Dynamic Programming (Ch. 15)

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method".
- Doesn't really refer to computer programming.

Dynamic Programming

Dynamic programming can provide a good solution for problems that take exponential time to solve by brute-force methods.

Typically applied to optimization problems, where there are many possible solutions, each solution has a particular value, and we wish to find the solution with an optimal (minimal or maximal) value.

For many of these problems, we must consider all subsets of a possibly very large set, so there are $2^n$ possible solutions – too many to consider sequentially for large $n$.

Divide-and-conquer algorithms find an optimal solution by partitioning a problem into independent subproblems, solving the subproblems recursively, and then combining the solutions to solve the original problem.

Dynamic programming is applicable when the subproblems are not independent, i.e. when they share subsubproblems.

Dynamic Programming

This process takes advantage of the fact that subproblems have optimal solutions that combine to form an overall optimal solution.

DP is often useful for problems with overlapping subproblems. The technique involves solving each subproblem once, recording the result in a table and using the information from the table to solve larger problems.

For example, computing the nth Fibonacci number is an example of a non-optimization problem to which dynamic programming can be applied.

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

$$F(0) = 0 \text{ and } F(1) = 1.$$  

Fibonacci Numbers

A straightforward, but inefficient algorithm to compute the nth Fibonacci number would use a top-down approach:

1. if $n = 0$ return 0
2. else if $n = 1$ return 1
3. else return $RecFibonacci(n-1) + RecFibonacci(n-2)$

A more efficient, bottom-up approach starts with 0 and works up to $n$, requiring only $n$ values to be computed (because the rest are stored in an array):

1. create array $f[0...n]$
2. $f[0] = 0$
3. $f[1] = 1$
4. for $i = 2 \ldots n$
5. \hspace{1em} $f[i] = f[i-1] + f[i-2]$
6. return $f[n]$

Dynamic Programming

Development of a dynamic-programming algorithm can be broken into 4 steps:

1) Characterize the structure of an optimal solution (in words)
2) Recursively define the value of an optimal solution
3) Compute the value of an optimal solution from the bottom up
4) Construct an optimal solution from computed information

Rod Cutting Problem

Problem: Find most profitable way to cut a rod of length $n$
Given: rod of length $n$ and an array of prices

Assume that each length rod has a price $p$,

Find best set of cuts to get maximum price, where

- Each cut is integer length in inches
- Can use any number of cuts, from 0 to $n - 1$
- No cost for a cut

Example cuts for $n = 4$ (prices shown above each example)
**Rod Cutting Problem**

Problem: Find most profitable way to cut a rod of length $n$

Given: rod of length $n$ and an array of prices $p_i$

Output: The maximum revenue obtainable for rods whose lengths sum to $n$, computed as the sum of the prices for the individual rods.

- If $p_i$ is large enough, an optimal solution might require no cuts, i.e., just leave the rod as $i$ inches long.
- Every optimal solution has a leftmost cut. In other words, there's some cut off the left end, and a remaining piece of length $n-i$ on the right.

**Example cuts for $n=4$ (prices shown above each example)**

- 1,1,5 (prices 1,1,2)
- 1,5,1 (prices 1,2,1)
- 5,1,1 (prices 1,1,2)
- 1,1,1,1 (prices 1,1,1,1)

**Calculating Maximum Revenue**

Compute the maximum revenue ($r_i$) for rods of length $i$

<table>
<thead>
<tr>
<th>Length (inches) $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

**Top-down approach:**

- 1: $r_1 = 1$
- 2: $r_2 = \max(p_1, r_1 + p_2)$
- 3: $r_3 = \max(p_1, p_2, r_2 + p_3)$
- 4: $r_4 = \max(p_1, p_2, p_3, r_3 + p_4)$
- ...  

<table>
<thead>
<tr>
<th>$i$</th>
<th>$r_i$</th>
<th>optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 (no cuts)</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2 (no cuts)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3 (no cuts)</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2 + 2</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>2 + 3</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>6 (no cuts)</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>1 + 6 or 2 + 2 + 3</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>2 + 6</td>
</tr>
</tbody>
</table>

**Simpler way to decompose the problem**

Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length $i$ cut off the left end, and a remaining piece of length $n-i$ on the right.

- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than 2 subproblems.
- Say that the solution with no cuts has first piece size $i$ with revenue $p_i$, and remainder size 0 with revenue $r_{n-i} = 0$.

Gives a simpler version of the equation for $r_n$:

$$r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$$

**Rod Cutting Problem**

Example rod lengths and values:

<table>
<thead>
<tr>
<th>Length (in inches) $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

Can cut rod in $2^{n-1}$ ways since each inch can have a cut or no cut and there are at most $n-1$ places to cut.

Example: rod of length 4:

- $p_1 + p_2 = 5 + 5 = 10$

**Calculating Maximum Revenue**

<table>
<thead>
<tr>
<th>Length (inches) $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

Optimal revenue $r_n$ can be determined by taking the maximum of:

- $p_i$: The revenue for the entire rod, with no cuts,
- $r_i + r_{n-i}$: the max revenue from a rod of 1 and a rod of $n-1$,
- $r_i + r_{n-i}$: the max revenue from a rod of 2 and a rod of $n-2$,
- ...

That is, $r_n = \max(r_i + r_{n-i}, r_{i+1} + r_{n-i-1}, \ldots, r_{n-1} + r_0)$

Example: For $n=7$, one of the optimal solutions makes a cut at 3 inches, giving 2 subproblems of lengths 3 and 4. We need to solve both of the subproblems optimally. The optimal solution for a problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the problem of length 7.

**Recursive, Top-down Solution to Rod Cutting**

Input: price array $p$, length $n$

Output: maximum revenue for rod of length $n$

```
1. if n == 0 return 0
2. q = -∞
3. for i = 1 to n
4. q = max(q, p(i) + Cut-Rod(p, n-i))
5. return q
```

This procedure works, but it is very inefficient. If you code it up and run it, it could take more than an hour for $n=40$. The running time almost doubles each time $n$ increases by 1.
Recursive, Top-down Solution to Rod Cutting

Input: price array \( p \), length \( n \)
Output: maximum revenue for rod of length \( n \)

\[ \text{Cut-Rod}(p, n) \]
1. \( \text{if } n = 0 \) return 0
2. \( q = -\infty \)
3. for \( i = 1 \) to \( n \)
   4. \( q = \max(q, p(i) + \text{Cut-Rod}(p, n-i)) \)
5. return \( q \)

Why so inefficient?
Cut-Rod calls itself repeatedly, even on subproblems it has already solved. Here’s a tree of recursive calls for \( n = 4 \). Inside each node is the value of \( n \) for the call represented by the node. The subproblem for size 2 is solved twice, the one for size 1 four times, and the one for size 0 eight times.

Exponential growth: Let \( T(n) \) equal the number of calls to Cut-Rod with second parameter equal to \( n \). Then

\[
T(n) = \begin{cases} 
1 & \text{if } n = 0, \\
1 + \sum_{j=1}^{n} T(j) & \text{if } n > 1 
\end{cases}
\]

The summation counts calls where second parameter is \( j = n - i \). The solution to this recurrence is \( T(n) = 2^n \).

Dynamic-programming Solution

- Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- Save the solution to a subproblem in a table, and refer back to the table whenever the subproblem is revisited.
- “Store, don’t recompute” = a time/memory trade-off.
- Turns an exponential-time solution into a polynomial-time solution.
- Two basic approaches: top-down with memoization, and bottom-up.

Rod Cutting Problem

The bottom-up approach uses the natural ordering of subproblems: a problem of size \( i \) is “smaller” than a problem of size \( j \) if \( i < j \). Solve subproblems \( j=0,1,...,n \) in that order.

Input: price array \( p \), length \( n \)
Output: maximum revenue for rod of length \( n \)

\[ \text{Bottom-Up-Cut-Rod}(p, n) \]
1. let \( r[0...n] \) and \( s[0...n] \) be new arrays
2. \( r[0] = 0 \)
3. for \( j = 1 \) to \( n \)
   4. \( q = -\infty \)
   5. for \( i = 1 \) to \( j \)
      6. \( q = \max(q, p[i] + r[j-i]) \)
   7. \( s[j] = i \)
   8. \( r[j] = q \)
8. return \( r \) and \( s \)

Running time? \( T(n) = \Theta(n^2) \)

Printing Optimal Solution

The following pseudocode uses the \( s \) array to print the optimal solution.

Input: price array \( p \), length \( n \)
Output: side-effect printing of optimal cuts

\[ \text{Print-Cut-Rod-solution}(p, n) \]
1. \((r, s) = \text{Bottom-Up-Cut-Rod}(p, n)\)
2. while \( n > 0 \)
   3. print \( s[n] \)
   4. \( n = n - s[n] \)

Running time? \( T(n) = \Theta(n) \)
Matrix-Chain Product

If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then
\[ A \cdot B = C \text{ is } m \times p \text{ matrix} \]
and the time needed to compute C is $O(mnp)$.

- there are $mp$ elements of C
- each element of C requires $n$ scalar multiplications and $n-1$ scalar additions

Matrix-Chains Multiplication Problem:

Given matrices $A_1, A_2, A_3, ..., A_n$, where the dimension of $A_i$ is $p_{i-1} \times p_i$, determine the minimum number of multiplications needed to compute the product $A_1 \cdot A_2 \cdot ... \cdot A_n$. This involves finding the optimal way to parenthesize the matrices.

Since multiplication is associative, for more than 2 matrices, there exists more than one order of multiplication to get the same end result.

Matrix-Chains Product Example

<table>
<thead>
<tr>
<th>$A_1$ (4x2)</th>
<th></th>
<th>$A_2$ (2x5)</th>
<th>$A_3$ (5x1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two ways to parenthesize this product:

$(A_1 \cdot A_2) \cdot A_3$: requires $4 \cdot 2 \cdot 5 = 40$ multiplications, $M_{1,3}$ is $4 \times 5$ matrix

$A_1 \cdot (A_2 \cdot A_3)$: requires $4 \cdot 5 \cdot 1 = 20$ multiplications, $M_{2,3}$ is $4 \times 1$ matrix

$\Rightarrow$ total multiplications = 40 + 20 = 60

$A_1 \cdot (A_2 \cdot A_3)$

$M_1 = A_1 \cdot A_2$: requires $2 \cdot 5 \cdot 1 = 10$ multiplications, $M_2$ is $2 \times 1$ matrix

$M_2 = A_2 \cdot A_3$: requires $4 \cdot 2 \cdot 1 = 8$ multiplications, $M_3$ is $4 \times 1$ matrix

$\Rightarrow$ total multiplications = 10 + 8 = 18

Matrix-Chains Product – Recursive Solution

Let $M[i,j] =$ min number of multiplications to compute $A_i \cdot A_{i+1} \cdot ... \cdot A_j$, where dimension of $A_i \cdot A_{i+1} \cdot ... \cdot A_j$ is $p_{i-1} \times p_j$, $M[i,j] = 0$ for $i$ to $n$, and $M[1, n]$ is the solution we want.

- $p$ is indexed from 0 to $n$ and $p_i = p[i]$.

$M[i,j]$ can be determined as follows:

$M[i, j] = \min(M[i,k] + M[k+1, j] + p_i \cdot p_k \cdot p_j)$, where $i \leq k < j$.

where $M[i,j]$ equals the minimum cost for computing subproducts $A_i \cdot A_{i+1} \cdot ... \cdot A_j$ plus the cost of multiplying these two matrices together. Each matrix $A_i$ is dimension $p_{i-1} \times p_i$, so computing matrix product $A_i \cdot A_{i+1} \cdot ... \cdot A_j$ takes $p_i \cdot p_k \cdot p_j$ scalar multiplications.

Matrix-Chains Product – Recursive Solution

Let $M[1, n] = \min(M[1,1] + M[2,3] + p_1 \cdot p_2 \cdot p_3, M[1,2] + M[3,3] + p_1 \cdot p_4 \cdot p_5, M[1,3] + M[4,1] + p_1 \cdot p_6 \cdot p_7)$

$= \min((2 \times 5 \times 1 + 4 \times 2 \times 1) = 18, (4 \times 2 \times 5 + 4 \times 5 \times 1) = 60)$

$M[1,3] = \min(M[1,1] + M[2,3] + p_1 \cdot p_2 \cdot p_3, M[1,2] + M[3,3] + p_1 \cdot p_4 \cdot p_5, M[1,3] + M[4,1] + p_1 \cdot p_6 \cdot p_7)$

$= \min((2 \times 5 \times 1 + 4 \times 2 \times 1) = 18, (4 \times 2 \times 5 + 4 \times 5 \times 1) = 60)$

There is redundant computation because no intermediate results are stored.
Matrix-Chain Product – Recursive Solution

RMC(p, i, j)
1. if i = j return 0
2. M[i,j] = ∞
3. for k = i to j-1
4. q = RMP(p, i, k) + RMP(p, k+1, j) + p_i-1p_kp_j
5. if q < M[i,j] then M[i,j] = q
6. return M[i,j]

Matrix-Chain Product – Bottom-up solution

Matrix-Chain-Order (p) // p is array of dimensions
1. n = p.length – 1
2. let M[1…n, 1…n] and s[1…n-1,2…n] be new tables
3. for i = 1 to n
4. M[i,i] = 0                     /** fill in main diagonal with 0s **/
5. for d = 2 to n                    /** d is chain length **/
6. for i = 1 to n – d + 1
7. j = i + d - 1
8. M[i,j] = #
9. for k = i to j-1
10. q = M[i,k]+ M[k+1,j] + p_i-1p_kp_j
11. if q < M[i,j] then M[i,j] = q
12. s[i,j] = k
14. return M and s

Input:   s array, dimensions of M matrix
Output: side-effect printing of optimal parenthesization
Print-Optimal-Parens(s, i, j)  // initially, i = 1 and j = n
1. if i == j
2.      print "A"
3. else
4.      print "(
5.      Print-Optimal-Parens(s, i, s[i, j])
6.      Print-Optimal-Parens(s, s[i, j] + 1, j)
7.      print ")

Matrix-Chain Product – Bottom-Up Solution

Complexity:
• O(n^3) time because of the nested for loops with each of d, i, and k taking on at most n-1 values.
• O(n^2) space for two n x n matrices M and s

Longest Common Subsequence Problem (§ 15.4)

Problem: Given X = < x_1, x_2, ..., x_m> and Y = <y_1, y_2, ..., y_n>, find a longest common subsequence (LCS) of X and Y.

Example:
X = ( A, B, C, D, A, B )
Y = ( B, D, C, A, B, A ) (or also LCS_XY = ( B, D, A, B ))

Brute-Force solution:
1. Enumerate all subsequences of X and check to see if they appear in the correct order in Y (chars in sequences are not necessarily consecutive).
2. Each subsequence of X corresponds to a subset of the indices (1,2,...,m) of the elements of X, so there are 2^m subsequences of X. The time to check whether each of these is a subsequence of Y is θ(n^2).
3. Clearly, this is not a good approach...time to try dynamic programming!
Notation

- \( X_i = \langle x_i, x_{i+1}, \ldots, x_n \rangle \)
- \( Y_j = \langle y_j, y_{j+1}, \ldots, y_{n} \rangle \)

For example, if \( X = (A, C, G, T, T, C, A) \), then

\[ X_2 = (A, C, G, T) \]

Example LCSs:
- springtime
- horseback
- pioneer
- snowflake
- magistrat
- heroically
- becalm
- scholarly

Recursive Solution to LCS Problem

The recursive LCS Formulation

- Theorem: Let \( X = \langle x_1, x_2, \ldots, x_n \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \) be sequences and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) be any LCS of \( X \) and \( Y \).

Case 1: If \( x_n = y_n \), then let \( Z = \langle z_1, z_2, \ldots, z_k, \ldots, z_{n-1} \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

Case 2: If \( x_n \neq y_n \), then let \( z_k \neq y_k \), then let \( Z' = \langle z_1, z_2, \ldots, z_{k-1}, y_k \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

Case 3: If \( x_n \neq y_n \), then let \( z_k = y_k \), then let \( Z' = \langle z_1, z_2, \ldots, z_{k-1}, x_k \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

Thus \( Z' \) is a longer sequence than \( Z \). Contradiction to \( Z \) being an LCS.

Proof: First, show that \( Z \) is an LCS of \( X \) and \( Y \).

Now suppose there exists a subsequence \( W \) of \( X_{m-1} \) and \( Y_{n-1} \) that's longer than \( Z \).

So if \( x_n = y_n \), then let \( Z' = \langle z_1, z_2, \ldots, z_k, y_n \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

If \( x_n \neq y_n \), then let \( Z' = \langle z_1, z_2, \ldots, z_{k-1}, x_k \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).


Top-Down Recursive Solution to LCS

- initialize \( C[0,0] = C[0,j] = 0 \) for \( i = 0 \ldots m \) and \( j = 1 \ldots n \)
- initialize \( C[i, j] = NIL \) for \( i = 1 \ldots m \) and \( j = 1 \ldots n \)

LCS(\( i, j \) ) //input is index into 2 strings of length \( i \) and \( j \)

1. if \( C[0, j] = 0 \)
2. if \( x_i = y_j \)
3. \( C[i, j] = LCS(\( i-1, j-1 \)) + 1 \)
4. else
5. \( C[i, j] = \max(LCS(\( i, j-1 \)), LCS(\( i-1, j \))) \)

C is a two-dimensional array holding the solutions to subproblems.

Recursive Solution to LCS Problem

Optimal Substructure of an LCS

Theorem: Let \( X = \langle x_1, x_2, \ldots, x_n \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \) be sequences and let \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) be any LCS of \( X \) and \( Y \).

Case 1: If \( x_n = y_n \), then let \( Z = \langle z_1, z_2, \ldots, z_k, \ldots, z_{n-1} \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

Case 2: If \( x_n \neq y_n \), then let \( z_k \neq y_k \), then let \( Z' = \langle z_1, z_2, \ldots, z_{k-1}, y_k \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

Case 3: If \( x_n \neq y_n \), then let \( z_k = y_k \), then let \( Z' = \langle z_1, z_2, \ldots, z_{k-1}, x_k \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

Thus \( Z' \) is a longer sequence than \( Z \). Contradiction to \( Z \) being an LCS.

Proof: First, show that \( Z \) is an LCS of \( X \) and \( Y \).

Now suppose there exists a subsequence \( W \) of \( X_{m-1} \) and \( Y_{n-1} \) that's longer than \( Z \).

So if \( x_n = y_n \), then let \( Z' = \langle z_1, z_2, \ldots, z_k, y_n \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).

If \( x_n \neq y_n \), then let \( Z' = \langle z_1, z_2, \ldots, z_{k-1}, x_k \rangle \) be any LCS of \( X_{n-1} \) and \( Y_{n-1} \).
Top-Down Recursive Solution to LCS

LCS(i, j) \ // input is index into 2 strings
1. if C[i, j] == NIL
2. if x_i == y_j
3. \ C[i, j] = LCS(i-1, j-1) + 1
4. else
5. \ C[i, j] = max(LCS(i, j-1), LCS(i-1, j))
6. return C[i, j]

Bottom-Up DP Solution to LCS Problem

To compute c[i, j], we need the solutions to:
- c[i-1, j-1] (when x_i = y_j)
- c[i-1, j] and c[i, j-1] (when x_i ≠ y_j)

LCS(X, Y), where X and Y are strings
1. m = length[X]
2. n = length[Y]
3. let b[1...m,1...n] and c[0...m,0...n] be new tables
4. for i = 0 to m do  c[0, i] = 0
5. for j = 0 to n do  c[i, 0] = 0
6. for i = 1 to m
7. for j = 1 to n
8. if x_i = y_j
9. \ c[i, j] = c[i-1, j-1] + 1
10. b[i, j] = Up&Left
11. else if c[i-1, j] ≥ c[i, j-1]
12. \ c[i, j] = c[i-1, j]
13. \ b[i, j] = Up
14. else
15. \ c[i, j] = c[i, j-1]
16. \ b[i, j] = Left
17. return b and c

Bottom-Up LCS DP

Running time = O(mn) (constant time for each entry in c)

This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?

We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).
- if x_i = y_j, the answer came from the upper left (diagonal)
- if x_i ≠ y_j, the answer came from above or to the left, whichever value is larger (if equal, default to above).

Array b keeps "pointers" indicating how to traverse the c table.

Constructing the LCS

Print-LCS(b,X,i,j)
1. if i == 0 or j == 0 then return
2. if b[i,j] = Up&Left
3. \ Print-LCS(b,X,i-1,j-1)
4. print x_i
5. else if b[i,j] = Up
6. \ Print-LCS(b,X,i-1,j)
7. else Print-LCS(b,X,j-1,j)

Initial call is Print-LCS(b,X,len(X),len(Y)), where b is the table of pointers

Complexity of LCS Algorithm

The running time of the LCS algorithm is O(mn), since each table entry takes O(1) time to compute.

The running time of the Print-LCS algorithm is O(m + n), since one of m or n is decremented in each step of the recursion.
End of Dynamic Programming Lectures

Binomial Coefficient

Bottom-up dynamic programming can be applied to this problem for a polynomial-time solution.

\[ C(n,k) = C(n-1, k-1) + C(n-1,k) \text{ for } n > k > 0 \]

and

\[ C(n, 0) = C(n,n) = 1 \]

Binomial(n,k)
1. for i = 0 to n
2. for j = 0 to min(i,k)
3. if j = 0 or j = i
4. \[ C[i,j] = 1 \]
5. else \[ C[i,j] = C[i-1, j-1] + C[i-1,j] \]
6. return \[ C[n,k] \]

Rod Cutting Problem

Dynamic Programming (dynamic table lookup) has both a top-down and a bottom-up solution.

The top-down approach uses memoization (recording a value so we can look it up later)

Input: price array \( p[1.\ldots n] \), length \( n \)
Output: maximum revenue for rod of length \( n \)

Memoized-Cut-Rod(p, n)
1. let \( r[0..n] \) be new array
2. for i = 0 to n
3. \( r[i] = \text{MinInt} \text{ // negative infinity} \)
4. return Memoized-Cut-Rod-Aux(p, n, r)

Memoized-Cut-Rod-Aux(p, n, r)
1. if \( r[n] \neq 0 \) return \( r[n] \)
2. else \( q = \text{MinInt} \text{ // negative infinity} \)
3. for i = 1 to n
4. \( q = \max(q, p[i] + \text{Memoized-Cut-Rod-Aux(p, n-i, r)}) \)
5. \( r[n] = q \)
6. return q