# CS 2500: Algorithms

## Lecture 20: Greedy Algorithms: Huffman Coding and Fractional Knapsack

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

October 24, 2024

# Fractional Knapsack Problem

- We are given $n$ objects and a knapsack (or bag).
- Object $i$ has a weight $w_i$ and the knapsack has a capacity $m$.
- If a fraction $x_i$, $0 \leq x_i \leq 1$, of object $i$ is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to fill the knapsack to maximize the total profit.
- The total weight of chosen objects must not exceed the knapsack capacity $m$.

# Fractional Knapsack Problem

**Objective:**

$$\text{Maximize} \sum_{1 \le i \le n} p_i x_i \tag{1}$$

**Subject to:**

$$\sum_{1 \le i \le n} w_i x_i \le m \tag{2}$$

$$0 \le x_i \le 1, \quad 1 \le i \le n \tag{3}$$

- Profits and weights are positive numbers.
- A feasible solution is any set $(x_1, \ldots, x_n)$ satisfying the constraints.
- An optimal solution maximizes the total profit.

# Greedy Strategies for the Fractional Knapsack Problem

- The problem involves selecting a subset of objects and fractions $x_i$.
- Greedy strategies:
  - Fill the knapsack starting with the object that **provides the highest profit** $p_i$.
  - Use the object with the **lowest weight** $w_i$.
  - Consider the **highest profit-to-weight ratio** $p_i/w_i$.
- Only the third strategy (highest $p_i/w_i$) produces an optimal solution.

# Fractional Knapsack Problem: Example 1

Consider the following instance:

- $n = 3, m = 20$
- Profits: $(p_1, p_2, p_3) = (25, 24, 15)$
- Weights: $(w_1, w_2, w_3) = (18, 15, 10)$

| | $x_1, x_2, x_3$ | $\sum w_i x_i$ | $\sum p_i x_i$ | Comments |
|---|---|---|---|---|
| 1 | (1/2, 1/3, 1/4) | 16.5 | 24.25 | Random |
| 2 | (1, 2/15, 0) | 20 | 28.2 | According to profit |
| 3 | (0, 2/3, 1) | 20 | 31 | By least weight |
| 4 | (0, 1, 1/2) | 20 | 31.5 | By profit-to-weight ratio |

Table: Feasible Solutions

- Solution 4 is optimal with a profit of 31.5.

# Observations and Lemmas

**Lemma 4.1:** If the sum of all weights $\leq m$, then $x_i = 1$ for all $i$ is an optimal solution.

- If weights fit within the capacity, choose all items fully.

**Lemma 4.2:** All optimal solutions will fill the knapsack exactly.

- True because we can increase contributions of objects fractionally until reaching full capacity.

# Greedy Strategy: Adjusting for Maximum Profit

- Each time an object is included, except possibly the last one, we attempt to maximize profit by filling the knapsack.
- If only a fraction of the last object is included, sometimes a better increase can be obtained by using a different object.
- **Example:**
  - Suppose there are 2 units of space left.
  - Two objects: $(p_i = 4, w_i = 4)$ and $(p_j = 3, w_j = 2)$.
  - Using $j$ is better than using half of $i$.
- This strategy ensures that at each step, the largest possible increase in profit value is obtained.

# Greedy Strategy: Adjusting for Maximum Profit

- Object 1 has the largest profit $p_1 = 25$.
- It is placed into the knapsack first: $x_1 = 1$, earning 25.
- 2 units of capacity remain.
- Object 2 has the next highest profit $p_2 = 24$, but $w_2 = 15$ does not fit.
- Using $x_2 = 2/15$, we can fill the knapsack exactly.
- The result is a total profit of 28.2, which is suboptimal.
- The method described is termed a greedy approach, but it **does not always yield the optimal solution**.

# Other Greedy Approaches

- Different strategies for selecting objects include:
  - Considering objects in order of nonincreasing profit values.
  - Considering objects in order of nondecreasing weights (assumption: more items = more profit)
- Both strategies can lead to suboptimal solutions as demonstrated in previous examples.
- Optimal solution strategy:
  - Balance the rate at which profit increases and the rate at which capacity is used.
  - Choose objects ordered by the **profit-to-weight ratio** $p_i/w_i$.
- Example 1 produces the optimal solution when using this strategy.

# Explanation of Profit-to-Weight Ratio Strategy

- This strategy ensures that the objects with the maximum profit per unit weight are included first.
- Objects are sorted in descending order of $p_i/w_i$, ensuring that at each step, the profit increase is maximized while using capacity efficiently.
- Note that solutions corresponding to this strategy can be obtained using Algorithm GreedyKnapsack.

# Why the Profit-to-Weight Strategy is Optimal

- Choosing objects in order of $p_i/w_i$ ensures that each unit of capacity is used to maximize profit.
- This is particularly effective when the knapsack must be filled precisely, as it balances the rate of profit increase and capacity consumption.
- Disregarding the initial sort, the algorithm runs in $O(n)$ time.

# Greedy Knapsack Algorithm

**Algorithm GreedyKnapsack(p,w,m)**

1: Sort objects by $p_i/w_i$ in descending order
2: $profit \leftarrow 0, capacity \leftarrow m$
3: **for** each object $i$ in sorted order **do**
4:     **if** $w_i \leq capacity$ **then**
5:         Take full object $i$
6:         $profit \leftarrow profit + p_i$
7:         $capacity \leftarrow capacity - w_i$
8:     **else**
9:         Take fraction of $i$ to fill $capacity$
10:         $profit \leftarrow profit + p_i \cdot (capacity/w_i)$
11:         $capacity \leftarrow 0$
12:         **break**
13:     **end if**
14: **end for**

# Fractional Knapsack Problem: Example 2

**Problem Statement:**

- There are three sugar bottles with their weights and profits as shown.
- The capacity of the bigger bottle (or knapsack) is 20 kg.
- Apply a knapsack greedy algorithm and show the optimal order of items.

**Data:**

| Item | Weight (kg) | Profit ($) |
| --- | --- | --- |
| 1 | 14 | 24 |
| 2 | 18 | 20 |
| 3 | 10 | 16 |

**Approach:**

- Calculate the profit-to-weight ratio $(p_i/w_i)$ for each item.
- Sort the items in descending order based on $p_i/w_i$.
- Select items in this order, filling the knapsack until it is full.

# Fractional Knapsack Problem: Example 2

**Step-by-Step Process:**

1. **Profit-to-Weight Ratios:**

$$\frac{24}{14} = 1.71,\ \frac{20}{18} = 1.11,\ \frac{16}{10} = 1.6$$

2. **Order of Items:** 1, 3, 2
3. **Load Items:**
   - Load item 1 (14 kg, $24).
   - Remaining capacity: $20 - 14 = 6$ kg.
   - Load fraction of item 3, i.e., $\frac{6}{10} = 0.6$ of item 3.
   - Profit from item 3: $0.6 \times 16 = 9.6$.
4. **Total Profit:**

$$24 + 9.6 = 33.6$$

# Huffman Coding

- Developed by David Huffman in 1951.
- Variable-length code.
- Frequently used in data compression to minimize the average length of codes.
- Efficiently assigns shorter codes to symbols with higher frequencies and longer codes to those with lower frequencies.

**Applications:**

- Data compression (e.g., ZIP files, MP3, JPEG).
- Efficient communication encoding.

# Huffman Coding

**Data Compression:**

- Huffman coding is a method used for compressing data efficiently.
- Typical savings range from 20% to 90% depending on the data characteristics.
- Data arrives as a sequence of characters, and Huffman's algorithm builds an optimal way of representing each character as a binary string.

**Key Concept:**

- Frequent characters are assigned shorter codes.
- Infrequent characters are given longer codes.
- This approach minimizes the total number of bits required to store the data.

# Huffman Coding

**Procedure:**

1. Sort symbols by frequency.
2. Combine the two symbols with the lowest frequencies.
3. Assign a '0' to the left branch and '1' to the right branch.
4. Repeat until only one node remains, forming the Huffman Tree.

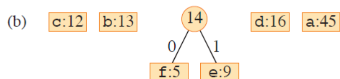
**Properties:**

- Prefix-free code: No code is a prefix of another.
- Optimal for minimizing average code length.

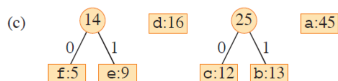
# Huffman Code: Example

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

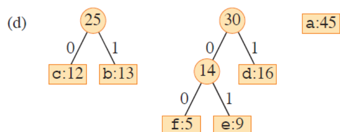
(b) c:12 b:13



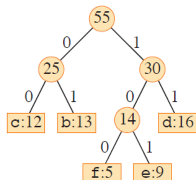

(c)



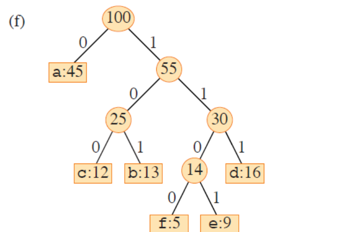

(d)




(e) a:45




(f)



# Huffman Code: Priority Queue

**What is a Priority Queue?**

- A **priority queue** is a data structure where each element has a priority associated with it.
- Elements are served based on their priority, not the order they arrive.
- Higher priority elements are processed before lower priority ones.
- Can be implemented using **binary heaps**.

**Priority Queue in Huffman Coding:** The priority queue is used to store characters (or nodes) based on their frequencies. Characters with lower frequencies are given higher priority, so they are processed first when building the Huffman tree.

# Huffman Code: Algorithm

**Algorithm HuffmanCoding(frequencies)**

1: Initialize a priority queue $Q$ with all characters and their frequencies
2: **while** $Q$ has more than one node **do**
3: Extract two nodes $x$ and $y$ with the smallest frequencies from $Q$
4: Create a new node $z$ with $\mathit{frequency}(z) = \mathit{frequency}(x) + \mathit{frequency}(y)$
5: Set $x$ as the left child of $z$ and $y$ as the right child of $z$
6: Insert $z$ back into $Q$
7: **end while**
8: The remaining node in $Q$ is the root of the Huffman Tree
9: **Return** the Huffman Tree

# Huffman Codes: Complexity

- Construction of the priority queue: $O(n)$
- Extraction and merging operations: $O(n \log n)$
- Total Time Complexity: $O(n \log n)$