Dynamic Programming

Rana Barua

Visiting Scientist, IAI, TCG CREST Kolkata

1 Dynamic Programming

Typically, dynamic programming apply to optimization problems in which we try to solve a reasonable number of (overlapping) subproblems whose optimal solutions hopefully yield the intended optimal solutions. The subproblems are solved recursively via some recursive relations. Straight-forward algorithms for solving these recursive relations will generally yield highly inefficient algorithms. In the dynamic programming method, the solutions to the recursive calls are stored in a tabular form. Unlike the Divide-and Conquer paradigm, the subproblems we consider are generally overlapping and the structure of the optimal solution will suggest what subproblems we need to consoder.

1. **Longest Common Subsequence(LCS)**
   The first problem we shall consider is the longest-common-subsequnce problem.

   Definition 1. *Given a sequence* $X =< x_1, \ldots, x_m >$*, another sequence* $Z =< z_1, \ldots, z_k >$ *is said to be a subsequence of* X *if there exists* $1 \leq i_1 < i_2 < \ldots < i_k \leq m$ *such that* $x_{i_1} = z_1, x_{i_2} = z_2, \ldots, x_{i_k} = z_k$.
   Given two sequences $X < x_1, \ldots, x_m >$ *and* $Y =< y_1, \ldots, y_n >$*, a sequence* Z *is a* **common subsequence** *of* X *and* Y *if* Z *is a subsequence of both* X *and* Y*..*
   The longest-common-subsequence problem *is the following.*
   Given two sequences X *and* Y *as above, find a maximum-length common subsequence of* X *and* Y*.*

   Characterizing an LCS

   Given a sequence $X =< x_1, \ldots, x_m >$, we define the ith prefix of X as $X_i =< x_1 \ldots, x_i >$ for $i = 0, 1, \ldots, m$. Here X_0 is the empty sequence. Given two sequences $X =< x_1, \ldots, x_m >$ and $Y =< y_1, \ldots, y_n >$, let $Z =< z_1, \ldots, z_k >$ be any LCS of X and Y. Then it is not hard to see that the following properties hold.
   - 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 $x_m \neq y_n$ and $y_k \neq x_m$ then Z is an LCS of X_{m-1} and Y.
   - If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}.

   The above characterization tells us that the LCS problem has, what is called, an optimal substructure property. It also gives rise to the following recursive relations for the subproblems that we need to consider.
   Let $c[i, j]$ denote the *length* of an LCS of the sequences X_i and Y_j for $0 \leq i \leq m; 0 \leq j \leq n$. If either $i = 0$ or $j = 0$, then clearly the LCS has length 0. The above characterization yields the recursive formula.

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

Computing the Length of an LCS

Based on equation (1), one could easily write an exponential-time recursive algorithm to compute the length an LCS of two sequences. However, we use dynamic programming to compute the solutions bottom-up.

Method. Algorithm **LCS-Length** takes as inputs two sequences $X =< x_1, \ldots, x_m >, Y =< y_1, \ldots, y_n >$. It stores the values $c[i, j]$ in a table $c[0..m; 0..n]$ whose entries are computed in a row-major order. It also maintains table $b[0..m; 0..n]$ to facilitate the construction of an optimal solution $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen while computing $c[i, j]$. The algorithm returns the tables b and c. $c[m, n]$ contains the length of an LCS of X and Y.

LCS-Length(X, Y)

```
1.      m ← Length(X)
2.      n ← Length(Y)
3.      for i ← 1 to m
4.          do c[i, 0] ← 0
5.       for j ← 0 to n
6.          do c[0, j] ← 0
7.      for i ← 1 to m
8.          do for j ← 1 to n
9.                  do if x_i = x_j
10.                     then c[i, j] ← c[i − 1, j − 1] + 1
11.                          b[i, j] ←"↖"
12.                     else  if c[i − 1, j] ≥ c[i, j − 1]
13.                           then 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
```

Constructing an LCS

The following procedure prints out an LCS of X and Y in proper order. The initial invocation is **Print-LCS**$(b, X, Length(X), Length(Y))$

Print-LCS(b, X, i, j)

```
1.      if i = 0 or j = 0
2.      then return
3.          if b[i, j] ="↖"
4.          then Print-LCS(b, X, i − 1, j − 1)
5.                  print x_i
6.          else if b[i, j] ="↑"
7.               then Print-LCS(b, X, i − 1, j)
8.               else Print-LCS(b, X, i, j − 1)
```

Exercise 1. (a) Determine an LCS of $< 1, 0, 0, 1, 0, 1, 0, 1 >$ and $< 0, 1, 0, 1, 1, 0, 1, 1, 0 >$.

(b) Show how one can construct an LCS from the completed c table and the sequences $X =< x_1, \ldots, x_m >$ and $< y_1, \ldots, x_n >$ in $O(m+n)$ time, without using table b.
(c) Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

2. **Matrix-Chain Multiplication**

If we are given two matrices of dimension, say, $p \times q$ and $q \times r$ then the number of multiplications required to form the product AB is pqr. If, however, we are given three matrices A, B, C of order $p \times q, q \times r, r \times s$ respectively, then to form the product ABC we may first mutiply A and B to obtain the product AB and then AB is multiplied with C; or we may multiply B and C to obtain the product BC and finally mutiply A with BC. It is easy to see that the number of mutiplications required in the first case is $pqr + prs$ while the number of mutiplications required in the second case will be $qrs + pqs$. One of these can be substantially larger than the other. For instance, if $p = 10, q = 100, r = 5$ and $s = 50$, then the number of mutiplications in the first case is $7,500$ while the number of multiplications in the second case will be $75,000$! Thus the manner in which we obtain the product matters.
Matrix-Chain Multiplication
Given a sequence or chain $< A_1, \ldots, A_n >$ of n matrices, we wish to form the product

$$A_1 A_2 \ldots A_n.$$

We assume that each $A_i, 1 \leq i \leq n$ is of dimension $p_{i-1} \times p_i$. It is not hard to check that the number of ways of obtaining this product is the same as the number of ways of fully parenthesizing $A_1 \ldots A_n$. A product of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by the parentheses $(,)$. For example, $(A_1.((A_2.A_3).A_4))$ is fully parenthesized. The way the product is fully parenthesized gives us a way to form the matrix product.
The matrix-chain multiplication problem is the following.
Given a chain $< A_1, \ldots, A_n >$ of n matrices, where $A_i, 1 \leq i \leq n$ has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

The Number of Parenthesizations

Let $P(n)$ denote the number of parenthesizations of a sequence of n matrices. Now observe that we can split a sequence of n matrices between the kth and $k+1$st matrices for any $k = 1, \ldots, n$ and then parenthesize the two resulting subsequence independently to obtain the final parenthesization. Thus we have the following recurrence formula.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}.$$

The solution to this recurrence is $P(n) = C(n-1)$, where

$$C(n) = \frac{1}{n+1} \binom{2n}{n},$$

is the nth Catalan number.

Structure of Optimal Parenthesization

Let $A_{[i,j]}$ denote the product $A_i \ldots A_j, i < j$. Note that an optimal parenthesization of $A_1 \ldots A_n$

splits the product between A_k and A_{k+1} for some $k, 1 \leq k < n$. Thus for some k, we first compute $A_{[1,k]}$ and $A_{[k+1,n]}$ and then multiplying them to obtain the final product $A_{[1,n]}$. Thus the cost of this optimal parenthesization is the cost of computing the matrix $A_{[1,k]}$ + the cost of computing $A_{[k+1,n]}$ + the cost of multiplying them together. Thus to obtain an optimal parenthesization, we must have used an optimal parenthesizations of both $A_{[1,k]}$ and $A_{[k+1,n]}$.
This leads us to the following recursive solution.
Let $m[i,j], i \leq i \leq n$, be the minimum number of scalar multiplications required to compute $A_{[i,j]}$. Then $m[1,n]$ would be the optimal cost to compute $A_{[1,n]}$. We define $m[i,j]$ recursively as follows.

$$m[i,j] = \begin{cases} 0 & \text{if } \ i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } \ i < j \end{cases}.$$

The $m[i,j]$ values give the costs of optimal solutions to subproblems. To keep track of how to construct an optimal solution we also define

$$s[i,j] = k \ \leftrightarrow \ \mathrm{m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j}.$$

Computing the Optimal costs

A recursive algorithm based upon the above recurrence to compute $A_{[1,n]}$ will take exponential time. Instead we compute the optimal cost by using a bottom-up approach. The procedure assumes that each matrix A_i has dimension $p_{i-1} \times p_i, 1 \leq i \leq n$. The input is a sequence $p = (p_0, \ldots, p_n)$ of length $lth[p] = n + 1$. The procedure uses an auxiliary table $m[1..n; 1..n]$ for storing the costs $m[i,j]$ and an auxiliary table $s[1..n; 1..n]$ that records which index k achieved the optimal cost in computing $m[i,j]$.

Matrix-Chain-Product(p)

```
1.    n ← lth(p) − 1
2.    for i ← 1 to n
3.       do m[i.i] ← 0
4.    for l ← 2 to n
5.       do for i ← 1 to n − l + 1
6.          do j ← i + l − 1
7.             m[i, j] ← ∞
8.             for k ← i to j − 1
9.                do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10.                  if q < m[i, j]
11.                     then m[i, j] ← q
12.                        s[i, j] ← k
13.   return m and s
```

The algorithm first computes $m[i,i] = 0, 1 \leq i \leq n$. It then computes $m[i,i+1], 1 \leq i \leq n-1$, during the first execution of the **for** loop. During the second execution of the loop, it computes $m[i,i+2], 1 \leq i \leq n-2$ and so forth.The loops are nested three deep and each loop index takes at most n values. The running time is $O(n^3)$.

Constructing the Optimal Solution

We use the table $s[1..n; 1..n]$ to find the best way to mutiply the matrices. Each entry $s[i,j]$ records the value k such that the optimal parenthesization of $A_{[i,j]}$ splits the product between A_k and A_{k+1}. Thus if $s[1,n] = k$ then we know that the final matrix mutiplication in computing $A_{[1,n]}$ optimally is $A_{[1,k]}.A_{[k+1,n]}$. The earlier ones can be obtained recursively. Thus we have the

following recursive procedure, given the matrices $A = (A_1, \ldots, A_n)$. The initial call is Matrix-Chain-Multiply$(A, s, 1, n)$.

Matrix-Chain-Multiply(A, s, i, j)

```
1.    if j > i
2.       then X ← Matrix − Chain − Multiply(A, s, i, s[i, j])
3.            Y ← Matrix − Chain − Multiply(A, s, s[i, j] + 1, j)
4.            return X × Y
5.       else return A_i.
```

Exercise 2. (a) Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $< 5, 10, 3, 12, 5, 50, 6 >$.

(b) Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referred by Matrix-Chain-Product in computing other table entries. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n}\sum_{j=1}^{n} R(i, j) = \frac{n^3 - n}{3}.$$

(c) Show that a full parenthesization of an n-element expression has exactly $n - 1$ pair of parentheses.

3. **Subset Sums and Knapsacks**

Subset Sum Problem. Given a bound W and n items $1, 2, \ldots, n$ with non-negative weights $w_1, w_2, \ldots, w_n$ respectively, find a subset $S \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in S} w_i \leq W,$$

and $\sum_{i \in S} w_i$ is as large as possible.
A related problem is the *knapsack problem.*
Knapsack Problem. Given a bound W and n items $1, \ldots, n$ of weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, find a subset $S \subseteq \{1, \ldots, n\}$ such that

$$\sum_{i \in S} w_i \leq W$$

and $\sum_{i \in S} v_i$ is as large as possible.
We shall first consider the subset sum problem and look at the structure of an optimal solution. Assume that the weights are intergers as also the bound W. Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution. If $n \notin S$ then clearly S is an optimal solution of the corresponding knapsack problem consisting of the first $n - 1$ items. If $n \in S$, then $S' = S - \{n\}$ is an optimal solution of the knapsack problem with $n - 1$ items $1, \ldots, n - 1$ with weight-bound $W - w_n$.(Why?) Thus we need to consider subproblems consisting of the first i items $1, \ldots, i$, for $1 \leq i \leq n$ with maximum allowed weight w, where $1 \leq w \leq W$. Let $M[i, w]$ denote the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight w. Thus

$$M[i, w] = \max_{T} \sum_{j \in T} w_j,$$

where the maximum is taken over all subsets $T \subseteq \{1, \ldots, i\}$ that satisfy $\sum_{j \in T} w_j \leq w$. The final solution to our problem would be $M[n, W]$. A similar argument as above yields the following

recurrence for $M[i, w]$.

$$M[i,w] = \begin{cases} M[i-1,W] & \text{if } W < w_i \\ \max\{M[i-1,w], w_i + M[i-1,w-w_i]\} & \text{if } otherwise \end{cases}.$$

As before, we will design an algorithm to build up a table of all $M[i, w]$ values while computing each of them at most once.

Subset-Sum(n, W)

```
Array M[0..n; 0..W]
Initilize M[0, w] := 0 for all w = 0, ..., W.
for i ← 1 to n
    for w ← 0 to W
        do if W < w_i
            then M[i, w] ← M[i − 1, w]
            else M[i, w] ← max{M[i − 1, w], w_i + M[i − 1, w − w_i]}
    endfor
endfor
return M[n, W].
```

Using the above recurrence, one can show that $M[n, W]$ is the optimal solution weight for the items $1, \ldots, n$ with available weight W. Also, each value $M[i, w]$ is computed in constant time using the previous values. Thus the running time is $O(nW)$. Thus we have

Theorem 1. *The Subset-Sum(n, W) correctly computes the value of the optimal solution in time $O(nW)$.*

Exercise 3. Obtain a recurrence relation for the knapsack problem. Show that the knapsack problem can be solved in $O(nW)$ time