|  |  |
| --- | --- |
| **NAME** | **Ms. Neerja Doshi** |
| **UID** | **2021300029** |
| **DIV** | **CE - SE – A DIV (B BATCH)** |
| **DAA EXPT - 4** | |

|  |  |
| --- | --- |
| **AIM** | (Dynamic Programming -Matrix Chain Multiplication) |
| **THEORY** | 1. Dynamic programming is a method for solving optimization problems by breaking them down into smaller subproblems and solving each subproblem only once. 2. The key idea behind dynamic programming is to store the solutions to the subproblems in a table, so that they can be reused when needed. This is known as memoization, and it can significantly reduce the time complexity of an algorithm. 3. Matrix Chain Multiplication(MCM) is a problem in computer science that involves finding the most efficient way to multiply a series of matrices. The objective is to minimize the total number of scalar multiplications required to multiply the matrices together. 4. **ALGORITHM :**    1. **For Matrix Chain Multiplication:**        1. Start.       2. Take the range of matrices from the user, say from 𝐴𝑖 to 𝐴𝑗.       3. For i<=k<j, divide the provided range of matrices into two parts having matrices 𝐴𝑖 to 𝐴𝑘 and 𝐴𝑘+1 to 𝐴𝑗 respectively.       4. For each set of divisions, calculate the number of scalar products required using dynamic programming approach.       5. Store the minimum scalar products required in a table. Also, store the k value at which the minimum was obtained in a separate table.       6. End.    2. **For Parenthezation :**        1. Start.       2. Iterate over the k value table and recursively store the number of opening and closing brackets for each matrix.       3. Use the stored information while printing the final parenthesized expression.       4. End.    3. **Time Complexity :**        1. There could be O(n^2) unique sub-problems to any MCM given problem and for every such sub-problem there could be O(n) splits possible.       2. So it is O(n^3). |
| **CODE** | // matrix chain multiplication + parenthezation  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  // function for creation and destruction of 2D arrays  *int* \*\*createArr(*int* *row*, *int* *column*)  {  *int* \*\*arr = (*int* \*\*)calloc(*row*, sizeof(*int* \*));      for (*int* i = 0; i < *row*; i++)          arr[i] = (*int* \*)calloc(*column*, sizeof(*int*));      return arr;  }  *void* destroyArr(*int* \*\**arr*, *int* *row*)  {      for (*int* i = 0; i < *row*; i++)          free(*arr*[i]);      free(*arr*);  }  // function for randomly populating the dimension array  *int* \*generateDimensions(*int* *size*, *int* *startVal*, *int* *endVal*)  {  *int* \*dim = (*int* \*)malloc(*size* \* sizeof(*int*));      for (*int* i = 0; i < *size*; i++)          dim[i] = *startVal* + rand() % (*endVal* - *startVal* + 1);      return dim;  }  // functions for finding optimal number of scalar products and corresponsing k values  *void* matrixChainMul(*int* \**dim*, *int* \*\**optimalVal*, *int* \*\**kVal*, *int* *i*, *int* *j*)  {      if (*i* != *j* && *optimalVal*[*i*][*j*] == 0)      {  *int* tempVal = 0, optimal, kOpt = *i*, k = *i*;          optimal = *optimalVal*[*i*][k] + *optimalVal*[k + 1][*j*] + *dim*[*i* - 1] \* *dim*[k] \* *dim*[*j*];          k++;          while (k < *j*)          {              tempVal = *optimalVal*[*i*][k] + *optimalVal*[k + 1][*j*] + *dim*[*i* - 1] \* *dim*[k] \* *dim*[*j*];              if (tempVal < optimal)              {                  optimal = tempVal;                  kOpt = k;              }              k++;          }  *optimalVal*[*i*][*j*] = optimal;  *kVal*[*i*][*j*] = kOpt;      }  }  *void* fillOptimalSolution(*int* \**dim*, *int* \*\**optimalVal*, *int* \*\**kVal*, *int* *numOfMat*)  {  *int* offset;      for (*int* d = *numOfMat* - 1; d > 0; d--)      {          offset = *numOfMat* - d;          for (*int* i = 1; i <= d; i++)          {              matrixChainMul(*dim*, *optimalVal*, *kVal*, i, i + offset);          }      }  }  // function for printing the required tables  *void* printTab(*int* \*\**table*, *int* *size*)  {      printf("\t");      for (*int* i = 1; i < *size*; i++)      {          printf("%d\t", i);      }      printf("\n");      for (*int* i = 0; i < *size*; i++)      {          printf("--------");      }      printf("\n");      for (*int* i = 1; i < *size*; i++)      {          printf("%d\t", i);          for (*int* j = 1; j < *size*; j++)          {              if (*table*[i][j] == 0)                  printf("-\t");              else                  printf("%d\t", *table*[i][j]);          }          printf("\n");      }  }  // functions for determining parenthesization  *void* findParenthesisInfo(*int* \*\**parenthesis*, *int* \*\**kVal*, *int* *i*, *int* *j*)  {  *int* k = *kVal*[*i*][*j*];      if (*j* - *i* + 1 > 2)      {          if (k - *i* + 1 > 1)          {  *parenthesis*[*i*][0]++;  *parenthesis*[k][1]++;              findParenthesisInfo(*parenthesis*, *kVal*, *i*, k);          }          if (*j* - k > 1)          {  *parenthesis*[k + 1][0]++;  *parenthesis*[*j*][1]++;              findParenthesisInfo(*parenthesis*, *kVal*, k + 1, *j*);          }      }  }  *void* printMatMulExp(*int* \*\**parenthesis*, *int* *numOfMat*)  {      for (*int* i = 1; i <= *numOfMat*; i++)      {          for (*int* j = 0; j < *parenthesis*[i][0]; j++)          {              printf("(");          }          printf("M%d", i);          for (*int* j = 0; j < *parenthesis*[i][1]; j++)          {              printf(")");          }      }  }  // function to calculate the number of scalar products under trivial matrix multiplication  *int* trivialMatMul(*int* \**dim*, *int* *numOfMat*)  {  *int* sum = 0;      for (*int* i = 1; i <= *numOfMat* - 1; i++)          sum += *dim*[0] \* *dim*[i] \* *dim*[i + 1];      return sum;  }  // main function  *int* main()  {      srand(time(0));      // taking user input  *int* num;      printf("\nEnter the number of matrices that you want to multiply : ");      scanf("%d", &num);      // displaying the input configuration the program will be dealing with  *int* \*dim = generateDimensions(num + 1, 15, 46);      printf("\nThe following dimension matrix was randomly generated having values between 15 and 46 -\n ");      for (*int* i = 0; i <= num; i++)          printf(" % d\t ", dim[i]);      printf("\n\nThat is, the following matrices are taken into consideration -\n\n ");      for (*int* i = 1; i <= num; i++)          printf(" M % d - order(% d x % d)\n ", i, dim[i - 1], dim[i]);      printf("\n ");      // calculating the optimal multiplication order using dynamic programming approach  *int* \*\*optimalVal = createArr(num + 1, num + 1);  *int* \*\*kVal = createArr(num + 1, num + 1);      fillOptimalSolution(dim, optimalVal, kVal, num);      // displaying the results      printf("Following tabular data was obtained-\n\n");      printf("I. Table showing the optimal number of multiplications required at each step : \n\n");      printTab(optimalVal, num + 1);      printf("\nII. Table showing the k values at which optimal solution was obtained at each step-\n\n");      printTab(kVal, num + 1);      printf("\nOptimal Parenthesization is as follows-\n\n");  *int* \*\*parenthesis = createArr(num + 1, 2);      findParenthesisInfo(parenthesis, kVal, 1, num);      printMatMulExp(parenthesis, num);      printf("\n\n");      printf("Summary-\n\n");  *int* sum = trivialMatMul(dim, num);      printf("Number of scalar products required under trivial matrix chain multiplication:  %d\n", sum);      printf("Number of scalar products required under optimal matrix chain multiplication:  %d\n", optimalVal[1][num]);      printf("Hence, optimal solution is %.2lf times faster than the trivial solution\n\n", (*double*)sum / optimalVal[1][num]);      // de-allocating all the used locations      destroyArr(optimalVal, num + 1);      destroyArr(kVal, num + 1);      destroyArr(parenthesis, num + 1);      free(dim);      return 0 ;  } |
| **OUTPUT** |  |
| **CONCLUSION** | By performing the above experiment , Ive succefully understood coding Matrix Chain Multiplicatioon and its Algorithm.  I observed that optimal order of multiplying a chain of matrices is a crucial factor in reducing the time an algorithm takes to multiply matrices |