Batch:T6

Practical No.4

Title of Assignment: Divide and conquer strategy Strassen's Matrix Multiplication

Student Name: Parshwa Herwade

Student PRN: 22510064

1) Implement Naive method multiply two matrices and Justify complexity is o(n3)

Ans:

Matrix Multiplication Function:

multiplyMatrices: This function takes two matrices A and B, multiplies them using three nested loops, and returns the resulting matrix C.

- The time complexity of this function is O(n·m·p), where n is the number of rows in A, mmm is the number of columns in A (and rows in B), and p is the number of columns in B. For square matrices where n=m=p, this complexity simplifies to O(n3).

Input and Output:

- The program prompts the user to enter matrix dimensions and elements.
- It then computes the matrix multiplication and prints the resulting matrix.

Complexity Justification

The naive matrix multiplication algorithm has a time complexity of O(n3)

when both matrices are n×n. This is because:

- We have three nested loops: the outer loop runs n times (for each row of the resulting matrix), the middle loop runs n times (for each column of the resulting matrix), and the innermost loop runs n times (to compute the dot product for each element).

Thus, the total number of operations is proportional to n·n·n=, justifying the O(n3) complexity for square matrices.

Time Complexity:

- Best Case: O(n3)
- Worst Case: O(n3)
- Average Case: O(n3)
- Space Complexity: O(n.m+m.p)

Psedocode:
Matrix Multiplication

1. Function multiplyMatrices(A, B):
   - Input: Matrices A of size n x m and B of size m x p
   - Output: Matrix C of size n x p
   - Initialize matrix C with zeros
   - For each row i of A from 0 to n-1:
     - For each column j of B from 0 to p-1:
       - For each inner index k from 0 to m-1:
         - C[i][j] += A[i][k] * B[k][j]
   - Return C

Print Matrix

1. Function printMatrix(matrix):
   - Input: Matrix matrix
   - For each row in matrix:
     - For each element val in the row:
       - Print val followed by a space
     - Print a newline

CODE:

```
#include <iostream>
#include <vector>

using namespace std;
```

```
vector<vector<int>> multiplyMatrices(const vector<vector<int>>& A,
const vector<vector<int>>& B) {
    int n = A.size();
    int m = A[0].size();
    int p = B[0].size();

    vector<vector<int>> C(n, vector<int>(p, 0));

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

    return C;
}

void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

int main() {
    int n, m, p;

    cout << "Enter the number of rows and columns for the first
matrix (n m): ";
    cin >> n >> m;

    cout << "Enter the number of columns for the second matrix (p):
";
    cin >> p;

    vector<vector<int>> A(n, vector<int>(m));
    vector<vector<int>> B(m, vector<int>(p));
```

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

    cout << "Enter elements for matrix B (" << m << "x" << p <<
"):\n";
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < p; ++j) {
            cin >> B[i][j];
        }
    }

    vector<vector<int>> C = multiplyMatrices(A, B);

    cout << "Resulting matrix C (" << n << "x" << p << "):\n";
    printMatrix(C);

    return 0;
}
```

OUPUT:

```
Enter the number of rows and columns for the first matrix (n m): 2 2
Enter the number of columns for the second matrix (p): 2
Enter elements for matrix A (2x2):
2 2
3 3
Enter elements for matrix B (2x2):
4 4
5 5
Resulting matrix C (2x2):
18 18
27 27
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign\DAA\22510064_4_(1_4)> cd "
 -o 1 } ; if ($?) { .\1 }
Enter the number of rows and columns for the first matrix (n m): 3 2
Enter the number of columns for the second matrix (p): 3
Enter elements for matrix A (3x2):
1 2
3 4
5 6
Enter elements for matrix B (2x3):
7 8 9
10 11 12
```

```
Resulting matrix C (3x3):
27 30 33
61 68 75
95 106 117
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign\DAA\22510064_4_(1_4)> cd
 -o 1 } ; if ($?) { .\1 }
Enter the number of rows and columns for the first matrix (n m): 1 1
Enter the number of columns for the second matrix (p): 1
Enter elements for matrix A (1x1):
4
Enter elements for matrix B (1x1):
6
Resulting matrix C (1x1):
24
```

2) Implement Strassen's matrix multiplication for 3*3 matrix.

Do analysis of algorithm with respect to time complexity.

Ans:

For implementing Strassen's algorithm specifically for 3×33 \times 33×3 matrices, we'll need to adapt the approach since Strassen’s algorithm is best suited for matrices where dimensions are powers of 2 (like 2×22 \times 22×2, 4×44 \times 44×4, etc.). However, it is possible to adapt Strassen's algorithm to work with 3×33 \times 33×3 matrices by breaking them down into smaller matrices and using recursive calls.

Here's a simplified implementation that will handle 3×33 \times 33×3 matrices by padding them to 4×44 \times 44×4 for practical purposes, or using a direct approach if you're focused on pure 3×33 \times 33×3 multiplication.

Pseudocode:
Pseudocode:
Matrix Addition

1. Function add(A, B):

   - Input: Matrices A and B of size n x n

   - Output: Matrix C of size n x n

   - For each row i from 0 to n-1:

     - For each column j from 0 to n-1:

       - C[i][j] = A[i][j] + B[i][j]

   - Return C

Matrix Subtraction

1. Function subtract(A, B):

   - Input: Matrices A and B of size n x n

   - Output: Matrix C of size n x n

   - For each row i from 0 to n-1:

     - For each column j from 0 to n-1:

       - C[i][j] = A[i][j] - B[i][j]

   - Return C

Strassen's Matrix Multiplication (2x2)

1. Function strassenMultiply2x2(A, B):
   - Input: Matrices A and B of size 2 x 2
   - Output: Matrix C of size 2 x 2
   - Calculate intermediate values:
     - a = A[0][0], b = A[0][1], c = A[1][0], d = A[1][1]
     - e = B[0][0], f = B[0][1], g = B[1][0], h = B[1][1]
   - Calculate elements of C:
     - C[0][0] = a*e + b*g
     - C[0][1] = a*f + b*h
     - C[1][0] = c*e + d*g
     - C[1][1] = c*f + d*h
   - Return C

Standard Matrix Multiplication (3x3)

1. Function multiply3x3(A, B):
   - Input: Matrices A and B of size 3 x 3
   - Output: Matrix C of size 3 x 3
   - For each row i from 0 to 2:
     - For each column j from 0 to 2:
       - For each inner index k from 0 to 2:
         - C[i][j] += A[i][k] * B[k][j]
   - Return C

Code:

```
#include <iostream>
#include <vector>

using namespace std;

```

```
vector<vector<int>> add(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> 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;
}

vector<vector<int>> subtract(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> 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;
}

vector<vector<int>> strassenMultiply2x2(const vector<vector<int>>&
A, const vector<vector<int>>& B) {
    vector<vector<int>> C(2, vector<int>(2));

    int a = A[0][0], b = A[0][1], c = A[1][0], d = A[1][1];
    int e = B[0][0], f = B[0][1], g = B[1][0], h = B[1][1];

    C[0][0] = a*e + b*g;
    C[0][1] = a*f + b*h;
    C[1][0] = c*e + d*g;
    C[1][1] = c*f + d*h;

    return C;
}

vector<vector<int>> multiply3x3(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
```

```
    int n = 3;
    vector<vector<int>> C(n, vector<int>(n, 0));

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

    return C;
}

void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

int main() {
    int n = 3;
    vector<vector<int>> A(n, vector<int>(n));
    vector<vector<int>> B(n, vector<int>(n));

    cout << "Enter elements for matrix A (3x3):\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> A[i][j];
        }
    }

    cout << "Enter elements for matrix B (3x3):\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> B[i][j];
        }
    }
```

```
    vector<vector<int>> C = multiply3x3(A, B);

    cout << "Resulting matrix C (3x3):\n";
    printMatrix(C);

    return 0;
}
```

OUTPUT:

```
Enter elements for matrix A (3x3):
1 2 4
1 3 5
3 5 6
Enter elements for matrix B (3x3):
2 4 65
12 4 54
23 54 67
Resulting matrix C (3x3):
118 228 441
153 286 562
204 356 867
```

```
Enter elements for matrix A (3x3):
1 2 3
0 1 0
0 0 3
Enter elements for matrix B (3x3):
3 43 6
1 3
2 4 5
2
Resulting matrix C (3x3):
17 64 16
1 3 2
12 15 6
```

Explanation

1. Naive Multiplication (multiply3x3):
    - Directly multiplies two 3×3 matrices using the standard triple nested loop approach.

- Complexity: O(3^3)=O(27), which is constant time for fixed-size matrices, but scales as O(n3) for general n×n matrices.

2. Strassen's Algorithm (for 2×2 matrices):
   - Included for educational purposes, showing how Strassen's method applies to 2×2 matrices.
3. Padding or Decomposition:
   - Strassen's algorithm would typically require padding the 3×3 matrix to 4×4 or using specific techniques to adapt the method. For educational purposes, direct multiplication is used here.

Time Complexity

- Naive Multiplication for 3×3matrices:
  - Complexity: O(n3) which is constant O(27) for 3×3 matrices.
- Strassen's Algorithm for 2×2 matrices:
  - Complexity: O(n^$\log_2$7), approximately O(n^2.81) for n×n matrices.

Space Complexity

- Naive Multiplication:
  - Space Complexity: O(n^2) for storing matrices A, B, and C.
- Strassen's Algorithm:
  - Space Complexity: O(n^2), similar to naive, but may involve additional temporary storage for intermediate matrices.

For 3×3 matrices, using direct multiplication is the simplest approach and does not introduce the overhead of padding or complex recursion.