Fractional Knapsack

• Given a set $S$ of $n$ items, such that each item $i$ has a positive benefit $b_i$ and a positive weight $w_i$; the size of the knapsack $W$

• The problem is to find the amount $x_i$ of each item $i$ which maximizes the total benefit

$$\sum_i b_i \left( \frac{x_i}{w_i} \right)$$

under the condition that $0 \leq x_i \leq w_i$ and

$$\sum_i x_i \leq W$$
Fractional Knapsack - Example

Fractional Knapsack in Python

```python
def fractional_knapsack(S, W):
    v = []
    for item in S:
        value = float(item[1]) / float(item[0])
        v.append((value, item[1], item[2]))
    v.sort(key=itemgetter(0))
    w, result = 0, []
    while w < W:
        high = v[-1]
        v.pop()
        a = min(high[1], W-w)
        w += a
        result.append((a, high[2]))
    return result
```

Remark: sort v by value = benefit/weight

Remark: select and remove the highest value (high)

Remark: a is how much item high we took
Fractional Knapsack

- Time complexity is $O(n \log n)$
- Fact: Greedy strategy is optimal for the fractional knapsack problem
- Proof: We will show that the problem has the optimal substructure and the algorithm satisfies the greedy-choice property

Greedy Choice

Items (decr. order $b_i/w_i$): $1 \quad 2 \quad 3 \quad \ldots \quad j \quad \ldots \quad n$

“Optimal” solution $A$: $x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_j \quad \ldots \quad x_n$

Greedy solution $G$: $x_1^G \quad x_2^G \quad x_3^G \quad \ldots \quad x_j^G \quad \ldots \quad x_n^G$

Fact: There exists an optimal solution $A$ in which we take as much as possible from item 1.

Proof: If in $A$ we have $x_j = \min(w_j, W)$, then $A$ is what we are looking for.
If in $A$ we have $x_j < \min(w_j, W)$, then do the following
- while $x_j < \min(w_j, W)$:
  
  for some small $d > 0$, increase $x_j$ by $d$, decrease some $x_j$ by $d$

Observations
- In each step, we increase the total benefit by $d(b_j/w_j - b_j/w_i)$, which is non-negative
- We can keep doing this until when $x_j = \min(w_j, W)$, we call this solution $B$
- The quality of solution $B$ with $x_j = \min(w_j, W)$ is as good (if not better) than $A$

$B$ is what we are looking for.

[That’s why the greedy choice $x_j^G = \min(w_j, W)$ is optimal]
Optimal substructure

- Let $x_1 = \min(w_1, W)$ be the greedy choice for the first step
- $(S, W)$ is the original problem
- $(S', W')$ is the subproblem, where $S' = \{2, 3, \ldots, n\}$, $W' = W - x_1$

items (decr. order $b_i/w_i$)

\begin{array}{cccccc}
1 & 2 & 3 & \ldots & n \\
\hline
\text{solution for } (S', W') & \text{?} & \text{?} & \ldots & \text{?} \\
\text{solution for } (S, W) & x_1 & x_2 & x_3 & \ldots & x_n \\
\end{array}

Fact: $x_2, x_3, \ldots, x_n$ is an optimal solution to $(S', W')$ $\iff$ $x_1, x_2, x_3, \ldots, x_n$ is an optimal solution to $(S, W)$

**Proof** ($\Rightarrow$) Assume that $x_2, x_3, \ldots, x_n$ is an optimal solution to $(S', W')$ but $x_1, x_2, x_3, \ldots, x_n$ is not an optimal solution to $(S, W)$.

If the latter is not optimal, a better solution $y_1, y_2, y_3, \ldots, y_n$ exists for $(S, W)$, that is

$$\sum_{1 \leq i \leq n} y_i \left(\frac{b_i}{w_i}\right) > \sum_{1 \leq i \leq n} x_i \left(\frac{b_i}{w_i}\right)$$

But $y_j = x_j$ (because an optimal solution has to contain $x_j$ as the greedy choice) thus

$$\sum_{2 \leq i \leq n} y_i \left(\frac{b_i}{w_i}\right) > \sum_{2 \leq i \leq n} x_i \left(\frac{b_i}{w_i}\right)$$

which implies that $x_2, x_3, \ldots, x_n$ was not optimal for $(S, W)$ $\rightarrow$ contradiction.
Optimal substructure

Proof \( (\leq) \) Assume that \( x_1, x_2, x_3, \ldots, x_n \) is an optimal solution to \((S, W)\), but \( x_2, x_3, \ldots, x_n \) is not an optimal solution to \((S', W')\).

If the latter is not optimal, a better solution \( y_2, y_3, \ldots, y_n \) exists for \((S', W')\), that is

\[
\sum_{2 \leq i \leq n} y_i \left( \frac{b_i}{w_i} \right) > \sum_{2 \leq i \leq n} x_i \left( \frac{b_i}{w_i} \right)
\]

But

\[
\sum_{1 \leq i \leq n} x_i \left( \frac{b_i}{w_i} \right) = x_1 \left( \frac{b_1}{w_1} \right) + \sum_{2 \leq i \leq n} x_i \left( \frac{b_i}{w_i} \right) < x_1 \left( \frac{b_1}{w_1} \right) + \sum_{2 \leq i \leq n} y_i \left( \frac{b_i}{w_i} \right)
\]

which implies that \( x_1, x_2, x_3, \ldots, x_n \) was not optimal for \((S,W)\)

\( \rightarrow \) contradiction

Huffman codes
Data Compression

• Text files are usually stored by representing each character with an 8-bit ASCII code.

• The ASCII encoding is an example of fixed-length encoding, where each character is represented with the same number of bits.

• In order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others.

Data Compression

• Variable-length encoding uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.

• Huffman coding (section 5.2)
File Compression: Example

• An example
  – text: “java”
  – encoding: a = “0”, j = “11”, v = “10”
  – encoded text: 110100 (6 bits)
• How to decode in the case of ambiguity?
  – encoding: a = “0”, j = “11”, v = “00”
  – encoded text: 110000 (6 bits)
  – could be “java”, or “jvv”, or “jaaa”, or …

Encoding

• To prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule: no code is a prefix of another
• Example
  – a = “0”, j = “11”, v = “10” satisfies the prefix rule
  – a = “0”, j = “11”, v = “00” does not satisfy the prefix rule (the code of ’a’ is a prefix of the codes of ’v’)
Trie

- We use an encoding trie to satisfy this prefix rule
  - the characters are stored at the external nodes
  - a left child (edge) means 0
  - a right child (edge) means 1

Example of Decoding

- encoded text:
  010111010001010010111011010

- text: ABRACADABRA (11 bytes=88 bits)
Data Compression

- **Problem:** We want the encoded text as short as possible
- **Example:** ABRACADABRA
  \[0101101101000101001011011010\] 29 bits

![Diagram of a binary tree encoding ABRACADABRA with 29 bits]

Data Compression

- **Example 2:** ABRACADABRA
  \[0010110010001100101100\] 24 bits

![Diagram of a binary tree encoding ABRACADABRA with 24 bits]
Optimization problem

• Given a character $c$ in the alphabet $\Sigma$
  – let $f(c)$ be the frequency of $c$ in the file
  – let $d_T(c)$ be the depth of $c$ in the tree = the length of the codeword

• We want to minimize the number of bits required to encode the file, that is

$$\min_{\text{binary trees } T \text{ with } |\Sigma| \text{ leaves } \sum_{c \in \Sigma} f(c)d_T(c)}$$

Huffman Encoding: Example

<table>
<thead>
<tr>
<th>Step 0</th>
<th>ABRACADABRA</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>character frequency</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 1</th>
<th>5</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>R</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Huffman Encoding: Example

Step 2

A 5
B 2
R 2
C 1
D 1

Step 3

A 5
B 2
R 2
C 1
D 1

Step 4

A 5
B 2
R 2
C 1
D 1

Step 5

A 5
B 2
R 2
C 1
D 1

Huffman Encoding: Example
Final Huffman Trie

A B R A C A D A B R A
0 1 0 0 1 0 1 0 0 1 1 0 0 1 1 1 0 0 1 0 0 1 0
(23 bits)

Another Example

<table>
<thead>
<tr>
<th>Step</th>
<th>ABRACADABRA character frequency</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 0</td>
<td></td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Step 1</td>
<td>5 A</td>
<td>2 B</td>
<td>2 R</td>
<td>C</td>
<td>1 D</td>
<td>1</td>
</tr>
<tr>
<td>Step 2</td>
<td>5 A</td>
<td>2 B</td>
<td>2 R</td>
<td>C</td>
<td>1 D</td>
<td>1</td>
</tr>
</tbody>
</table>
Another Example

Step 3

Another Example

Step 4

Another Example

Step 5

Another Example

Step 6
Final Trie

A B R A C   A D   A B R A
0 10 110 0 1110 0 1111 0 10 110 0 (23 bits)

Priority queue

• Use a priority queue for storing the nodes
• Priority queue is a queue ordered by priority (heap)
• For our application, priority = frequency
• If there are \( k \) elements in the queue:
  – Extracting the lowest priority is \( O(\log k) \)
  – Inserting takes \( O(\log k) \)
Huffman algorithm in Python

```python
def makeHuffTree(t):
    heapq.heapify(t)  # transforms list t in a heap in linear time
    while len(t) > 1:
        L, R = heapq.heappop(t), heapq.heappop(t)
        parent = (L[0] + R[0], L, R)
        heapq.heappush(t, parent)
    return t[0]  # returns the tree represented a nested tuple

def printHuffTree(t, prefix = ' '):
    if len(t) == 2:
        print t[1], prefix
    else:
        printHuffTree(t[1], prefix + '0')
        printHuffTree(t[2], prefix + '1')
```

Huffman Algorithm

- Running time for a text of length $n$ with $k$ distinct characters: $O(n + k \log k)$
- If we assume $k$ to be a constant (i.e., not a function of $n$) then the algorithm runs in $O(n)$ time
- Fact: Using a Huffman encoding trie, the encoded text has minimal length
- Proof: omitted
Greedy method: summary

- Task scheduling
- Fractional knapsack
- Huffman encoding (section 5.2)

- Other greedy algorithms that will be covered later: Prim (section 5.1.5), Kruskal (section 5.1.3) and Dijkstra (section 4.4)