**Dynamic programming vs Greedy algo - cont'**

**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>Item</th>
<th>$w_i$</th>
<th>$c_i$</th>
<th>$c_i/w_i$</th>
<th>Take:</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk</td>
<td>10</td>
<td>20</td>
<td>2</td>
<td>all of gold</td>
</tr>
<tr>
<td>flour</td>
<td>20</td>
<td>45</td>
<td>2.25</td>
<td>flour</td>
</tr>
<tr>
<td>gold</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>4 pounds of milk</td>
</tr>
</tbody>
</table>

$W = 25$
KNAPSACK - divisible: a greedy solution

**KNAPSACK-DIVISIBLE** \((n,c,w,W)\)

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

Find a counterexample.
The heart of the algorithm:

\[
S[k][v] = \begin{cases} 
0 & \text{if } k = 0 \text{ or } v = 0 \\
\max \{ S[k-1][v], c_k + S[k-1][v-w_k] \} & \text{if } v \geq w_k 
\end{cases}
\]

2nd part of the heart:

\[S[k,v] = \begin{cases} 
0 & \text{if } k = 0 \text{ or } v = 0 \\
\max \{ S[k-1][v], c_k + S[k-1][v-w_k] \} & \text{if } v \geq w_k 
\end{cases}\]

3rd part:

return \( S[n,W] \)
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:

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

**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 \( (w_k \leq v) \) 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 \( S[n][W] \)
KNAPSACK – indivisible: a dyn-prog solution

The heart of the algorithm:

\[ S[k][v] = \text{maximum cost of a subset of the first } k \text{ items,} \]

\[ \text{where the weight of the subset is at most } v \]

\[
\text{Running time: } \quad 0(nW)
\]

\[
\text{"Weakly polynomial" (not polynomial running time because it is exponential in the length of } W\text{)}
\]

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

\[
\begin{align*}
A &= 01000001 \\
B &= 01000010 \\
C &= 01000011 \\
& \quad \vdots
\end{align*}
\]

fixed-length code

How to decode: ?

\[
0100000101000010100001101000001
\]

A   B   C   A
Problem: Huffman Coding

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

e.g. code

\[
\begin{align*}
A &= 0 \\
B &= 10 \\
C &= 11 \\
\ldots
\end{align*}
\]

variable-length code

How to decode: ?

0101001111
A B B A C C
Problem: Huffman Coding

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

e.g. code

\[
\begin{align*}
A &= 0 \\
B &= 10 \\
C &= 11 \\
\vdots
\end{align*}
\]

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

e.g. another code:

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

variable-length code

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

How to decode: 

\[
10101111
\]

B B C C ?

AAAA ?
Def:

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</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>

Optimization problems

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

Def:
Huffman coding is an optimal prefix-free code.

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

<table>
<thead>
<tr>
<th>Letter</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>
Problem: Huffman Coding

Example:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>60%</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>20%</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>10%</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>10%</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Is fixed-width coding optimal?

Huffman coding

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

expected codeword length $= 2$
Problem: Huffman Coding

Example:

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

0 10 110 111

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

Example:

Is fixed-width coding optimal?

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

Expected codeword length:

\[0.6 \times 1 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 1.6\]
Problem: Huffman Coding

Huffman ( \([a_1,f_1],[a_2,f_2],...,[a_n,f_n]\))

1. if n=1 then
2. \(\text{code}[a_1] \leftarrow \text{""}\)
3. else
4. let \(f_i,f_j\) be the 2 smallest \(f\)'s
5. Huffman ( \([a_i,f_i+f_j],[a_1,f_1],...,[a_n,f_n]\) )
   \[\text{omits } a_i,a_j\]
6. \(\text{code}[a_j] \leftarrow \text{code}[a_i] + \text{"0"}\)
7. \(\text{code}[a_i] \leftarrow \text{code}[a_i] + \text{"1"}\)

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. we make \(n\) nested recursive calls

O(\(\log n\)) steps with the heap (need to put all elements/frequencies to a heap before calling Huffman)
Problem: Huffman Coding

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

By contradiction, assume that \( \text{length}(C_x) > \text{length}(C_y) \). Let's create code \( D \) s.t. \( D_x = C_y \) and \( D_y = C_x \) (we swapped \( x \)'s and \( y \)'s codewords and all other codewords of \( D \) are same as \( C \) codewords).

Expected codeword length for code \( C \):
\[
A = \sum_{i=1}^{n} f_i \cdot \text{length}(C_{a_i})
\]

Expected codeword length for code \( D \):
\[
B = \sum_{i=1}^{n} f_i \cdot \text{length}(D_{a_i})
\]

\[
A - B = f_x \cdot \text{length}(C_x) + f_y \cdot \text{length}(C_y) - f_x \cdot \text{length}(D_x) - f_y \cdot \text{length}(D_y) > 0
\]

This contradicts the optimality of \( C \), hence the assumption is false.
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 no other codeword of the same length. Let \(y\) be the second longest codeword.

Then, let's cut (shorten) the \(x\)'s codeword to the length of \(y\)'s codeword.

Observe:
1) code is still prefix-free (bec. no prefixes of the long \(x\)'s codeword existed in the code, hence no prefixes of the new codeword exist)
2) expected codeword length decreased \(\Rightarrow\) original code 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.
Problem: Huffman Coding

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.

Let's create a new code: y like x but the last bit is swapped

if there is a z with \( x \) codeword (same as new y's codeword), give z y's old codeword

Observe:
1) prefix-free because no prefixes of x's codeword existed in the old code
2) expected codeword length is the same

Lemma 1: the lowest frequency \( \Rightarrow \) longest codeword

Lemma 2: the 2nd lowest frequency \( \Rightarrow \) longest codeword
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.