Dynamic programming vs Greedy algo – con’t

**KNAPSACK**

**Input:** a number \( W \) and a set of \( n \) items, the \( i \)-th item has a weight \( w_i \) and a cost \( c_i \)

**Output:** a subset of items with total weight \( \leq W \)

**Objective:** maximize cost

**Version 1:** Items are **divisible**.

<table>
<thead>
<tr>
<th></th>
<th>( w_i )</th>
<th>( c_i )</th>
<th>( c_i/w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk</td>
<td>10</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>flour</td>
<td>20</td>
<td>45</td>
<td>2.25</td>
</tr>
<tr>
<td>gold</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

\( W = 25 \)
KNAPSACK - divisible: a greedy solution

KNAPSACK-DIVISIBLE(n,c,w,W)
1. sort items in decreasing order of \( \frac{c_i}{w_i} \)
2. \( i = 1 \)
3. \( \text{currentW} = 0 \)
4. while \( \text{currentW} + w_i < W \) {
5. take item of weight \( w_i \) and cost \( c_i \)
6. \( \text{currentW} += w_i \)
7. \( i++ \)
8. }
9. take \( W - \text{currentW} \) portion of item \( i \)

Correctness:

Running time: \( O(n \log n) \)
KNAPSACK – indivisible

Version 2: Items are indivisible.

Does previous algorithm work for this version of KNAPSACK?

No. Think about a counterexample.
KNAPSACK - indivisible: a dyn-prog solution

The heart of the algorithm:

\[ S[k][v] = \max \text{ cost of a subset of items of the first k items of weight of the subset } \leq V \]

1st part

\[ k \text{ ranges } 0 \ldots n \]
\[ v \text{ ranges } 0 \ldots W \]

assumption:
weights are integers!

2nd part of the heart:

\[ S[k,v] = \begin{cases} 0 & \text{if } k=0 \text{ or } v=0 \\ S[k-1,v] & \text{if } w_k > v \\ \max \left\{ S[k-1,v], \right. \\ \left. c_k + S[k-1, v-w_k] \right\} & \text{if } w_k \leq v \end{cases} \]
The heart of the algorithm:

$$S[k][v] = \text{maximum cost of a subset of the first } k \text{ items, where the weight of the subset is at most } v$$
KNAPSACK – indivisible: a dyn-prog solution

The heart of the algorithm:

\[ S[k][v] = \text{maximum cost of a subset of the first } k \text{ items, where the weight of the subset is at most } v \]

\[
\begin{align*}
\text{KNAPSACK-INDIVISIBLE}(n, c, w, W) & \\
& \text{1. init } S[0][v] = 0 \text{ for every } v = 0, \ldots, W \\
& \text{2. init } S[k][0] = 0 \text{ for every } k = 0, \ldots, n \\
& \text{3. for } v = 1 \text{ to } W \text{ do} \\
& \quad \text{for } k = 1 \text{ to } n \text{ do} \\
& \quad \quad S[k][v] = S[k-1][v] \\
& \quad \text{6. if } (w_k \leq v) \text{ and } \\
& \quad \quad (S[k-1][v-w_k] + c_k > S[k][v]) \\
& \quad \quad \text{then} \\
& \quad \quad S[k][v] = S[k-1][v-w_k] + c_k \\
& \quad \text{7. RETURN } S[n][W]
\end{align*}
\]
The heart of the algorithm:

\[ S[k][v] = \text{maximum cost of a subset of the first } k \text{ items,} \]
\[ \text{where the weight of the subset is at most } v \]

```
KNAPSACK-INDIVISIBLE(n, c, w, W)
1. init S[0][v]=0 for every v=0,...,W
2. init S[k][0]=0 for every k=0,...,n
3. for v=1 to W do
4.   for k=1 to n do
5.     S[k][v] = S[k-1][v]
6.     if (w_k ≤ v) and
         (S[k-1][v-w_k]+c_k > S[k][v])
         then
7.         S[k][v] = S[k-1][v-w_k]+c_k
8.     RETURN S[n][W]
```
KNAPSACK – indivisible: a dyn-prog solution

The heart of the algorithm:

\[ S[k][v] = \text{maximum cost of a subset of the first } k \text{ items, where the weight of the subset is at most } v \]

KNAPSACK-INDIVISIBLE \((n,c,w,W)\)

1. init \(S[0][v]=0\) for every \(v=0,\ldots,W\)
2. init \(S[k][0]=0\) for every \(k=0,\ldots,n\)
3. for \(v=1\) to \(W\) do
4. 
   for \(k=1\) to \(n\) do
5. 
   \(S[k][v] = S[k-1][v]\)
6. 
   if \((w_k \leq v)\) and
   \((S[k-1][v-w_k]+c_k > S[k][v])\)
   then
7. 
   \(S[k][v] = S[k-1][v-w_k]+c_k\)
8. RETURN \(S[n][W]\)
Problem: Huffman Coding

Def: **binary character code** = assignment of binary strings to characters

e.g. ASCII code

A = 01000001  
B = 01000010  
C = 01000011  
...  

fixed-length code

How to decode: ?

01000001010000101000001101000001

A  B  C  A
Problem: Huffman Coding

Def: **binary character code** =
assignment of binary strings to characters

e.g. code

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
</tbody>
</table>

variable-length code

How to decode: ?

0101001111
ABBACCC
Def: **binary character code** = assignment of binary strings to characters

e.g. code

A = 0
B = 10
C = 11
...

variable-length code

Def:

A code is **prefix-free** if no codeword is a prefix of another codeword.

How to decode:  0101001111
### Problem: Huffman Coding

**Def:** *binary character code* = assignment of binary strings to characters

*Example another code:

\[
\begin{align*}
A &= 1 \\
B &= 10 \\
C &= 11 \\
&\quad \ldots
\end{align*}
\]

**Def:** A code is *prefix-free* if no codeword is a prefix of another codeword.

**How to decode:** ?

\[10101111\]

- B  B  C  C  A  A  A
Problem: Huffman Coding

Def:

**Huffman coding** is an optimal prefix-free code.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>11.1607%</td>
<td>M</td>
<td>3.0129%</td>
</tr>
<tr>
<td>A</td>
<td>8.4966%</td>
<td>H</td>
<td>3.0034%</td>
</tr>
<tr>
<td>R</td>
<td>7.5809%</td>
<td>G</td>
<td>2.4705%</td>
</tr>
<tr>
<td>I</td>
<td>7.5448%</td>
<td>B</td>
<td>2.0720%</td>
</tr>
<tr>
<td>O</td>
<td>7.1635%</td>
<td>F</td>
<td>1.8121%</td>
</tr>
<tr>
<td>T</td>
<td>6.9509%</td>
<td>Y</td>
<td>1.7779%</td>
</tr>
<tr>
<td>N</td>
<td>6.6544%</td>
<td>W</td>
<td>1.2899%</td>
</tr>
<tr>
<td>S</td>
<td>5.7351%</td>
<td>K</td>
<td>1.1016%</td>
</tr>
<tr>
<td>L</td>
<td>5.4893%</td>
<td>V</td>
<td>1.0074%</td>
</tr>
<tr>
<td>C</td>
<td>4.5388%</td>
<td>X</td>
<td>0.2902%</td>
</tr>
<tr>
<td>U</td>
<td>3.6308%</td>
<td>Z</td>
<td>0.2722%</td>
</tr>
<tr>
<td>D</td>
<td>3.3844%</td>
<td>J</td>
<td>0.1965%</td>
</tr>
<tr>
<td>P</td>
<td>3.1671%</td>
<td>Q</td>
<td>0.1962%</td>
</tr>
</tbody>
</table>

Optimization problems

- Input:
- Output:
- Objective:
Problem: Huffman Coding

Def:

**Huffman coding** is an optimal prefix-free code.

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>11.1607%</td>
</tr>
<tr>
<td>A</td>
<td>8.4966%</td>
</tr>
<tr>
<td>R</td>
<td>7.5809%</td>
</tr>
<tr>
<td>I</td>
<td>7.5448%</td>
</tr>
<tr>
<td>O</td>
<td>7.1635%</td>
</tr>
<tr>
<td>T</td>
<td>6.9509%</td>
</tr>
<tr>
<td>N</td>
<td>6.6544%</td>
</tr>
<tr>
<td>S</td>
<td>5.7351%</td>
</tr>
<tr>
<td>L</td>
<td>5.4893%</td>
</tr>
<tr>
<td>C</td>
<td>4.5388%</td>
</tr>
<tr>
<td>U</td>
<td>3.6308%</td>
</tr>
<tr>
<td>D</td>
<td>3.3844%</td>
</tr>
<tr>
<td>P</td>
<td>3.1671%</td>
</tr>
<tr>
<td>M</td>
<td>3.0129%</td>
</tr>
<tr>
<td>H</td>
<td>3.0034%</td>
</tr>
<tr>
<td>G</td>
<td>2.4705%</td>
</tr>
<tr>
<td>B</td>
<td>2.0720%</td>
</tr>
<tr>
<td>F</td>
<td>1.8121%</td>
</tr>
<tr>
<td>Y</td>
<td>1.7779%</td>
</tr>
<tr>
<td>W</td>
<td>1.2899%</td>
</tr>
<tr>
<td>K</td>
<td>1.1016%</td>
</tr>
<tr>
<td>V</td>
<td>1.0074%</td>
</tr>
<tr>
<td>X</td>
<td>0.2902%</td>
</tr>
<tr>
<td>Z</td>
<td>0.2722%</td>
</tr>
<tr>
<td>J</td>
<td>0.1965%</td>
</tr>
<tr>
<td>Q</td>
<td>0.1962%</td>
</tr>
</tbody>
</table>

**Huffman coding**

- **Input:** an alphabet with frequencies
- **Output:** a prefix-free code
- **Objective:** minimize expected number of bits per character
Problem: Huffman Coding

Example: 

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>60%</td>
</tr>
<tr>
<td>B</td>
<td>20%</td>
</tr>
<tr>
<td>C</td>
<td>10%</td>
</tr>
<tr>
<td>D</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Is fixed-width coding optimal?

Expected codeword length is 2.

Ex: 

\[
\text{expected codeword length} = 0.6 \times 2 + 0.2 \times 2 + 0.1 \times 2 + 0.1 \times 2 = 2
\]

Ex: 

\[
\text{expected codeword length} = 0.6 \times 1 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 1.6
\]

Huffman coding
- Input: an alphabet with frequencies
- Output: a prefix-free code
- Objective: minimize expected number of bits per character
**Problem: Huffman Coding**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>60%</td>
</tr>
<tr>
<td>B</td>
<td>20%</td>
</tr>
<tr>
<td>C</td>
<td>10%</td>
</tr>
<tr>
<td>D</td>
<td>10%</td>
</tr>
</tbody>
</table>

Is fixed-width coding optimal?

**NO, exists a prefix-free code using 1.6 bits per character!**

**Huffman coding**

- **Input:** an alphabet with frequencies
- **Output:** a prefix-free code
- **Objective:** minimize expected number of bits per character
Problem: Huffman Coding

Huffman ( [a₁,f₁],[a₂,f₂],…,[aₙ,fₙ] )
1. if n=1 then
2. code[a₁] ← “”
3. else
4. let fᵢ, fⱼ be the 2 smallest f’s
5. Huffman ( [aᵢ, fᵢ+fⱼ], [a₁, f₁],…,[aₙ, fₙ] ) omits aᵢ, aⱼ
6. code[aⱼ] ← code[aᵢ] + “0”
7. code[aᵢ] ← code[aᵢ] + “1”

Huffman coding
- Input: an alphabet with frequencies
- Output: a prefix-free code
- Objective: minimize expected number of bits per character

Running time: $O(n \log n)$ bec. $n$ nested recursive calls, $O(\log n)$ steps each
$T(n) = T(n-1) + O(\log n)$
Problem: Huffman Coding

Lemma 1: Let \( x, y \) be the symbols with frequencies \( f_x > f_y \). Then in an optimal prefix code \( \text{length}(C_x) \leq \text{length}(C_y) \).

\[
\text{by contradiction, assume } \quad \text{length}(C_x) > \text{length}(C_y)
\]

Let's create a new code \( D \) by swapping \( C_x \) and \( C_y \) i.e.:

\[
\begin{align*}
D_x &= C_y \\
D_y &= C_x
\end{align*}
\]

Expected codeword length for \( C \):

\[
\sum_{z} f_z \cdot \text{length}(C_z) = A
\]

\[
D: \quad \sum_{z} f_z \cdot \text{length}(D_z) = B
\]

What is:

\[
A - B = f_x \cdot \text{length}(C_x) + f_y \cdot \text{length}(C_y) - f_x \cdot \text{length}(D_x) - f_y \cdot \text{length}(D_y)
\]

\[
= (f_x - f_y) (\text{length}(C_x) - \text{length}(C_y)) > 0
\]

Contradiction. A > B and thus \( C \) could not have been an optimal prefix free code.
Problem: Huffman Coding

Lemma 1: Let \( x, y \) be the symbols with frequencies \( f_x > f_y \). Then in an optimal prefix code \( \text{length}(C_x) \leq \text{length}(C_y) \).

Lemma 2: If \( w \) is a longest codeword in an optimal code then there exists another codeword of the same length.

by contradiction, assume the second longest codeword is shorter than \( w \)

\[
\begin{align*}
\text{x} & \quad \text{w} \\
\text{y} & \quad \\
\end{align*}
\]

the longest codeword (for symbol \( x \))
the 2nd longest (for \( y \))

we will create a new code by shortening \( x \)'s codeword to the length of \( y \)'s codeword

we observe:
1) expected codeword length decreased
2) the new code is prefix-free (i.e., no prefix of \( w \) is a codeword)

\[ \Rightarrow \text{the original code was not optimal} \]
Problem: Huffman Coding

Lemma 1: Let $x,y$ be the symbols with frequencies $f_x > f_y$. Then in an optimal prefix code $\text{length}(C_x) \leq \text{length}(C_y)$.

Lemma 2: If $w$ is a longest codeword in an optimal code then there exists another codeword of the same length.

Lemma 3: Let $x,y$ be the symbols with the smallest frequencies. Then there exists an optimal prefix code such that the codewords for $x$ and $y$ differ only in the last bit.
Lemma 3: Let $x, y$ be the symbols with the smallest frequencies. Then there exists an optimal prefix code such that the codewords for $x$ and $y$ differ only in the last bit.

Create another code:

$y$'s codeword same as $x$'s but the last bit is flipped:

if there is a symbol $z$ with the same codeword as $y$'s new codeword then, let $z$'s new codeword $=$ $y$'s old codeword.

Observe:
- new code is prefix-free
- exp. codeword length is the same, i.e. optimal.

$\square$
Problem: Huffman Coding

Lemma 1: Let \( x, y \) be the symbols with frequencies \( f_x > f_y \). Then in an optimal prefix code
\[ \text{length}(C_x) \leq \text{length}(C_y). \]

Lemma 2: If \( w \) is a longest codeword in an optimal code then there exists another codeword of the same length.

Lemma 3: Let \( x, y \) be the symbols with the smallest frequencies. Then there exists an optimal prefix code such that the codewords for \( x \) and \( y \) differ only in the last bit.

Theorem: The prefix code output by the Huffman algorithm is optimal.