Chapter 17: Greedy Algorithms

Greedy is a strategy that works well on problems with the following characteristics:

1. Greedy-choice property:
   A global optimum can be arrived at by selecting a local optimum.

2. Optimal substructure:
   An optimal solution to the problem contains an optimal solution to subproblems.

The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

"Ex. 1: An activity-selection problem"

$S = \{1, 2, \ldots, n\}$ is a set of activities needing to use a resource. Each activity $i$ has its starting time $s_i$ and finish time $f_i$ with $s_i \leq f_i$, namely, if selected, $i$ takes place during time $[s_i, f_i]$. $i$ and $j$ are compatible if their time periods are disjoint.

The activity-selection problem is the problem of selecting a largest set of mutually compatible activities.

Let $R, S$, and $T$ be the set of activities selected by the algorithm for the elements up to $k$, up to $n - 1$, and up to $n$, respectively.

By induction hypothesis, $R$ and $S$ are optimal for $\{1, \ldots, k\}$ and $\{1, \ldots, n - 1\}$, respectively.

So, an optimal solution for $\{1, \ldots, n\}$ is either $S$ or $R \cup \{n\}$. Note

$S \supset R \Rightarrow S$ is optimal,

$S = R \Rightarrow R \cup \{n\}$ is optimal,

$S \supset R \Rightarrow T = S$, and

$S = R \Rightarrow T = R \cup \{n\}$.

Thus the algorithm correctly computes an optimal solution.
Ex. 2: Knapsack

0-1 knapsack problem: Given items 1, . . . , n with values and weights, and given an integer W, find a selection of items with total weight \( \leq W \) that maximizes the sum of values.

<table>
<thead>
<tr>
<th>item</th>
<th>oz.</th>
<th>val.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Candy</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Chips</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Juice</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>Cookie</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Yo-Yo</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>Frisbee</td>
<td>10</td>
<td>30</td>
</tr>
</tbody>
</table>

Q2. What do you put in your knapsack if its capacity is 30 ounce?

Ex. 3: Huffman codes

Storage space for files can be saved by compressing them—by replacing each symbol with a unique binary string.

Here no codeword can be a prefix of any other code prefix-free. Otherwise, decoding is impossible.

The character coding problem: Given an alphabet \( C = \{a_1, \ldots, a_n\} \) and its frequencies \( f_1, \ldots, f_n \), find a set of prefix-free binary codes \( w_1, \ldots, w_n \) that minimizes the average code length

\[
\sum_{i=1}^{n} f_i \cdot w_i.
\]

* There is an \( O(nW) \) step algorithm based on dynamic programming

A greedy approach might be:

- scan items 1 . . . n and add whatever items that can squeeze in.

This strategy does not work, mostly because it does not care how much value will be gained.

<table>
<thead>
<tr>
<th>#</th>
<th>weight</th>
<th>val.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>120</td>
</tr>
</tbody>
</table>

W = 50

Q3. What is the output of the above greedy method?

Q4. What is the optimal solution?

Depict a prefix-free binary code using a binary tree, where left-branches (right-branches) are labeled by 0 (1) and leaves are uniquely labeled by the symbols in \( C \).

The code of \( a \in C \) is the label of the path from the root to \( a \).

Each node \( v \) is labeled by the frequency sum of the symbols in \( \text{subtree}(v) \).
The Huffman coding is a greedy method for obtaining an optimal prefix-free binary code.

Idea: Starting with \( D = C \), repeat the following until \( ||D|| = 1 \).
- Pick up from \( D \) two elements \( x \) and \( y \) with the lowest frequencies.
- Generate a node \( z \) whose left child is \( x \) and right child is \( y \).
- Set \( f[z] \) to \( f[x] + f[y] \).
- Replace \( x \) and \( y \) by \( z \).
- The replacement will force the codeword of \( x \) (or \( y \)) to be that of \( z \) followed by a 0 (or a 1).

An example: \( a:1, b:3, c:2, d:4, e:5 \)

1. \( a \& c \rightarrow x: \)

\[
\begin{array}{c}
\text{ } \\
\text{ } \\
0 & \text{1} \\
\text{a} & \text{c}
\end{array}
\]

2. \( x \& b \rightarrow y: \)

\[
\begin{array}{c}
\text{ } \\
\text{ } \\
0 & \text{1} \\
\text{x} & \text{b}
\end{array}
\]

The resulting tree

3. \( d \& e \rightarrow z: \)

\[
\begin{array}{c}
\text{ } \\
\text{ } \\
0 & \text{1} \\
\text{d} & \text{e}
\end{array}
\]

4. \( y \& z \rightarrow w: \)

\[
\begin{array}{c}
\text{ } \\
\text{ } \\
0 & \text{1} \\
\text{y} & \text{z}
\end{array}
\]

The idea can be implemented using a priority queue that is keyed on \( f \).

The correctness of the greedy method

**Lemma B** If \( x \) and \( y \) have the lowest frequencies in an alphabet \( C \), then \( C \) has an optimal prefix code in which \( x \) and \( y \) are sibling leaves.

**Proof** There are at least two nodes with the lowest depth. Take two of them and exchange them with \( x \) and \( y \) if they are not at the lowest depth. This will not increase the average code length.

**Lemma C** Create an alphabet \( D \) from \( C \) by replacing \( x \) and \( y \) by a single letter \( z \) such that \( f[z] = f[x] + f[y] \). Then there is a one-to-one correspondence between
- the set of code trees for \( D \) in which \( z \) is a leaf; and
- the set of code trees for \( C \) in which \( x \) and \( y \) are siblings.

**Proof** Omitted

Now suppose \( x \) and \( y \) are letters with the lowest frequencies in \( C \). Obtain an optimal code \( T \) for \( D \) and replace \( z \) by a depth-one binary tree with \( x \) and \( y \) as the leaves. Then we obtain an optimal code for \( C \).
Summary

Greedy algorithm

- arrives at an global optimum by selecting local optima
- applies to problems with the optimal substructure property

Examples: Activity Selection, Fractional Knapsack, and Huffman Code

Suggested section exercises

- 17.2-1 and 17.3-2