Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	4
Aim	To implement dynamic algorithms.
Name	Harsh Mukesh Jain
UID No.	2021300048
Class & Division	SE Comps A (C – Batch)
Date of Performance	5-02-2023
Date of Submission	18-02-2023

Theory/Experiment:

Dynamic Programming :

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.

The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

The approach of solving problems using dynamic programming algorithm has following 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.

Matrix Chain Multiplication :

The main aim is that for the given dimension of a sequence of matrices in an array dim[], where the dimension of the ith matrix is (dim[i-1] * dim[i]), the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

Two matrices of size m*n and n*p when multiplied, they generate a matrix of size m*p and the number of multiplications performed are m*n*p.

Now, for a given chain of N matrices, the first partition can be done in N-1 ways. For example, sequence of matrices A, B, C and D can be grouped as (A)(BCD), (AB)(CD) or (ABC)(D) in these 3 ways.

So a range [i, j] can be broken into two groups like {[i, i+1], [i+1, j]}, {[i, i+2], [i+2, j]}, . . . , {[i, j-1], [j-1, j]}.

Each of the groups can be further partitioned into smaller groups and we can find the total required multiplications by solving for each of the groups.
The minimum number of multiplications among all the first partitions is the required answer.

Algorithm :

```
MCM(dim)
for i = 0 to n – 1
        c[i][j] = 0;
        for j = 0 to n – 1
                c[i][j] = 0;
        for k = 0 to n – 2
                d = d + 1;
                j = d;
        for i = 1 to n – d
                if(i == j)
                        dp[i][j] = 0;
                        paranthesis[i][j] = 0;
                else
                        for m = i + 1 to j
                                min = dp[i][m] + dp[m+1][j] + dim[i-1]*dim[m]*dim[j];
                        dp[i][j] = min;
```

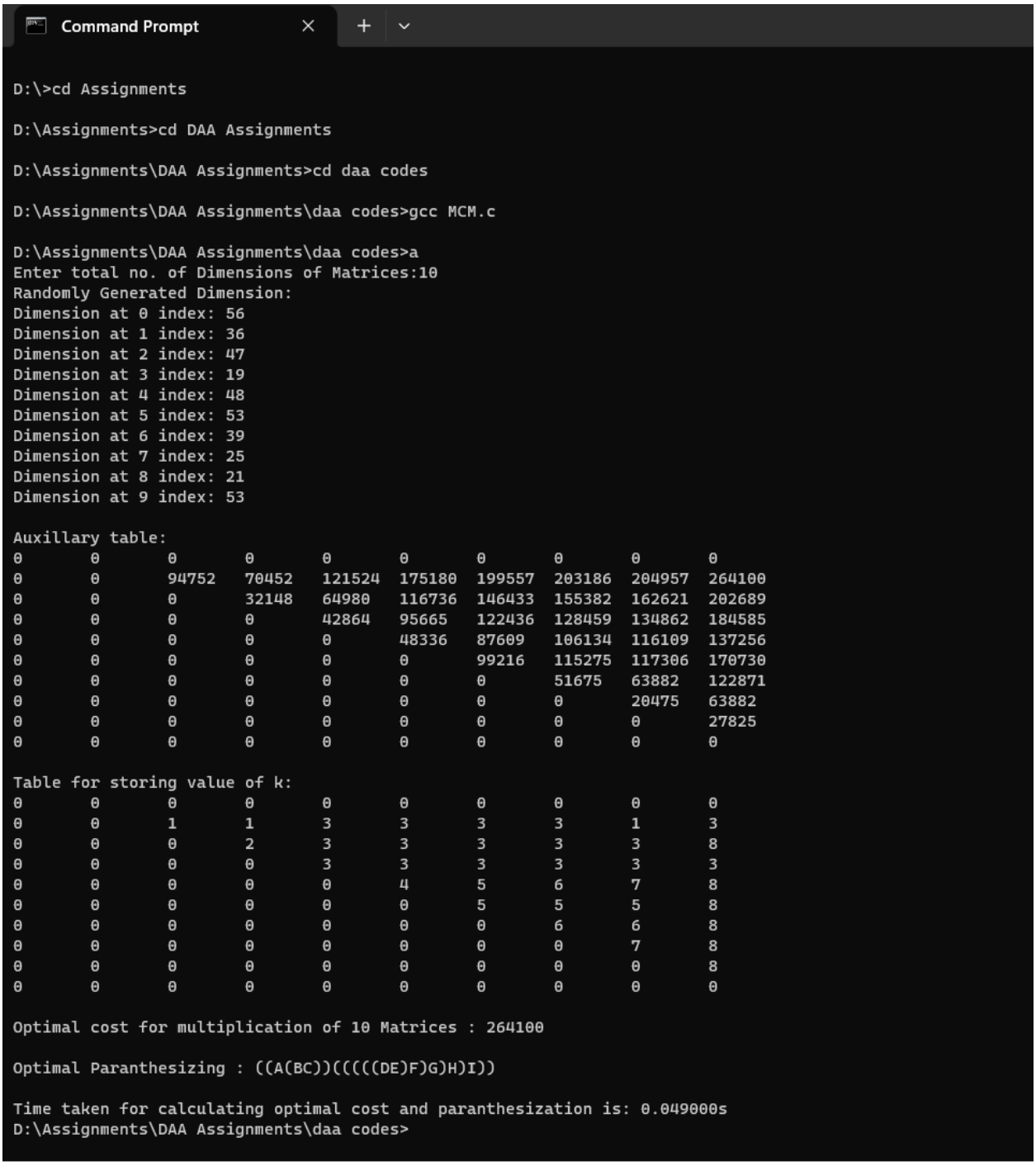
Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Print paranthesis recursive function that divides matrices sequence into two parts
void printParenthesis(int i, int j, int n, int paranthesis[][10], char *matrices)
{
   if(i < 0 || j > n) return;

   if(i == j) {
      printf("%c",matrices[i-1]);
      return;
   }

   printf("(");
   printParenthesis(i,paranthesis[i][j],n,paranthesis,matrices);
   printParenthesis(paranthesis[i][j] + 1,j,n,paranthesis,matrices);
   printf(")");

}
int main()
{
   int n; //n = 10
   printf("Enter total no. of Dimensions of Matrices:");
   scanf("%d",&n);
   int *dim = (int *)malloc(sizeof(int) * n);
   //Randomly generated number between 15 and 46
   printf("Randomly Generated Dimension:\n");
   for(int i = 0; i < n; i++) {
      dim[i] = rand() % 46 + 15;
      printf("Dimension at %d index: %d\n",i,dim[i]);
   }
   int dp[n][n], paranthesis[n][n];
   for(int i = 0 ; i < n ; i++) {
      for(int j = 0 ; j < n; j++) {
         dp[i][j] = 0;
         paranthesis[i][j] = 0;
      }
   }
   clock_t start, end;
```

```
double cpu_time_used;
start = clock();
int d = 0;
for(int k = 0; k < n - 1; k++) {
   d = d + 1;
   int j = d;
   for(int i = 1 ; i <= n - d; i++, j++) {
      if(i == j) {
         dp[i][j] = 0;
         paranthesis[i][j] = 0;
      }
      else {
         int m = i;
         int min = dp[i][m] + dp[m+1][j] + dim[i-1]*dim[m]*dim[j];
         paranthesis[i][j] = i;
         for(m = i + 1; m < j; m++) {
            if(min > dp[i][m] + dp[m+1][j] + dim[i-1]*dim[m]*dim[j]) {
               min = dp[i][m] + dp[m+1][j] + dim[i-1]*dim[m]*dim[j];
               paranthesis[i][j] = m;
            }
         }
         dp[i][j] = min;
      }
   }
}

printf("\nAuxillary table:\n");
for(int i = 0 ; i < n ; i++) {
   for(int j = 0 ; j < n; j++) {
      printf("%d\t",dp[i][j]);
   }
   printf("\n");
}
printf("\nTable for storing value of k:\n");
for(int i = 0 ; i < n ; i++) {
   for(int j = 0 ; j < n; j++) {
      printf("%d\t",paranthesis[i][j]);
   }
   printf("\n");
}
```

```
    int i = 1, j = n - 1;
    char matrices[9] = "ABCDEFGHI";
    printf("\nOptimal cost for multiplication of 10 Matrices : %d\n",dp[1][9]);
    printf("\nOptimal Paranthesizing : ");
    printParenthesis(i, j, n, paranthesis,matrices);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\n\nTime taken for calculating optimal cost and paranthesization is:
%fs",cpu_time_used);
    return 0;
}
```

Output :



Time Complexity : $O(n^3)$

CONCLUSION :

By performing this experiment, I understood the concept of matrix chain multiplication which is used to find an optimal cost and parenthesization order in which the matrix should be multiply. The time complexity of this algorithm is $O(n^3)$, where n is the number of matrices, but it can be reduced to $O(n^2)$ using memoization.