Ahsania Mission University of Science & Technology

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional

Experiment No: 06

Experiment Date: 30.04.25

Submitted By:

Mehrin Nusrat Chowdhury

Roll: 1012320005101026

1st Batch, 2nd Year, 1st Semester

Department of Computer Science and Engineering

Ahsania Mission University of Science & Technology

Submitted To:

Md. Fahim Faisal

Lecturer

Department of Computer Science and Engineering

Faculty of Engineering, Ahsania Mission University of Science & Technology

Submission Date: 07.05.25

Task 01: A C++ program that will implement matrix multiplication using the Naive approach.

Theory:

Matrix multiplication is a fundamental operation in linear algebra and computer science, frequently used in graphics processing, scientific computing, and machine learning. The naive approach to matrix multiplication follows a straightforward triple nested loop structure, iterating through rows of the first matrix and columns of the second matrix to compute the dot product. Given two matrices **A** of size *(m × n)* and **B** of size *(n × p)*, the resulting matrix **C** of size *(m × p)* is obtained by multiplying corresponding elements and summing them. This method follows **O(m × n × p) = O(n^3)** time complexity, making it computationally intensive for large matrices. Despite its simplicity, this approach lays the groundwork for optimized algorithms such as Strassen's multiplication and parallel computing techniques.

Code:

```
#include<iostream>
using namespace std;

int main()
{
   int m , n , p;
   cout << "Enter rows and columns of matrix A: ";
   cin >> m >> n;
   cout << "Enter columns of matrix B: ";
   cin >> p;

   int A[m][n] , B[n][p] , C[m][p] = {0};

   int multiplicationCount = 0;
   int additionCount = 0;

   cout << "Mat A: ";
   for(int i = 0 ; i < m ; i++)
   {
      for(int j = 0 ; j < n ; j++)
      {
         cin >> A[i][j];
      }
   }
```

```
    cout << "Mat B: ";
    for(int i = 0 ; i < n ; i++)
    {
        for(int j = 0 ; j < p ; j++)
        {
            cin >> B[i][j];
        }
    }

    for(int i = 0 ; i < m ; i++)
    {
        for(int j = 0 ; j < p ; j++)
        {
            C[i][j] = 0;
            for(int k = 0 ; k < n ; k++)
            {
                C[i][j] = C[i][j] + (A[i][k] * B[k][j]);

                multiplicationCount++;

                if(k > 0)
                {
                    additionCount++;
                }
            }
        }
    }

    cout << "\nMatrix multiplication of A and B: ";
    for(int i = 0 ; i < m ; i++)
    {
        for(int j = 0 ; j < p ; j++)
        {
            cout << C[i][j] << " ";
        }
    }

    cout << "\n";

    cout << "Total multiplication operations: " << multiplicationCount << "\n";
    cout << "Total addition operations: " << additionCount << "\n";

    return 0;
}
```

Output:

```
Enter rows and columns of matrix A: 2 3
Enter columns of matrix B: 2
Mat A: 1 2 3
       4 5 6
Mat B: 7 8
       9 10
       11 12

Matrix multiplication of A and B: 58 64 139 154
Total multiplication operations: 12
Total addition operations: 8

Process returned 0 (0x0)   execution time : 36.998 s
Press any key to continue.
```

Conclusion:

In this lab, we implemented matrix multiplication using the naive approach, demonstrating the fundamental process of combining two matrices into a single resultant matrix. The program successfully computes and tracks the number of multiplication and addition operations, highlighting the computational workload involved. While effective for small matrices, the naive method becomes inefficient for large-scale data, emphasizing the importance of optimized algorithms in real-world applications.

Task 02: A C++ program that will implement matrix multiplication using the Divide-and-Conquer approach.

Theory:

Matrix multiplication is a critical operation in various fields such as engineering, data science, and computer graphics. The **Divide-and-Conquer** approach optimizes matrix multiplication by breaking larger matrices into smaller submatrices, solving them recursively, and then combining the results. Unlike the naive approach, which runs in $\mathbf{O(n^3)}$ time complexity, this method reduces operations by dividing the problem into smaller, manageable parts. The standard implementation partitions matrices into four quadrants and computes their products recursively, reducing computational overhead and improving efficiency. While this method enhances performance for larger matrices, further optimizations, such as Strassen's algorithm, further improve speed by reducing the number of multiplications required.

Code:

```
#include<iostream>
#include<vector>
using namespace std;
typedef vector<vector<int>> Matrix;

Matrix add(const Matrix &A , const Matrix &B)
{
   int n = A.size();
   Matrix C(n , vector<int>(n));
   for(int i = 0 ; i < n ; i++)
   {
      for(int j = 0 ; j < n ; j++)
      {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
   return C;
}

Matrix subtract(const Matrix &A , const Matrix &B)
{
   int n = A.size();
   Matrix C(n , vector<int>(n));
   for(int i = 0 ; i < n ; i++)
```

```
    {
        for(int j = 0 ; j < n ; j++)
        {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}

Matrix multiply(const Matrix &A , const Matrix &B)
{
    int n = A.size();
    Matrix C(n , vector<int>(n , 0));
    if(n == 1)
    {
        C[0][0] = A[0][0] * B[0][0];
    }
    else
    {
        int k = n / 2;
        Matrix A11(k , vector<int>(k)) , A12(k , vector<int>(k)) , A21(k , vector<int>(k)) ,
A22(k , vector<int>(k));
        Matrix B11(k , vector<int>(k)) , B12(k , vector<int>(k)) , B21(k , vector<int>(k)) ,
B22(k , vector<int>(k));

        for(int i = 0 ; i < k ; i++)
        {
            for(int j = 0 ; j < k ; j++)
            {
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j + k];
                A21[i][j] = A[i + k][j];
                A22[i][j] = A[i + k][j + k];

                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j + k];
                B21[i][j] = B[i + k][j];
                B22[i][j] = B[i + k][j + k];
            }
        }
        Matrix C11 = add(multiply(A11 , B11) , multiply(A12 , B21));
        Matrix C12 = add(multiply(A11 , B12) , multiply(A12 , B22));
        Matrix C21 = add(multiply(A21 , B11) , multiply(A22 , B21));
        Matrix C22 = add(multiply(A21 , B12) , multiply(A22 , B22));

        for(int i = 0 ; i < k ; i++)
```

```
        {
            for(int j = 0 ; j < k ; j++)
            {
                C[i][j] = C11[i][j];
                C[i][j + k] = C12[i][j];
                C[i + k][j] = C21[i][j];
                C[i + k][j + k] = C22[i][j];
            }
        }
    }
    return C;
}
void printMatrix(const Matrix &M)
{
    for(const auto &row : M)
    {
        for(int num : row)
        {
            cout << num << "\t";
        }
        cout << "\n";
    }
}

int main()
{
    int n;
    cout << "Enter size: ";
    cin >> n;

    cout << "\n";
    Matrix A(n , vector<int>(n)) , B(n , vector<int>(n));
    cout << "Mat A: ";
    for(int i = 0 ; i < n ; i++)
    {
        for(int j = 0 ; j < n ; j++)
        {
            cin >> A[i][j];
        }
    }
    cout << "\n";
    cout << "Mat B: ";
    for(int i = 0 ; i < n ; i++)
    {
        for(int j = 0 ; j < n ; j++)
        {
```

```
            cin >> B[i][j];
        }
    }
    Matrix C_add = add(A , B);
    Matrix C_subtract = subtract(A , B);
    Matrix C_multiply = multiply(A , B);

    cout << "\nMultiplication Result:\n\n";
    printMatrix(C_multiply);
    return 0;
}
```

Output:

```
Enter size: 4

Mat A: 1 5 3 4
       0 6 2 8
       7 0 1 2
       5 4 5 0

Mat B: 3 7 0 1
       6 0 2 1
       0 2 4 3
       1 9 0 9

Multiplication Result:

37      49      22      51
44      76      20      84
23      69      4       28
39      45      28      24

Process returned 0 (0x0)   execution time : 138.344 s
Press any key to continue.
```

Conclusion:

This lab experiment demonstrated matrix multiplication using the **Divide-and-Conquer** approach. By recursively breaking matrices into submatrices and computing smaller multiplications, we achieved an efficient multiplication process. This technique offers advantages over the naive approach, particularly for large datasets, by minimizing redundant operations and optimizing memory usage. While powerful, this method highlights the importance of balancing recursion depth and computational efficiency, paving the way for more advanced matrix multiplication algorithms.

Task 03: A C++ program that will implement matrix multiplication using Strassen's Matrix Multiplication algorithm.

Theory:

Strassen's Matrix Multiplication algorithm is an advanced technique that optimizes matrix multiplication by reducing the number of required multiplications compared to the conventional method. Introduced by Volker Strassen in 1969, this approach decomposes matrices into submatrices and applies recursive calculations using only **seven** multiplications instead of the standard **eight**, which improves computational efficiency. By utilizing **addition and subtraction** operations strategically, Strassen's method achieves a **time complexity of O(n2.81)**, outperforming the naive **O(n^3)** approach for large matrices. This algorithm is widely used in image processing, cryptography, and scientific computing due to its ability to handle massive datasets efficiently.

Code:

```
#include<iostream>
#include<vector>
using namespace std;
typedef vector<vector<int>> Matrix;

Matrix add(const Matrix &A , const Matrix &B)
{
   int n = A.size();
   Matrix C(n , vector<int>(n));
   for(int i = 0 ; i < n ; i++)
   {
      for(int j = 0 ; j < n ; j++)
      {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
   return C;
}

Matrix subtract(const Matrix &A , const Matrix &B)
{
   int n = A.size();
   Matrix C(n , vector<int>(n));
   for(int i = 0 ; i < n ; i++)
```

```
    {
        for(int j = 0 ; j < n ; j++)
        {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}

Matrix strassen(const Matrix &A , const Matrix &B)
{
    int n = A.size();
    Matrix C(n , vector<int>(n , 0));

    if(n == 1)
    {
        C[0][0] = A[0][0] * B[0][0];
    }
    else
    {
        int k = n / 2;
        Matrix A11(k , vector<int>(k)) , A12(k , vector<int>(k)) , A21(k , vector<int>(k)) ,
A22(k , vector<int>(k));
        Matrix B11(k , vector<int>(k)) , B12(k , vector<int>(k)) , B21(k , vector<int>(k)) ,
B22(k , vector<int>(k));

        for(int i = 0 ; i < k ; i++)
        {
            for(int j = 0 ; j < k ; j++)
            {
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j + k];
                A21[i][j] = A[i + k][j];
                A22[i][j] = A[i + k][j + k];

                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j + k];
                B21[i][j] = B[i + k][j];
                B22[i][j] = B[i + k][j + k];
            }
        }

        Matrix M1 = strassen(A11 , subtract(B12 , B22));
        Matrix M2 = strassen(add(A11 , A12) , B22);
        Matrix M3 = strassen(add(A21 , A22) , B11);
        Matrix M4 = strassen(A22 , subtract(B21 , B11));
```

```
        Matrix M5 = strassen(add(A11 , A22) , add(B11 , B22));
        Matrix M6 = strassen(subtract(A12 , A22) , add(B21 , B22));
        Matrix M7 = strassen(subtract(A11 , A21) , add(B11 , B12));

        Matrix C11 = subtract(add(add(M4 , M5) , M6) , M2);
        Matrix C12 = add(M1 , M2);
        Matrix C21 = add(M3 , M4);
        Matrix C22 = subtract(add(subtract(M1 , M3) , M5) , M7);

        for(int i = 0 ; i < k ; i++)
        {
            for(int j = 0 ; j < k ; j++)
            {
                C[i][j] = C11[i][j];
                C[i][j + k] = C12[i][j];
                C[i + k][j] = C21[i][j];
                C[i + k][j + k] = C22[i][j];
            }
        }
    }
    return C;
}

void printMatrix(const Matrix &M)
{
    for(const auto &row : M)
    {
        for(int num : row)
        {
            cout << num << "\t";
        }
        cout << "\n";
    }
}

int main()
{
    int n;
    cout << "Enter size: ";
    cin >> n;

    cout << "\n";
    Matrix A(n , vector<int>(n)) , B(n , vector<int>(n));

    cout << "Mat A: ";
    for(int i = 0 ; i < n ; i++)
```

```
    {
        for(int j = 0 ; j < n ; j++)
        {
            cin >> A[i][j];
        }
    }

    cout << "\n";
    cout << "Mat B: ";
    for(int i = 0 ; i < n ; i++)
    {
        for(int j = 0 ; j < n ; j++)
        {
            cin >> B[i][j];
        }
    }
    Matrix C_add = add(A , B);
    Matrix C_subtract = subtract(A , B);
    Matrix C_strassen_multiplication = strassen(A , B);

    cout << "\nMultiplication Result Using Strassen's Idea:\n\n";
    printMatrix(C_strassen_multiplication);
    return 0;
}
```

Output:

```
Enter size: 4

Mat A: 1 2 3 4
       5 6 7 8
       9 0 1 2
       3 4 5 6

Mat B: 3 7 0 1
       6 3 2 1
       1 2 4 5
       8 9 0 9

Multiplication Result Using Strassen's Idea:

50      55      16      54
122     139     40      118
44      83      4       32
86      97      28      86

Process returned 0 (0x0)   execution time : 140.498 s
Press any key to continue.
```

Conclusion:

This lab experiment successfully implemented Strassen's Matrix Multiplication, demonstrating its ability to optimize matrix computations. By reducing the number of multiplications through recursive submatrix decomposition, this method proves effective in handling large matrices efficiently. While powerful, Strassen's algorithm comes with increased complexity in implementation and memory usage, highlighting the trade-off between computational speed and resource consumption. Understanding and applying such optimized algorithms is crucial in modern computing applications where performance matters.