Introduction

- Greedy methods: A technique for solving optimization problems
  - Choose a solution to a problem that is best per an objective function
- Similar to dynamic programming (covered later) in that we examine subproblems, exploiting optimal substructure property
- Key difference: In dynamic programming we considered all possible subproblems
- In contrast, a greedy algorithm at each step commits to just one subproblem, which results in its greedy choice (locally optimal choice)
- Examples: Minimum spanning tree, single-source shortest paths
Activity Selection (1)

- Consider the problem of scheduling classes in a classroom
- Many courses are candidates to be scheduled in that room, but not all can have it (can’t hold two courses at once)
- Want to maximize utilization of the room
- This is an example of the activity selection problem:
  - Given: Set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ proposed activities that wish to use a resource that can serve only one activity at a time
  - $a_i$ has a start time $s_i$ and a finish time $f_i$, $0 \leq s_i < f_i < \infty$
  - If $a_i$ is scheduled to use the resource, it occupies it during the interval $[s_i, f_i)$ ⇒ can schedule both $a_i$ and $a_j$ iff $s_i \geq f_j$ or $s_j \geq f_i$ (if this happens, then we say that $a_i$ and $a_j$ are compatible)
  - Goal is to find a largest subset $S' \subseteq S$ such that all activities in $S'$ are pairwise compatible
  - Assume that activities are sorted by finish time:

$$f_1 \leq f_2 \leq \cdots \leq f_n$$
Activity Selection (2)

Sets of mutually compatible activities: \{a_3, a_9, a_{11}\}, \{a_1, a_4, a_8, a_{11}\}, \{a_2, a_4, a_9, a_{11}\}
Optimal Substructure of Activity Selection

- Let $S_{ij}$ be set of activities that start after $a_i$ finishes and that finish before $a_j$ starts
- Let $A_{ij} \subseteq S_{ij}$ be a largest set of activities that are mutually compatible
- If activity $a_k \in A_{ij}$, then we get two subproblems: $S_{ik}$ and $S_{kj}$
- If we extract from $A_{ij}$ its set of activities from $S_{ik}$, we get $A_{ik} = A_{ij} \cap S_{ik}$, which is an optimal solution to $S_{ik}$
  - If it weren’t, then we could take the better solution to $S_{ik}$ (call it $A_{ik}'$) and plug its tasks into $A_{ij}$ and get a better solution
- Thus if we pick an activity $a_k$ to be in an optimal solution and then solve the subproblems, our optimal solution is $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, which is of size $|A_{ik}| + |A_{kj}| + 1$
Let $c[i, j]$ be the size of an optimal solution to $S_{ij}$

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

We try all $a_k$ since we don’t know which one is the best choice...

...or do we?
Greedy Choice

- What if, instead of trying all activities $a_k$, we simply chose the one with the earliest finish time of all those still compatible with the scheduled ones?
- This is a **greedy choice** in that it maximizes the amount of time left over to schedule other activities.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be set of activities that start after $a_k$ finishes.
- If we greedily choose $a_1$ first (with earliest finish time), then $S_1$ is the only subproblem to solve.
Greedy Choice (2)

**Theorem:** Consider any nonempty subproblem \( S_k \) and let \( a_m \) be an activity in \( S_k \) with earliest finish time. Then \( a_m \) is in some maximum-size subset of mutually compatible activities of \( S_k \).

**Proof:**
- Let \( A_k \) be an optimal solution to \( S_k \) and let \( a_j \) have earliest finish time of all in \( A_k \).
- If \( a_j = a_m \), we’re done.
- If \( a_j \neq a_m \), then define \( A'_k = A_k \setminus \{a_j\} \cup \{a_m\} \).
- Activities in \( A' \) are mutually compatible since those in \( A \) are mutually compatible and \( f_m \leq f_j \).
- Since \( |A'_k| = |A_k| \), we get that \( A'_k \) is a maximum-size subset of mutually compatible activities of \( S_k \) that includes \( a_m \).

What this means is that there is an optimal solution that uses the greedy choice.
Recursive-Activity-Selector \((s, f, k, n)\)

1. \(m = k + 1\)
2. \(\textbf{while} \ m \leq n \text{ and } s[m] < f[k] \ \textbf{do} \)
   3. \(m = m + 1\)
4. \(\textbf{end}\)
5. \(\textbf{if} \ m \leq n \ \textbf{then}\)
   6. \(\textbf{return} \ \{a_m\} \cup \text{Recursive-Activity-Selector}(s, f, m, n)\)
7. \(\textbf{else return} \ \emptyset\)
Recursive Algorithm Example

```
\begin{array}{c|c|c}
  k & s_k & f_k \\
  \hline
  0 & 0 & a_0 \\
  1 & 1 & 4 & a_1 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 0, 11) \\
  2 & 3 & 5 & a_2 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 1, 11) \\
  3 & 0 & 6 & a_3 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 1, 11) \\
  4 & 5 & 7 & a_4 \text{ \hspace{1cm} } m = 4 \\
  5 & 3 & 9 & a_5 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 4, 11) \\
  6 & 5 & 9 & a_6 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 4, 11) \\
  7 & 6 & 10 & a_7 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 4, 11) \\
  8 & 8 & 11 & a_8 \text{ \hspace{1cm} } m = 8 \\
  9 & 8 & 12 & a_9 \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 8, 11) \\
 10 & 2 & 14 & a_{10} \text{ \hspace{1cm} } \text{Recursive-Activity-Selector}(s, f, 8, 11) \\
11 & 12 & 16 & a_{11} \text{ \hspace{1cm} } m = 11
\end{array}
```

```
  time
  0   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16
```
Greedy-Activity-Selector($s, f, n$)

1 $A = \{a_1\}$
2 $k = 1$
3 for $m = 2$ to $n$ do
4     if $s[m] \geq f[k]$ then
5         $A = A \cup \{a_m\}$
6         $k = m$
7     end
8 end
9 return $A$

What is the time complexity?
Greedy vs Dynamic Programming (1)

- When can we get away with a greedy algorithm instead of DP?
- When we can argue that the greedy choice is part of an optimal solution, implying that we need not explore all subproblems

Example: The knapsack problem

- There are $n$ items that a thief can steal, item $i$ weighing $w_i$ pounds and worth $v_i$ dollars
- The thief’s goal is to steal a set of items weighing at most $W$ pounds and maximizes total value
- In the 0-1 knapsack problem, each item must be taken in its entirety (e.g., gold bars)
- In the fractional knapsack problem, the thief can take part of an item and get a proportional amount of its value (e.g., gold dust)
There’s a greedy algorithm for the fractional knapsack problem:
- Sort the items by \( v_i/w_i \) and choose the items in descending order.
- Has greedy choice property, since any optimal solution lacking the greedy choice can have the greedy choice swapped in.
  - Works because one can always completely fill the knapsack at the last step.

Greedy strategy does not work for 0-1 knapsack, but do have \( O(nW) \)-time dynamic programming algorithm:
- Note that time complexity is pseudopolynomial.
- Decision problem is NP-complete.
Greedy vs Dynamic Programming (3)

Problem instance 0-1 (greedy is suboptimal) Fractional
Huffman Coding

- Interested in encoding a file of symbols from some alphabet
- Want to minimize the size of the file, based on the frequencies of the symbols
- A **fixed-length code** uses $\lceil \log_2 n \rceil$ bits per symbol, where $n$ is the size of the alphabet $C$
- A **variable-length code** uses fewer bits for more frequent symbols

<table>
<thead>
<tr>
<th>Frequency (in thousands)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

Fixed-length code uses 300k bits, variable-length uses 224k bits
Huffman Coding (2)

Can represent any encoding as a binary tree

If \( c.freq \) = frequency of codeword and \( d_T(c) \) = depth, cost of tree \( T \) is

\[
B(T) = \sum_{c \in C} c.freq \cdot d_T(c)
\]
Algorithm for Optimal Codes

- Can get an optimal code by finding an appropriate **prefix code**, where no codeword is a prefix of another
- Optimal code also corresponds to a full binary tree
- Huffman’s algorithm builds an optimal code by greedily building its tree
- Given alphabet $C$ (which corresponds to leaves), find the two least frequent ones, merge them into a subtree
- Frequency of new subtree is the sum of the frequencies of its children
- Then add the subtree back into the set for future consideration
Huffman($C$)

1. $n = |C|$
2. $Q = C$                 // min-priority queue
3. for $i = 1$ to $n - 1$ do
6.     allocate 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.     $\text{Insert}(Q, z)$
4. end
9. return $\text{Extract-Min}(Q)$       // return root

Time complexity: $n - 1$ iterations, $O(\log n)$ time per iteration, total $O(n \log n)$
Huffman Example

(a) f:5  e:9  c:12  b:13  d:16  a:45

(b) c:12  b:13

(c) 14
d:16

14

(a:45)

(d) 30
14
d:16

(b:13  c:12)

0 1

14

(f) 55
100

55

(a:45)

(c:12  b:13  d:16)

14

0 1

14

0 1

(f:5  e:9)

(c:12  b:13)

0 1

14

0 1

(f:5  e:9)
Optimal Coding Has Greedy Choice Property (1)

Lemma: Let $C$ be an alphabet in which symbol $c \in C$ has frequency $c.freq$ and let $x, y \in C$ have lowest frequencies. Then there exists an optimal prefix code for $C$ in which codewords for $x$ and $y$ have same length and differ only in the last bit.

Proof: Let $T$ be a tree representing an arbitrary optimal prefix code, and let $a$ and $b$ be siblings of maximum depth in $T$

Assume, w.l.o.g., that $x.freq \leq y.freq$ and $a.freq \leq b.freq$

Since $x$ and $y$ are the two least frequent nodes, we get $x.freq \leq a.freq$ and $y.freq \leq b.freq$

Convert $T$ to $T'$ by exchanging $a$ and $x$, then convert to $T''$ by exchanging $b$ and $y$

In $T''$, $x$ and $y$ are siblings of maximum depth
Optimal Coding Has Greedy Choice Property (2)
Optimal Coding Has Greedy Choice Property (3)

Cost difference between $T$ and $T'$ is $B(T) - B(T')$:

\[
\begin{align*}
&= \sum_{c \in C} c.\text{freq} \cdot d_T(c) - \sum_{c \in C} c.\text{freq} \cdot d_{T'}(c) \\
&= x.\text{freq} \cdot d_T(x) + a.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_{T'}(x) - a.\text{freq} \cdot d_{T'}(a) \\
&= x.\text{freq} \cdot d_T(x) + a.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_T(x) \\
&= (a.\text{freq} - x.\text{freq})(d_T(a) - d_T(x)) \geq 0
\end{align*}
\]

since $a.\text{freq} \geq x.\text{freq}$ and $d_T(a) \geq d_T(x)$

Similarly, $B(T') - B(T'') \geq 0$, so $B(T'') \leq B(T)$, so $T''$ is optimal
**Lemma:** Let $C$ be an alphabet in which symbol $c \in C$ has frequency $c.freq$ and let $x, y \in C$ have lowest frequencies. Let $C' = C \setminus \{x, y\} \cup \{z\}$ and $z.freq = x.freq + y.freq$. Let $T'$ be any tree representing an optimal prefix code for $C'$. Then $T$, which is $T'$ with leaf $z$ replaced by internal node with children $x$ and $y$, represents an optimal prefix code for $C$.

**Proof:** Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$,

$$x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1)$$

$$= z.freq \cdot d_{T'}(z) + (x.freq + y.freq)$$

Also, since $d_T(c) = d_{T'}(c)$ for all $c \in C \setminus \{x, y\}$,

$B(T) = B(T') + x.freq + y.freq$ and $B(T') = B(T) - x.freq - y.freq$
Optimal Coding Has Optimal Substructure Property (2)

- Assume that $T$ is not optimal, i.e., $B(T'') < B(T)$ for some $T''$
- Assume w.l.o.g. (based on previous lemma) that $x$ and $y$ are siblings in $T''$
- In $T''$, replace $x$, $y$, and their parent with $z$ such that $z.freq = x.freq + y.freq$, to get $T'''$:

  $$B(T''') = B(T'') - x.freq - y.freq$$  (from prev. slide)
  $$< B(T) - x.freq - y.freq$$  (from $T$ suboptimal assumption)
  $$= B(T')$$  (from prev. slide)

- This contradicts assumption that $T'$ is optimal for $C'$