Muhammad Ammar

BSCS_F19_M_63
Section B - Morning

Matrix Multiplication

Question #1 - Part #1

Dynamic Programming

Solving a problem in particular steps at runtime, using the techniques of **Memoization** and **Tabulation** is called **Dynamic Programming**. Dynamic Programming adapts optimized recursive approach, which significantly reduces the Memory Usage, by utilizing it in a smarter way, which implicate Lesser CPU Cycles.

- **Memoization** is a technique used to temporarily store the results of recursive calls, and utilize them where needed in the future.
- **Tabulation**: is a Bottom-Up approach, in contrast to the Top-Bottom approach of regular Recursive methods. Tabulated algorithms start with base cases and work their way up to the final solution.

In Dynamic Programming, Memoization requires Tables, or Matrices to store the Intermediate results of recursive calls. This'll be illustrated by the examples below.

Matrix Chain Multiplication

As we are familiar with the Matrix Multiplication. Usually, two or more Matrices are multiplied and we get the answers. But sometimes, the number of calculations differ in the approaches to solve the matrices. This matters the most, when there are 'n' numbers of matrices, which requires a large amount of calculations. In that case, the number of calculations in different approaches may differ. Hence, we need a most efficient approach, which needs minimal

calculations to get the end result. For that purpose, we use Dynamic Approach to figure out which approach will imply minimal calculations.

Some properties of Matrix Chain multiplication are as follows;

Non-Commutative

Matrices can be multiplicated, if and only if, the Column of 1st Matrix and Row of 2nd one, are similar in magnitude. If their order is reversed and if the above condition gets unsatisfied, then matrices will be incompatible to get multiplied. Hence, their order always matters, until it's violation doesn't cause incompatibility issues.

Associative

Matrix Multiplication is possible, regardless of the parenthesization. For example, for Matrices A,B, and C, Multiplication of A x (B x C) and (A x B) x C, both are possible and will result in similar results.

The parenthesization of matrices also determines how many operations will get performed. For a multiplication chain of integers, parenthesizing them does not affect the number of operations performed, but for matrices the change is significant.

- ➔ Let us use the following example:
  - ◆ Let A be a **2 x 10** matrix
  - ◆ Let B be a **10 x 50** matrix
  - ◆ Let C be a **50 x 20** matrix
- ➔ Let's get back to our example: We will show that the way we group matrices when multiplying A, B, C matters:

Consider computing **A(BC):**

Multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix

Multiplications for A(BC) = 2x10x20 = 400

Total multiplications = 10000 + 400 = 10400.

Consider computing **(AB)C**:

Multiplications for (AB) = 2x10x50 = 1000, creating a '2 x 50' answer matrix

Multiplications for (AB)C = 2x50x20 = 2000,

Total multiplications = 1000 + 2000 = 3000

Dynamic Programming Formulation

The dynamic programming solution involves breaking up the problem into sub-problems whose solutions can be combined to solve the global problem. Let $A_{i..j}$ be the result of multiplying matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

$$A3_{4\times5} \qquad A4_{5\times2} \qquad A5_{2\times8} \qquad A6_{8\times7} = A3..6_{4\times7}$$

At the highest level of parenthesization, we multiply two matrices

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n} \qquad 1 \le k \le n - 1$$

Divide and conquer strategy can't be used as the value of Optimal **k** is unknown yet. So, all possible values of k have to be considered. To identify the best of them, Dynamic approach will be considered.

For that purpose, we will use **Memoization** to store the temporary results in a table, and **Tabulation** for Bottom-Up approach. For **1 ≤ i ≤ j ≤ n**, let M[i,j] denote the minimal number of multiplications needed to compute $A_{i..j}$. The optimal can be described by the following recursive formulation.

- ➔ If i = j, there is only one matrix and thus m[i,i] = 0 (the diagonal entries).
- ➔ If i < j, then we are asking for the product $A_{i..j}$.
- ➔ This can be split by considering each k, i ≤ k < j, as $A_{i..k}$ times $A_{k+1..j}$.

The optimal time to compute $A_{i..k}$ is m[i,k] and optimal time for $A_{k+1..j}$ is in m[k + 1,j]. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1} \times p_k \times p_j$. This suggests the following recursive rule:

★ **m[i,i] = 0**

★ **m[i,j] = min (m[i,k] + m[k + 1,j] + $p_{i-1}p_kp_j$) i≤k<j**

Now the 'm' will be filled along the diagonals as the Recursive approach shouldn't be used here. For that, We will, Set all m[i,i] = 0 using the base condition. Compute cost for multiplication of a sequence of 2 matrices. These are m[1, 2], m[2, 3], m[3, 4], ..., m[n − 1, n].

Let, m[1, 2] = m[1, 1] + m[2, 2] + $p_0 \cdot p_1 \cdot p_2$

m[1, 1]	m[1, 2]			
	m[2, ↓2]	←m[2, 3]		
		m[3, ↓3]	←m[3, 4]	
			m[4, ↓4]	←m[4, 5]
				m[5, ↓5]

Next, we compute the cost of multiplication for sequences of three matrices. These are m[1,3],m[2,4],m[3,5],...,m[2,n].m[1,3], for example is

m[1,3] =min (m[1,1] + m[2,3] + $p_0 \cdot p_1 \cdot p_3$)

m[1,2] + m[3,3] + $p_0 \cdot p_2 \cdot p_3$

The process is repeated for 4,5, and higher matrices, until the final result ends up in m[1,n].