Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Dynamic programming is used when the sub problems are not independent. Dynamic Programming is a bottom – up approach – we solve all possible small problems and then combine them to obtain solutions for bigger problems.

Dynamic Programming is often used in optimization problems (A problem with many possible solutions for which we want to find an optimal solution)
Dynamic Programming works when a problem has the following two main properties.

- Overlapping Subproblems
- Optimal Substructure

Overlapping Subproblems: When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point in storing the solutions if they are not needed again.

Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

The Principle of Optimality
To use dynamic programming the problem must observe the principle of optimality, that whatever the initial state is, remaining decisions must be optimal with regard the state following from the first decision.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Chain – Matrix multiplication problem

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix.

There are $p \cdot r$ total entries in C and each takes $O(q)$ time to compute, thus the total time to multiply these two matrices is dominated by the number of scalar multiplication, which is $p \cdot q \cdot r$.

The time to compute C is dominated by the number of scalar multiplications which is pqr. We shall express costs of multiplying two matrices in terms of the number of scalar multiplications.

Given following matrices $\{A_1, A_2, A_3, \ldots, A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$$A_1 \times A_2 \times A_3 \times \ldots \times A_n$$

Matrix multiplication is an associative operation, but not a commutative operation. By this, we mean that we have to follow the above matrix order for multiplication, but we are free to **parenthesize** the above multiplication depending upon our need.

For example,

if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct ways:

$(A_1(A_2(A_3 A_4)))$,
$(A_1((A_2 A_3)A_4))$,
$((A_1 A_2)(A_3 A_4))$,
$((A_1(A_2 A_3))A_4)$,
$(((A_1 A_2)A_3)A_4)$.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain <A1, A2, A3> of three matrices. Suppose that the dimensions of the matrices are 10 × 100, 100 × 5, and 5 × 50, respectively. The possible order of multiplication are

a) If we multiply according to the parenthesization *((A1 A2)A3)*

- to compute the 10 × 5 matrix product *A*1 *A*2, we perform 10 · 100 · 5 = 5000 scalar multiplications
- to multiply this matrix product *A*1 *A*2 by matrix *A*3, we perform another 10 · 5 · 50 = 2500 scalar multiplications
- Hence, to compute the product *((A*1 *A*2*)A*3*),* a total of 7500 scalar multiplications.

b) If instead we multiply according to the parenthesization *(A*1*(A*2 *A*3*))*,

- to compute the 100 × 50 matrix product *A*2 *A*3 we perform 100 · 5 · 50 = 25,000 scalar multiplications
- to multiply this matrix product *A*2 *A*3 by matrix *A*1, we perform another 10 · 100 · 50 = 50,000 scalar multiplications
- Hence, to compute the product *(A*1*(A*2 *A*3*))*, a total of 75,000 scalar multiplications.

Thus, computing the product according to the first parenthesization is 10 times faster.

The ***matrix-chain multiplication problem*** can be stated as follows: Given a sequence of *n* matrices

$A_1, A_2, \ldots A_n$, and their dimensions $p_0, p_1, p_2, \ldots, p_n$, where where $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Step 1: The structure of an optimal parenthesization

Our first step in the dynamic-programming paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. For the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the notation $A_{i..j}$, where $i \le j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then any parenthesization of the product $A_i A_{i+1} \cdots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$.

That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of this parenthesization is thus the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

Step 2: Recursively define the value of an optimal solution

To help us keep track of solutions to subproblems, we will use a table, and build the table in a bottomup manner. For $1 \le i \le j \le n$, let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the $A_{i..j}$. The optimum cost can be described by the following recursive formulation.

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

To keep track of optimal subsolutions, we store the value of k in a table $s[i, j]$. Recall, k is the place at which we split the product $A_{i..j}$ to get an optimal parenthesization.

That is, $s[i, j] = k$ such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j$.

Step 3: Computing the value of an optimal costs

we perform the third step of the dynamic-programming paradigm and compute the optimal cost by using a tabular, bottom-up approach. The following pseudo-code assumes that matrix Ai has dimensions $pi-1 \times pi$ for $i = 1, 2, \ldots, n$. The input is a sequence $p = p_0, p_1, \ldots, p_n$, where $length[p] = n + 1$. The procedure uses an auxiliary table $m[1 \ldots n, 1 \ldots n]$ for storing the $m[i, j]$ costs and an auxiliary table $s[1 \ldots n, 1 \ldots n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We will use the table s to construct an optimal solution.

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n          ▷ l is the chain length.
 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
```

Time Complexity of matrix chain multiplication is **: O(n^3)**

Step 4: Constructing an optimal solution

The array $s[i, j]$ can be used to extract the actual sequence. The basic idea is to keep a split marker in $s[i, j]$ that indicates what is the best split. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $A_1, A_2, \ldots, A_n$

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2     then print "A"_i
3     else print "("
4          PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5          PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6          print ")"
```

Example problem:

Example 1 *We are given the sequence {4, 10, 3, 12, 20, 7}. The matrices have sizes 4×10, 10×3, 3×12, 12×20, 20×7. We need to compute M[i, j], 0 ≤ i, j ≤ 5. We know M[i, i] = 0 for all i.* **(UPTU MCA 2013)**

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Now products of 3 matrices

$$M[1,3]=\min\begin{cases} M[1,2]+M[3,3]+p_0p_2p_3 = 120+0+4\cdot3\cdot12 = 264 \\ M[1,1]+M[2,3]+p_0p_1p_3 = 0+360+4\cdot10\cdot12 = 840 \end{cases} = \mathbf{264}$$

$$M[2,4]=\min\begin{cases} M[2,3]+M[4,4]+p_1p_3p_4 = 360+0+10\cdot12\cdot20 = 2760 \\ M[2,2]+M[3,4]+p_1p_2p_4 = 0+720+10\cdot3\cdot20 = 1320 \end{cases} = \mathbf{1320}$$

$$M[3,5]=\min\begin{cases} M[3,4]+M[5,5]+p_2p_4p_5 = 720+0+3\cdot20\cdot7 = 1140 \\ M[3,3]+M[4,5]+p_2p_3p_5 = 0+1680+3\cdot12\cdot7 = 1932 \end{cases} = \mathbf{1140}$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 4 matrices

$$M[1,4]=\min\begin{cases} M[1,3]+M[4,4]+p_0p_3p_4 = 264+0+4\cdot12\cdot20 = 1224 \\ M[1,2]+M[3,4]+p_0p_2p_4 = 120+720+4\cdot3\cdot20 = 1080 \\ M[1,1]+M[2,4]+p_0p_1p_4 = 0+1320+4\cdot10\cdot20 = 2120 \end{cases} = \mathbf{1080}$$

$$M[2,5]=\min\begin{cases} M[2,4]+M[5,5]+p_1p_4p_5 = 1320+0+10\cdot20\cdot7 = 2720 \\ M[2,3]+M[4,5]+p_1p_3p_5 = 360+1680+10\cdot12\cdot7 = 2880 \\ M[2,2]+M[3,5]+p_1p_2p_5 = 0+1140+10\cdot3\cdot7 = 1350 \end{cases} = \mathbf{1350}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 5 matrices

$$M[1,5]=\min\begin{cases} M[1,4]+M[5,5]+p_0p_4p_5 = 1080+0+4\cdot20\cdot7 = 1544 \\ M[1,3]+M[4,5]+p_0p_3p_5 = 264+1680+4\cdot12\cdot7 = 2016 \\ M[1,2]+M[3,5]+p_0p_2p_5 = 120+1140+4\cdot3\cdot7 = 1344 \\ M[1,1]+M[2,5]+p_0p_1p_5 = 0+1350+4\cdot10\cdot7 = 1630 \end{cases} = \mathbf{1344}$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL-PARENS procedure.

Now for optimal parenthesization , Each time we find the optimal value for M[i,j] we also store the value of ***k*** that we used. If we did this for the example, we would get

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The *k* value for the solution is 2, so we have $((A_1\ A_2)(A_3\ A_4\ A_5))$. The first half is done. The optimal solution for the second half comes from entry *M*[3,5].The value of *k* here is 4, so now we have $((A_1\ A_2)((A_3\ A_4)\ A_5))$.Thus the optimal solution is to parenthesize $((A_1\ A_2)((A_3\ A_4)\ A_5))$.