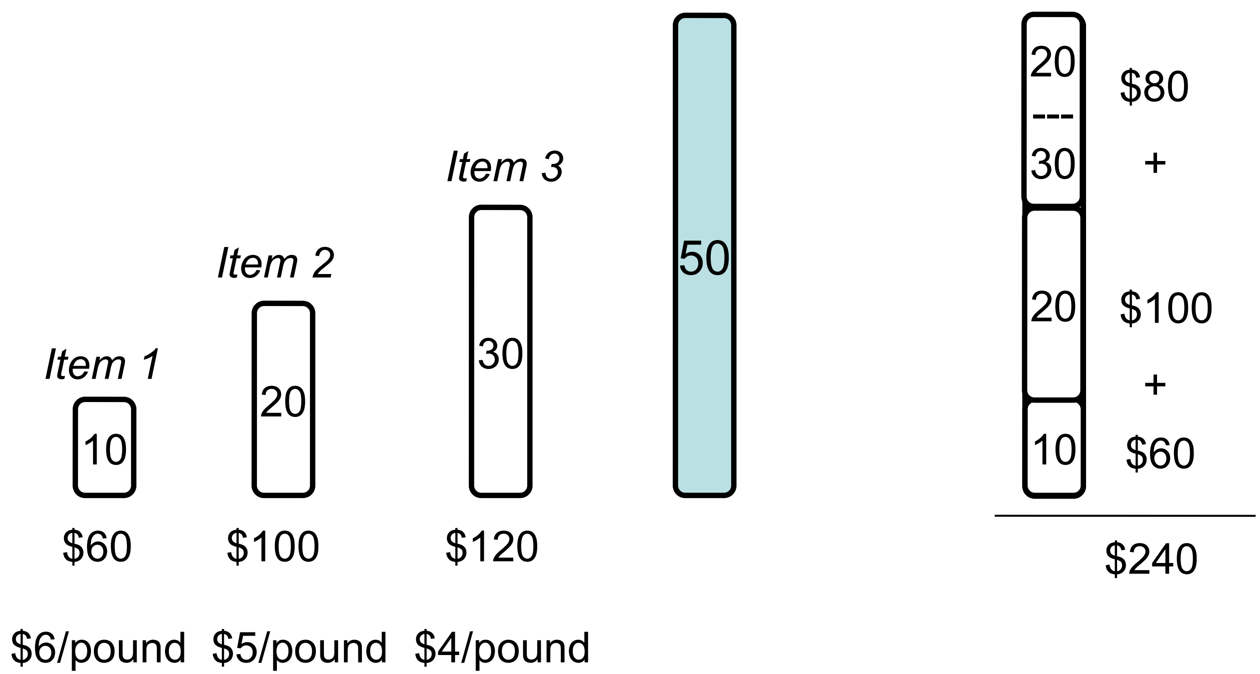
Alg.: Fractional-Knapsack (**W, v[n], w[n]**)

1. w = W
2. While **w > 0** and there are items remaining
3. pick item **i** with maximum v_i/w_i
4. $x_i \leftarrow \min(1, w/w_i)$
5. remove item **i** from list
6. $w \leftarrow w - x_i w_i$

- **w** – the amount of space remaining in the knapsack
- Running time: $\Theta(n)$ if items already ordered; else $\Theta(n\lg n)$

Fractional Knapsack - Example

- *E.g.:*



Greedy Choice

Items: 1 2 3 … j … n

Optimal solution: x_1 x_2 x_3 x_j x_n

Greedy solution: x_1' x_2' x_3' x_j' x_n'

- We know that: $x_1' \geq x_1$
  - greedy choice takes as much as possible from item 1
- Modify the optimal solution to take x_1' of item 1
  - We have to decrease the quantity taken from some item j: the new x_j is decreased by: $(x_1' - x_1)\ w_1\ /w_j$
- Increase in profit: $(x_1' - x_1)\ v_1$
- Decrease in profit: $(x_1' - x_1)\ w_1\ v_j/w_j$

$$(x_1' - x_1)\ v_1 \geq (x_1' - x_1)\ w_1\ v_j/w_j$$

$$v_1 \geq w_1\ v_j/w_j \quad \Rightarrow \quad v_1/w_1 \geq v_j/w_j$$

True, since x_1 had the best value/pound ratio

The 0-1 Knapsack Problem

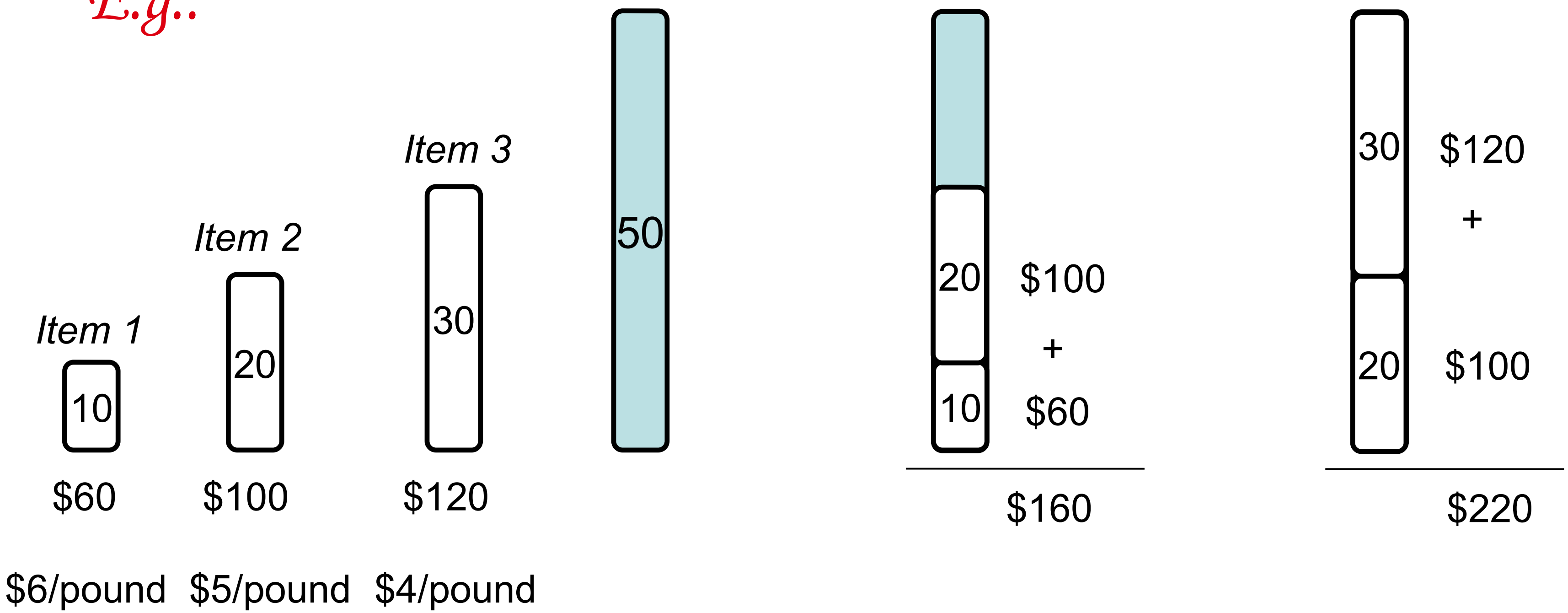
- Thief has a knapsack of capacity W
- There are n items: for i-th item value v_i and weight w_i
- Goal:
  - Find coefficients x_i so that for all $x_i = \{0, 1\}$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

0-1 Knapsack - Greedy Strategy

- *E.g.:*



- None of the solutions involving the greedy choice (item 1) leads to an optimal solution
  - The greedy choice property does not hold

0-1 Knapsack - Dynamic Programming

- $P(i, w)$ – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

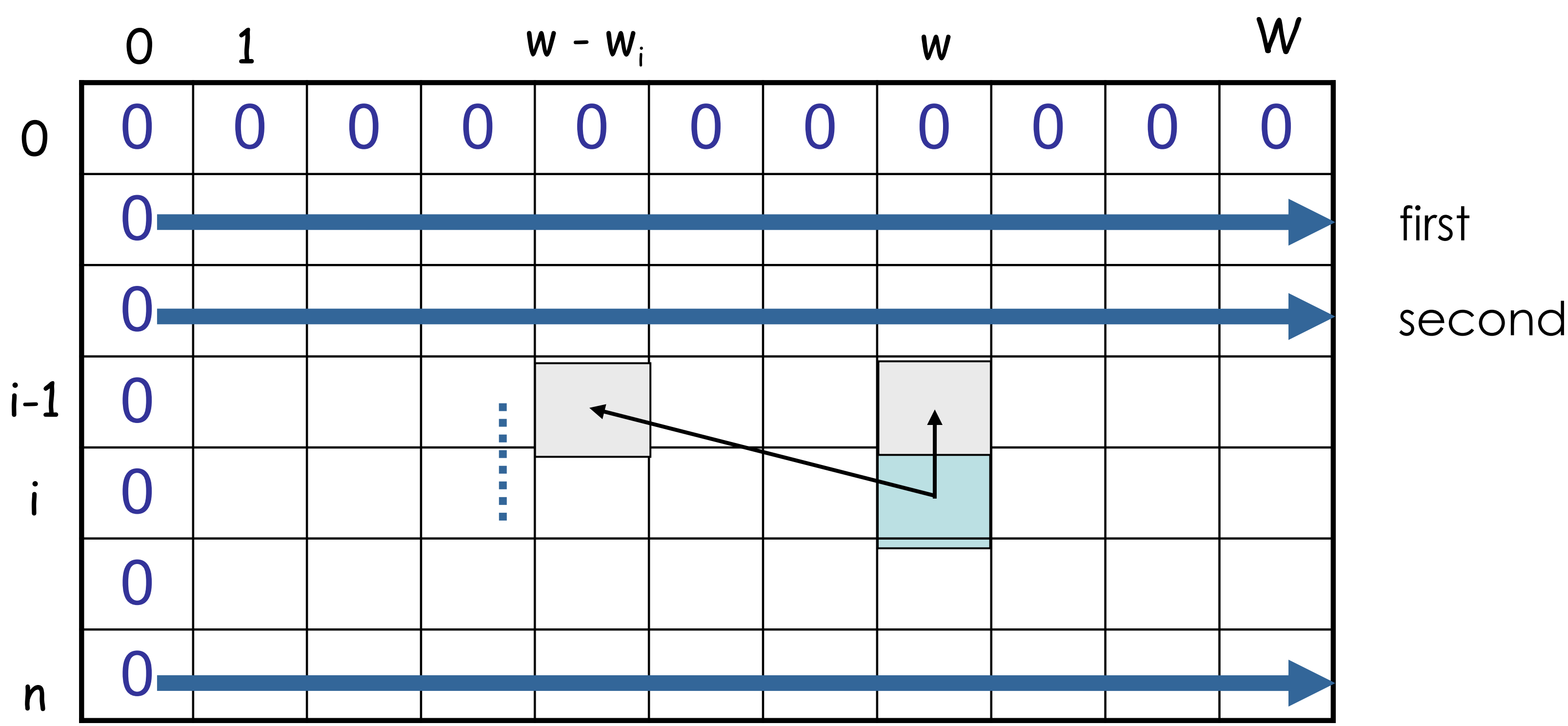
$$P(i, w) = v_i + P(i - 1, w-w_i)$$

- Case 2: thief does not take item i

$$P(i, w) = P(i - 1, w)$$

0-1 Knapsack - Dynamic Programming

$$P(i, w) = \max\{\underbrace{v_i + P(i-1, w-w_i)}_{\text{Item i was taken}}, \underbrace{P(i-1, w)}_{\text{Item i was not taken}}\}$$



Example:

$$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(3, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at P(n, W)
- When you go left-up ⇒ item i has been taken
- When you go straight up ⇒ item i has not been taken

Optimal Substructure

- Consider the most valuable load that weights at most **W** pounds
- If we remove item **j** from this load

⇒ The remaining load must be the most valuable load weighing at most **$W - w_j$** that can be taken from the remaining **$n - 1$** items

Overlapping Subproblems

$$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

E.g.: all the subproblems shown in grey may depend on **P(i-1, w)**

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 × 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

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

$= 224{,}000$ 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

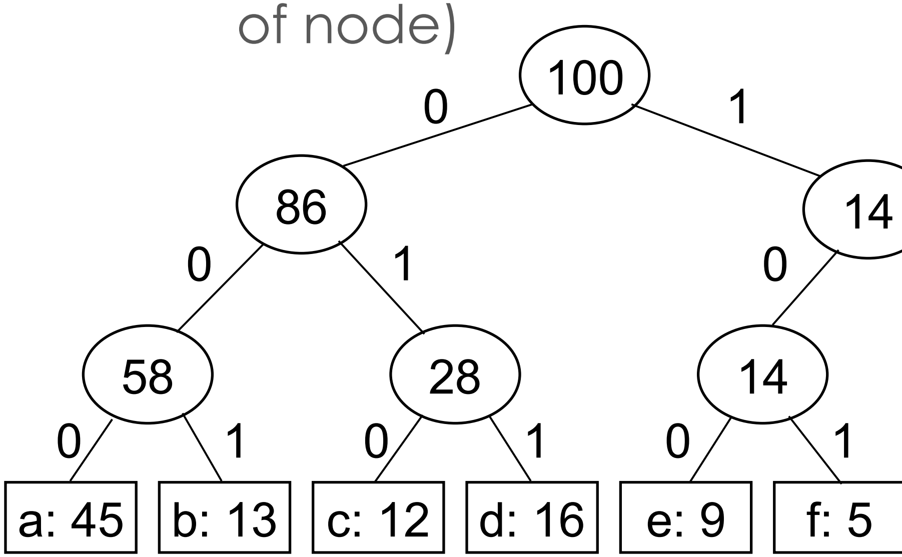
- Encoding
  - Concatenate the codewords representing each character in the file
- *E.g.*:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - abc = 0 × 101 × 100 = 0101100

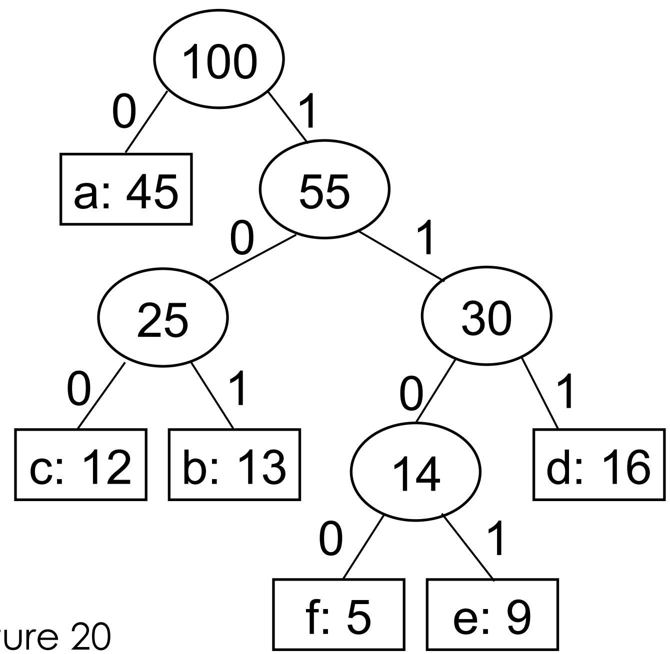
Decoding with Binary Character Codes

- 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 = $0 \times 0 \times 101 \times 1101$ = 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

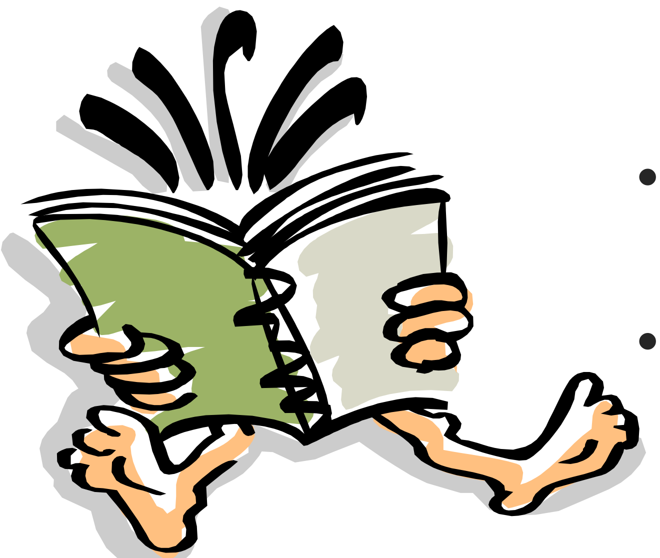
Constructing a Huffman Code

- Let's build 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 $f(c)$
- Idea:

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

  - The tree T is built in a bottom up manner
  - Start with a set of $|C| = n$ 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

Readings



- For this lecture
  - Chapter 15
- Coming next
  - Chapter 15