VIT

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Ex.No: 5
Date : 05.09.2024

Implementation of matrix-chain multiplication algorithm with experiments on analysis and efficiency

Name : Venkatesan M
Reg.No : 22BAI1259

Aim

Given the dimension of a sequence of matrices in an array arr[], where the dimension of the ith matrix is (arr[i-1] * arr[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.

When 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.

Algorithm

Matrix-Chain Multiplication (Dynamic Programming Approach)

1. **Initialization:**

   Let the chain of matrices be A1,A2,…,An

   Define a table `m` where `m[i][j]` represents the minimum number of scalar multiplications needed to multiply matrices Ai to Aj

   Let `s` be the table to store the index k at which the optimal split occurs.

2. **Compute Cost**:

   For each chain length l from 2 to n:

   For each starting index i from 1 to n−l+1:

   Set the ending index j=i+l−1.

   Initialize `m[i][j]` to infinity.

   For each possible split point k from gi to j−1:

   Compute the cost of multiplying matrices Ai to Ak and Ak+1 to Aj, plus the cost of multiplying the two resulting matrices.

   Update `m[i][j]` if a lower cost is found.

   Store the split point in `s[i][j]`.

3. **Parenthesization**:

   Use the table `s` to determine the optimal parenthesization of the matrices.

4. **Output**:

   The minimum number of scalar multiplications is stored in `m[1][n]`.

   The optimal parenthesization is obtained using the `s` table.

Example

Matrix-chain multiplication problem Example

- Given: n = 3, A_1 (10x100), A_2: (100x5), A_3: (5x50)

  i.e. Given n=3, and $p_0 = 10, p_1 = 100, p_2 = 5, p_3 = 50$

- To compute: $A_1 A_2 A_3$
- Option 1: $(A_1 A_2) A_3$
- Option 2: $A_1 (A_2 A_3)$
- Would the result be the same?

Matrix-chain multiplication problem Example

- Given: n = 3, A_1 (10x100), A_2: (100x5), A_3: (5x50) *$p_0 = 10, p_1 = 100, p_2 = 5, p_3 = 50$*
- To compute: $A_1 A_2 A_3$

- Option 1: $(A_1 A_2) A_3$
  - Total multiplications = $(A_1 A_2)$ + A + $(A_1 A_2) A_3$
  - $(A_1 A_2)$: multiplications = 10 x 100 x 5 = 5000
  - 10 x 5 resulting matrix
  - $(A_1 A_2) A_3$: multiplications = 10 x 5 x 50 = 2500

    = 5000 + 0 + 2500 = 7,500

- Option 2: $A_1 (A_2 A_3)$
  - $(A_2 A_3)$: multiplications = 100 x 5 x 50 = 25,000
  - $(A_2 A_3)$: 100 x 50 resulting matrix
  - $A_1 (A_2 A_3)$: multiplications = 10 x 100 x 50 = 50,000
  - Total multiplications = 25,000 + 50,000 = 75,000

- Hence Option 1 is 10 times faster than Option 2!

MATRIX-CHAIN-ORDER(p)

```
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n                ▷ l is the chain length.
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              m[i, j] ← ∞
              for k ← i to j − 1
                  do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                     if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
    then print "A"_i
    else print "("
         PRINT-OPTIMAL-PARENS(s, i, s[i, j])
         PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
         print ")"
```

C++ Implementation

```cpp
#include <iostream>

#include <limits.h>

#include <vector>

#include <chrono>


using namespace std;


// Function to print optimal parenthesization

void printOptimalParens(vector<vector<int>>& s, int i, int j) {

  if (i == j)

    cout << "A" << i;

  else {

    cout << "(";

    printOptimalParens(s, i, s[i][j]);

    printOptimalParens(s, s[i][j] + 1, j);

    cout << ")";

  }

}
```

```
// Matrix Chain Multiplication using Dynamic Programming
int matrixChainOrder(vector<int>& p, int n) {
    vector<vector<int>> m(n, vector<int>(n, 0));
    vector<vector<int>> s(n, vector<int>(n, 0));

    for (int l = 2; l < n; l++) { // l is the chain length
        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; 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;
                }
            }
        }
    }

    cout << "Optimal Parenthesization: ";
    printOptimalParens(s, 1, n - 1);
    cout << "\nMinimum number of multiplications is " << m[1][n - 1] << endl;

    return m[1][n - 1];
}

int main() {
    vector<int> p = {40, 20, 30, 10, 30};
    int n = p.size();
```

```
    auto start = chrono::high_resolution_clock::now();

    int minMultiplications = matrixChainOrder(p, n);

    auto end = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    cout << "Time taken by function: " << duration.count() << " microseconds" << endl;
    return 0;
}
```

Input

Array P = {40, 20, 30, 10, 30};

Size N = 5

Output

Optimal Parenthesization: ((A1(A2A3))A4)

Minimum number of multiplications is 26000

Time taken by function: 65 microseconds

Random Sampling

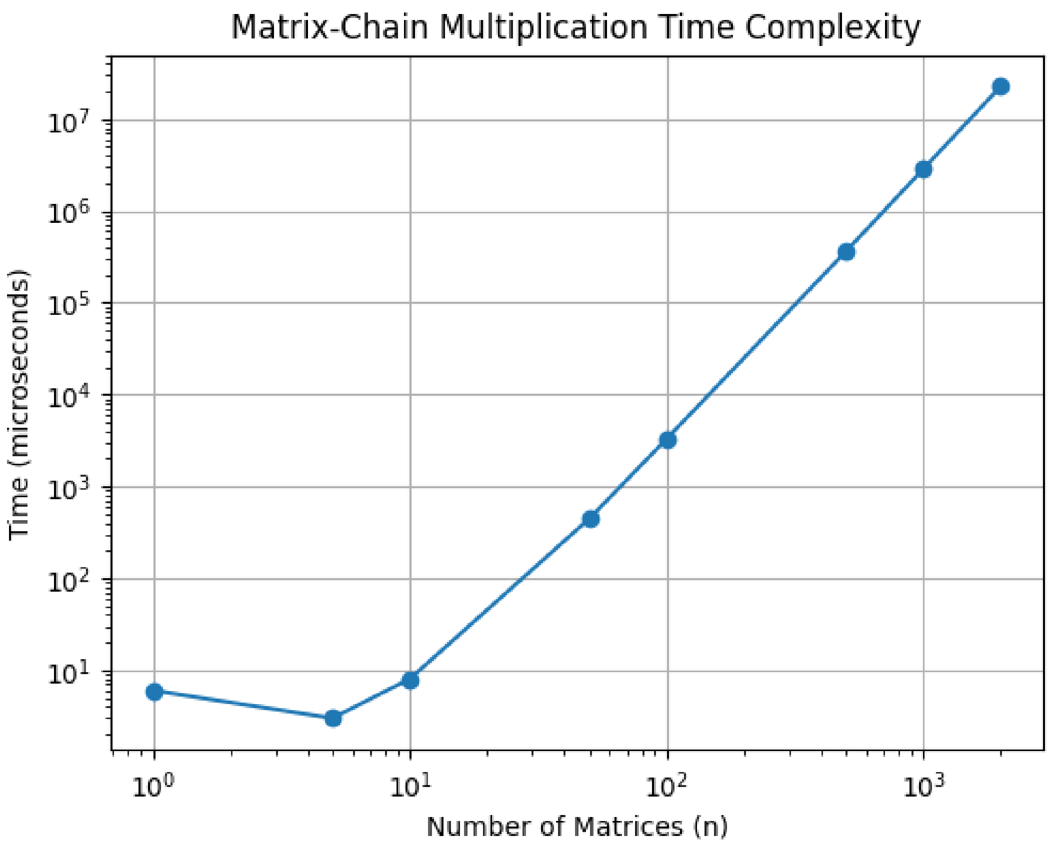
```
    vector<int> sizes = {1,5, 10,50, 100,500, 1000, 2000}; // Different sizes of input
    vector<long long> times;
    for (int n : sizes) {
        // Generate random matrix dimensions
        vector<int> p(n + 1);
        for (int i = 0; i <= n; ++i) {
            p[i] = rand() % 100 + 1; // Random dimensions between 1 and 100
        }
```

```
    auto start = chrono::high_resolution_clock::now();
    int minMultiplications = matrixChainOrder(p, n + 1); // n+1 as p has n+1 elements
    auto end = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    times.push_back(duration.count());

    cout << "Input size: " << n << " -> Time: " << duration.count() << " microseconds" << endl;
}
```

Time Complexity Analysis

Input	Time (Microseconds)
1	6
5	3
10	8
50	445
100	3291
500	363912
1000	2870975
2000	22679698

Graph between varying size of inputs and time



Complexity Analysis:

- **Time Complexity:** The time complexity of the matrix-chain multiplication algorithm is $O\left(n^{3}\right)$, which explains the exponential increase in execution time as observed in the plot.
- **Space Complexity:** The space complexity is $O\left(n^{2}\right)$, reflecting the storage required for the dynamic programming tables used in the computation.