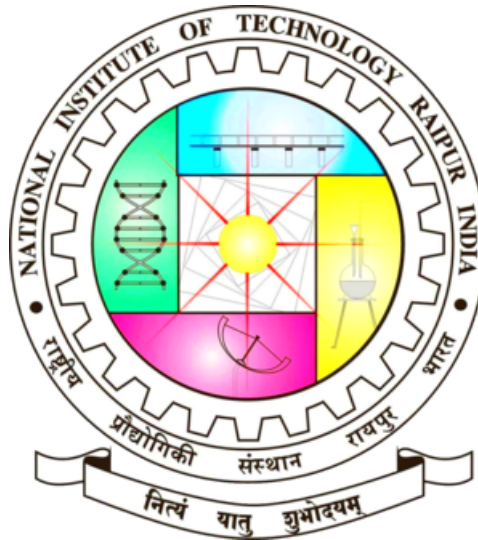


NATIONAL INSTITUTE OF TECHNOLOGY RAIPUR



DESIGN AND ANALYSIS OF ALGORITHMS

Topic: Dynamic Programming & Implementation of Matrix Chain Multiplication problem using Bottom-up DP Approach

Name: Ketaki

Roll no:

Branch: Information Technology (5th sem)

Subject Code:

Submitted to:

INDEX

Sr. No.	Topic	Pg. No.
1	Introduction	3
2	Dynamic Programming	3,4
3	Bottom-up DP Approach	4
4	MCM Problem	5,6
5	Algorithm with Example	6,7,8
6	Complexity Analysis	9
7	Source code	9
8	Output	10

INTRODUCTION

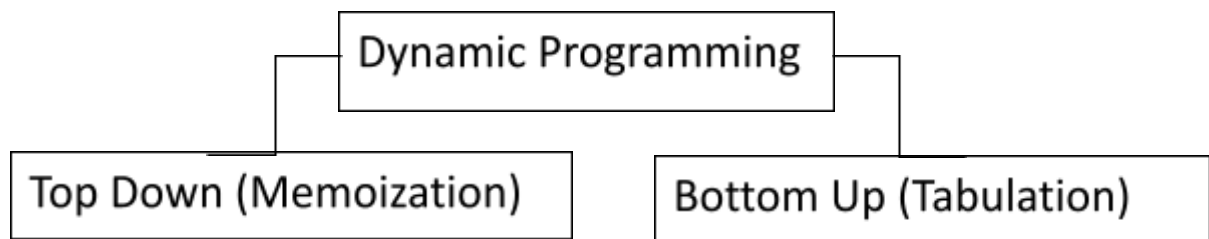
In this document we will see how problems are solved using dynamic programming. Then we will discuss the bottom-up approach in brief. After that we will try to understand working of this approach by taking one of the famous dynamic programming problems i.e., Matrix Chain Multiplication. By doing step by step analysis with the help of an example we will figure out time and space complexities along with source code and output.

DYNAMIC PROGRAMMING

Dynamic Programming is a technique used to do optimization over plain recursive problems. It helps to reduce time complexity from exponential to polynomial. The idea is that, if we carefully observe a recursive tree for a particular problem then in most of the cases, we are solving the same problems again and again, which results in exponential time complexity for the whole computation. But what if we store the result of these sub problems and reuse it whenever required? It will not only reduce time complexity but also make calculations simpler and easier. Thus, by using this technique, we are not required to solve sub-problems repeatedly and it guarantees to find optimal solutions if they exist.

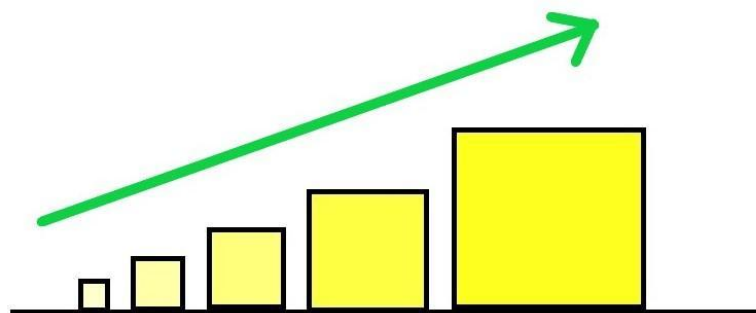
Thus, we break complex problems into many overlapping subproblems and do the same to them until we can solve them easily. Results of these sub-problems are remembered and then combined to achieve the best possible solution.

There are two approaches to solve a problem using dynamic programming:



BOTTOM-UP APPROACH

In this technique, we remove recursion and try to fix it with a tabulation approach. We start solving from base cases and then use its results to solve sub-problems and then main problems respectively. Here we use arrays to store the result of base cases and sub-problems so that they become easier to access further. We can use loops to iterate over sub-problems. Because there is no recursion, the problem of stack overflow and overhead of recursive calls is solved. In this way, as we start from small problems and then solve bigger and bigger ones, it is an ascending approach of solving step by step, and eventually called bottom up.



MCM PROBLEM

The Matrix Chain Multiplication Problem is one of the most famous problems of dynamic programming. Here, a sequence of matrices is given and we have to find the most optimal way to multiply all of them so that the number of operations are reduced. In simple terms we have to find the order in which we should multiply them and then return the total number of operations.

Let say we have given three matrices A, B and C. So, the ways in which we can multiply them is as follows:

$$ABC = (AB)C = A(BC)$$

If we change the parenthesis then the number of operations also changes, we can see this by example.

Matrix	Dimensions
A	3 X 4
B	4 X 5
C	5 X 6

$$(AB)C = (3*4*5) + (3*5*6) = 60 + 90 = 150$$

$$A(BC) = (3*4*6) + (4*5*6) = 72 + 120 = 192$$

From the above example we can see that the number of operations differ from case to case.

So, we have to find out the most efficient way in which we have to return the minimum number of operations to be performed in order to multiply all the given matrices.

INPUT

Given an array of numbers which represents dimensions of matrices. Ex:

4	6	9	3	5	8
M1	M2	M3	M4	M5	

If array size is n , then it represents $n-1$ matrices.

Here $n=6$ and matrices are 5.

ALGORITHM

Step 1: To solve the problem with a Bottom-Up approach let us first think about the base cases. So, the minimum size of the array to form a matrix is 2. The array of size 2 represents a single matrix and thus the operations required are zero.

Here we will make a two-dimensional array named 'dp' where $dp[i][j]$ represents minimum operations needed to multiply from matrix i to matrix j . Ex. $dp[2][4]$ represents minimum operations required to multiply M2, M3 and M4.

Now filling the dp matrix with base case conclusion:

	0	1	2	3	4	5
		0				
			0			
				0		
					0	
						0

From above figure,

$$dp[1][1] = dp[2][2] = dp[3][3] = dp[4][4] = dp[5][5] = dp[6][6] = 0$$

Green blocks do not represent any matrix, so there is no need to fill them.

Step 2: Now moving forward from base cases the next problem is when array size is 3. In this case two matrices are represented by an array. There is only one way to partition in between them so number of operations can be calculated as: \Rightarrow operations required to form M1 and M2 + Operation required to multiply them together

$$dp[1][2] = dp[1][1] + dp[2][2] + (arr[0]*arr[1]*arr[2])$$

Now filling the blocks of dp table which represents 2 matrices.

0	1	2	3	4	5
	0	216			
		0	162		
			0	135	
				0	120
					0

Calculations:

$$dp[1][2] = M1M2 = 0+0+ (arr[0]*arr[1]*arr[2]) = 4*6*9 = 216$$

$$dp[2][3] = M2M3 = 0+0+ (arr[1]*arr[2]*arr[3]) = 6*9*3 = 162$$

Similarly other calculations are done.

Step 3: Now if the size of the array becomes 4 then it will represent 3 matrices. Here the number of partitions we can do is 2 so the final result will be the minimum of both.

So, $M1M2M3 = (M1M2)M3 = M1(M2M3)$

$(M1M2)M3 = dp[1][2] + dp[3][3] + \text{cost to multiply resultant of } M1M2 \text{ with } M3$ i.e., $arr[0]*arr[2]*arr[3]$

$$= 216 + 0 + (4*9*3) = 216 + 108 = 324$$

$M1(M2M3) = dp[1][1] + dp[2][3] + \text{cost to multiply } M1 \text{ with resultant of } M2M3$ i.e., $arr[0]*arr[1]*arr[3]$

$$= 0 + 162 + (4*6*3) = 162 + 72 = 234$$

Hence, $dp[1][3] = \min(324, 234) = 234$

In this way, we have to make possible partitions and then calculate the effective number of operations by taking a minimum from all of them.

Now, filling all the cells by using this method.

0	1	2	3	4	5
	0	216	234	294	450
		0	162	252	426
			0	135	336
				0	120
					0

Finally, we have to compute the result for $M1M2M3M4M5$ which is represented by cell $dp[1][5]$. Hence the answer to the above problem will be **450**.

COMPLEXITY ANALYSIS

TIME COMPLEXITY

If an array of size n is given then it represents $n-1$ matrices. Now the cells in dp matrix we have to fill are:

$$= n*(n-1)/2 = \frac{1}{2}(n^2 - n) = O(n^2)$$

For each cell we have to partition in all possible ways. Now, considering partitions for all cells they will range from 0 to $n-1$. So, the time complexity of program is:

= No of cells to be filled X cost to fill each cell

$$= O(n^2) \times O(n) = O(n^3)$$

SPACE COMPLEXITY

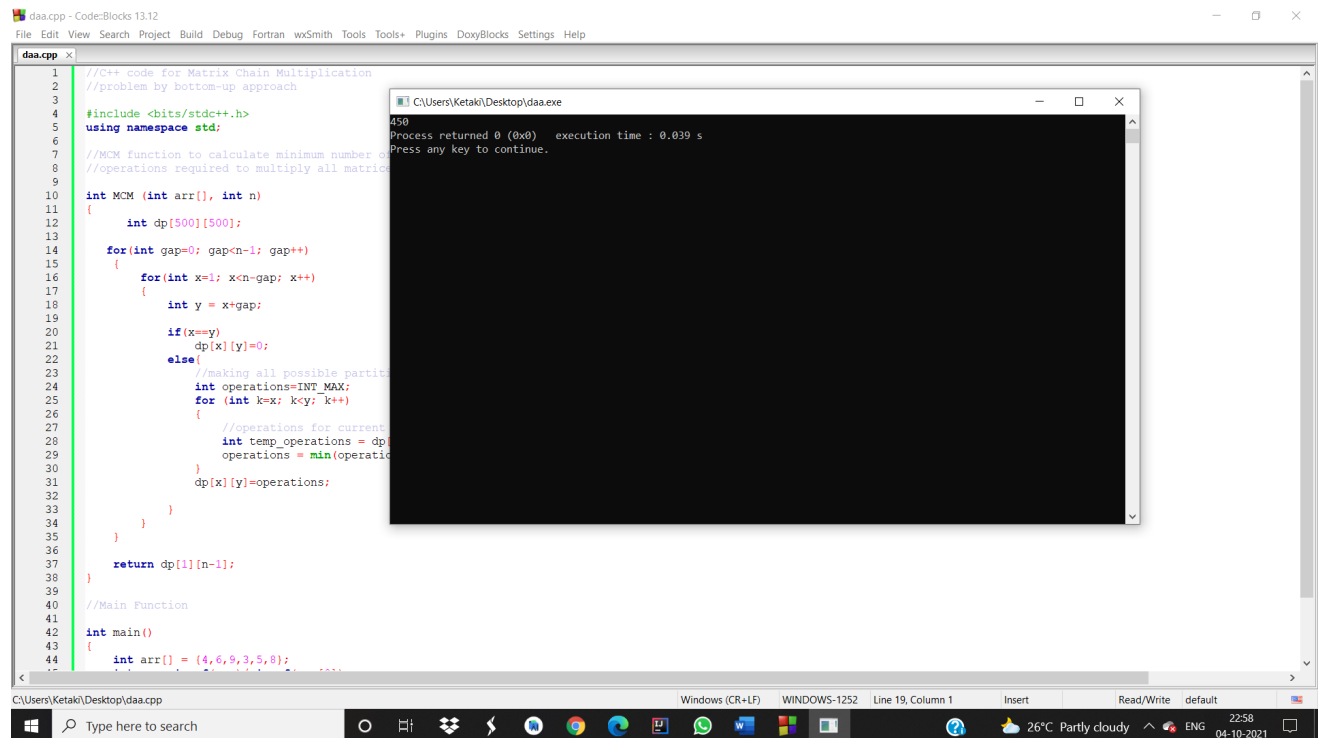
Extra space required is for the dp matrix which is of dimensions $n \times n$. So, the space complexity will be:

$$= O(n^2)$$

SOURCE CODE

<https://github.com/Ketaki-Gangadhar/DAA-Term-Project>

OUTPUT



The screenshot shows a C++ IDE with a file named `daa.cpp` open. The code implements a Matrix Chain Multiplication (MCM) algorithm using dynamic programming. The main function calls `MCM(arr, 8)` with the array `{4, 6, 9, 3, 5, 8}`. The output window shows the result `450` and the execution time `0.039 s`.

```
1 //C++ code for Matrix Chain Multiplication
2 //problem by bottom-up approach
3
4 #include <bits/stdc++.h>
5 using namespace std;
6
7 //MCM function to calculate minimum number of
8 //operations required to multiply all matrices
9
10 int MCM (int arr[], int n)
11 {
12     int dp[500][500];
13
14     for(int gap=0; gap<n-1; gap++)
15     {
16         for(int x=1; x<n-gap; x++)
17         {
18             int y = x+gap;
19
20             if(x==y)
21                 dp[x][y]=0;
22             else{
23                 //making all possible partitions
24                 int operations=INT_MAX;
25                 for (int k=x; k<y; k++)
26                 {
27                     //operations for current partition
28                     int temp_operations = dp[x][k] + dp[k+1][y] + arr[x]*arr[k+1]*arr[y];
29                     operations = min(operations, temp_operations);
30                 }
31                 dp[x][y]=operations;
32             }
33         }
34     }
35
36     return dp[1][n-1];
37 }
38
39 //Main Function
40
41 int main()
42 {
43     int arr[] = {4, 6, 9, 3, 5, 8};
44 }
```

Output window content:

```
C:\Users\Ketaki\Desktop\daa.exe
450
Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

THANK YOU