16 Greedy Algorithms

- Optimization algorithms typically go through a sequence of steps, with a set of choices at each
- For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do
- A greedy algorithm always makes the choice that looks best at the moment
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution

16.1 An activity-selection problem

- Suppose we have a set \( S = \{a_1, a_2, \ldots, a_n\} \) of \( n \) proposed activities that wish to use a resource (e.g., a lecture hall), which can serve only one activity at a time
- Each activity \( a_i \) has a start time \( s_i \) and a finish time \( f_i \), where \( 0 \leq s_i < f_i < \infty \)
- If selected, activity \( a_i \) takes place during the half-open time interval \([s_i, f_i)\)
Activities $a_i$ and $a_j$ are \textit{compatible} if the intervals $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap.

I.e., $a_i$ and $a_j$ are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.

We wish to select a maximum-size subset of mutually compatible activities.

We assume that the activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n$.

Consider, e.g., the following set $S$ of activities:

<table>
<thead>
<tr>
<th>i</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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.

It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger.

In fact, it is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.
The optimal substructure of the activity-selection problem

- Let $S_{ij}$ be the set of activities that start after $a_i$ finishes and that finish before $a_j$ starts.
- We wish to find a maximum set of mutually compatible activities in $S_{ij}$.
- Suppose that such a maximum set is $A_{ij}$, which includes some activity $a_k$.
- By including $a_k$ in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set $S_{ik}$ and finding mutually compatible activities in the set $S_{kj}$.

- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that
  - $A_{ik}$ contains the activities in $A_{ij}$ that finish before $a_k$ starts and
  - $A_{kj}$ contains the activities in $A_{ij}$ that start after $a_k$ finishes.
- Thus, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set $A_{ij}$ in $S_{ij}$ consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.
- The usual cut-and-paste argument shows that the optimal solution $A_{ij}$ must also include optimal solutions for $S_{ik}$ and $S_{kj}$.
This suggests that we might solve the activity-selection problem by dynamic programming.

If we denote the size of an optimal solution for the set $S_{ij}$ by $c[i, j]$, then we would have the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1$$

Of course, if we did not know that an optimal solution for the set $S_{ij}$ includes activity $a_k$, we would have to examine all activities in $S_{ij}$ to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i, j] = c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Making the greedy choice

For the activity-selection problem, we need consider only the greedy choice.

We choose an activity that leaves the resource available for as many other activities as possible.

Now, of the activities we end up choosing, one of them must be the first one to finish.

Choose the activity in $S$ with the earliest finish time, since that leaves the resource available for as many of the activities that follow it as possible.

Activities are sorted in monotonically increasing order by finish time; greedy choice is activity $a_1$. 
If we make the greedy choice, we only have to find activities that start after $a_1$ finishes.

$s_1 < f_1$ and $f_1$ is the earliest finish time of any activity $\Rightarrow$ no activity can have a finish time $\leq s_1$.

Thus, all activities that are compatible with activity $a_1$ must start after $a_1$ finishes.

Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after $a_k$ finishes.

Optimal substructure: if $a_1$ is in the optimal solution, then an optimal solution to the original problem consists of $a_1$ and all the activities in an optimal solution to the subproblem $S_1$.

Theorem 16.1 Consider any nonempty subproblem $S_k$, and let $a_m$ be an activity in $S_k$ with the earliest finish time. Then $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$.

Proof Let $A_k$ be a max-size subset of mutually compatible activities in $S_k$, and let $a_j$ be the activity in $A_k$ with the earliest finish time. If $a_j = a_m$, we are done, since $a_m$ is in a max-size subset of mutually compatible activities of $S_k$.

If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$. The activities in $A'_k$ are disjoint because the activities in $A_k$ are disjoint, $a_j$ is the first activity in $A_k$ to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that $A'_k$ is a maximum-size subset of mutually compatible activities of $S_k$ and includes $a_m$. $\blacksquare$
• We can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain.

• Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase.

• We can consider each activity just once overall, in monotonically increasing order of finish times.

A recursive greedy algorithm

\textbf{RECURSIVE-ACTIVITY-SELECTOR}(s, f, k, n)

1. \( m \leftarrow k + 1 \)

2. while \( m \leq n \) and \( s[m] < f[k] \) // find the first // activity in \( S_k \) to finish

3. \( m \leftarrow m + 1 \)

4. if \( m \leq n \)

5. return \{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)

6. else return \( \emptyset \)
An iterative greedy algorithm

Algorithm: $\text{GREEDY-ACTIVITY_SELECTOR}(s, f)$

1. $n \leftarrow s.\text{length}$
2. $A \leftarrow \{a_1\}$
3. $k \leftarrow 1$
4. for $m \leftarrow 2$ to $n$
5. if $s[m] \geq f[k]$
6. $A \leftarrow A \cup \{a_m\}$
7. $k \leftarrow m$
8. return $A$
• The set $A$ returned by the call
  \textsc{Greedy-Activity-Selector}(s, f)
  is precisely the set returned by the call
  \textsc{Recursive-Activity-Selector}(s, f, k, n)

• Both the recursive version and the iterative
  algorithm schedule a set of $n$ activities in $\theta(n)$
  time, assuming that the activities were already
  sorted initially by their finish times

### 16.3 Huffman codes

• Huffman codes compress data very effectively
  – savings of 20% to 90% are typical, depending on
    the characteristics of the data being compressed
• We consider the data to be a sequence of
  characters
• Huffman’s greedy algorithm uses a table giving
  how often each character occurs (i.e., its
  frequency) to build up an optimal way of
  representing each character as a binary string
We have a 100,000-character data file that we wish to store compactly.

We observe that the characters in the file occur with the frequencies given in the table above.

That is, only 6 different characters appear, and the character $a$ occurs 45,000 times.

Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a codeword.

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>45</td>
</tr>
<tr>
<td>$b$</td>
<td>13</td>
</tr>
<tr>
<td>$c$</td>
<td>12</td>
</tr>
<tr>
<td>$d$</td>
<td>16</td>
</tr>
<tr>
<td>$e$</td>
<td>9</td>
</tr>
<tr>
<td>$f$</td>
<td>5</td>
</tr>
</tbody>
</table>

Using a fixed-length code, requires 3 bits to represent 6 characters:

\[ a = 000, b = 001, \ldots, f = 101 \]

We now need 300,000 bits to code the entire file.

A variable-length code gives frequent characters short codewords and infrequent characters long codewords.

Here the 1-bit string 0 represents $a$, and the 4-bit string 1100 represents $f$.

This code requires

\[
(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits (savings } \approx 25\%) \]
Prefix codes

- We consider only codes in which no codeword is also a prefix of some other codeword.
- A prefix code can always achieve the optimal data compression among any character code, and so we can restrict our attention to prefix codes.
- Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file.
- E.g., with the variable-length prefix code, we code the 3-character file $abc$ as $0 \cdot 101 \cdot 100 = 0101100$.

Prefix codes simplify decoding
- No codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.
- We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file.
- In our example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to $aabe$. 

The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword.

A binary tree whose leaves are the given characters provides one such representation.

We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.”

Note that the trees are not BSTs — the leaves need not appear in sorted order and internal nodes do not contain character keys.

The trees corresponding to the fixed-length code \( a = 000, \ldots, f = 101 \) and the optimal prefix code \( a = 0, b = 101, \ldots, f = 1100 \).
• An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children
• The fixed-length code in our example is not optimal since its tree is not a full binary tree: it contains codewords beginning 10..., but none beginning 11...
• Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and
  – all character frequencies are positive, then
  – the tree for an optimal prefix code has exactly |C| leaves, one for each letter of the alphabet, and
  – exactly |C| − 1 internal nodes

• Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file
• For each character c in the alphabet C, let the attribute c.freq denote the frequency of c and let d_T(c) denote the depth of c’s leaf
• d_T(c) is also the length of the codeword for c
• Number of bits required to encode a file is thus
  \[ B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \]
  which we define as the cost of the tree T
Constructing a Huffman code

- Let $C$ be a set of $n$ characters and each character $c \in C$ be an object with an attribute $c.freq$
- The algorithm builds the tree $T$ corresponding to the optimal code bottom-up
- It begins with $|C|$ leaves and performs $|C| - 1$ “merging” operations to create the final tree
- We use a min-priority queue $Q$, keyed on $freq$, to identify the two least-frequent objects to merge
- The result is a new object whose frequency is the sum of the frequencies of the two objects

HUFFMAN($C$)
1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. for $i \leftarrow 1$ to $n - 1$
4. allocate a new node $z$
5. $z.left \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
6. $z.right \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$
7. $z.freq \leftarrow x.freq + y.freq$
8. $\text{INSERT}(Q, z)$
9. return $\text{EXTRACT-MIN}(Q)$ // return the root of the tree
To analyze the running time of HUFFMAN, let $Q$ be implemented as a binary min-heap.

For a set $C$ of $n$ characters, we can initialize $Q$ (line 2) in $O(n)$ time using the BUILD-MIN-HEAP.

The for loop executes exactly $n - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time.

Thus, the total running time of HUFFMAN on a set of $n$ characters is $O(n \lg n)$.

We can reduce the running time to $O(n \lg \lg n)$ by replacing the binary min-heap with a van Emde Boas tree.
Correctness of Huffman's algorithm

- We show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

**Lemma 16.2** Let $C$ be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ only in the last bit.

**Lemma 16.3** Let $C$, $c.freq$, $x$, and $y$ be as in Lemma 16.2. Let $C' = C - \{x, y\} \cup \{z\}$. Define $freq$ for $C'$ as for $C$, except that $z.freq = x.freq + y.freq$. Let $T'$ be any tree representing an optimal prefix code for the alphabet $C'$. Then the tree $T$, obtained from $T'$ by replacing the leaf node for $z$ with an internal node having $x$ and $y$ as children, represents an optimal prefix code for $C$.

**Theorem 16.4** Procedure HUFFMAN produces an optimal prefix code.