Lecture: Analysis of Algorithms (CS583 - 004)

Amarda Shehu

Spring 2019

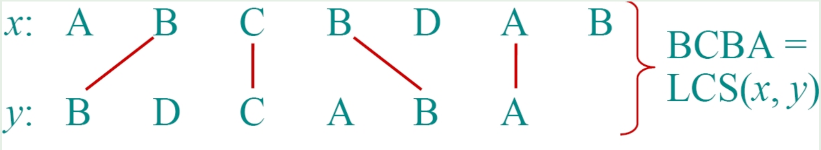
1. Dynamic Programming
   - Longest Common Subsequence
   - Dynamic Programming Hallmark # 1: Optimal Substructure
   - Dynamic Programming Solution to LCS
   - Dynamic Programming Hallmark # 2: Overlapping subproblems

Dynamic Programming

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:



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

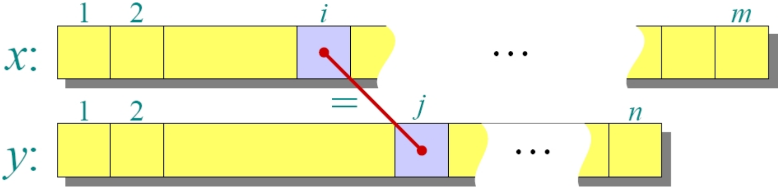
- Define $c[i, j] = |\mathrm{LCS}(\mathrm{x}[1 \ldots \mathrm{i}], \mathrm{y}[1 \ldots \mathrm{j}])|$
- Then, $\mathrm{LCS}(\mathrm{x}, \mathrm{y}) = \mathrm{c}[\mathrm{m}, \mathrm{n}]$

Recursive Formulation

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 $z[1\ldots k] = \mathrm{LCS}(\mathrm{x}[1\ldots \mathrm{i}], \mathrm{y}[1\ldots \mathrm{j}])$, 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

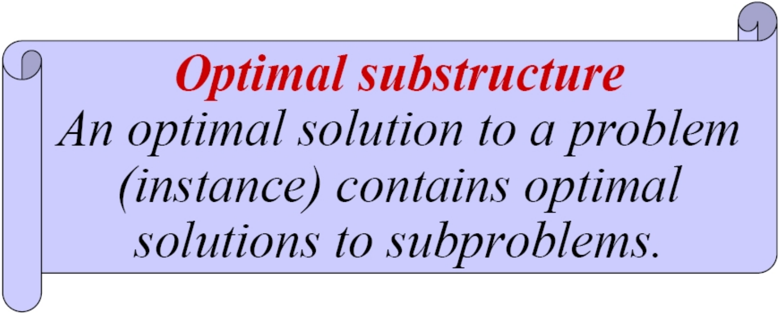
Claim: $z[1 \ldots k-1] = \mathrm{LCS(x[1 \ldots i-1], y[1 \ldots j-1])}$

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.

Dynamic Programming: Hallmark # 1



Optimal substructure

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

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

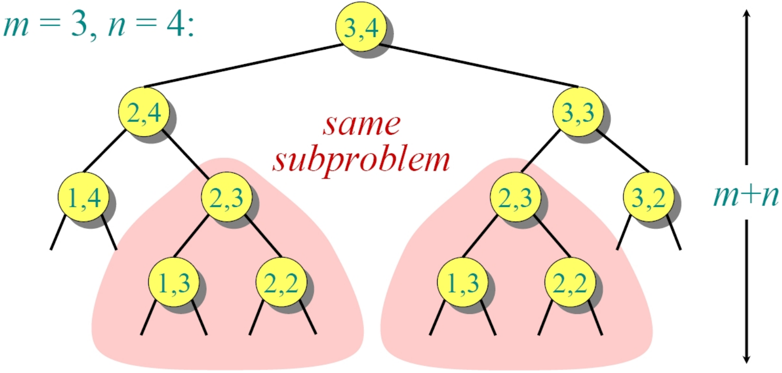
Recursive Algorithm for LCS

LCS(x, y, i, j)

```
if $x[i] = y[j]$ then
    $c[i,j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$
else $c[i,j] = \max\{\mathrm{LCS(x,y,i-1,j)}, \mathrm{LCS(x,y,i,j-1)}\}$
```

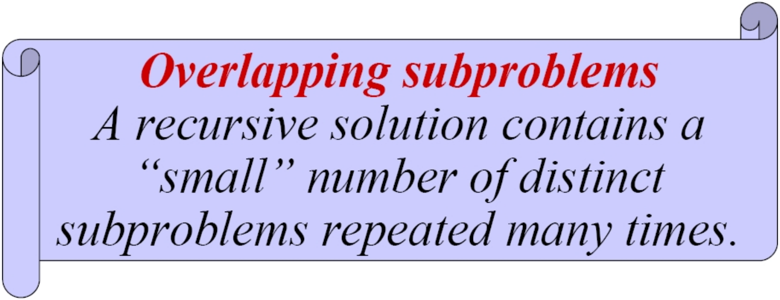
Worst-case: When $x[i] \neq y[j]$, the algorithm evaluates two subproblems, each one with only one parameter decremented.

Analysis of Recursion Tree



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)

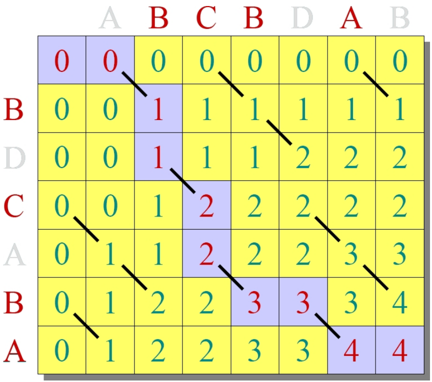
```
if $c[i, j] = NIL$ then
    if $x[i] = y[j]$ then
        $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$
    else $c[i, j] = \max\{\mathrm{LCS}(x, y, i-1, j), \mathrm{LCS}(x, y, i, j-1)\}$
```

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 $O(\min\{m, n\})$



		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Lecture: Analysis of Algorithms (CS583 - 004)

Amarda Shehu

Spring 2019

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 $M = 20$:

Object	i	1	2	3	4	5	6
Profit	p_i	7	6	12	3	12	6
Weight	w_i	2	8	10	4	14	5

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?

Mathematical Formulation of the Optimization Problem

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 $i \in \{1, 2, \ldots, n\}$

A Dynamic Programming Solution

Define $f_i(y)$ to be the optimal solution to the subproblem:

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

$$\text{such that } w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots w_i \cdot x_i \leq y$$

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

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

- $f_1(y)$ = the maximum profit for capacity y considering only object 1, where $x_1 \in \{0, 1\}$
- $f_2(y)$ = the maximum profit for capacity y considering only objects 1 and 2, 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?

Dynamic Programming Solution in Action

Let $p = (7, 6, 12, 3, 12, 16)$, $w = (2, 8, 10, 4, 14, 5)$, and $M = 20$

	0	1	2	3	4	...	10	...	20
f_0	0	0	0	0	0	...	0	...	0
f_1	0	0	7	7	7	...	7	...	7
f_2	0	0	7	7	7	...	13	...	13
f_3	0	0	7	7	7	...	13		
f_4									
f_5									
f_6									

A Greedy Approach for the Knapsack Problem

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

Object	i	1	4	6	2	3	5
Profit	p_i	7	3	16	6	12	12
Weight	w_i	2	4	5	8	10	14

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?

Greedy Approach: Not Optimal for 0/1 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:

Object	i	1	2	3
Profit	p_i	6	10	12
Weight	w_i	1	2	3
Profit/Weight	p_i/w_i	6	5	4

- 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 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.

The Coding Problem

- 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:

Letter	a	b	c	d	e	f
Frequency ($\times 10^3$)	45	13	12	16	9	5
Fixed-length encoding	000	001	010	011	100	101

- 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:

Letter	a	b	c	d	e	f
Frequency ($\times 10^3$)	45	13	12	16	9	5
Fixed-length encoding	000	001	010	011	100	101
Variable-length encoding	0	101	100	111	1101	1100

- 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

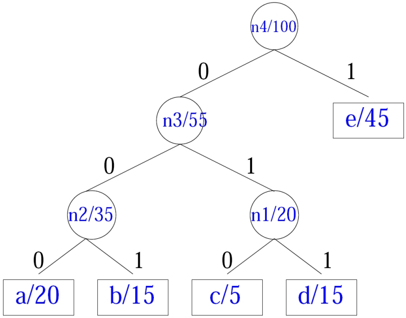
$$B(C) = \sum_{i=1}^{n} f(a_i) \cdot L(c(a_i))$$

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)

```
$n \leftarrow |A|$
$Q \leftarrow A$
for all $i = 1$ to $n - 1$ do
    allocate a new node $z$
    $\mathrm{left[z]} \leftarrow \mathrm{x} \leftarrow$ EXTRACT-MIN($Q$)
    $\mathrm{right[z]} \leftarrow \mathrm{y} \leftarrow$ EXTRACT-MIN($Q$)
    $f[z] \leftarrow f[x] + f[y]$
    INSERT($Q, z$)
return EXTRACT-MIN($Q$)
```

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