The Greedy Method and Text Compression

The Greedy Method Technique

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize

- It works best when applied to problems with the **greedy-choice** property:
  - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.
Text Compression

- Given a string $X$, efficiently encode $X$ into a smaller string $Y$
  - Saves memory and/or bandwidth
- A good approach: **Huffman encoding**
  - Compute frequency $f(c)$ for each character $c$.
  - Encode high-frequency characters with short code words
  - No code word is a prefix for another code
  - Use an optimal encoding tree to determine the code words

Encoding Tree Example

- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)
Encoding Tree Optimization

- Given a text string $X$, we want to find a prefix code for the characters of $X$ that yields a small encoding for $X$.
  - Frequent characters should have long code-words.
  - Rare characters should have short code-words.
- Example
  - $X = \text{abracadabra}$
  - $T_1$ encodes $X$ into 29 bits
  - $T_2$ encodes $X$ into 24 bits

Huffman’s Algorithm

- Given a string $X$, Huffman’s algorithm construct a prefix code that minimizes the size of the encoding of $X$.
- It runs in time $O(n + d \log d)$, where $n$ is the size of $X$ and $d$ is the number of distinct characters of $X$.
- A heap-based priority queue is used as an auxiliary structure.
Huffman’s Algorithm

**Algorithm** Huffman($X$):

**Input:** String $X$ of length $n$ with $d$ distinct characters

**Output:** Coding tree for $X$

1. Compute the frequency $f(c)$ of each character $c$ of $X$.
2. Initialize a priority queue $Q$.
3. **for each** character $c$ in $X$ **do**
   - Create a single-node binary tree $T$ storing $c$.
   - Insert $T$ into $Q$ with key $f(c)$.
4. **while** $\text{len}(Q) > 1$ **do**
   - $(f_1, T_1) = Q.\text{remove}\_\text{min}()$
   - $(f_2, T_2) = Q.\text{remove}\_\text{min}()$
   - Create a new binary tree $T$ with left subtree $T_1$ and right subtree $T_2$.
   - Insert $T$ into $Q$ with key $f_1 + f_2$.
5. $(f, T) = Q.\text{remove}\_\text{min}()$

**return** tree $T$

---

Example

$X = $ abracadabra

Frequencies

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Initial Frequencies

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Final Frequencies

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
The Fractional Knapsack Problem (not in book)

- **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 $\sum b_i \left( \frac{x_i}{w_i} \right)$
    - **Constraint:** $\sum x_i \leq W$

© 2014 Goodrich, Tamassia, Goldwasser  Greedy Method 10
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 weight at most $W$.

<table>
<thead>
<tr>
<th>Items</th>
<th>Weight</th>
<th>Benefit</th>
<th>Value ($/ml$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 ml</td>
<td>$12$</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8 ml</td>
<td>$32$</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2 ml</td>
<td>$40$</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>6 ml</td>
<td>$30$</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1 ml</td>
<td>$50$</td>
<td>50</td>
</tr>
</tbody>
</table>

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

"knapsack"

The Fractional Knapsack Algorithm

Greedy choice: Keep taking item with highest value (benefit to weight ratio)
- Since $\sum b_i x_i / w_i = \sum (b_i / w_i) x_i$
- Run time: $O(n \log n)$. Why?

Correctness: Suppose there is a better solution
- there is an item $i$ with higher value than a chosen item $j$, but $x_i < w_i$ $x_j > 0$ and $v_i < v_j$
- If we substitute some $i$ with $j$, we get a better solution
- How much of $i$: $\min\{w_i - x_j, x_j\}$
- Thus, there is no better solution than the greedy one

Algorithm $\text{fractionalKnapsack}(S, W)$

- 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 $S$

\[
\begin{align*}
    x_i &\leftarrow 0 \\
    v_i &\leftarrow b_i / w_i \quad \{\text{value}\} \\
    w &\leftarrow 0 \quad \{\text{total weight}\} \\
\end{align*}
\]

while $w < W$

- remove item $i$ w/ highest $v_i$
- $x_i \leftarrow \min\{w_i, W - w\}$
- $w \leftarrow w + \min\{w_i, W - w\}$
Task Scheduling
(not in book)

- Given: a set \( T \) of \( n \) tasks, each having:
  - A start time, \( s_i \)
  - A finish time, \( f_i \) (where \( s_i < f_i \))

- Goal: Perform all the tasks using a minimum number of "machines."

Task Scheduling Algorithm

- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
  - Run time: \( O(n \log n) \). Why?
  - Correctness: Suppose there is a better schedule.
    - We can use \( k-1 \) machines
    - The algorithm uses \( k \)
    - Let \( i \) be first task scheduled on machine \( k \)
    - Machine \( i \) must conflict with \( k-1 \) other tasks
    - But that means there is no non-conflicting schedule using \( k-1 \) machines

Algorithm \( \text{taskSchedule}(T) \)

Input: set \( T \) of tasks w/ start time \( s_i \) and finish time \( f_i \)

Output: non-conflicting schedule with minimum number of machines

\[
m \leftarrow 0
\]

while \( T \) is not empty

- remove task \( i \) w/ smallest \( s_i \)

if there's a machine \( j \) for \( i \) then

- schedule \( i \) on machine \( j \)

else

\[
m \leftarrow m + 1
\]

schedule \( i \) on machine \( m \)
Example

- Given: a set T of n tasks, each having:
  - A start time, s
  - A finish time, f (where \( s < f \))
  - \([1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8]\) (ordered by start)
- Goal: Perform all tasks on min. number of machines

![Diagram showing task allocation on machines](image)