|  |  |
| --- | --- |
| **NAME:** | Akshaya Karande |
| **UID:** | 2021300056 |
| **SUBJECT** | Design and Analysis of Algorithms |
| **EXPERIMENT NO:** | 4 |
| **AIM:** | To implement Matrix Chain Multiplication |
| **Theory:** | **Matrix chain multiplication**(or the **matrix chain ordering problem**) is an [optimization problem](https://en.wikipedia.org/wiki/Optimization_problem) concerning the most efficient way to [multiply](https://en.wikipedia.org/wiki/Matrix_multiplication) a given sequence of [matrices](https://en.wikipedia.org/wiki/Matrix_(mathematics)). The problem is not actually to *perform* the multiplications, but merely to decide the sequence of the matrix multiplications involved. The problem may be solved using [dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming).  There are many options because matrix multiplication is [associative](https://en.wikipedia.org/wiki/Associativity). In other words, no matter how the product is [parenthesized](https://en.wikipedia.org/wiki/Bracket_(mathematics)), the result obtained will remain the same. For example, for four matrices *A*, *B*, *C*, and *D*, there are five possible options:  ((*AB*)*C*)*D* = (*A*(*BC*))*D* = (*AB*)(*CD*) = *A*((*BC*)*D*) = *A*(*B*(*CD*)).  Although it does not affect the product, the order in which the terms are parenthesized affects the number of simple arithmetic operations needed to compute the product, that is, the [computational complexity](https://en.wikipedia.org/wiki/Computational_complexity). The straightforward multiplication of a matrix that is *X* × *Y* by a matrix that is *Y* × *Z* requires *XYZ* ordinary multiplications and *X*(*Y* − 1)*Z* ordinary additions. In this context, it is typical to use the number of ordinary multiplications as a measure of the runtime complexity.  If *A* is a 10 × 30 matrix, *B* is a 30 × 5 matrix, and *C* is a 5 × 60 matrix, then computing (*AB*)*C* needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while computing *A*(*BC*) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.  Clearly the first method is more efficient. With this information, the problem statement can be refined as "how to determine the optimal parenthesization of a product of *n* matrices?" Checking each possible parenthesization ([brute force](https://en.wikipedia.org/wiki/Brute-force_search)) would require a [run-time](https://en.wikipedia.org/wiki/Time_complexity) that is exponential in the number of matrices, which is very slow and impractical for large *n*. A quicker solution to this problem can be achieved by breaking up the problem into a set of related subproblems. |
| **Algorithm:** | **MATRIX-CHAIN-ORDER (p)**  1. n length[p]-1  2. for i ← 1 to n  3. do m [i, i] ← 0  4. for l ← 2 to n // l is the chain length  5. do for i ← 1 to n-l + 1  6. do j ← i+ l -1  7. m[i,j] ← ∞  8. for k ← i to j-1  9. do q ← m [i, k] + m [k + 1, j] + pi-1 pk pj  10.If q < m [i,j]  11.then m [i,j] ← q  12.s [i,j] ← k  13.return m and s.  **PRINT-OPTIMAL-PARENS (s, i, j)**  1. if i=j  2. then print "A"  3. else print "("  4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])  5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)  6. print ")" |
| **Code:** | #include <iostream>  #include <climits>  #include <random>  #include <ctime>  using namespace std;  void matrixChainOrder(int p[], int n, int m[][100], int s[][100])  {  for(int i=1; i<=n; i++)  m[i][i] = 0; for(int l=2; l<=n; l++)  {  for(int i=1; i<=n-l+1; i++)  { int j = i+l-1;  m[i][j] = INT\_MAX;  for(int k=i; k<=j-1; k++)  {  int q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j]; if(q < m[i][j])  {  m[i][j] = q;  s[i][j] = k;  }  }  }    }  }  void printOptimalParenthesis(int s[][100], int i, int j)  {  if(i == j)  cout << "A" << i;  else  {  cout << "("; printOptimalParenthesis(s, i, s[i][j]); printOptimalParenthesis(s, s[i][j]+1, j); cout << ")";  }  }  int main()  {  int p[8];  srand ( time(NULL) );  random\_device rd;  mt19937 gen(rd());  uniform\_int\_distribution<> distr(15, 46);  for(int i=0; i<10; ++i)  p[i] = distr(gen);  int n = sizeof(p)/sizeof(p[0]) - 1;    int m[100][100];  int s[100][100];  matrixChainOrder(p, n, m, s);  cout << "\nOptimal Parenthesization: "; printOptimalParenthesis(s, 1, n);  cout << endl;  cout << "\nMinimum Number of Scalar Multiplications: " << m[1][n] << endl;  cout << "\n\nm table:";  for(int a = 0; a < 8; a++)  {  for(int b = 0; b < 8; b++)  {  if(m[a][b] == 0){continue;}  cout << m[a][b] << " ";    }  cout << endl;  }  cout << "\n\ns table:";  for(int a = 0; a < 10; a++)  {  for(int b = 0; b < 10; b++)  {  if(s[a][b] == 0){continue;}  cout << s[a][b] << " ";  }  cout << endl;  }  return 0;  } |
| **Output** |  |
| **Conclusion:** | By applying dynamic programming techniques to matrix chain multi-plication, we can solve the problem in polynomial time instead of exponential time. This has important implications for a wide range of applications, from computer graphics and image processing to scientific simulations and machine learning. |