Fundamentals of Algorithm Design and Machine Learning

Name	**:**	Bansode Vasu Satyawan	**Date**	:	21-01-2025
Roll No.	**:**	24BM6JP13	**Lecture slot**	:	10:00 hrs to 11:00 hrs.

Algorithm Design Principle

Step 1. Initial Solution
a. Recursive Formulation
b. Correctness
c. Complexity Analysis

Step 2. Exploration of Structure
a. Decomposition
b. Analyse recursive structure
c. Re-composition

Step 3. Solution Refinement
a. Balance/split
b. Analysis of recurrence
c. Identical subproblems (overlapping substructure)

Step 4. Data structuring and Complexity
a. Reuse memory/saved info (memorization)
b. Analysis of space complexity

Step 5. Final solution
a. Traversal of recursive struct
b. Complexity
c. Pruning/backtracking

Step 6. Implementation

Algorithm Paradigms

1. Divide and conquer
2. Dynamic programming
3. Greedy approach
4. Branch and bound
5. Combinatorial optimisation
6. Approximation
7. Randomisation

Problems Discussed

1.	Max	**2.**	Max and Min	**3.**	Max and 2nd Max	**4.**	Sort (tournament)	**5.**	Coin exchange
6.	Searching	**7.**	Sorting	**8.**	Fibonacci	**9.**	Median Finding	**10.**	Closest pair

Dynamic Programming

Dynamic programming (DP) is required to solve problems that exhibit overlapping subproblems and optimal substructure. It works by breaking a problem into smaller subproblems, solving each just once, and storing their results to avoid redundant calculations. Unlike the divide and conquer paradigm, which solves each subproblem independently, DP reuses solutions to overlapping subproblems, making it more efficient. Memoization is incorporated in DP by storing intermediate results in a data structure and retrieving them when needed, significantly reducing time complexity.

Matrix Chain Multiplication Problem

Problem Statement:

Given the matrix multiplication chain of j matrices, $M_1M_2M_3 \dots \dots M_j$, find out the most efficient way to group the matrices (by parenthesization) that minimizes the total number of scalar multiplications.

Goal:

To determine the optimal order of matrix multiplication while keeping the sequence of matrices unchanged (since matrix multiplication is not commutative) to minimize computational cost.

Initial Solution:

For Demonstration, consider the matrix multiplication chain of 4 matrices $M_1M_2M_3M_4$. Parenthesization of this chain can be done in five distinct ways: $M_1(M_2(M_3M_4))$, $M_1((M_2M_3)M_4)$, $(M_1M_2)(M_3M_4)$, $(M_1(M_2M_3))M_4$ and $((M_1M_2)M_3)M_4$. The way of parenthesization can have a dramatic impact on the computation cost. Let us assume the dimensions of the matrices $M_1, M_2, M_3 \; and \; M_4$ be 10 x 100, 100 x 5, 5 x 50 and 50 x 5 respectively. The computational cost of matrix multiplication (using a standard algorithm) for matrices of size let's say p x q and q x r is dominated by the number of scalar multiplications which is p x q x r. Thus, the computational cost of matrix multiplication is O(p x q x r). Therefore, the computation cost of evaluating the matrix multiplication chain using the five parenthesization would be:

$M_1(M_2(M_3M_4))$	5 x 50 x 5 + 100 x 5 x 5 + 10 x 100 x 5 = 8750
$M_1((M_2M_3)M_4)$	100 x 5 x 50 + 100 x 50 x 5 + 10 x 100 x 5 = 55000
$(M_1M_2)(M_3M_4)$	10 x 100 x 5 + 5 x 50 x 5 + 10 x 5 x 5 = 6500
$(M_1(M_2M_3))M_4$	100 x 5 x 50 + 10 x100 x 50 + 10 x 50 x 5 = 77500
$((M_1M_2)M_3)M_4$	10 x 100 x 5 + 10 x 5 x 50 + 10 x 50 x 5 = 10000

The computation cost of evaluating the matrix multiplication chain as per $(M_1(M_2M_3))M_4$ is around 12 times than evaluating as per $(M_1M_2)(M_3M_4)$.

Consider a matrix multiplication chain of j matrices, $M_1M_2M_3 \dots \dots M_j$. Let P(j) denote the number of parenthesizations possible. When j = 1, the sequence consists of only one

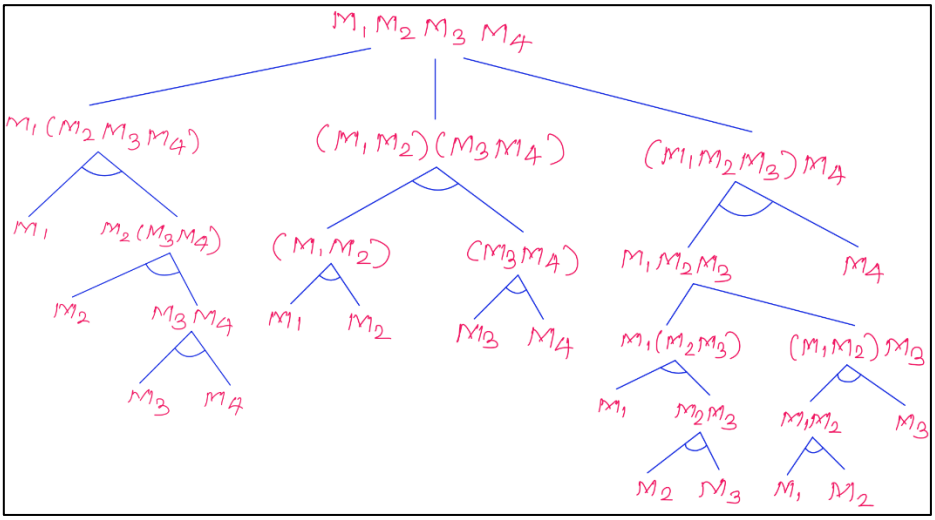
matrix, and there is only one way to parenthesize the matrix product fully. When j ≥ 2, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the k th and (k+1)st matrices for any k = 1, 2, 3, …..j – 1. Thus, we obtain a recurrence,

$$P(j) = \begin{cases} 1 & , \quad if\ j = 1 \\ \sum_{k=1}^{j-1} P(k)\, P(j-k)\,, & if\ j \geq 2 \end{cases}$$

This recurrence grows as $\Omega(2^j)$ (refer to Catalan numbers). Thus, exhaustively checking the computation cost for all possible parenthesizations is exponential in j. Therefore, the brute-force exhaustive search method makes for a poor strategy when determining how to parenthesize a matrix chain optimally.

Exploration of Recurrence Structure:

For simplicity, consider the recurrence structure for the matrix multiplication chain $M_1M_2M_3M_4$, as shown in the figure.



As we can already see, multiple identical sub-problems exist. The problem can be solved more efficiently using dynamic programming incorporating memoization to store the results of prior computations.

Solution Refinement:

For a matrix multiplication chain of n matrices: $M_1M_2M_3 \ldots \ldots M_n$. Where a matrix M_p is of dimensions $m_{p-1} \times m_p$ for $p = 1, 2, 3, \ldots \ldots n$. Let $C[i,j]$ denote the computation cost of evaluating the chain $M_iM_{i+1}M_{i+2} \ldots \ldots M_j$ most efficiently.

We have the following three base case conditions,

- When $i > j$, we have $C[i,j] = 0$ since the matrix production is not commutative
- When $i = j$, we have $C[i,j] = 0$, since there is only one matrix in the chain

- When $i < j$, we have $C[i,j] = \min_{i \le k < j} (C[i,k] + C[k+1,j] + n_{i-1} \times n_k \times n_j)$

Therefore, we have the following recurrence,

$$C[i,j] = \begin{cases} 0 & if\ i > j \\ 0 & if\ i = j \\ \min\limits_{i \le k < j} (C[i,k] + C[k+1,j] + n_{i-1} \times n_k \times n_j) & if\ i < j \end{cases}$$

The memoization can be done recursively (top-down approach). In this approach, $C[i,j]$ recursively calls $C[i,k]$ and $C[k+1,j]$, where k is the index at which the chain is partitioned for parenthesization. The recursion continues until the base case $i = j$ is reached, meaning there is only a single matrix. The solutions for subproblems are stored in a memoization table to avoid redundant computations. However, the drawback of the recursive approach is that storing function calls in the stack during recursion can lead to increased memory usage, especially for deep recursion trees. To avoid this, memoization can be done iteratively (bottom-up approach).

To demonstrate how this recursion works, let us consider the example of a matrix multiplication chain of four matrices $M_1M_2M_3M_4$. Our objective is to evaluate $C[1,4]$. Consider,

- $C[1,1] = C[2,2] = C[3,3] = C[4,4] = 0$.
- $C[1,2] = C[1,1] + C[2,2] + n_0 \times n_1 \times n_2 = n_0 \times n_1 \times n_2$
- $C[2,3] = C[2,2] + C[3,3] + n_1 \times n_2 \times n_3 = n_1 \times n_2 \times n_3$
- $C[3,4] = C[3,3] + C[4,4] + n_2 \times n_3 \times n_4 = n_2 \times n_3 \times n_4$
- $C[1,3] = \min \begin{cases} C[1,1] + C[2,3] + n_0 \times n_1 \times n_3 \\ C[1,3] + C[3,3] + n_0 \times n_2 \times n_3 \end{cases} = \min \begin{cases} n_1 \times n_2 \times n_3 + n_0 \times n_1 \times n_3 \\ n_0 \times n_1 \times n_2 + n_0 \times n_2 \times n_3 \end{cases}$
- $C[2,4] = \min \begin{cases} C[2,2] + C[3,4] + n_1 \times n_2 \times n_4 \\ C[2,3] + C[4,4] + n_1 \times n_3 \times n_4 \end{cases} = \min \begin{cases} n_2 \times n_3 \times n_4 + n_1 \times n_2 \times n_4 \\ n_1 \times n_2 \times n_3 + n_1 \times n_3 \times n_4 \end{cases}$
- $C[1,4] = min \begin{cases} C[1,1] + C[2,4] + n_0 \times n_1 \times n_4 \\ C[1,2] + C[3,4] + n_0 \times n_2 \times n_4 \\ C[1,3] + C[4,4] + n_0 \times n_3 \times n_4 \end{cases}$

$$= min \begin{cases} C[2,4] + n_0 \times n_1 \times n_4 \\ n_0 \times n_1 \times n_2 + n_2 \times n_3 \times n_4 + n_0 \times n_2 \times n_4 \\ C[1,3] + n_0 \times n_3 \times n_4 \end{cases}$$

The memoization table for this is as follows:

		i ⟶			
j ↓		**1**	**2**	**3**	**4**
	1	0			
	2	C[1,2]	0		
	3	C[1,3]	C[3,2]	0	
	4	C[4,4]	C[2,4]	C[3,4]	0

Notice that for computing C[1,3] we need values of only C[1,1], C[2,3], C[1,3] and C[3,3]. Similarly, for computing C[2,4], we need values of only C[2,2], C[3,4], C[2,3] and C[4,4], and so on and so forth. We can generalise from this observation that for computing any $C[i,j]$ we need values from the j^{th} row and i^{th} column. Hence, we can fill the table iteratively, starting from the diagonal and approaching the left-down corner.

Data structuring and Complexity:

Since there are $O(n^2)$ values to be filled (subproblems), and for each subproblem, O(n) possible values of k. Hence, the time complexity of the algorithm is $O(n^3)$ (which is much better than $O(2^n)$) and total space complexity is $O(n^2)$.

Longest Common Subsequence (LCS) Problem

A subsequence is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements. More formally, given a sequence $X = < x_1, x_2, x_3,....x_m >$ and another sequence $Y = < y_1, y_2, y_3,....y_n >$ is a subsequence of X if there exists a strictly increasing sequence $<i_1, i_2, i_3,....i_k>$ of indices of X such that for all j = 1,2,3,..k, we have $x_{i_j} = y_j$.

Problem Statement:

Given two sequences, X of length m and Y of length n, find the longest subsequence common to both X and Y.

Goal:

To determine the maximum length of a common subsequence between the two sequences using dynamic programming to minimize computational cost.

Initial solution:

We need to find LCS of $X = < x_1, x_2, x_3,....x_m >$ and $Y = < y_1, y_2, y_3,....y_n >$. A naïve approach as an initial solution would be to search for every element of X iteratively in Y. This would yield us time complexity of $O(n^2)$.

Solution Refinement:

When a $x_m = y_n$, length of LCS would be length of LCS of X_{m-1} and Y_{n-1} plus 1. When $x_m \neq y_n$, the LCS would be the longest LCS out of LCS of 1) X_{m-1} and Y 2) Y_{n-1} and X. This shows that LCS has many overlapping subproblems since finding LCS of X_{m-1}, Y and Y_{n-1}, X share a common subproblem of finding LCS of X_{m-1} and Y_{n-1}.

Let L[i, j] be the length of LCS of X_i and Y_j where i = 0,1,2,3,….m and j = 0,1,2,3,….n. The base case is if either of i and j is 0, i.e. one of the sequences has length 0, then LCS would also be 0. Therefore, we have,

$$L[i,j] = \begin{cases} 0 \; if \; i = 0 \; or \; j = 0 \\ 1 + \; L[i-1, j-1] \; if \; i,j > 0 \; and \; x_i = \; y_j \\ max \begin{cases} L[i-1,j] \\ L[i,j-1] \end{cases} when \; x_i \neq \; y_j \end{cases}$$

The problem can be either solved recursively where each L[i, j] iteratively calls L[i, j-1] and L[i-1, j] until a base case is reached, then the solutions to subproblems are recombined from bottom to top and the results are stored in the memoization table. However, storing function calls in the stack itself consumes some storage. Iteratively computing the solutions from the base case is another approach that is equally efficient and more intuitive.

Consider the example where X = <Y, E, L, L, O, W> and Y = <H, E, L, L, O>. As we can see that LCS is <E, L, L, O>. Consider the memoization table for L[i, j] . We aim to evaluate L[6,5] and find out the LCS. We can do this iteratively by starting with L[0,0] = L[0,:] = L[:,0] = 0. Since $x_1 \neq y_1$, L[1,1] would be maximum of L[0,1] and L[1,0] which is 0. Similarly, we have L[:, 1] = L[1,:] = 0 since 'H' from sequence Y is not in sequence X and 'Y' from sequence X is not in sequence Y. L[2,1] = 0 since <Y, E> doesn't share have any common sequence with <H>. Similarly, we can fill the rest of the table either row-wise or column-wise.

				Y					
	i		j →						
				0	**1**	**2**	**3**	**4**	**5**
					H	**E**	**L**	**L**	**O**
		0		0	0	0	0	0	0
X	↓	**1**	**Y**	0	0	0	0	0	0
		2	**E**	0	0	1	1	1	1
		3	**L**	0	0	1	2	2	2
		4	**L**	0	0	1	2	3	3
		5	**O**	0	0	1	2	3	4
		6	**W**	0	0	1	2	3	4

As we can see from the table, L[6,5] = 4; hence, the length of LCS of X = <Y, E, L, L, O, W> and Y = <H, E, L, L, O> is 4. To find LCS, we need to find the indices i and j where the increment occurred. This can be done by moving backwards from L[6,5] towards L[0,0] by following the direction where the increment occurred (as shown by the red arrow). Hence, we can see that the LCS is <E, L, L, O>.