Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
Department of Master of Computer Application

Experiment	3
Aim	To understand and implement Dynamic Programming Approach
Objective	1) Write Pseudocode for given problems and understanding the implementation of Dynamic Programming 2) Solve Matrix Multiplication Problem using Dynamic Programming 3) Calculating time complexity of the given problems
Name	Vivek Tiwari
UCID	2023510059
Class	FYMCA
Batch	C
Date of Submission	28-02-24

Algorithm and Explanation of the technique used	1. Craft a recursive method accepting start and end indices to delineate a matrix group's bounds. 2. Within this method, loop through intermediate indices, splitting the given range into two subgroups. 3. Invoke the recursive method recursively on these two subgroups. 4. Compute the minimal scalar multiplication cost across all possible splits by amalgamating the costs of the two subgroups and the multiplication cost of the current split. Return this minimum value. 5. The ultimate answer is the minimum value yielded by the recursive method when considering the entire range from the first to the last matrix.

Matrix Chain Multiplication

Given: $A_{3\times4} \times B_{4\times5} \times C_{5\times6}$

$\{3, 4, 5, 6\}$

$p_0 \quad p_1 \quad p_2 \quad p_3$

$M_1 \quad M_2 \quad M_3$

$[3\times4] \quad [4\times5] \quad [5\times6]$

(1)

	1	2	3
1	0	60	150
2		0	120
3			0

(2) $M[1,2] = M_1 \times M_2 = 3 \cdot 4 \cdot 5 = 60$

$M[2,3] = M_2 \times M_3 = 4 \cdot 5 \cdot 6 = 120$

(3) $M[1,3] = M_1 M_2 M_3$

$\rightarrow \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 \end{cases}$

$= \min \begin{cases} 60 + 0 + 90 \\ 0 + 120 + 72 \end{cases}$

$= \min \begin{cases} 150 \\ 192 \end{cases}$

$\therefore$ Min of $M[1,3]$ is 150.

Time Complexity

Consider the recurrence relation of computing $M[i][j]$

$$M[i][j] = \min_{i \le k < j} \{ M[i][k] + M[k+1][j] + p_{i-1} p_k p_j \}$$

Here,

(i) i & j represent start & end of indices of the subchain of matrices being considered

(ii) k represents the index at which the chain is split.

(iii) i represents the number of rows in the matrix & p_{i-1} represents no. of columns

The complexity to compute each cell $M[i][j]$ is $O(j-i)$ & there are $O(n^2)$ cells to compute in total (since there are $O(n^2)$ subproblem).

Therefore, overall complexity is close to $O(n^3)$.

Program(Code)

```java
public class  dplab3 {
    static int matrixChainOrder(int p[], int i, int j) {
        if (i == j)
            return 0;
        int mini = Integer.MAX_VALUE;
        for (int k = i; k < j; k++) {
            int count = matrixChainOrder(p, i, k)
                    + matrixChainOrder(p, k + 1, j)
```

<table>
<tr>
<td></td>
<td>

```java
                    + p[i - 1] * p[k] * p[j];
                mini = Math.min(count, mini);
            }
            return mini;
        }

    public static void main(String[] args) {
        int arr[] = {3, 4, 5, 6};
        int N = arr.length;
        System.out.println("Minimum number of
multiplications is " + matrixChainOrder(arr, 1, N - 1));
    }
}
```

</td>
</tr>
<tr>
<td>Output</td>
<td>

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Minimum number of multiplications is 150
```

</td>
</tr>
<tr>
<td>Justification of the complexity calculated</td>
<td>The time complexity analysis correctly examines the number of sub-tasks,
the effort required for each sub-task, and their combined impact. It notes that the algorithm tackles a quadratic number of sub-tasks, as it populates
an n x n matrix, with each cell representing a sub-problem. For every sub-
problem, it iterates through a linear number of possibilities, performing constant-time operations for each iteration. Consequently, the overall effort
to populate the entire matrix is cubic in the number of matrices. Thus, the
algorithm's time complexity when leveraging dynamic programming for matrix chain multiplication is O(n^3), where n denotes the number of matrices.</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Dynamic programming is a powerful technique that offers benefits like optimal substructure identification, memoization capabilities, time-efficient
solutions, versatility across domains, and deterministic outcomes. It finds
applications in diverse areas such as recursive function evaluation, path optimization, combinatorial problems, sequence analysis, matrix operations, and currency exchange optimization. By methodically decomposing problems into overlapping subproblems and reusing solved
subproblems, dynamic programming effectively tackles complex optimization challenges with enhanced efficiency.</td>
</tr>
</table>