Take-Aways

• Have optimal substructure
  - Like Dynamic Programming, and Divide and Conquer

• Divides into a single subproblem
  - DP and DC might split into one or more subproblems

• Choose how to divide into subproblem without considering all possible subproblems - make a local choice
  - Might be able to prove that the choice is optimal
    or it might be good enough (heuristic)
  - Unlike Dynamic Programming, which must consider all possible subproblems

• Similar to Divide and Conquer but there the choice is obvious or immaterial
Overview

• Greedy Algorithms
  ⇒ Activity Selection Problem
• Elements of the Greedy Strategy
• Huffman Codes

Activity Selection Problem

• Set $S = \{a_1, a_2, ..., a_n\}$ is $n$ proposed activities
  - Each activity has a start and end time $0 \leq s_i < f_i < \infty$
    + $a_i$ takes place during half-open time interval $[s_i, f_i)$
  - What is the maximum number of events that can be scheduled in a
    lecture hall (only one event at a time)?
    + Select a maximum sized subset of mutually compatible activities

• Arrange events by finish time

<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>
Framing Problem to Find Optimal Substructure

- Key insight
  - Not all time points are interesting
  - Only time points where an activity ends or starts is of interest
- Define \( S_{ij} = \{ a_k \in S | f_i \leq s_k \text{ and } f_k \leq s_j \} \)
  - activities that start after \( a_i \) finishes and that end before \( a_j \) starts
- Optimal Substructure:
  - Suppose \( A_{ij} \) is a max set of mutually compatible activities in \( S_{ij} \)
  - Suppose that \( a_k \in A_{ij} \)
  - Can now divide problem into finding max sets for \( S_{ik} \) and \( S_{kj} \)
  - Easy to verify optimal substructure
    + \( A_{ij} \cap S_{ik} \) is an optimal solution for \( S_{ik} \) and \( A_{ij} \cap S_{kj} \) for \( S_{kj} \)
- Solutions?
  - \( \{ a_3, a_9, a_{11} \} \) are mutually compatible, but is not a maximum subset
  - \( \{ a_1, a_4, a_9, a_{11} \} \) and \( \{ a_2, a_4, a_9, a_{11} \} \) are both maximum
Greedy Choice

- What if we could choose an activity to add to our optimal solution without having to first solve all the subproblems?
- Can we just use local information to pick an activity?
  - Hopefully guarantee that the activity is part of an optimal solution
- Pick one that leaves open as many other activities as possible
  - Pick the first one to finish: $a_1$
- Now just one problem left to solve, rather than two
- All activities that end after $a_1$ are still eligible for inclusion

Defining the Recursion (Similar to Step 2 of DP)

- Define $c[i, j]$ as size of optimal solution for $S_{ij}$

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

- Dynamic programming:
  - top-down recursive algorithm with memoization
  - bottom-up algorithm ordered by distance between $i$ and $j$
Proof that Greedy Choice is Optimal

- Is this optimal?
- Proof by Contradiction:
  - Assume \( a_1 \) is not part of any optimal solution
  - What activities are in optimal solution that are not compatible with \( a_1 \)?
  - If more than one, one must finish before \( a_1 \) finished. Contradiction
  - Must have finished after \( a_1 \)
    + Can replace it by \( a_1 \) giving set of same size of compatible activities: Contradiction

Recursive Algorithm

- Called with array start time array \( s \), finish time array \( f \),
  start point \( k \), end point \( n \)
  - Initial called with \( k=0, n=\text{end time} \)

\[
\text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, k, n)
\]

1. \( m = k + 1 \)
2. \( \text{while } m \leq n \text{ and } s[m] < f[k] \) \( \rightarrow \) find the first activity in \( S_k \) to finish
3. \( m = m + 1 \)
4. \( \text{if } m \leq n \)
5. \( \text{return } \{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n) \)
6. \( \text{else return } \emptyset \)

- Running time?
Iterative Algorithm

\[ \text{GREEDY-ACTIVITY-SELECTOR}(s, f) \]

1. \( n = s.\text{length} \)
2. \( A = \{a_1\} \)
3. \( k = 1 \)
4. \( \text{for } m = 2 \text{ to } n \)
5. \( \quad \text{if } s[m] \geq f[k] \)
6. \( \quad A = A \cup \{a_m\} \)
7. \( \quad k = m \)
8. \( \text{return } A \)

Overview

- Greedy Algorithms
- Activity Selection Problem
- Elements of the Greedy Strategy
- Huffman Codes
Steps

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (16.2), but we bypassed developing a recursive algorithm based on this recurrence.)
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

More Directly

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve
- Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is always safe
- Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem
- Underneath every greedy algorithm is a almost always a more cumbersome DP solution
Greedy Choice Property

- In DP, choice depends on solutions to subproblems
  - Usually solve DP bottom-up
- In Greedy, make choice that seems best (hopefully it is best)
  - Then solve the remaining problem
  - Top-down approach
- Can often put choices into some order to allow very efficient choosing of greedy choice
  - Last example: sort by finish times

Contrast: Knapsack problem

A thief robbing a store finds \( n \) items. The \( i \)th item is worth \( v_i \) dollars and weighs \( w_i \) pounds, where \( v_i \) and \( w_i \) are integers. Thief can just carry \( W \) pounds. What items should he take?

Example:
- item 1: weighs 10 pounds worth $60
- item 2: weighs 20 pounds worth $100
- item 3: weighs 30 pounds worth $120

Version 1: Must take it or leave it
Version 2: Can take fractional amounts
Overview

- Greedy Algorithms
- Activity Selection Problem
- Elements of the Greedy Strategy
  ⇒ Huffman Codes

Binary Knapsack Problem

- Step 1: Does it have optimal substructure?
  - If you have an optimal solution for weight $W$:
    + and you take out an item from optimal solution, say $i$ that weights $w_i$
    + Remaining items are optimal solution for weight $W-w_i$ excluding item $i$
    + So exhibits optimal substructure!

- Step 2: Recurrence relationship
  - Can consider items in order
    + Let $b(0, i, w)$ be best value for items 0 through $i$ having weight at most $w$
    + $b(0, 0, w) = 0$ if $w_0 < w$ otherwise $v_0$
    + $b(0, i, w)$ is either $b(0, i-1, w)$ or $b(0, i-1, w-w_i) + v_i$

- For bottom-up, how do we order the subproblems?
- Running Time?
Encoding Data

- Supposed we have a 100,000 character data file we wish to store compactly
- There are only 6 different characters, and we can store characters in as many bits as we want
- Fixed Length Code:
  - Each character encoded in same number of bits
  - How many bits are needed for 6 characters?
- Variable length code:
  - Each character can use different number of bits
  - Each code needs to be prefix-free
    - No code can be the prefix of another code
    - Ensures that decoding is unambiguous starting at the beginning of the file

Example

- a:45 means character 'a' appeared 45,000 times
  - How much storage does fixed length code need versus variable?
  
* Prefix free: only leaf nodes can have a code
Example (Continued)

- Fixed
  \[(45+13+12+16+9+5)\times 3 = 300,000 \text{ bits}\]

- Variable
  \[45+(12+13+16)\times 3 + (5+9)\times 4 = 45 + 41\times 3 + 14\times 4 = 45 + 123 + 56 = 224,000 \text{ bits}\]

Code

\begin{verbatim}
HUFFMAN(C)
1    n = |C|
2    Q = C
3    for i = 1 to n - 1
4        allocate a new node \(z\)
5        \(z.left = x = \text{EXTRACT-MIN}(Q)\)
6        \(z.right = y = \text{EXTRACT-MIN}(Q)\)
7        \(z.freq = x.freq + y.freq\)
8        \text{INSERT}(Q, z)
9    return \text{EXTRACT-MIN}(Q)  // return the root of the tree
\end{verbatim}
Example Run

HUFFMAN (C)
1 n = |C|
2 Q = C
3 for i = 1 to n - 1
4 allocate a new node z
5 z.left = x = EXTRACT-MIN(Q)
6 z.right = y = EXTRACT-MIN(Q)
7 z.freq = x.freq + y.freq
8 INSERT(Q, z)
9 return EXTRACT-MIN(Q) // return the root of the tree

Example Run

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

Efficiency of Code

HUFFMAN (C)
1 n = |C|
2 Q = C
3 for i = 1 to n - 1
4 allocate a new node z
5 z.left = x = EXTRACT-MIN(Q)
6 z.right = y = EXTRACT-MIN(Q)
7 z.freq = x.freq + y.freq
8 INSERT(Q, z)
9 return EXTRACT-MIN(Q) // return the root of the tree

- How much time does extract min and insert take?
  - Could take $O(n)$ using a simple array of freq.
  - With min heap?
- Overtime?
  - How much time to build the initial heap?
  - How much time to run the rest?
Preliminaries

- A binary tree gives a prefix tree iff all characters are on leaves (nodes with no children), and not on internal nodes.
- In an optimal prefix tree, every node will either have no children (a leaf) or both children.
- Every non-trivial code will have an optimal prefix tree in which there are at least two nodes at maximum depth that are siblings.
Lemma 16.2: Case 2

Case 2: \( x.\text{freq} \neq b.\text{freq} \). So \( x \neq y \)

Create \( T' \) by swamping \( a \) and \( x \)

Create \( T'' \) by swamping \( b \) and \( y \)

Since \( x \neq b \), \( x \) and \( y \) will be siblings in \( T'' \)

\[
\begin{array}{cccccccc}
T & / & \backslash & T' & / & \backslash & T'' & / \\
\backslash & \backslash & x & y & / & \backslash & a & y \\
\backslash & \backslash & a & b & x & b & 90 & 85 \\
\backslash & \backslash & 90 & 95 & 80 & 85 & 90 & 95 \\
\end{array}
\]

\[ d_T(x) \leq d_T(a) \text{ and } x.\text{freq} \leq a.\text{freq}, \text{ so } B(T') \leq B(T) \]

Similarly \( B(T'') \leq B(T') \)

Since \( T \) is optimal \( B(T'') \geq B(T) \)

So \( T \) and \( T'' \) must have same costs so \( T'' \) is also optimal

Remember: \( d_T(c) \) depth of leaf with character \( c \) in tree \( T \)
Lemma 16.2

Claim: There is an optimal prefix tree in which the lowest two frequency characters are siblings at maximum depth.

Proof:
Let \( x \) and \( y \) have the lowest frequencies such that \( x.\text{freq} \leq y.\text{freq} \). Let \( T \) be optimal prefix tree, with \( a \) and \( b \) siblings of deepest node, with \( a.\text{freq} \leq b.\text{freq} \). So \( x.\text{freq} \leq a.\text{freq} \) and \( y.\text{freq} \leq b.\text{freq} \).

\[
\begin{array}{cccc}
/ & / & \backslash \\
/ & \backslash & x & y \\
a & b & 100 & 92 \\
90 & 85 & \\
\end{array}
\]

Case 1: \( x.\text{freq} = y.\text{freq} = a.\text{freq} = b.\text{freq} \)
Trivially true as \( T \) has lowest two frequency characters \( a \) and \( b \) at max depth.

Lemma 16.3

Claim: \( T_{xy} \) is an optimal prefix tree for \( C_{xy} \).

Proof by contradiction:
Assume that \( T_{xy} \) is not optimal.
Let \( T'_{xy} \) be an optimal prefix tree for \( C_{xy} \).
So \( B(T'_{xy}) < B(T_{xy}) \).
Due to Lemma 16.2, WLG, assume that \( T \) has \( x \) and \( y \) under the same node.
Construct \( T'_z \) from \( T'_{xy} \) by replace \( x \) and \( y \) by \( z \) as above.

\[
B(T'_z) = B(T'_{xy}) - x.\text{freq} - y.\text{freq} \\
\leq B(T_{xy}) - x.\text{freq} - y.\text{freq} \\
= B(T_z)
\]

\( T'_z \) is a prefix tree with lower cost than \( T_z \), but \( T_z \) is optimal. Contradiction.
Preface to Lemma 16.3

Let $C_{xy}$ be a given alphabet with frequency $c$.freq for each character $c \in C$ where $x$ and $y$ be the least two frequent characters in $C$.

Let $C_z$ be same as $C_{xy}$, but with $x$ and $y$ replaced by $z$ and $z$'s frequency sum of $x$ and $y$'s. Let $T_z$ be an optimal prefix tree for $D$.

Create $T_{xy}$ by replacing $z$’s leaf by a node with $x$ and $y$ as its children.

$B(T_z) = B(T_{xy}) - x$.freq $- y$.freq

Claim: $T_{xy}$ is an optimal prefix tree for $C_{xy}$. 