Lecture: Analysis of Algorithms (CS583 - 002)

Amarda Shehu

Fall 2017
Dynamic Programming

- Longest Common Subsequence
- Dynamic Programming Hallmark #1: Optimal Substructure
- Dynamic Programming Solution to LCS
- Dynamic Programming Hallmark #2: Overlapping subproblems
Dynamic Programming is a design technique like divide-and-conquer.

Example: Longest Common Subsequence (LCS)

Given two sequences \(x[1 \ldots m]\) and \(y[1 \ldots n]\), find a longest subsequence common to them both:

\[
x: \quad A \quad B \quad C \quad B \quad D \quad A \quad B
\]
\[
y: \quad B \quad D \quad C \quad A \quad B \quad A
\]

\(BCBA = LCS(x, y)\)
Brute-force LCS Algorithm

Check every subsequence of $x[1 \ldots m]$ to see if it is also a subsequence of $y[1 \ldots n]$.

Analysis:

- There are $2^m$ possible subsequences of $x$, since each bit-vector of length $m$ represents a distinct subsequence of $x$
- Checking each one of them into $y$ takes $O(n)$ time
- So, worst-case running time is $O(n \cdot 2^m)$
- An exponential running time is impractical
A Better Algorithm

Simplification:
- Look at the length of a longest common subsequence
- Extend the algorithm to find the LCS itself

Notation: Let $|s|$ denote the length of a sequence $s$

Proposed Strategy: Consider prefixes of $x$ and $y$
- Define $c[i, j] = |\text{LCS}(x[1 \ldots i], y[1 \ldots j])|$ 
- Then, $\text{LCS}(x, y) = c[m, n]$
Theorem:

\[
c[i, j] = \begin{cases} 
c[i-1, j-1] + 1 & \text{if } x[i] = y[j] \\
\max\{c[i-1, j], c[i, j-1]\} & \text{otherwise}
\end{cases}
\]

Proof: Case \(x[i] = y[j]\)

Let \(z[1 \ldots k] = \text{LCS}(x[1 \ldots i], y[1 \ldots j])\), where \(c[i, j] = k\). Then \(z[k] = x[i]\). Otherwise, \(z\) could be extended by \(x[i]\). Moreover, \(z[1 \ldots k - 1] = \text{LCS}(x[1 \ldots i - 1], y[1 \ldots j - 1])\).
Continuing Proof in Case 1

Claim: \[ z[1 \ldots k - 1] = LCS(x[1 \ldots i - 1], y[1 \ldots j - 1]) \]

Proof of Claim by Contradiction:

- Suppose \( w \) is a longer common subsequence of \( x[1 \ldots i - 1] \) and \( y[1 \ldots j - 1] \). That is, \( |w| > k - 1 \).

- Then, cut and paste: \( w \cdot z[k] \) (\( w \) concatenated by \( z[k] \)) is also a common subsequence of \( x[1 \ldots i] \) and \( y[1 \ldots j] \). Since \( |w \cdot z[k]| > k \), we have reached a contradiction, proving the above claim.

- So, \( c[i - 1, j - 1] = k - 1 \), which implies that \( c[i, j] = c[i - 1, j - 1] + 1 \).

Case 2 is proven with a similar argument.
**Optimal substructure**

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of $z$ is an LCS of a prefix of $x$ and a prefix of $y$. 
Recursive Algorithm for LCS

\[ \text{LCS}(x, y, i, j) \]

1. \( \text{if } x[i] = y[j] \text{ then} \)

2. \( c[i, j] \leftarrow \text{LCS}(x, y, i - 1, j - 1) + 1 \)

3. \( \text{else } c[i, j] = \max \{ \text{LCS}(x, y, i - 1, j), \text{LCS}(x, y, i, j - 1) \} \)

**Worst-case:** When \( x[i] \neq y[j] \), the algorithm evaluates two subproblems, each one with only one parameter decremented.
The height of the recursion tree is $m + n$. It seems that the work is exponential because we are solving the same subproblems over and over. We need to remember subproblems once we solve them!
Dynamic Programming: Hallmark # 2

**Overlapping subproblems**

A recursive solution contains a “small” number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths $m$ and $n$ is only $mn$. 
Memoization Algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
1: if \( c[i,j] = NIL \) then
2: if \( x[i] = y[j] \) then
3: \( c[i,j] \leftarrow LCS(x, y, i - 1, j - 1) + 1 \)
4: else \( c[i,j] = \max\{LCS(x, y, i - 1, j), LCS(x, y, i, j - 1)\} \)

Running Time Analysis: \( T(n, m) \in \theta(m \cdot n) \) since the amount of work per table entry is constant.
Space Analysis: \( S(n, m) \in \theta(m \cdot n) \) since we only store the table.
Dynamic Programming Algorithm

**Idea:**
- Fill the table top left to bottom right
- $T(n, m) \in \theta(m \cdot n)$
- Reconstruct the LCS by tracing backwards
- $S(n, m) \in \theta(m \cdot n)$
- Exercise: reduce $S(n, m)$ to $O(\min\{m, n\})$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Amarda Shehu  
Lecture: Analysis of Algorithms (CS583 - 002)
Lecture: Analysis of Algorithms (CS583 - 002)

Amarda Shehu

Fall 2017
1 Greedy Algorithms
   - In the Context of the Following Problems
     - The 0/1 Integer Knapsack Problem
     - The Fractional Knapsack Problem
     - Huffman Coding
Greedy Algorithms

- Used to solve optimization problems
- A greedy algorithm builds a solution one step at a time
- At each step, the algorithm makes the *currently* best choice from a small number of choices
- The currently best choice is also referred to as the *locally* optimal choice
- Greedy algorithms are similar to DP algorithms in:
  - the solution is efficient if the problem exhibits substructure
- BUT
  - The greedy solution may not be optimal even if the problem exhibits optimal substructure
When to Apply the Greedy Approach

When to Design Greedy Algorithms

- On problems with optimal substructure where the greedy approach is the optimal approach
- These problems are said to have the greedy-choice property: a “locally optimal” choice leads to a “globally optimal” solution
- Applying the greedy approach to other problems that do not have this property can yield suboptimal solutions
- Suboptimal solutions may be good enough approximations of the optimal solution on some applications
  - Instance: when globally optimal solution is too expensive to compute
Sample Problems to Illustrate Greedy Algorithms

- The 0/1 Integer Knapsack Problem
- The Fractional Knapsack Problem
- Variable-length (Huffman) Coding
The 0/1 Integer Knapsack Problem

- Given \( n \) objects
- Each object has an integer weight \( w_i \) and integer profit \( p_i \)
- You have a knapsack with an integer weight capacity \( M \)
- Problem: Find the subset of \( n \) objects that fits in the knapsack and gives the maximum total profit
Examples of Possible Solutions

Say the knapsack has capacity $M = 20$:

<table>
<thead>
<tr>
<th>Object $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit $p_i$</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td>3</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>Weight $w_i$</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>4</td>
<td>14</td>
<td>5</td>
</tr>
</tbody>
</table>

Possible solutions:
- Put items 1-3 in knapsack: Total weight is 20, and profit is 25
- Put items 1, 2, 4, and 6: Total weight now is 19, profit is 32
- Other possible solutions ...

How long does it take to evaluate all *feasible* solutions?
MAXIMIZE

\[ p_1 \cdot x_1 + p_2 \cdot x_2 \ldots p_n \cdot x_n \]

such that (SUBJECT TO CONSTRAINT)

\[ w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots w_n \cdot x_n \leq M \]

where \( x_i \in \{0, 1\} \) for \( i \in \{1, 2, \ldots, n\} \)
Define $f_i(y)$ to be the optimal solution to the subproblem:

$$\text{MAXIMIZE } p_1 \cdot x_1 + p_2 \cdot x_2 \cdots p_i \cdot x_i$$

such that $w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots w_i \cdot x_i \leq y$

where $x_j \in \{0, 1\}$ for $j \in \{1, 2, \ldots, i\}$

Then we see the optimal substructure of the solution:

$$f_i(y) = \begin{cases} 
\max\{f_{i-1}(y), p_i + f_{i-1}(y - w_i)\} & \text{if } y \geq w_i \\
 f_{i-1}(y) & \text{if } y < w_i 
\end{cases}$$
Seeing the Optimal Substructure

- \[ f_1(y) = \text{the maximum profit for capacity } y \text{ considering only object 1}, \text{ where } x_1 \in \{0,1\} \]
- \[ f_2(y) = \text{the maximum profit for capacity } y \text{ considering only objects 1 and 2}, \text{ where } x_1, x_2 \in \{0,1\} \]
- Consider what happens when we consider object 3:
  - If \( x_3 = 0 \), this means we do not choose to include object 3 in the knapsack. So, maximum profit is what it used to be using objects 1, 2: \( f_3(y) = f_2(y) \)
  - Else, we choose to include, which means we only have \( y - w_3 \) capacity for objects 1, 2:
    - We do not know a priori whether \( x_3 \) should be 0 or 1
    - The only criterion is that \( f_3(y) = \max\{f_2(y), f_2(y - w_3)\} \)
Computing $f_i(y)$

The optimal substructure dictates that we compute $f_{i-1}(y)$ for all capacities $y \in \{0, 1, \ldots, M\}$

The recursion shows it is only necessary to save $f_i(y)$ and $f_{i-1}(y)$ for all possible values of $y$

Basic Idea:

- Set $f_0(y) = 0 \ \forall y \in \{0, 1, \ldots, M\}$
- Compute $f_1(y) \ \forall y \in \{0, 1, \ldots, M\}$
- ... 
- Compute $f_n(y) \ \forall y \in \{0, 1, \ldots, M\}$

Question: How big is the matrix that stores solutions to subproblems?
Let $p = (7, 6, 12, 3, 12, 16)$, $w = (2, 8, 10, 4, 14, 5)$, and $M = 20$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>10</th>
<th>...</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>$f_1$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>...</td>
<td>7</td>
<td>...</td>
<td>7</td>
</tr>
<tr>
<td>$f_2$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>...</td>
<td>13</td>
<td>...</td>
<td>13</td>
</tr>
<tr>
<td>$f_3$</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>...</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Greedy Approach for the Knapsack Problem

Reorder the objects by increasing weight (focus on feasible solutions):

<table>
<thead>
<tr>
<th>Object</th>
<th>Profit ( p_i )</th>
<th>Weight ( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>7 3 16 6 12 12</td>
<td>2 4 5 8 10 14</td>
</tr>
</tbody>
</table>

A potential greedy solution:
- Put object with smallest weight in knapsack first
- Add objects (according to sorted order of weights) into knapsack as long as there is capacity
- What is the resulting greedy solution when \( M = 20 \)?
- What is the time complexity of this approach?
Another Greedy Approach

- Instead, sort the items by descending $p_i/w_i$ ratios (focusing on maximizing profit while minimizing weight)
- Examine each object $i \in \{1, \ldots, n\}$ in this order
- If object fits in knapsack, take it
- What is the time complexity now?
- Does this greedy approach find the optimal solution to the 0/1 Integer Knapsack Problem?
The 0/1 Knapsack problem can be solved optimally by Dynamic Programming, as illustrated.

The problem cannot be solved optimally by the Greedy Approach.

Why? Because the 0/1 knapsack problem does not have the greedy-choice property.

To show that the greedy approach does not work, we have to provide a counterexample.
Greedy Approach: Not Optimal for 0/1 Knapsack Problem

Say knapsack has capacity $M = 5$ and there are $n = 3$ items:

<table>
<thead>
<tr>
<th>Object</th>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit</td>
<td>$p_i$</td>
<td>6</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Weight</td>
<td>$w_i$</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Profit/Weight</td>
<td>$p_i/w_i$</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- A greedy algorithm that chooses by highest profit/weight chooses items 1 and 2 for a total profit of 16.
- Optimal solution: items 2 and 3 for a total value of 22.
- Hence, greedy algorithm does not give optimal solution.
- However, the greedy approach gives an optimal solution to the fractional knapsack problem.
The Fractional Knapsack Problem

- Given $n$ objects
- Each object has an integer profit $p_i$
- Each object has a fractional weight $w_i$
- You can take fractions of an object
- You have a knapsack with weight capacity $M$, where $M$ is not necessarily an integer
- Problem: Fit objects (taking even fractions of them) that give the maximum total profit
An Optimal Greedy Solution to the Fractional Knapsack Problem

- Sort the items by descending $p_i/w_i$ ratios (focusing on maximizing profit while minimizing weight)
- Examine each object $i \in \{1, \ldots, n\}$ in this order
- If object fits in knapsack, take it
- What is the time complexity?
- Why does this greedy approach find the optimal solution to the Fractional Knapsack Problem?
Proof of Correctness

Let $X \in \{1, 2, \ldots, k\}$ be the optimal items taken

- Consider item $j$ with associated $(p_j, w_j)$ that has the highest $p_j/w_j$ ratio
- If $j$ is not used in $X$, then $X$ is not optimal: We can remove portions of items with a total weight of $w_j$ from $X$ and add $j$ instead.
- Repeating this process, you see that the greedy approach changes $X$ considering all items without decreasing the total value of $X$. 
Consider a message consisting of \( k \) characters (with known frequencies).

We want to encode this message using a binary cipher.

That is, we want to assign \( d \) bits to each letter:

<table>
<thead>
<tr>
<th>Letter</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency ( \times 10^3 )</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length encoding</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
</tbody>
</table>

A message consisting of 100,000 \( a-f \) characters would require 300,000 bits of storage!!!
How about Variable-length Encoding?

- We could assign a variable-length encoding instead:

<table>
<thead>
<tr>
<th>Letter</th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (\times 10^3)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length encoding</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length encoding</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

- A message like 001011101 parses uniquely
  - That is to say that one can decode this cipher uniquely
  - This result is based on the fact that no code is a prefix of another for the encoded characters

- Only 9 bits are used instead.
Optimum Source Coding Problem

**Problem:** Given an alphabet \( A = \{a_1, \ldots, a_n\} \) with frequency distribution \( f(a_i) \), find a binary prefix code \( C \) for \( A \) that minimizes the number of bits

\[
B(C) = \sum_{i=1}^{n} f(a_i) \cdot L(c(a_i))
\]

needed to encode a message of \( \sum_{i=1}^{n} f(a_i) \) characters, where \( c(a_i) \) is the codeword/code for encoding \( a_i \), and \( L(c(a_i)) \) is the length of this code.

**Solution:** Huffman developed a greedy algorithm for producing a minimum-cost prefix code. The code that is produced is called a *Huffman Code*. 
Basic Idea Behind Huffman Coding

- A binary tree constructs codes
- 1-1 correspondence between the leaves and the characters
- The label of each leaf is the frequency of each character
- Left edges are labeled 0, right edges are labeled 1
- Path from root to leaf is the code associated with the character at that leaf

\{a = 000, b = 001, c = 010, d = 011, e = 1\}
**Basic Idea Behind Huffman Coding**

**Step 1.** Pick two letters $x, y$ from alphabet $A$ with the smallest frequencies and create a subtree that has these two characters as leaves. This is the greedy idea. Label the root of this subtree as $z$.

**Step 2.** Set frequency $f(z) = f(x) + f(y)$. Remove $x$ and $y$ and add $z$, creating a new alphabet $A' = A \cup z - \{x, y\}$. Note that $|A'| = |A| - 1$.

Repeat this procedure, called *merge*, creating new alphabet $A'$ until only one symbol is left. The resulting tree is the **Huffman Code**.
Huffman Code Algorithm

HuffmanCoding(C)
1: $n \leftarrow |A|$
2: $Q \leftarrow A$
3: for all $i = 1$ to $n - 1$ do
4: allocate a new node $z$
5: left[$z$] $\leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
6: right[$z$] $\leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$
7: $f[z] \leftarrow f[x] + f[y]$
8: INSERT($Q, z$)
9: return EXTRACT-MIN($Q$)

Can you see why the time complexity of this algorithm is $O(n \cdot \log n)$?