Dynamic Programming

Analysis of Algorithms
Introduction

• Dynamic programming is a tabular method of solving problems, based on solutions to sub-problems that are stored in a table

• Similar to a divide-and-conquer approach, however with dynamic programming the sub-problems are not independent
  – With divide-and-conquer, all sub-problems are independent of one another
Introduction

• All sub-problems are solved
  – Even those which may not be needed
  – Sub-problem solutions are stored in a table
  – This avoids having to re-solve the same sub-problem over and over again - just look them up in the table

• Usually used for optimization problems
  – Which of many alternative solutions is the best?
Problem Characteristics

• What are the characteristics of a typical problem that would benefit from a dynamic programming approach?
  – Problems that have an *optimal sub-structure*
    • Problems that can be broken into multiple pieces
  – Problems that have *overlapping sub-problems*
    • The information from one sub-problem can be used to solve another sub-problem
    • For example, the solution for N-2 items can be used to solve for N-1 items, and N-1 can be used to solve for N
Array-Based Methods

• In general, use a look-up table to keep track of information instead of recalculating it

• For example, the classic recursive Fibonacci algorithm recalculates values many times

\[
\begin{align*}
\text{fib}(1) &= 1 \\
\text{fib}(2) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \text{ for } n \geq 3
\end{align*}
\]

• An table-based alternative would save the smaller Fibonacci numbers in a table as they are calculated, and then look those values up in the table instead of recalculating them each time
What if we want the solve \( \text{fib} \ (7) \)? We must create a tree that is \textit{twice} the size of \( \text{fib} \ (6) \). Worse yet, we will be re-computing the same results over and over again. How many times did we have to compute \( \text{fib} \ (3) \) to solve \( \text{fib} \ (6) \)?

This is highly inefficient! There are an \textit{exponential} number of recursive calls!
Recursion Warnings

• Do not use recursive algorithms for these kinds of problems!
  – Very inefficient for both time and space!
  – Examples such as Fibonacci, factorial, etc. are usually only shown to illustrate the algorithm or to illustrate the concept of recursion
  – Better to use an *iterative* solution for problems that have overlapping sub-problems instead
When to Use Recursion

• Use recursion if:
  – The recursive solution is natural and easier to understand
    • Recall the code for the “find” part of union-find and the code for insertion into a red-black tree
  – The recursive solution does not result in excessive duplication of computations
  – The iterative solution is too complex
Memoization

• Memoization is a technique falling somewhere between recursion and dynamic programming

• Memoization is a recursive technique that attempts to relieve the potential inefficiency of recursion by using a look-up table
  – “memo” -ize it

• Memoization allows a recursive solution to be used, once a good way has been found to break up a large problem into sub-problems
Memoization

1. Create a table (an array) that is indexed by the possible inputs to the recursive function
2. Change the code of the recursive function so that it first checks the table to see if the value is already stored there
3. If it is, simply return the value without re-computing it, otherwise call the function recursively and add the value to the table for future reference

```c
int fib (array A, int n) { // Assume elements in the array are initialized to -1
    int value = 0;
    if (A[n] != -1) return A[n];
    if (n == 1) or (n == 2)
        value = 1;
    else
        value = fib(A, n-2) + fib(A, n-1);
    A[n] = value;
    return value;
}
```
Dynamic Fibonacci Numbers

Dynamic programming solution:
Store *all* of the intermediate results in the array.

```c
int fib (int n) {
    A[1] = 1;
    for (int i = 3; i <= n; i++ ) {
    }
    return A[n];
}
```
Max-Weight Independent Set

• Problem: Given a linear graph with non-negative weights on the vertices (not the edges), find the subset of non-adjacent vertices (an independent set) of maximum total weight

1 4 5 4

• Brute-force search?
• Greedy?
• Divide-and-conquer?
Max-Weight Independent Set

- Consider the very last vertex in the graph
- Either it is not, or it is in the optimal solution
  - Case 1 (it is not in the optimal solution): discard it then recursively look at the remaining vertices
  - Case 2 (it is in the optimal solution): save it then recursively look at the remaining vertices; add it into the solution when you are finished with the recursion
Max-Weight Independent Set

• So, the optimal solution must be either:
  – A max-weight independent set of the graph with the last vertex deleted (case 1, it is not in the solution), or
  – The last vertex plus a max-weight independent set with the second-to-last vertex either deleted or not deleted (case 2, the last vertex is in the solution)

• If we knew whether or not the last vertex was in the optimal solution, then we could just recursively compute the max-weight independent set as above

• Why not try both possibilities (case 1 and case 2), and return the better solution?
Max-Weight Independent Set

• It appears to be a brute-force algorithm
• However, the actual number of distinct sub-problems that get solved by this algorithm is linear in the number of vertices
• Recall what happens before the recursion – a single vertex (the last one) is removed from the end of the graph
• This eliminates the redundancy of the brute-force approach
Max-Weight Independent Set

- Idea: perform a brute-force search in a forward fashion, saving “mini” solutions as you proceed
- Populate an array $A$ from the left to the right, where each $A[i] = \text{value of the max-weight independent set so far}$
- $A[0] = 0$, $A[1] = w_1$ (the weight of the first vertex)
- For $i = 2, 3, 4, \ldots N$:
  \[ A[i] = \max(A[i-1], A[i-2] + w_i) \]
- Running time is linear (proportional to $N$)
Max-Weight Independent Set

- For our example,

The “winner” (max value) of each pass of the for loop is circled. This is the value that is saved at that array location for determining future values in the array. The other value is saved also, but will be used only after we have filled in the entire array.
Max-Weight Independent Set

• In addition to the value of the optimal solution (which is stored in the last location of the array), we’d like to know which vertices belong to the optimal solution

• For this problem, as we fill in the array, we can simply note which vertex was the winner of each “contest”

• When we are done, scan backwards through the array from right to left, starting with $i = N$:
  – if $A[i-1] \geq A[i-2] + w_i$ (case 1 was winner) decrease $i$ by 1, else add $v_i$ to the solution and decrease $i$ by 2 (case 2 was winner)
  – if ($i == 1$) { add $v_i$ to the solution and terminate }
  else if ($i == 0$) { terminate }
Matrix Chain Multiplication

• Consider the problem of multiplying together a sequence of matrices $A_1, A_2, \ldots, A_n$ to form the product $A = A_1 A_2 \ldots A_n$
  – Since the multiplication of matrices is associative, the exact order in which the multiplications is done does not affect the answer
  – In other words, $(A_1 A_2) A_3 = A_1 (A_2 A_3)$
  – Although the answer is the same, the amount of work required to do the multiplications varies
Matrix Chain Multiplication

• Suppose A is a $p \times q$ matrix and B is a $q \times r$ matrix

• Then to compute the product $AB$ requires $p \times q \times r$ multiplications
  – There are $p \times r$ entries in the product $AB$, and each entry needs $q$ multiplications
Matrix Chain Multiplication

• For example, suppose we have 3 matrices with dimensions: \(A_1 = 5 \times 10\), \(A_2 = 10 \times 100\), \(A_3 = 100 \times 20\)

• What is the best way to parenthesize the multiplications, \((A_1 A_2) A_3\) or \(A_1 (A_2 A_3)\), to minimize the number of operations that need to be performed?

• Computing \((A_1 A_2) A_3\) takes \(5 \cdot 10 \cdot 100 + 5 \cdot 100 \cdot 20 = 15000\) multiplications, whereas computing \(A_1 (A_2 A_3)\) takes \(10 \cdot 100 \cdot 20 + 5 \cdot 10 \cdot 20 = 21000\) multiplications

• Therefore, \((A_1 A_2) A_3\) is the more efficient parenthization
Matrix Chain Multiplication

- Let the matrices be denoted by $A_1, A_2, \ldots, A_N$
  - Assume that matrix $A_i$ has dimension $p_{i-1} \times p_i$
  - We can thus specify the entire problem by giving the sequence $p_0, p_1, \ldots, p_N$
- Let $A_{i\ldots j}$ denote the product $A_{i\ldots j} = A_i A_{i+1} \ldots A_j$
- Let $m(i,j)$ be the min number of multiplications necessary to compute the product $A_{i\ldots j}$
- Therefore the entire problem can be phrased as that of finding the value $m(1,N)$
Matrix Chain Multiplication

• Consider the very last multiplication that needs to be done

• Then there are N-1 possibilities for this very last multiplication:

\[ A = (A_1) (A_2 A_3 A_4 \ldots A_{N-2} A_{N-1} A_N) \]
\[ A = (A_1 A_2) (A_3 A_4 \ldots A_{N-2} A_{N-1} A_N) \]
\[ A = (A_1 A_2 A_3) (A_4 \ldots A_{N-2} A_{N-1} A_N) \]
\[ \ldots \]
\[ A = (A_1 A_2 A_3 A_4 \ldots A_{N-2}) (A_{N-1} A_N) \]
\[ A = (A_1 A_2 A_3 A_4 \ldots A_{N-2} A_{N-1}) (A_N) \]
Matrix Chain Multiplication

• The costs of these N-1 possibilities are:

\[
A = (A_1) (A_2 A_3 A_4 \ldots A_{N-2} A_{N-1} A_N) \rightarrow m(1, 1) + m(2, N) + p_0 p_1 p_N
\]
\[
A = (A_1 A_2) (A_3 A_4 \ldots A_{N-2} A_{N-1} A_N) \rightarrow m(1, 2) + m(3, N) + p_0 p_2 p_N
\]
\[
\ldots
\]
\[
A = (A_1 A_2 A_3 A_4 \ldots A_{N-2}) (A_{N-1} A_N) \rightarrow m(1, N-2) + m(N-1, N) + p_0 p_{N-2} p_N
\]
\[
A = (A_1 A_2 A_3 A_4 \ldots A_{N-2} A_{N-1}) (A_N) \rightarrow m(1, N-1) + m(N, N) + p_0 p_{N-1} p_N
\]

• After finding all of these values, select the minimum
  – The minimum value will be stored as \( m(1, N) \), which is the fewest number of computations required to parenthesize all N matrices
Matrix Chain Multiplication

- We must at some stage compute all of the values $m(i,j)$ where $i \leq j$, so let’s try to schedule these computations as efficiently as possible.
- We know all the values $m(i,i)$ (they are all equal to 0) so we shall use these values to compute all the values $m(i, i+1)$ and then use these to find the values $m(i, i+2)$, etc.
Matrix Chain Multiplication

- This corresponds to solving the problem for chains of increasing length, where at each stage we use the results for chains of shorter lengths as the sub problems.
- Remember, we are not actually multiplying any matrices, we are only trying to figure out the best way to parenthesize the product.
**Matrix Chain Multiplication**

**Example**: Consider the following $N = 6$ matrix problem:

<table>
<thead>
<tr>
<th></th>
<th>$A_1$ 10 x 20</th>
<th>$A_2$ 20 x 5</th>
<th>$A_3$ 5 x 15</th>
<th>$A_4$ 15 x 50</th>
<th>$A_5$ 50 x 10</th>
<th>$A_6$ 10 x 15</th>
</tr>
</thead>
</table>

The problem can be phrased as one of filling in the following table representing the values $m$:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Matrix Chain Multiplication

Chains of length 2 are easy, as there is no minimization required, so \( m(i, j) = p_{i-1} p_i p_j \) or \( m(1,2) = 10 \times 20 \times 5 = 1000 \)

\[ p_0 = 10, \ p_1 = 20, \ p_2 = 5, \ p_3 = 15, \ p_4 = 50, \ p_5 = 10, \ p_6 = 15, \]

<table>
<thead>
<tr>
<th>i/j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>10 x 20</td>
<td>1</td>
<td>0</td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_2</td>
<td>20 x 5</td>
<td>2</td>
<td>0</td>
<td>1500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_3</td>
<td>5 x 15</td>
<td>3</td>
<td>0</td>
<td>3750</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_4</td>
<td>15 x 50</td>
<td>4</td>
<td>0</td>
<td>7500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_5</td>
<td>50 x 10</td>
<td>5</td>
<td>0</td>
<td>7500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_6</td>
<td>10 x 15</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Chain Multiplication

The chains of length 3 require some minimization - but only one each.

\[
\begin{align*}
\text{m}(1,3) &= \min \left\{ \text{m}(1,1) + \text{m}(2,3) + p_0 p_1 p_3 = 0 + 1500 + (10)(20)(15) = 4500, \\
\text{m}(1,2) + \text{m}(3,3) + p_0 p_2 p_3 = 1000 + 0 + (10)(5)(15) = \boxed{1750} \right\} \\
\text{m}(2,4) &= \min \left\{ \text{m}(2,2) + \text{m}(3,4) + p_1 p_2 p_4 = 0 + 3750 + (20)(5)(50) = \boxed{8750}, \\
\text{m}(2,3) + \text{m}(4,4) + p_1 p_3 p_4 = 1500 + 0 + (20)(15)(50) = 16500 \right\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>i/j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 x 20</td>
<td>20 x 5</td>
<td>5 x 15</td>
<td>15 x 50</td>
<td>50 x 10</td>
<td>10 x 15</td>
<td></td>
<td></td>
<td></td>
<td>1750</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1000</td>
<td>1750</td>
<td>8750</td>
<td>3750</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7500</td>
<td>7500</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td>3750</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td>7500</td>
<td>0</td>
<td>7500</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td>7500</td>
<td>0</td>
<td>7500</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Chain Multiplication

When we reach the last entry in the table, m(1,6), we have:

<table>
<thead>
<tr>
<th>i/j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1000</td>
<td>1750</td>
<td>7250</td>
<td>7750</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1500</td>
<td>8750</td>
<td>7250</td>
<td>8500</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3750</td>
<td>6250</td>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>7500</td>
<td>9750</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>7500</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Chain Multiplication

Compute $m(1,6)$ as the following minimization:

$$m(1,6) = \min \begin{cases} 
    m(1,1) + m(2,6) + p_0 p_1 p_6 = 11500 \\
    m(1,2) + m(3,6) + p_0 p_2 p_6 = 8750 \\
    m(1,3) + m(4,6) + p_0 p_3 p_6 = 13750 \\
    m(1,4) + m(5,6) + p_0 p_4 p_6 = 22250 \\
    m(1,5) + m(6,6) + p_0 p_5 p_6 = 10000
\end{cases}$$
Matrix Chain Multiplication

• So far we have decided that the best way to parenthesize the entire expression results in 8750 multiplications
  – We have not yet addressed how we should actually do the multiplication to achieve this value
  – Looking at the last computation we did, the minimum value came from m(1,2) and m(3,6)
  – This tells us that the penultimate step split the matrices as:
    \[ A = (A_1 \ A_2) \ (A_3 \ A_4 \ A_5 \ A_6) \]
Matrix Chain Multiplication

• Therefore, whenever we find the best way to split the matrices, store that value in an auxiliary array, called $s$
  – For example, store the value $s(1,6) = 2$ because the split $A = (A_1 \ A_2 \ )\ (A_3 \ A_4 \ A_5 \ A_6)$ came after $A_2$

• In general, as we proceed with filling in the $m$ table, if we find that the best way to compute $A_{i...j}$ is $A_{i...k} \ A_{k+1...j}$ then we set $s(i,j) = k$

• From the values of $k$, we can backtrack through the $s$ table to construct the optimal way to parenthesize the entire expression
Matrix Chain Multiplication

From our example problem, the $s$ matrix looks like:

<table>
<thead>
<tr>
<th>i/j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Matrix Chain Multiplication

Recall we found that the split $A = (A_1 A_2) (A_3 A_4 A_5 A_6)$ came after $A_2$.

By looking at $s(3,6) = 5$ we discover that we should compute $A_{3...6}$ as $A_{3...5} A_{6...6}$ and then by seeing that $s(3,5) = 4$ we get the final parenthesization: $A = (A_1 A_2) ((A_3 A_4) A_5) A_6$.

A quick check reveals that this indeed takes the required 8750 multiplications.

<table>
<thead>
<tr>
<th>i/j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Approximate String Matching

• Goal: Suppose you wanted to know how similar the DNA of two different organisms are
  – A strand of DNA consists of an ordered sequence of molecules called bases - adenine (A), guanine (G), cytosine (C), and thymine (T)
  – This ordered sequence could be represented as a string over the finite alphabet \{A,G,C,T\}

Organism #1 (S1): ACCGGTGTGCAGTCGCAAGCCGGAAGCCGGGCCC
Organism #2 (S2): GTCGTTTCGGAATGCGCGGTTCGGTA

• One way to determine similarity is to define a measure of similarity between the two strings.
Approximate String Matching

- We could find a third string, $S_3$ which is a substring of both $S_1$ and $S_2$, and count the number of characters in $S_3$
  - $S_3$ must have characters in the same order as $S_1$ and $S_2$, but the characters do not necessarily have to be consecutive
  - The longer the strand $S_3$ is, the more similar $S_1$ and $S_2$ are
  - The longest such $S_3$ is called the *longest common subsequence* (LCS) of $S_1$ and $S_2$

Organism #1 ($S_1$): ACCGGTTCGAGTGCGCGGAAGCCGGGCGCGAA
Organism #2 ($S_2$): GTCGTTTCGGAATGCGCGGCTCTGTAAA

$$(S_3) = \text{GTCGTCGGAAGCCGGGCGCGAA} = \text{LCS}(S_1, S_2)$$
Longest Common Subsequence

• How can we efficiently find the longest common subsequence of two strings?

• Consider the following example
  – Suppose string $X = \text{ABCBDAB}$ and string $Y = \text{BDCABA}$. To find the LCS($X,Y$), line up the two strings and insert spaces between contiguous characters that do not match the other string.
  – Wherever the two strings have characters in common defines the LCS.

\[
\begin{align*}
X &= \text{A B C B D } \_ \_ \text{A B } \_ \\
Y &= \_ \text{B } \_ \_ \text{D C } \text{A B A} \\
\text{LCS}(X,Y) &= \_ \_ \text{B } \_ \_ \text{D } \text{A B } \_ \\
\end{align*}
\]
Longest Common Subsequence

• This is easy to do for very short strings, but becomes inefficient as the strings become longer because it is not obvious where the spaces should be placed

• Another idea is to use a brute-force approach:
  – Enumerate all possible subsequences of X, and then for each one, check to see if it is also a subsequence of Y
  – Keep track of the longest one found so far
  – When you are done, report to the user the longest one found overall
LCS and Dynamic Programming

• Another possibility is to use a dynamic programming approach

• To see why this will work, consider that an LCS of two strings contains within it an LCS of the prefixes of the two sequences

  – For example, if $X = ABCBDAB$, then $X_0$ is the 0th prefix of $X$ consisting of the empty string, and $X_4$ is the 4th prefix of $X$ consisting of the string ABCB
LCS and Dynamic Programming

• Using the concept of a prefix, if $X = x_1 x_2 ... x_m$ and $Y = y_1 y_2 ... y_n$, recursively examine two sub-problems consisting of the strings’ prefixes using these rules:
  - If $x_m = y_n$: find an LCS of $X_{m-1}$ and $Y_{n-1}$ (Case 1)
  - If $x_m \neq y_n$: find an LCS of $X_{m-1}$ and $Y_n$, (Case 2)
    find an LCS of $X_m$ and $Y_{n-1}$,
    save the longer of the two
LCS and Dynamic Programming

• This allows us to rule out certain sub-problems, and drastically cuts down on the number of substrings we need to consider.

• There are now only $m \times n$ sub-problems to consider rather than $2^m$.

• Use a table, $c$, to compute the answer
  
  • Table entry $c[i,j]$ contains the length of an LCS of the sequence $X_i$ and $Y_j$.
  
  • The length of the LCS is found in $c[m,n]$.  

LCS and Dynamic Programming

if $i = 0$ or $j = 0$ then $c[i,j] = 0$       // Initialize c table

if both $i,j > 0$ and $x_i = y_j$               // Case 1
    then $c[i,j] = c[i-1, j-1] + 1$

if both $i,j > 0$ and $x_i \neq y_j$               // Case 2
    then $c[i,j] = \max(c[i, j-1], c[i-1, j])$
Constructing an LCS

LCS-LENGTH(X,Y)
1. m := length(X)
2. n := length(Y)
3. for i := 1 to m do
4.     c[i][0] := 0
5. for j := 0 to n do
6.     c[0][j] := 0
7. for i := 1 to m do
8.     for j := 1 to n do
9.         if x_i == y_j then
10.            c[i][j] := c[i-1][j-1] + 1
11.            b[i][j] := ““
12.     else if c[i-1][j] >= c[i][j-1] then
13.            c[i][j] := c[i-1][j]
14.            b[i][j] := “↑”
15.     else c[i][j] := c[i][j-1]
16.            b[i][j] := “←”
17.     return c and b

Figure 15.8 from Introduction to Algorithms, 3rd edition, by Cormen, Leiserson, Rivest, and Stein.
Constructing an LCS

- Runtime is proportional to $m \times n$ (the size of the table), and each table entry takes constant time.
- The $b$ table is used to quickly construct the LCS:
  - Begin at $b[m,n]$ and trace through the table following the “arrows” (could be an enumerated type).
  - Whenever we encounter a “\" it implies that $x_i = y_j$ is an element of the LCS, so push it onto a stack.
  - When done, return the stack of the elements as the LCS.
Printing the LCS

- A recursive procedure will print out the LCS of X and Y in the proper, forward order
- The initial invocation of this procedure is: PRINT-LCS(b, X, length(X), length(Y))
- This procedure runs in time proportional to $m + n$ since at least one of $i$ or $j$ is decremented at each stage of the recursion
Printing the LCS

PRINT-LCS(b, X, i, j)
1. if i = 0 or j = 0 then
2. return
3. if b[i][j] == “↖“ then
4. PRINT-LCS(b, X, i-1, j-1)
5. print x_i
6. else if b[i][j] == “↑“ then
7. PRINT-LCS(b, X, i-1, j)
8. else PRINT-LCS(b, X, i, j-1)
Analysis of LCS

• We can reduce the space requirements for LCS-LENGTH, since it needs only 2 rows of the c table at any time:
  – We only need the row being computed and the previous row (in fact, we need only slightly more than the space for one row)
  – This improvement works if we need only the length of the LCS; if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in time proportional to \( m + n \)