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 $c_i > 0$.

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>
</tr>
</thead>
<tbody>
<tr>
<td>milk</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>sugar</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>gold</td>
<td>1</td>
<td>1800</td>
</tr>
</tbody>
</table>

$W = 4$

- take gold + 3 pounds of sugar: $1800 + \frac{3}{5} \cdot 2$
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: exchange argument

Running time: \( O(n \log n) \)
**KNAPSACK - indivisible**

Version 2: Items are indivisible.

Does previous algorithm work for this version of KNAPSACK?

**No**

- A 4 100
- B .1 10

**DP**

- Weighted interval scheduling
  
  \[ S[j] = \text{max cost of a non-overlap. subset of the first } j \text{ intervals} \]

- Longest incr. subseq.
  
  \[ S[j] = \text{max length of an incr. subseq. of the first } j \text{ items, including the } j\text{-th} (a_j) \]

- **KNAPSACK**
  
  \[ S[v] = \text{max cost when taking all items and capacity } v \]

  \[ S[j][v] = \text{max cost of a subset of the first } j \text{ items which fits in capacity } v \]
The heart of the algorithm:

- \( S[k][v] = \max \text{ cost of a subset of the first } k \text{ items where subset is of total weight } \leq v \)
- \( S[k][v] = \begin{cases} 
0 & \text{if } k = 0 \\
0 & \text{if } v = 0 \\
\max \{ S[k-1][v], c_k + S[k-1][v-w_k] \} & \text{if } k > 0 \text{ and } v > 0
\end{cases} \)

- 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 ≤ v) and
6.         (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
7.             \( (S[k-1][v-w_k]+c_k > S[k][v]) \) then
8.             \( S[k][v] = S[k-1][v-w_k]+c_k \)
9.     RETURN \( S[n][W] \)

Running time:
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: ?

01000000101000011001000011101000001

A  B  C  A
Problem: Huffman Coding

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

e.g. code

A = 0  
B = 10  
C = 11  

variable-length code

A = 0  
B = 01  
C = 1

How to decode: ?

0101001111  

A B B A C C

0101001111  

A C ?  
B ?
Problem: Huffman Coding

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

e.g. code

A = 0  
B = 10  
C = 11

...  

How to decode:  ?

0101001111

Def:

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

easy to speedily decode (unambiguous)
**Problem: Huffman Coding**

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

E.g. another code

A = 1
B = 10
C = 11
...  # *not prefix-free*

**variable-length code**

**Def:**

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

How to decode: ?

10101111
Problem: Huffman Coding

Def:

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>11.1607%</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>8.4966%</td>
<td>00</td>
</tr>
<tr>
<td>R</td>
<td>7.5809%</td>
<td>01</td>
</tr>
<tr>
<td>I</td>
<td>7.5448%</td>
<td>110</td>
</tr>
<tr>
<td>O</td>
<td>7.1635%</td>
<td>111</td>
</tr>
<tr>
<td>T</td>
<td>6.9509%</td>
<td>10</td>
</tr>
<tr>
<td>N</td>
<td>6.6544%</td>
<td>101</td>
</tr>
<tr>
<td>S</td>
<td>5.7351%</td>
<td>000</td>
</tr>
<tr>
<td>L</td>
<td>5.4893%</td>
<td>1100</td>
</tr>
<tr>
<td>C</td>
<td>4.5388%</td>
<td>001</td>
</tr>
<tr>
<td>U</td>
<td>3.6308%</td>
<td>010</td>
</tr>
<tr>
<td>D</td>
<td>3.3844%</td>
<td>011</td>
</tr>
<tr>
<td>P</td>
<td>3.1671%</td>
<td>0100</td>
</tr>
<tr>
<td>M</td>
<td>3.0129%</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>3.0034%</td>
<td>0010</td>
</tr>
<tr>
<td>G</td>
<td>2.4705%</td>
<td>0110</td>
</tr>
<tr>
<td>B</td>
<td>2.0720%</td>
<td>0111</td>
</tr>
<tr>
<td>F</td>
<td>1.8121%</td>
<td>100</td>
</tr>
<tr>
<td>Y</td>
<td>1.7779%</td>
<td>1011</td>
</tr>
<tr>
<td>W</td>
<td>1.2899%</td>
<td>1101</td>
</tr>
<tr>
<td>K</td>
<td>1.1016%</td>
<td>1110</td>
</tr>
<tr>
<td>V</td>
<td>1.0074%</td>
<td>1111</td>
</tr>
<tr>
<td>X</td>
<td>0.2902%</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0.2722%</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>0.1965%</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>0.1962%</td>
<td></td>
</tr>
</tbody>
</table>

Optimization problems:

- Input: frequency table
- Output: prefix-free code for the symbols in the freq. table
- Objective: min expected #bits per symbol
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
Problem: Huffman Coding

Example:

<table>
<thead>
<tr>
<th></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>

Is fixed-width coding optimal? No

Huffman coding

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

Example:  

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

Is fixed-width coding optimal?

Claim:  \( f_A \geq f_B \) then \( |c_A| \leq |c_B| \)

Proof: By contradiction, assume \( |c_A| > |c_B| \) (\( \ast \))

Then, let's swap the codewords and show that expected # of bits decreases. 

\[
\begin{align*}
\text{Before:} & \quad \sum_{S: \text{symbols}} f_S |c_S| = \text{exp. # of bits} \\
& = \sum_{S: S \neq A, B} f_S |c_S| + f_A |c_A| + f_B |c_B| \\
\text{After swap:} & \quad \sum_{S: S \neq A, B} f_S |c_S| + f_A |c_B| + f_B |c_A| \\
& \geq ?
\end{align*}
\]

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>Symbol</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>

So far: higher frequency \(\rightarrow\) shorter codeword

Huffman coding

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

Is fixed-width coding optimal?

**Claim #2:** for the lowest 2 frequencies, we get codewords of the same length.

Let's look at the two symbols w. lowest frequency.

\[101 \quad 110 \quad 111\]

Suppose \(C, D\).

What can we say about \(c_C, c_D\) in an optimal prefix-free code?

Suppose \(f_C \geq f_D\). We know \(f_1 \geq f_C\). Why? By the previous claim, we know that \(|c_C| \leq |c_C| \leq |c_D|\).

By contradiction, suppose \(|c_C| < |c_D|\).

Remove yellow, get \(C_D\) where \(|c_C| = |c_D|\).

Since no prefix problems with \(C_D\), then no prefix problems w. \(C_D\).
Problem: Huffman Coding

Example:

<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?

- **Claim #3:**
  - Let $C_1, C_2$ be the two lowest frequencies.
  - We know that $|C_1| = |C_2|$ in an optimal prefix-free code.
  - We know that $|C_3| \leq |C_1|$ for $C_3 \neq C_1, C_2$.

There exist an optimal prefix-free code such that:

- $C_1$ and $C_2$ differ in only the last bit.

**Huffman coding**

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

Example:

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
Problem: Huffman Coding

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