Chapter 3: Dynamic programming

Calculating Fibonacci numbers

- The usual recurrence:

$$Fibo(n) = \begin{cases} 1 & if\ n = 0 \lor n = 1 \\ Fibo(n-1) + Fibo(n-2) & if\ n \geq 2 \end{cases}$$

- Algorithm that calculates using this recurrence
- Runtime recurrence: $T(n) = T(n-1) + T(n-2) + 1$
- Memoization (table of previously calculated values)
- Only keep track of last two values

Catalan numbers

- How many different strings of n pairs of balanced parentheses?
- Recurrence formula:

$$C(n) = \begin{cases} 1 & if\ n = 0 \\ \sum_{i=1}^{n} C_{i-1}C_{n-i} & otherwise \end{cases}$$

- $C(0) = 1$
- $C(1) = C(0)C(0) = 1$
- $C(2) = C(0)C(1) + C(1)C(0) = 2$
- $C(3) = C(0)C(2) + C(1)C(1) + C(2)C(0) = 5$
- $C(4) = C(0)C(3) + C(1)C(2) + C(2)C(1) + C(3)C(0) = 14$
- $C(5) = C(0)C(4) + C(1)C(3) + C(2)C(2) + C(3)C(1) + C(4)C(0) = 42$
- First few values of the sequence: 1, 1, 2, 5, 14, 42, 132, ….

Catalan numbers

- Naïve pseudocode:

```
CATALAN(n)
   if n <= 1 then return 1
   sum = 0
   for i = 0 to n-1
      sum += CATALAN(i-1) * CATALAN(n-i-1)
   return sum
```

- Top-down approach
- Bottom-up approach using memoization

Catalan numbers

- Dynamic programming pseudocode:

```
Catalan(n)
    table[0] = 1
    table[1] = 1
    for i = 2 to n
      sum = 0
      for j = 0 to i-1
         sum += table[j]*table[i-j-1]
      table[i] = sum
   return table[n]
```

Chessboard traversal

Given an $n \times n$ table p of profits. Top down definition of maximal profit:

$$q(i,j) = \begin{cases} 0 & j < 1 \text{ or } j > n \\ p[i,j] & \text{if } i = 1 \\ p[i,j] + \max\{q(i-1,j-1), q(i-1,j), q(i-1,j+1)\} & \text{otherwise} \end{cases}$$

Q(i,j)
  if j < 1 or j > n then return 0
  else if i = 1 then return p[i,j]
  else return p[i,j] + max(q(i-1,j-1), q(i-1,j), q(i-1,j+1))

This algorithm would take $\Theta(2^n)$ time.

Developing a dynamic programming solution

a) Formulate the problem recursively.
   a) Specification.
   b) Solution.
b) Build solutions to your recurrence from the bottom up.
   a) Identify the sub-problems.
   b) Choose a memoization data structure.
   c) Identify dependencies.
   d) Find a good evaluation order.
   e) Analyze space and running time.
   f) Write down the algorithm.

Edit distance

- This is a ***minimization optimization*** problem.
- Here’s a representation showing both strings and the edit operations (d=delete,s=substitute,i=insert), for example:

$$\begin{array}{ccccccccccc}
A & L & G & O & R & & I & & T & H & M \\
A & L & & T & R & U & I & S & T & I & C \\
& & d & s & & i & & i & & s & s
\end{array}$$

- Look at the steps in creating a dynamic programming solution to a problem.
- Specify those for the edit distance problem.

Edit distance

Recurrence computing the edit distance:

$$Edit(i,j) = \begin{cases} i & if\ j = 0 \\ j & if\ i = 0 \\ \min \begin{Bmatrix} Edit(i, j-1) + 1\ (insert) \\ Edit(i-1, j) + 1\ (delete) \\ Edit(i-1, j-1) + [A[i] \neq B[i]]\ (substitute) \end{Bmatrix} & otherwise \end{cases}$$

The parameter i is an index into the first string and j an index into the second string. So one reads $Edit(i, j)$ as the edit distance going from the first i characters of the first string to the first j characters of the second string.

Longest common subsequence

In the ***longest-common-subsequence problem***, we are given two sequences:

$$X = \langle x_1, x_2, \dots, x_m \rangle, \qquad Y = \langle y_1, y_2, \dots, y_n \rangle$$

We wish to find a **maximum-length** common subsequence Z of X and Y.

LCS: definition of subsequence

Given a sequence

$$X = \langle x_1, x_2, \dots, x_m \rangle$$

Another sequence

$$Z = \langle z_1, z_2, \dots, z_k \rangle$$

is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \dots, k$, we have $x_{i_j} = z_j$.

Notation: X_j is the first j characters of sequence X.

Also, assume that X has m characters, Y has n characters, and Z has k characters.

LCS: optimal substructure

Given sequences X and Y and longest common subsequence Z:

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The LCS problem has optimal substructure.

LCS: recursive solution

We can use the optimal substructure formulas to construct a recurrence relation for $c[i,j]$, which is the length of the LCS for the pair X_i and Y_j:

$$c[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ c[i-1, j-1] + 1 & if\ i, j > 0\ and\ x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & if\ i, j > 0\ and\ x_i \neq y_j \end{cases}$$

There are many possible sub-problems but the formula only considers a few.

LCS: computing using DP

Note that the c values may be placed into a table of size $m \times n$.

Index the rows from the top to bottom 0 to m, index the columns from left to right 0 to n. Note that computing $c[i, j]$ requires table values to its left, above it, or left and above.

So fill the table row-by-row starting at row 0.

On the board: An example with the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$.

LCS algorithm

$\text{LCS-L}\scriptstyle{\text{ENGTH}}\displaystyle(X, Y, m, n)$

let $b[1\,..\,m, 1\,..\,n]$ and $c[0\,..\,m, o\,..\,n]$ be new tables
for $i = 1$ **to** m
  $c[i, 0] = 0$
for $j = 0$ **to** n
  $c[0, j] = 0$
for $i = 1$ **to** m
  for $j = 1$ **to** n
    if $x_i == y_j$
      $c[i, j] = c[i-1, j-1] + 1$
      $b[i, j] =$ "$\nwarrow$"
    else if $c[i-1, j] \geq c[i, j-1]$
      $c[i, j] = c[i-1, j]$
      $b[i, j] =$ "$\uparrow$"
    else $c[i, j] = c[i, j-1]$
      $b[i, j] =$ "$\leftarrow$"
return c and b

LCS algorithm: running time

Nested loops, the outside one executing m iterations, the inside one executing n iterations: $\Theta(mn)$.