LPS-5 (MCM)

By:- Rachit Bhalla

21BAI1869

Q1.

In matrix chain multiplication problem, given the number of matrices to be multiplied, write a C++ program to find the number of ways it may be parenthesized. For example, when there are three matrices A1, A2 and A3 there are two ways to parenthesize them. (A1(A2A3)) and ((A1A2)A3). If there are four matrices A1, A2, A3 and A4 then there are five ways to parenthesize as shown (A1(A2(A3A4))), (A1((A2A3)A4)), ((A1,A2)(A3A4)), ((A1(A2A3))A4), (((A1A2)A3)A4).

Ans:

Problem Statement: To calculate the number of ways to parenthesize a chain of matrices using recursive function calls.

Logic:

The code uses a recursive approach to break down the problem into smaller subproblems. The number of ways to parenthesize the entire chain of matrices is then calculated by multiplying the number of ways on the left side by the number of ways on the right side for each potential split point k. The results are accumulated in the numWays variable.

Algorithm:

1.Import all the necessary libraries.

2.create a fxn matrix_chain where

if n<=2, ie. Only 1 matrix present return 1.

3.initialize numWays=0

4.iterate from i=0 to n and recursively call matrix_chain fxn for leftWays and RightWays.

5.Calculate numWays +=LeftWays *RightWays

6.return NumWays.

Pseudocode:

```
matrix_chain(int n)
    if (n <= 2)
        return 1;
   numWays = 0;
    for (int i = 1 to n){
        int leftWays = matrix_chain(i);
        int rightWays = matrix_chain (n - i);
        numWays += leftWays * rightWays;
}
    return numWays;
```

Proof of correctness:

Initialization: Before the first recursive call to matrix_chain, the code checks if n is less than or equal to 2. If this condition is satisfied, the function returns 1, which represents the correct number of ways to parenthesize the matrices in this base case. Therefore, the initialization step holds true, as the base case is correctly handled, providing the correct result for the smallest input values.

Maintenance: The recursive calls in the for loop ensure that the function is correctly invoked for smaller subproblems. For each split point i in the range 1 to n-1, the function recursively calculates the number of ways to parenthesize the matrices on the left side of i and the matrices on the right side of i. This recursive approach ensures that all possible combinations of parenthesizations are explored and considered when calculating the total number of ways.

Termination:

The termination condition is reached when n is less than or equal to 2, and the base case is triggered. In this case, the function immediately returns the correct value of 1, as there is only one way to parenthesize the matrices when there are 2 or fewer matrices. Hence it returns the numWays.

Time complexity:

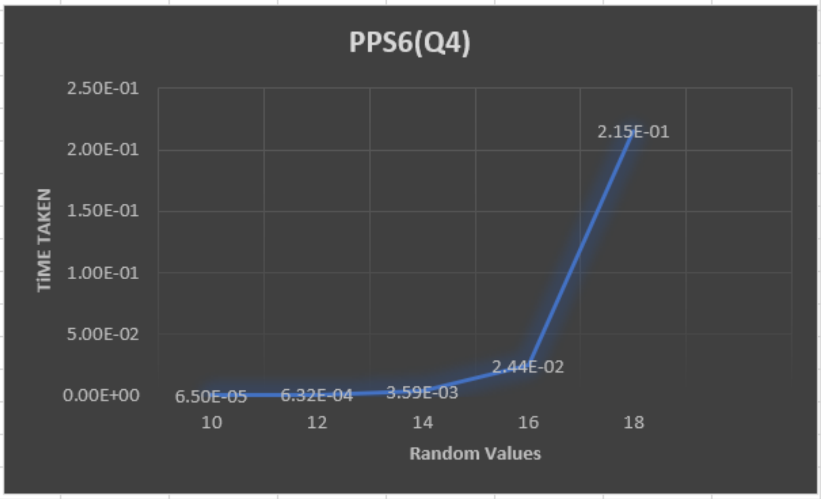
->The function matrix_chain is called recursively for each subproblem. The function has a loop that iterates from 1 to n-1. For each iteration, the function makes two recursive calls to matrix_chain with smaller subproblems of size i and n-i respectively.

->The time complexity of the function can be represented by the recurrence relation T(n) = T(1) + T(n-1) + T(2) + T(n-2) + ... + T(n-1) + T(1). The recurrence relation can be simplified to T(n) = 2T(n-1) + C, where C is a constant time taken by the loop.

->The time complexity of the function can be calculated using the master theorem, which gives a time complexity of O(2^n).

Graph:

Random n value	Time Taken
10	6.50E-05
12	6.32E-04
14	3.59E-03
16	2.44E-02
18	2.15E-01



Q2.

Develop a top down dynamic programming algorithm to find minimum cost for matrix chain multiplication. Print the tables maintained by the algorithm

Ans:

Problem Statement: Take the number of matrices and their dimensions as input, and as output, display the minimum cost required to multiply them, also print the table used to calculate the minimum cost.

Logic: It uses the top down Dynamic programming approach to calculate the minimum cost required to multiply 2 matrices.

Algorithm:

1. Import all the necessary libraries.

2. Create a table to store the minimum cost required to multiply each subchain of matrices.

3.Initialize the diagonal elements of the table to 0, since a single matrix requires no multiplication.

4.For each subchain length l, starting from 2 and up to n (the number of matrices), loop over all possible subchains of length l.

5.For each subchain (i, j) of length l, loop over all possible split points k within the subchain (i <= k < j).

6.Calculate the cost of multiplying the two subchains (i, k) and (k+1, j), and add it to the cost of multiplying the resulting matrices.

7.If this cost is less than the current minimum cost for subchain (i, j), update the minimum cost in the table.

8.After all subchains have been processed, the minimum cost for multiplying all matrices (1, n) will be stored in the top-right corner of the table.

9.Backtrack through the table to determine the optimal parenthesization of matrices that achieves this minimum cost.

10. Return the minimum cost.

Pseudocode:

```
void printTable(const vector<vector<int>>& table) {
    int n = table.size()
    for (int i = 1; i < n; i++)
        for (int j = 1; j < n; j++)
            if (i <= j)
                cout << table[i][j] << "\t";
            else
                cout << "-\t";
        cout << endl;
```

```
   int matrix_chain_mult(vector<pair<int, int>>& dimensions,
vector<vector<int>>& table, int i, int j) {
   if (i == j)
      return 0;
   if (table[i][j] != -1)
      return table[i][j];
   int min_cost = INT_MAX;
   for (int k = i; k < j; k++) {
      int cost = [matrix_chain (dimensions, table, i, k) +
              matrix_chain (dimensions, table, k + 1, j) +
              dimensions[i - 1].first * dimensions[k].second *
dimensions[j].second];
      if (cost < min_cost)
         min_cost = cost;
   table[i][j] = min_cost;
   return min_cost;
```

Proof of correctness:

Initialization: The table is initialized with -1 for all entries. The diagonal entries of the table are initialized to 0, since a single matrix requires no multiplication. This ensures that the base case is correctly handled when computing the minimum cost for multiplying subchains of matrices.

Maintenance:

The algorithm loops over all possible subchain lengths l, starting from 2 and up to n (the number of matrices). For each subchain (i, j) of length l, loop over all possible split points k within the subchain (i <= k < j). Calculate the cost of multiplying the two subchains (i, k) and (k+1, j), and add it to the cost of multiplying the resulting matrices. If this cost is less than the current minimum cost for subchain (i, j), update the minimum cost in the table. This ensures that for each subchain of matrices, the minimum cost required to multiply them is correctly computed and stored in the table.

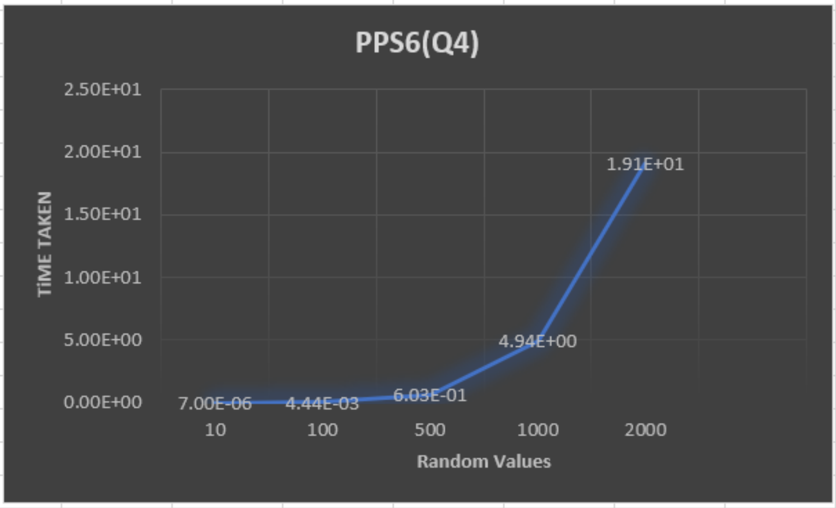
Termination:

After all subchains have been processed, the minimum cost for multiplying all matrices (1, n) will be stored in the top-right corner of the table. This value is returned as the final result.

Time complexity: The time complexity of the given code is O(n^3), where n is the number of matrices. This is because the algorithm uses dynamic programming to compute the minimum cost for multiplying each subchain of matrices, and there are O(n^2) subchains to consider. For each subchain, the algorithm loops over all possible split points, which adds an additional O(n) factor to the time complexity. Finally, computing the cost of multiplying two subchains requires constant time, so the overall time complexity is O(n^3).

Graph:

Random n value	Time Taken
10	7.00E-06
100	4.44E-03
500	6.03E-01
1000	4.94E+00
2000	1.91E+01



Q3.

MyBinomial coefficient is defined as follows:

C(n, 1) = 1

C(n, n) = 1

C(n, a) = 0 if a < 0 or n<0

If n is odd then C(n, k) = C(n-1, k-1) + C(n-1, k)

If n is even then C(n, k) = C(n-2, k-2) + C(n-2, k)

Given values of n and k, use dynamic programming to computeMyBinomial coefficient C(n,k).

Ans:

Problem Statement: Using DP, design and analyze an algorithm to find the modified binomial coefficient according to the given constraints.

Logic: Using memoization table(Dynamic Programming), we store the values obtained at each and every instance of binomial calculation to call back on them again when needed.Hence reducing the time taken and making our code efficient.

Algorithm:

1. import all the necessary libraries.

2. create a 2d matrix to store the values of coefficients calculated before.

3.iterate through rows and colums such that:

Btable[i][0] = 1;   Btable[i][i] = 1;

4.using nested loops satisfy the conditions of:

If n is odd then Btable[i][j] = Btable[i - 1][j - 1] + Btable[i - 1][j];

If n is even then Btable[i][j] = Btable[i - 2][j - 2] + Btable[i - 2][j];

5.return the value of Btable[n][k];

Pseudocode:

```
compute_binomial_coefficient(int n, int k) {
   // Create a table to store the computed values
   vector<vector<int>> Btable(n + 1, vector<int>(k + 1, 0));


   for (i=0 to n)
      Btable[i][0] = 1;




   for (i=1 to k)
      Btable[i][i] = 1;


  // using dynamic programming filling the table,
   for i=1 top n {
      for j = 1to min(i, k) {
         if (i % 2 == 1)
            Btable[i][j] = Btable[i - 1][j - 1] + Btable[i - 1][j];
         else
            Btable[i][j] = Btable[i - 2][j - 2] + Btable[i - 2][j];


   return Btable[n][k];
```

Proof of correctness:

Initialization: It means loop invariant holds true prior to first iteration, Inintially we have just initialized the memoization Table and made all its values as 0, we also specified base cases which will be true in all scenarios.

Maintainence:

This means showing that if first iteration holds true, all the succeeding iterations will hold true. Our code will fill the values in memoization table one by one and then recursively call back the obtained values to further compute the next values till we fill the whole memoization table.

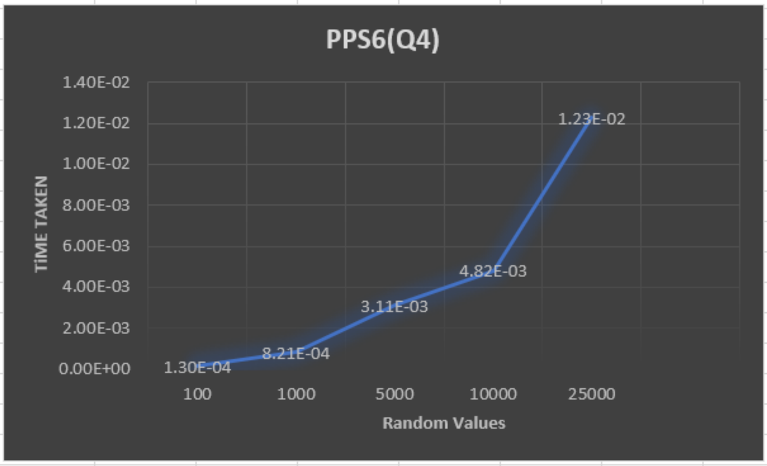
Termination: our loop will terminate when i<=n condition becomes false i.e when i becomes n+1 then we return value Btable[n][k].

Time Complexity: O[N^2]

->As the loop runs n*(n-1) times due to the nested loops being used to fill the Memoization table. Hence it is O[N^2].

Graph:

N value	Time Taken	k value
100	1.30E-04	20
1000	8.21E-04	20
5000	3.11E-03	20
10000	4.82E-03	20
25000	1.23E-02	20


PPS6(Q4)
TIME TAKEN
1.40E-02
1.20E-02
1.00E-02
8.00E-03
6.00E-03
4.00E-03
2.00E-03
0.00E+00
1.30E-04
8.21E-04
3.11E-03
4.82E-03
1.23E-02
Random Values