Greedy algorithms:

Examples: Kruskal’s MST algorithm
           Prim’s MST algorithm
           Dijkstra’s SSSP algorithm
           Selection sort
           Insertion sort

Format of greedy algorithms:

for each item X in some order
   decide whether/how to include item X in the solution;

For many problems it’s easy to design a greedy algorithm. We’ll see several more examples of greedy algorithms that solve various problems.

Unfortunately some “obvious” greedy algorithms are incorrect. We’ll also see examples of these and how to identify them.
Problem: Activity Selection

Activities \{1...n\}
Start times \(S[1...n]\)
Finish times \(F[1...n]\)

- Activities \(j\) and \(k\) are compatible if either \(F[j] \leq S[k]\) or \(F[k] \leq S[j]\).
- Otherwise activities \(j\) and \(k\) overlap or conflict.

Goal: Choose largest subset of mutually compatible activities.

Example:
Greedy Algorithm 1 for Activity Selection:

\[ A = \{ \}; \]

for each activity \( k \) in order of **ascending** \( S[k] \)
if (activity \( k \) is compatible with all activities in \( A \))
\[ A = A \cup \{k\}; \]

Correct for previous example, but not for all possible inputs.

Counterexample: add new 6\(^{th} \) activity as shown

\[
\begin{array}{ccccccc}
S & 1 & 2 & 3 & 4 & 5 & 6 \\
8 & 2 & 5 & 6 & 1 & 0 \\
F & 11 & 6 & 7 & 10 & 4 & 12 \\
\end{array}
\]

So Greedy Algorithm 1 above is NOT correct.
Greedy Algorithm 2 for Activity Selection:

\[ A = \{ \}; \]

for each activity \( k \) in order of **ascending lengths** \( F[k] - S[k] \)

if (activity \( k \) is compatible with all activities in \( A \))

\[ A = A \cup \{k\}; \]

Correct for previous examples, but not for all possible inputs.

Counterexample: stretch 1, 3, 5 and shrink 2, 4 as shown

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>12</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

So Greedy Algorithm 2 above is NOT correct.
Greedy Algorithm 3 for Activity Selection:

\[
A = \{\};
\]
for each activity \( k \) in order of ascending \( F[k] \)
if (activity \( k \) is compatible with all activities in \( A \))
\[
A = A \cup \{k\};
\]

Correct for previous examples.

Greedy algorithm 3 is ALWAYS correct for all possible inputs. Why?
("No known counterexample" isn’t a sufficient justification.)

Contradiction argument:
Suppose Greedy Algorithm 3 was not correct.
Consider its first incorrect choice of some activity \( k \).
So there must be a better next choice, say activity \( k' \),
which leads to a larger solution \( A' \).
By compatibility, every future activity \( k'' \) in \( A' \) has \( F[k'] \leq S[k''] \).
Also \( F[k] \leq F[k'] \) because \( k \) precedes \( k' \) in ascending order of \( F \).
Hence \( F[k] \leq S[k''] \).
Therefore solution \( A' - \{k'\} \cup \{k\} \) is an equally good solution,
thus contradicting that \( k' \) is better choice than \( k \).

So Greedy Algorithm 3 above is indeed correct.
Problem: Huffman Coding

Alphabet A[1…n] with n characters
Relative frequencies F[1…n] of each character

Example:

<table>
<thead>
<tr>
<th>A</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

- Assign binary codes C[1…n] to each character (either fixed-length or variable-length codes)
- To encode a message, replace each character A[k] by its corresponding binary code C[k]
- To decode a message, replace each binary code C[k] by its corresponding character A[k]

- We’ll use variable-length binary codes to obtain shortest encoded message length
- Larger frequency $\Rightarrow$ shorter binary code, and smaller frequency $\Rightarrow$ longer binary code
- To decode efficiently, the codes must satisfy the **prefix property**: No character’s code C[k] is a prefix of any other character’s code C[k’]
Goal: Map the characters to variable-length binary codes C[1...n] that satisfy the prefix property and that yield the shortest encoded message lengths

Example:

<table>
<thead>
<tr>
<th>A</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Greedy algorithm for Huffman coding:

1. First construct a Huffman tree (binary tree) as follows:

   ```
   H = new Heap( );       // min-ordered heap
   for j = 1 to n
       H.insert (new Leaf(F[j], A[j]));   // F[j] is key
   while (! H.size( ) >= 2) {
       left = H.removeMin( );
       right = H.removeMin( );
       sum = left.key + right.key;
       H.insert (new Node(sum, left, right));  // sum is key
   }
   root = H.removeMin( );
   ```
Next assign each character a binary code as follows:

Label each link to left child with 0.
Label each link to right child with 1.
Follow the path from root to each leaf node.
Concatenate the labels to obtain the binary code.

(2)
Encode a message:

\texttt{abigchefahead}

\texttt{111 00001 101 1100 1101 100 01 0001 111 100 01 111 001}

Decode the message:

\texttt{111000011011100110110001000111110001111001}

?
Why is prefix property needed to decode the message?

Why does Huffman code always satisfy the prefix property?

Why does Huffman coding yield shorter encoded messages?
  • Smaller frequency ⇒ added to tree earlier ⇒ deeper in tree ⇒ longer binary code.
  • Larger frequency ⇒ added to tree later ⇒ shallower in tree ⇒ shorter binary code.
Problem: Fractional Knapsack

Objects \{1…n\}
Profits \(P[1…n]\)
Weights \(W[1…n]\)
Maximum weight capacity \(M\)

For any fraction \(0 \leq f \leq 1\), if we take \(f \times W[k]\) weight of object \(k\), then we will obtain \(f \times P[k]\) profit.

Goal: Determine the fractions \(F[1…n]\) for each object so that \(\sum_{1 \leq k \leq n} F[k] \times W[k] \leq M\), and \(\sum_{1 \leq k \leq n} F[k] \times P[k]\) is as large as possible.

Example:

\[
\begin{array}{cccccccccc}
& 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
P & 14 & 30 & 36 & 9 & 33 & 40 & 12 & 42 & 35 \\
W & 4  & 5  & 8  & 2  & 6  & 10 & 3  & 12 & 7  \\
F &     &     &     &     &     &     &     &     &     \\
\end{array}
\]

\(M = 16\)
Greedy Algorithm 1 for Fractional Knapsack:

Total = 0
for each object k in order of descending $P[k]$ {
    $F[k] = \min \left( \frac{M}{W[k]}, 1 \right)$;
    $M = M - F[k] \times W[k]$;
    Total = Total + $F[k] \times P[k]$;
}

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>14</td>
<td>30</td>
<td>36</td>
<td>9</td>
<td>33</td>
<td>40</td>
<td>12</td>
<td>42</td>
<td>35</td>
</tr>
<tr>
<td>W</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2/5</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>Total</th>
<th>k</th>
<th>$F[k]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>42</td>
<td>6</td>
<td>2/5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>58</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All other $F[k] = 0$

Total profit = 58 (not optimal)

Greedy Algorithm 1 above is NOT correct.
Greedy Algorithm 2 for Fractional Knapsack:

Total = 0
for each object k in order of ascending W[k] {
    \( F[k] = \min \left( \frac{M}{W[k]}, 1 \right) \);
    \( M = M - F[k] \times W[k] \);
    Total = Total + F[k] \times P[k];
}

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>14</td>
<td>30</td>
<td>36</td>
<td>9</td>
<td>33</td>
<td>40</td>
<td>12</td>
<td>42</td>
<td>35</td>
</tr>
<tr>
<td>W</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1/3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>Total</th>
<th>k</th>
<th>F[k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>35</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>65</td>
<td>5</td>
<td>1/3</td>
</tr>
<tr>
<td>0</td>
<td>76</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All other \( F[k] = 0 \)

Total profit = 76  (not optimal)

Greedy Algorithm 2 above is NOT correct.
Greedy Algorithm 3 for Fractional Knapsack:

Total = 0
for each object k in order of descending ratios P[k]/W[k] {
    F[k] = min (M/W[k], 1);
    M = M – F[k]*W[k];
    Total = Total + F[k]*P[k];
}

\begin{array}{cccccccccc}
\hline
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
P & 14 & 30 & 36 & 9 & 33 & 40 & 12 & 42 & 35 \\
W & 4 & 5 & 8 & 2 & 6 & 10 & 3 & 12 & 7 \\
R & 3.5 & 6 & 4.5 & 4.5 & 5.5 & 4 & 4 & 3.5 & 5 \\
F & 1 & 1 & 5/7 & & & & & & \\
\hline
\end{array}

\begin{array}{cccc}
\hline
M & Total & k & F[k] \\
16 & 0 & 2 & 1 \\
11 & 30 & 5 & 1 \\
5 & 63 & 9 & 5/7 \\
0 & 88 & & \\
\hline
\end{array}

All other F[k] = 0

Total profit = 88 (optimal)

Greedy algorithm 3 is ALWAYS correct for all possible inputs. Why?
Greedy algorithm 3 is ALWAYS correct for all possible inputs. Why?

Contradiction argument:
Suppose Greedy Algorithm 3 is not optimal. That is, it takes too much of some object \( k \), and too little of some other object \( k' \), where \( k \) precedes \( k' \) in order of descending ratio.
So the optimal solution must have \( F[k] < 1 \) and \( F[k'] > 0 \), where \( P[k]/W[k] \geq P[k']/W[k'] \).

Now we can increase the amount of object \( k \) by \( Z \) units, and also decrease the amount of object \( k' \) by \( Z \) units, until either \( F[k] \) reaches 1 or \( F[k'] \) reaches 0.
Here \( Z = \min ((1-F[k])*W[k], F[k']*W[k']) \).
This change would increase the total profit by \( Z*(P[k]/W[k] - P[k']/W[k']) \geq 0 \).
So either Greedy Algorithm 3 does not take too much of object \( k \), or it does not take too little of other object \( k' \), or both.

Contradiction, so Greedy Algorithm 3 is correct.