Matrix Multiplication

By Manav Patel

<u>Basics of Matrix Multiplication.</u>

We can multiply matrices when,

Matrix A = [n * p], and Matrix B = [p * m]

n* p = p * m

So, the resulting Matrix C = [n * m]

We cannot multiply matrices if number of columns in the first matrix does not match the number of rows in the second matrix.

Function for Multiply matrices.

```
// Function to multiply two matrices
void multiplyMatrices(int A[][10], int B[][10], int C[][10], int r1, int c1, int r2, int c2) {

    // Matrix multiplication logic
    for (int i = 0; i < r1; i++) { // Iterate through rows of matrix A
        for (int j = 0; j < c2; j++) { // Iterate through columns of matrix B
            C[i][j] = 0;
            for (int k = 0; k < c1; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Time complexity of this algorithm:-

Outer loop:- n times.

Middle loop:- n times

Inner loop:- n times

Time complexity for Matrix Multiplication is O(n^3).

Another method for Matrix Multiplication.

Divide and conquer:-

- Divide the larger matrix into the smaller one and calculate for the smallest input that we know. This is our base case.
- Recursive call for these matrices. Divide the matrices n/2 every time.

Base case:

```cpp
// Function to multiply two 2x2 matrices
vector<vector<int>> multiply2x2(const vector<vector<int>>& A, const vector<vector<int>>& B)
    vector<vector<int>> C(2, vector<int>(2));
    C[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0];
    C[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1];
    C[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0];
    C[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1];
    return C;
}
```

Now, if the value of n(matrix size) is greater than 2 then we divide it into half until n is not equal to 2.

Ex. 4 * 4 → 2 * 2

8 * 8 → 4 * 4 → 2 * 2

And send 2 * 2 matrix into the recursive function.

Recursive Function:

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

    // Recursively multiply submatrices
    vector<vector<int>> C11 = addMatrix(divideAndConquer(A11, B11), divideAndConquer(A12, B21));
    vector<vector<int>> C12 = addMatrix(divideAndConquer(A11, B12), divideAndConquer(A12, B22));
    vector<vector<int>> C21 = addMatrix(divideAndConquer(A21, B11), divideAndConquer(A22, B21));
    vector<vector<int>> C22 = addMatrix(divideAndConquer(A21, B12), divideAndConquer(A22, B22));
```

Add Function:

To add 2 matrices, we need two for loops, and each for loop runs from 0 to n.

Aso time complexity for this function is O(n^2).

Time complexity of the function is O(n^2).

Recurrence relation:

Base case :- constant time → $O(1)$.

Recursion → 8 times, and every time divide by half = $8T(n/2)$.

Addition four times = $4(n^2)$

$T(n) = 8T(n/2) + 4n^2 \rightarrow 8T(n/2) + O(n^2)$.

$T(n) = 8T(n/2) + O(n^2)$.

Standard approach and Divide and conquer approach both will take $O(n^3)$ time.

Can we do better?

Strassen's algorithm is an efficient method for matrix multiplication that reduces the time complexity compared to the traditional $O(n^3)$ approach. It does this by breaking down the multiplication of two matrices into smaller subproblems, allowing for fewer total multiplications.

Base case:

```
// Function to multiply two 2x2 matrices
vector<vector<int>> multiply2x2(const vector<vector<int>>& A, const vector<vector<int>>& B)
    vector<vector<int>> C(2, vector<int>(2));
    C[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0];
    C[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1];
    C[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0];
    C[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1];
    return C;
}
```

Recursive call:

```
// Applying Strassen's formula
vector<vector<int>> M1 = strassen(addMatrix(a11, a22), addMatrix(b11, b22));
vector<vector<int>> M2 = strassen(addMatrix(a21, a22), b11);
vector<vector<int>> M3 = strassen(a11, subtractMatrix(b12, b22));
vector<vector<int>> M4 = strassen(a22, subtractMatrix(b21, b11));
vector<vector<int>> M5 = strassen(addMatrix(a11, a12), b22);
vector<vector<int>> M6 = strassen(subtractMatrix(a21, a11), addMatrix(b11, b12));
vector<vector<int>> M7 = strassen(subtractMatrix(a12, a22), addMatrix(b21, b22));
```

Calculating resulting matrix:-

```
// Calculating submatrices of the result matrix C
vector<vector<int>> c11 = addMatrix(subtractMatrix(addMatrix(M1, M4), M5), M7);
vector<vector<int>> c12 = addMatrix(M3, M5);
vector<vector<int>> c21 = addMatrix(M2, M4);
vector<vector<int>> c22 = addMatrix(subtractMatrix(addMatrix(M1, M3), M2), M6);
```

Recurrence relation:

Base case :- constant time → $O(1)$.

Recursion → 7 times, and every time divide by half = $7T(n/2)$.

Addition 6 times + Subtraction 4 times = $10(n^2)$.

$T(n) = 7T(n/2) + 10(n^2) \rightarrow 7T(n/2) + O(n^2)$.

$T(n) = 7T(n/2) + O(n^2)$.

By applying master theorem, The time complexity of this algorithm is $O(n^{2.81})$.

Thank you