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.

\[
\begin{array}{ccc}
\text{milk} & 3 & 1 \\
\text{sugar} & 5 & 2 \\
\text{gold} & 1 & 1800
\end{array}
\]

\( W = 4 \)

\( \text{get: gold (1), sugar (3)} \)

\( 1800 + \frac{3}{5} \times 2 \)
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: exchange argument: can make optimum look like our solution

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

Version 2: Items are indivisible.

Does previous algorithm work for this version of KNAPSACK?

\[ \text{NO} \quad W = 4 \quad \begin{array}{cc} \text{Wi} & \text{Ci} \\ 3 & 3.1 \end{array} \leftarrow \text{highest } \frac{c_i}{w_i} \]

\[ \begin{array}{cc} 2 & 2 \\ 2 & 2 \end{array} \]

**Weighted Interval Scheduling**

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

**Longest Incr. Subseq.**

\[ S[j] = \max \text{ length of an incr. subseq. of the first } j \text{ items, ending } w, cj \]

**KNAPSACK**

- \[ S[j][v] = \max \text{ value of a subset of items w.total weight } \leq v \text{, out of the first } j \text{ items} \]
  - \[ S[j][v] = \max \{ S[j-1][v], cj + S[j-1][v-w_j] \} \]
  - \[ S[0][v] = 0 \]
  - \[ S[j][0] = 0 \]
- \[ \text{return } S[n][W] \]
The heart of the algorithm:

\[ S[k][v] = \]
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:

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

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

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. \hspace{1cm} $S[k][v] = S[k-1][v]$
6. \hspace{1cm} if ($w_k \leq v$) and \hspace{1cm} ($S[k-1][v-w_k]+c_k > S[k][v]$) then
7. \hspace{1cm} \hspace{1cm} $S[k][v] = S[k-1][v-w_k]+c_k$
8. RETURN $S[n][W]$
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 \( v = 0, \ldots, W \)
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 \( S[n][W] \)
Def: **binary character code** = assignment of binary strings to characters

e.g. ASCII code

A = 01000001  \quad B = 01000010  \quad C = 01000011

...  \quad \text{fixed-length code}

How to decode: ?

01000001 10100001 00100001 1101000001
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: ?

variable-length code
Problem: Huffman Coding

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

e.g. code

\[
\begin{align*}
A &= 0 \\
B &= 10 \\
C &= 11 \\
D &= 100 \quad \text{(no longer prefix-free)}
\end{align*}
\]

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

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

variable-length code

Def:

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

How to decode: ?
Problem: Huffman Coding

Def:

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

How to decode:

- we have a prefix-free code
  - A: 01
  - B: 1001
  - C: 101

Optimization problems

- **Input:** a frequency table
- **Output:** a prefix-free code for the symbols in
- **Objective:** min weighted average of codeword length (or alternatively, min expected codeword length)
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>

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

Is fixed-width coding optimal?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>

2 bits per symbol (fixed-width)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>0</th>
<th>0.6 \cdot 1 +</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>10</td>
<td>0.2 \cdot 2 +</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>110</td>
<td>0.1 \cdot 3 +</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>111</td>
<td>0.1 \cdot 3</td>
</tr>
</tbody>
</table>

= 1.6 expected # bits for this 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>A</th>
<th>60%</th>
</tr>
</thead>
<tbody>
<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

Example: | A  | 60% |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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**

**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>

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

Example:

```
  A  B  C  D  E  F  G
  25 20 15 10 15 25 15
```

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

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

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

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-free code output by the Huffman algorithm is optimal.