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
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).

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>a</td>
</tr>
<tr>
<td>010</td>
<td>b</td>
</tr>
<tr>
<td>011</td>
<td>c</td>
</tr>
<tr>
<td>10</td>
<td>d</td>
</tr>
<tr>
<td>11</td>
<td>e</td>
</tr>
</tbody>
</table>
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.

Algorithm $HuffmanEncoding(X)$

**Input** string $X$ of size $n$

**Output** optimal encoding trie for $X$

1. $C \leftarrow distinctCharacters(X)$
2. $computeFrequencies(C, X)$
3. $Q \leftarrow$ new empty heap
4. for all $c \in C$
   - $T \leftarrow$ new single-node tree storing $c$
   - $Q.insert(getFrequency(c), T)$
5. while $Q.size() > 1$
   - $f_1 \leftarrow Q.minKey()$
   - $T_1 \leftarrow Q.removeMin()$
   - $f_2 \leftarrow Q.minKey()$
   - $T_2 \leftarrow Q.removeMin()$
   - $T \leftarrow join(T_1, T_2)$
   - $Q.insert(f_1 + f_2, T)$
6. return $Q.removeMin()$
Example

$X = \text{abracadabra}$

Frequencies

<table>
<thead>
<tr>
<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>2</td>
</tr>
</tbody>
</table>

© 2004 Goodrich, Tamassia
Extended Huffman Tree Example

String: a fast runner need never be afraid of the dark

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
<th>b</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>h</th>
<th>i</th>
<th>k</th>
<th>n</th>
<th>o</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</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
$$\sum_{i \in S} b_i \left( \frac{x_i}{w_i} \right)$$

Constraint:
$$\sum_{i \in S} x_i \leq W$$
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 ($ per 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
The Fractional Knapsack Algorithm

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

- $x_i \leftarrow 0$
- $v_i \leftarrow b_i / w_i$ \{value\}
- $w \leftarrow 0$ \{total weight\}

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.”
Greedy Method and Compression

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 `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

$m \leftarrow 0$ \hspace{1cm} \{no. of machines\}

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, $s_i$
- A finish time, $f_i$ (where $s_i < f_i$)

Goal: Perform all tasks on min. number of machines