Dynamic Programming 2

Problem Solving Club
November 23, 2016
What is dynamic programming?

- Dynamic programming requires recursive thinking
- Wikipedia: “a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions – ideally, using a memory-based data structure”
- Overall, a bit hard to define
Longest Common Subsequence (LCS)

Review from last week’s meeting

Given two strings:

X = bacda

Y = dbdc

$LCS(X, Y) = ?$

$LCS\text{-}length(X, Y) = ?$
General steps to solving a DP problem

1. **Formulate the problem in terms of a mathematical function**
   - Each input corresponds to exactly one output
   - The output of the function depends only on its inputs (no side effects)
   - What would be a function for *LCS-length*?
   - *LCS-length* : (X : string, Y : string) -> integer

2. **Find a recurrence formula for the problem in terms of smaller subproblem(s)**
   - What is the recurrence for *LCS-length*?
   - *LCS-length*(Xa, Ya) = *LCS-length*(X, Y) + 1
   - *LCS-length*(Xa, Yb) = max[ *LCS-length*(Xa, Y), *LCS-length*(X, Yb) ]

3. **Recognize and solve the base cases**
   - What are the base cases for *LCS-length*?
   - *LCS-length*(X, ε) = *LCS-length*(ε, Y) = 0

4. **Code it**
Coding

- Figure how many total states your function has
- This determines how much memory your program will need
- How many states does LCS-length have?
  - \( LCS-length : (X : \text{string}, Y : \text{string}) \rightarrow \text{integer} \)
  - \( LCS-length(Xa, Ya) = LCS-length(X, Y) + 1 \)
  - \( LCS-length(Xa, Yb) = \max[LCS-length(Xa, Y), LCS-length(X, Yb)] \)
  - \( LCS-length(X, \varepsilon) = LCS-length(\varepsilon, Y) = 0 \)
- In this particular recurrence, \( X \) and \( Y \) are always prefixes of the original string
- For better runtime performance, define an alternative recurrence:
  - \( LCS-length2 : (x : \text{integer}, y : \text{integer}) \rightarrow \text{integer} \)
  - \( LCS-length2(x, y) = LCS-length2(x-1, y-1) + 1 \) if \( X[x] = Y[y] \)
  - \( = \max[LCS-length2(x, y-1), LCS-length2(x-1, y)] \) otherwise
  - \( LCS-length2(x, 0) = LCS-length2(0, y) = 0 \)
- Is this a mathematical function?
Coding Bottom-up

- For **bottom-up** implementation, you must determine a correct iteration order that processes smaller subproblems before larger ones
  - $\text{LCS-length2} : (x : \text{integer}, y : \text{integer}) \rightarrow \text{integer}$
  - $\text{LCS-length2}(x, y) = \text{LCS-length2}(x-1, y-1) + 1$ if $X[x] = Y[y]$
  - $= \max[\text{LCS-length2}(x, y-1), \text{LCS-length2}(x-1, y)]$ otherwise
  - $\text{LCS-length2}(x, 0) = \text{LCS-length2}(0, y) = 0$

- What would be a correct iteration order for $\text{LCS-length2}$?

- **string** $X, Y$
- **int** $dp[|X| + 1][|Y| + 1]$
- **for** $0 \leq x \leq |X|$
  - **for** $0 \leq y \leq |Y|$
    - **if** $x == 0$ or $y == 0$: $dp[x][y] = 0$
    - **else if** $X[x] == Y[y]$: $dp[x][y] = dp[x-1][y-1] + 1$
    - **else**: $dp[x][y] = \max(dp[x][y-1] + dp[x-1][y])$

- What do we print as the answer?
Coding Top-down

- For **top-down** implementation, it is **not necessary** to find an iteration order.
- Allow the computer to do it for you (like Excel, functional programming).
- Implement a function in your program that matches the mathematical function:
  - \( \text{LCS-length2} : (x : \text{integer}, y : \text{integer}) \rightarrow \text{integer} \)
  - \( \text{LCS-length2}(x, y) = \text{LCS-length2}(x-1, y-1) + 1 \) if \( X[x] = Y[y] \)
    \( = \max[\text{LCS-length2}(x, y-1), \text{LCS-length2}(x-1, y)] \) otherwise
  - \( \text{LCS-length2}(x, 0) = \text{LCS-length2}(0, y) = 0 \)
- string \( X, Y \)
- int \( dp[|X| + 1][|Y| + 1] = \text{initialized to} -1 \)
- int \( \text{lcs}(\text{int } x, \text{ int } y) \)
  - int \( ans \)
  - if \( \text{dp}[x][y] \neq -1 \): \( \text{ans} = \text{dp}[x][y] \)
  - else if \( x == 0 \) or \( y == 0 \): \( \text{ans} = 0 \)
  - else if \( X[x] == Y[y] \): \( \text{ans} = \text{lcs}(x-1, y-1) + 1 \)
  - else:
    - \( \text{ans} = \max(\text{lcs}(x, y-1) + \text{lcs}(x-1, y)) \)
  - \( \text{dp}[x][y] = \text{ans} \)
  - return \( \text{ans} \)
Coding: Bottom-up vs. Top down

- **string** $X$, $Y$
- **int** $dp[|X| + 1][|Y| + 1]$
- **for** $0 \leq x \leq |X|$  
  - **for** $0 \leq y \leq |Y|$
    - **if** $x == 0$ or $y == 0$:  
      - $dp[x][y] = 0$
    - **else if** $X[x] == Y[y]$:  
      - $dp[x][y] = dp[x-1][y-1] + 1$
    - **else**:
      - $dp[x][y] = \max(dp[x][y-1] + dp[x-1][y])$

- **string** $X$, $Y$
- **int** $dp[|X| + 1][|Y| + 1]$ = initialized to -1
- **int** $lcs(int x, int y)$
  - **int** ans
  - **if** $dp[x][y] != -1$:
    - $ans = dp[x][y]$
  - **else if** $x == 0$ or $y == 0$:
    - $ans = 0$
  - **else if** $X[x] == Y[y]$:  
    - $ans = lcs(x-1, y-1) + 1$
  - **else**:
    - $ans = \max(lcs(x, y-1) + lcs(x-1, y))$
  - $dp[x][y] = ans$
  - **return** $ans$
Coding: Bottom-up vs. Top down

What might be some reasons to prefer one method over the other?

- **Runtime performance**
  - A complicated issue
  - Bottom-up computes all states, while top-down only computes relevant states
  - If most states are visited, top-down is usually slower than bottom-up due to call stack
  - Top-down approach can cause a stack overflow

- **Ease of coding**
  - Bottom-up requires determination of the iteration order
  - For some types of problems (e.g. travelling salesman), the iteration order is non-obvious

- **Personal preference**