<u>Design and Analysis of Algorithms (CS206)</u>

Assignment - 2

<u>**U19CS012**</u>

1. Assist an architect in *drawing the skyline* of a city given the locations of the buildings in the city. All buildings are **rectangular** in shape and they share a **common bottom** (a flat surface).

A building is specified by an **ordered triplet** (*Li*, *Ri*, *Hi*) where <u>*Li* and *Ri*</u> are the left and right (x) coordinates, respectively, of the building *i* (0 < *Li* < *Ri*) and *Hi* is the height of the building.

• For example, the input can be as follows.

(33, 41, 5)
(4, 9, 21)
(30, 36, 9)
(14, 18, 11)
(2, 12, 14)
(34, 43, 19)
(23, 25, 8)
(14, 21, 16)
(32, 37, 12)
(7, 16, 7)
(24, 27, 10)

The pseudocode/program should give the minimum number of points on graph (coordinates) as output to assist the architect in drawing the skyline.

<u>APPROACH</u>

We can use an <u>array of 10,000 elements</u> [**Aux_Hgt**] to represent the *height of each individual discrete x-coordinate.*
For each x-coordinate, we take the <u>highest of all the heights of all the buildings within the range</u>.
For each adjacent x-coordinates, report if there is a change in the height.

1.1. (T) Write a pseudocode (using an incremental/conventional approach) to find the skyline. Analyze the time complexity.

```
// Maximum X Co-ordinate for Right Edge [Constraint]
#define MAX_Ri 100000
int Aux_Hgt[MAX_Ri];

// Skyline Problem
• Sky_Line_Brute_Force(L, R, H)

n = H.size();
 Rmax = 0;
// Interating For Each Building
for i = 0 to n-1
// From Range [L[i],R[i])
// Check if H[i]>Aux_Hgt[j] [Building Bi is Taller]
 for j = L[i] to R[i]
     if (Aux_Hgt[j] < H[i])
         Aux_Hgt[j] = H[i];
     if (Rmax < R[i])
           Rmax = R[i];

// Print the Output
Old_Hgt = 0;

 for i = 1 to Rmax-1
     if (Old_Hgt != Aux_Hgt[i])
         cout << i << " " << Aux_Hgt[i] << endl;
     Old_Hgt = Aux_Hgt[i];

 cout << Rmax << " " << Aux_Hgt[Rmax] << endl;
```

Analysis:

Assume **n** = Number of Buildings in Input Sequence, **m** = rightmost x-coordinate [maximum Ri]

From Above Pseudo Code, We are Traversing from Left to Right to Update the Heights. For Worst Case, n Equal Sized Building with l=0 to r = m – 1 coordinates.

Therefore, Running Time = $O(n*m) = O(n^2)$, if (m>n)

1.2. (L) Write a program using an incremental (conventional) approach to find the skyline.

```
#include <bits/stdc++.h>

using namespace std;

typedef long long int ll;

// Maximum X Co-ordinate for Right Edge [Constraint]
#define MAX_Ri 10010
int Aux_Hgt[MAX_Ri];

// Skyline Problem
void Sky_Line_Brute_Force(vector<int> &L, vector<int> &R, vector<int> &H)
{
    int n = H.size();
    int Rmax = 0;
    // Interating For Each Building
    for (int i = 0; i < n; i++)
    {
        // From Range [L[i],R[i])
        // Check if H[i]>Aux_Hgt[j] [Building Bi is Taller]
        for (int j = L[i]; j < R[i]; j++)
        {
            if (Aux_Hgt[j] < H[i])
                Aux_Hgt[j] = H[i];
            if (Rmax < R[i])
                Rmax = R[i];
        }
    }

    int Old_Hgt = 0;
    for (int i = 1; i < Rmax; i++)
    {
        if (Old_Hgt != Aux_Hgt[i])
        {
            cout << i << " " << Aux_Hgt[i] << endl;
            Old_Hgt = Aux_Hgt[i];
        }
    }
    cout << Rmax << " " << Aux_Hgt[Rmax] << endl;
    return;
}

int main()
{
    // Number of Points
    ll n;
```

```
    cin >> n;

    vector<int> L(n, 0);
    vector<int> R(n, 0);
    vector<int> H(n, 0);
    // li = x-Position Of Left Edge
    // ri = x-Position Of Right Edge
    // hi = Building's Height

    for (int i = 0; i < n; i++)
    {
        cin >> L[i] >> R[i] >> H[i];
    }

    Sky_Line_Brute_Force(L, R, H);

    return 0;
}
```

Only Change in UVa Problem 105 is {Li,Hi,Ri} Instead if {Li,Ri,Hi} [As Mentioned in this Assignment] [Run-Time = 0.010 seconds]

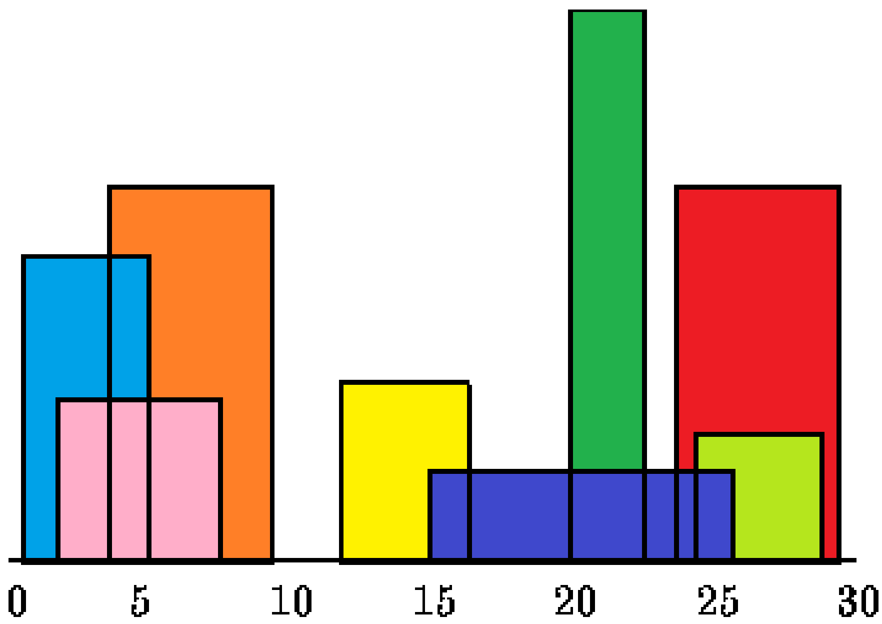
My Submissions

Online Judge Accepted Verdict

#	Problem	Verdict	Language	Run Time	Submission Date
26128577	105 The Skyline Problem	Accepted	C++11	0.010	2021-02-24 12:27:02

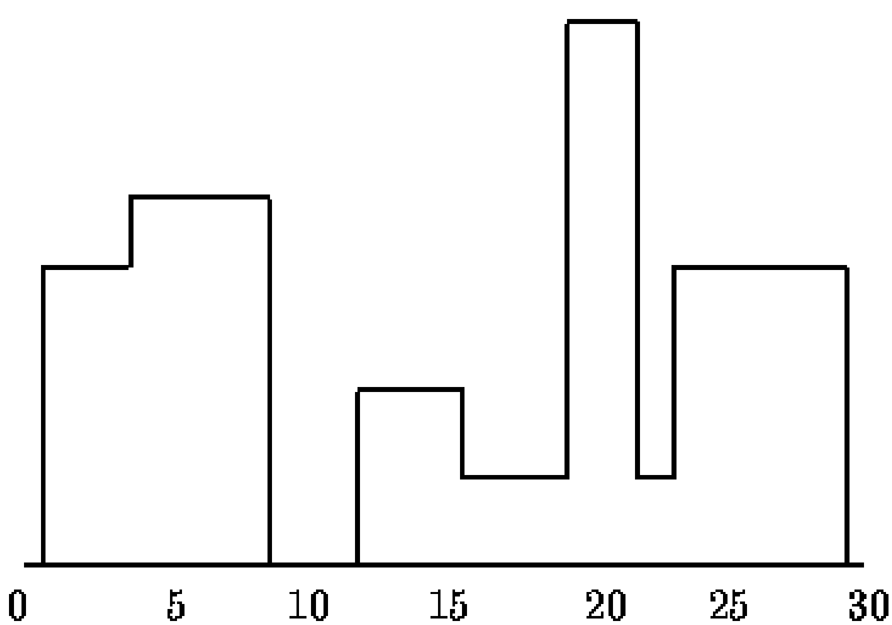
Sample Test Case: [Left Edge | Right Edge | Height]

```
8
1 5 11
2 7 6
3 9 13
12 16 7
14 25 3
19 22 18
23 29 13
24 28 4
```



Expected Output: [X Co-ordinate | Height]

1 11
3 13
9 0
12 7
16 3
19 18
22 3
23 13
29 0



1.3. (T) Write a pseudocode to find the skyline using the divide and conquer approach. Analyze the time complexity.

```
// Skyline Problem [Divide And Conquer Approach ~ Merge Sort]
• Skyline_DnC(buildings, start, end)

// Base Case
if start == end
    vector<pair<int,int>> ans;
    ans.push_back({buildings[start][0], buildings[start][2]});
    ans.push_back({buildings[start][1], 0});
    return ans;

mid = start + (end - start) / 2; // Avoid Overflow Errors

lft_skyline = Skyline_DnC(buildings, start, mid);
rgt_skyline = Skyline_DnC(buildings, mid + 1, end);
ans = merge_skyline(lft_skyline, rgt_skyline);
return ans;
```

MERGE PART

```
// Merging of Two Skylines Using Two-Pointer Approach
• merge_skyline(lft_skyline, rgt_skyline)

    vector<pair<int,int>> ans;
    // Two Pointers
    int i = 0, j = 0, curr_hgt1 = 0, curr_hgt2 = 0;
    int max_hgt = max(curr_hgt1, curr_hgt2);

// Iterating until any one of Skyline Ends
    while (i < lft_skyline.size() && j < rgt_skyline.size())

// Case 1 : X Co-Ordinate of Left Skyline is Smaller
        if (lft_skyline[i].first < rgt_skyline[j].first)
            curr_hgt1 = lft_skyline[i].second;
            if (max_hgt != max(curr_hgt1, curr_hgt2))
                ans.pb({lft_skyline[i].first, max(curr_hgt1, curr_hgt2)});
            max_hgt = max(curr_hgt1, curr_hgt2);
            i++;

// Case 2 : X Co-Ordinate of Right Skyline is Smaller
        else if (lft_skyline[i].first > rgt_skyline[j].first)
            curr_hgt2 = rgt_skyline[j].second;
            if (max_hgt != max(curr_hgt1, curr_hgt2))
                ans.pb({rgt_skyline[j].first, max(curr_hgt1, curr_hgt2)});
            max_hgt = max(curr_hgt1, curr_hgt2);
            j++;
```

```
// Case 3 : Both Have Same X Co-ordinate
        else
            curr_hgt1 = lft_skyline[i].second;
            curr_hgt2 = rgt_skyline[j].second;

    // Case 3 (a) : Height of Left Skyline is Greater
            if (lft_skyline[i].second >= rgt_skyline[j].second)
                if (max_hgt != max(curr_hgt1, curr_hgt2))
                    ans.pb({lft_skyline[i].first, max(curr_hgt1, curr_hgt2)});

    // Case 3 (b) : Height of Right Skyline is Greater
            else
                if (max_hgt != max(curr_hgt1, curr_hgt2))
                    ans.pb({rgt_skyline[j].first, max(curr_hgt1, curr_hgt2)});
            max_hgt = max(curr_hgt1, curr_hgt2);
            i++;
            j++;


// Remaining Elements in Left Skyline
    while (i < lft_skyline.size())
        ans.pb(lft_skyline[i]);
        i++;

// Remaining Elements in Right Skyline
    while (j < rgt_skyline.size())
        ans.pb(rgt_skyline[j]);
        j++;

    return ans;
```

APPROACH

- We can solve this problem by separating the buildings into two halves and solving those recursively and then Merging the 2 skylines.
  - Similar to merge sort.
  - Requires that we have a way to merge 2 skylines.

- Consider two skylines:
  - Skyline A: $a_1, h_{11}, a_2, h_{12}, a_3, h_{13}, \ldots, a_n, 0$
  - Skyline B: $b_1, h_{21}, b_2, h_{22}, b_3, h_{23}, \ldots, b_m, 0$

- Merge(list of a's, list of b's)
  - $(c_1, h_{11}, c_2, h_{21}, c_3, \ldots, c_{n+m}, 0)$

Complexity Analysis:
The Above Algorithm has Similar Structure as Merge Sort, Hence the Time Complexity of this Approach is O(n*log(n)). [Seen in Run-Time Difference]

1.4. (L) Write a program using the divide-and-conquer approach to find the skyline.

```
#include <bits/stdc++.h>

using namespace std;

//SHORT HAND
#define pb push_back
#define mp make_pair

typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int, int> pi;
typedef vector<pi> vpi;

// Merging of Two Skylines Using Two-Pointer Approach
vpi merge_skyline(vpi &lft_skyline, vpi &rgt_skyline);

// Skyline Problem [Divide And Conquer Approach ~ Merge Sort]
vpi Skyline_DnC(vvi &buildings, int start, int end);

int main()
{
    // Number of Points
    int n, lft, rgt, hgt;
    cin >> n;

    // lft = x-Position Of Left Edge
    // rgt = x-Position Of Right Edge
    // hgt = Building's Height

    vvi buildings;
    for (int i = 0; i < n; i++)
    {
        cin >> lft >> rgt >> hgt;
        vi tmp;
        tmp.push_back(lft);
        tmp.push_back(rgt);
        tmp.push_back(hgt);
        buildings.push_back(tmp);
    }

    vpi final_ans = Skyline_DnC(buildings, 0, buildings.size() - 1);
```

```
    for (auto pr : final_ans)
    {
        cout << pr.first << " " << pr.second << endl;
    }

    return 0;
}

// Skyline Problem [Divide And Conquer Approach ~ Merge Sort]
vpi Skyline_DnC(vvi &buildings, int start, int end)
{
    // Base Case
    if (start == end)
    {
        vpi ans;
        ans.pb({buildings[start][0], buildings[start][2]});
        ans.pb({buildings[start][1], 0});
        return ans;
    }

    int mid = start + (end - start) / 2; // Avoid Overflow Errors

    vpi lft_skyline = Skyline_DnC(buildings, start, mid);
    vpi rgt_skyline = Skyline_DnC(buildings, mid + 1, end);
    vpi ans = merge_skyline(lft_skyline, rgt_skyline);

    return ans;
}

// Merging of Two Skylines Using Two-Pointer Approach
vpi merge_skyline(vpi &lft_skyline, vpi &rgt_skyline)
{
    vpi ans;
    // Two Pointers
    int i = 0, j = 0, curr_hgt1 = 0, curr_hgt2 = 0;
    int max_hgt = max(curr_hgt1, curr_hgt2);

    // Iterating until any one of Skyline Ends
    while (i < lft_skyline.size() && j < rgt_skyline.size())
    {
        // Case 1 : X Co-Ordinate of Left Skyline is Smaller
        if (lft_skyline[i].first < rgt_skyline[j].first)
        {
            curr_hgt1 = lft_skyline[i].second;
            if (max_hgt != max(curr_hgt1, curr_hgt2))
            {
                ans.pb({lft_skyline[i].first, max(curr_hgt1, curr_hgt2)});
            }
            max_hgt = max(curr_hgt1, curr_hgt2);
            i++;
```

```
        }
        // Case 2 : X Co-Ordinate of Right Skyline is Smaller
        else if (lft_skyline[i].first > rgt_skyline[j].first)
        {
            curr_hgt2 = rgt_skyline[j].second;
            if (max_hgt != max(curr_hgt1, curr_hgt2))
            {
                ans.pb({rgt_skyline[j].first, max(curr_hgt1, curr_hgt2)});
            }
            max_hgt = max(curr_hgt1, curr_hgt2);
            j++;
        }
        // Case 3 : Both Have Same X Co-ordinate
        else
        {
            curr_hgt1 = lft_skyline[i].second;
            curr_hgt2 = rgt_skyline[j].second;
            // Case 3 (a) : Height of Left Skyline is Greater
            if (lft_skyline[i].second >= rgt_skyline[j].second)
            {
                if (max_hgt != max(curr_hgt1, curr_hgt2))
                {
                    ans.pb({lft_skyline[i].first, max(curr_hgt1, curr_hgt2)});
                }
            }
            // Case 3 (b) : Height of Right Skyline is Greater
            else
            {
                if (max_hgt != max(curr_hgt1, curr_hgt2))
                {
                    ans.pb({rgt_skyline[j].first, max(curr_hgt1, curr_hgt2)});
                }
            }
            max_hgt = max(curr_hgt1, curr_hgt2);
            i++;
            j++;
        }
    }
    // Remaining Elements in Left Skyline
    while (i < lft_skyline.size())
    {
        ans.pb(lft_skyline[i]);
        i++;
    }
    // Remaining Elements in Right Skyline
    while (j < rgt_skyline.size())
    {
        ans.pb(rgt_skyline[j]);
        j++;
    }
```

```
    return ans;
}
```

Only Change in UVa Problem 105 is {Li,Hi,Ri} Instead if {Li,Ri,Hi} [As Mentioned in this Assignment]

My Submissions

Online Judge Accepted Submission [Divide & Conquer]

#	Problem	Verdict	Language	Run Time	Submission Date
26128652	105 The Skyline Problem	Accepted	C++11	0.000	2021-02-24 12:48:29

[Run-Time = 0.000 seconds as Compared to **Brute Force** Solution whose Run-Time was 0.010 seconds.]

---

2. Given two matrices *A* and *B*, answer the following questions.

2.1. (T) Write a pseudocode (using an incremental/conventional approach) to multiply the given matrices. Analyze the time complexity.

Suppose we are multiplying 2 matrices A and B and both of them have dimensions n x n.
The resulting matrix C after multiplication in the naive algorithm is obtained by the formula:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

```
Matrix_Multiplication(A,B,C)

for i = 0 to n
  for j = 0 to n
     C[i][j] = 0;
     for k = 0 to n
        C[i][j] += A[i][k] * B[k][j]
```

In this algorithm, the statement “C[i][j] += A[i][k] * B[k][j]” executes n^3 times as evident from the three nested for loops and is the most costly operation in the algorithm.

Time Complexity of above method is $O(N^3)$.

2.2. (L) Write a program using an incremental (conventional) approach to multiply the given matrices.

```
#include <bits/stdc++.h>

using namespace std;

// Change this According to Your Requirement [Constraints on N]
#define MAX_N 100

// Dimensions of A[n1*m1] & B[n2*m2]
int n1, n2, m1, m2;

// 2 Dimensions Matrix [A,B & C]
vector<vector<int>> A(MAX_N, vector<int>(MAX_N, 0));
vector<vector<int>> B(MAX_N, vector<int>(MAX_N, 0));
vector<vector<int>> C(MAX_N, vector<int>(MAX_N, 0));

// Brute Force Method to Multiply Two Matrices O(N^3)
void matrix_multiply()
{
    for (int i = 0; i < n1; i++)
    {
        for (int j = 0; j < m2; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < m1; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return;
}

int main()
{
    // Dimensions of Matrix A
    cout << "Enter Dimensions of Matrix 1 [row col]: " << endl;
    cin >> n1 >> m1;
    cout << "Enter the Values in Matrix 1:" << endl;
```

```
    for (int i = 0; i < n1; i++)
    {
        for (int j = 0; j < m1; j++)
        {
            cout << "A[" << i << "][" << j << "] = ";
            cin >> A[i][j];
        }
    }

    cout << "Enter Dimensions of Matrix 2 [row col]: " << endl;
    // Dimensions of Matrix B
    cin >> n2 >> m2;
    cout << "Enter the Values in Matrix 2:" << endl;
    for (int i = 0; i < n2; i++)
    {
        for (int j = 0; j < m2; j++)
        {
            cout << "B[" << i << "][" << j << "] = ";
            cin >> B[i][j];
        }
    }

    if (m1 != n2)
    {
        cout << "Matrix Can't Be Multiplied!" << endl;
        cout << "For Matrix Multiplication,\n No. Of Columns [Matrix-
1] Must be Equal No. Of Rows [Matrix-2]!" << endl;
    }
    else
    {
        // n1 * m2 = Dimensions of C
        matrix_multiply();

        cout << "MATRIX A:" << endl;
        for (int i = 0; i < n1; i++)
        {
            for (int j = 0; j < m1; j++)
            {
                cout << A[i][j] << " ";
            }
            cout << endl;
        }

        cout << "MATRIX B:" << endl;
        for (int i = 0; i < n2; i++)
        {
            for (int j = 0; j < m2; j++)
            {
                cout << B[i][j] << " ";
            }
```

```
            cout << endl;
        }

        cout << "MATRIX C [AXB]:" << endl;
        for (int i = 0; i < n1; i++)
        {
            for (int j = 0; j < m2; j++)
            {
                cout << C[i][j] << " ";
            }
            cout << endl;
        }
    }

    return 0;
}
```

Sample Test Case:

```
Enter Dimensions of Matrix 1 [row col]:
3 4
Enter the Values in Matrix 1:
1 3 1 2
2 1 2 3
3 2 1 3
Enter Dimensions of Matrix 2 [row col]:
4 3
Enter the Values in Matrix 2:
1 2 1
2 3 1
4 2 1
3 1 3
MATRIX A:
1 3 1 2
2 1 2 3
3 2 1 3
MATRIX B:
1 2 1
2 3 1
4 2 1
3 1 3
MATRIX C [AXB]:
17 15 11
21 14 14
20 17 15
```

2.3. (T) Write a pseudocode to multiply the given matrices using the divide and conquer approach. Analyze the time complexity.

[Note: I have Implemented in C++, But Explaining Pseudo-Code in Python was Easy]

```python
def strassen(x, y):

    # Base case when size of matrices is 1x1
    if len(x) == 1:
        return x * y

    # Splitting the matrices into quadrants. This will be done recursively
    # untill the base case is reached.
    a, b, c, d = split(x)
    e, f, g, h = split(y)

    # Computing the 7 products, recursively (p1, p2...p7)
    p1 = strassen(a, f - h)
    p2 = strassen(a + b, h)
    p3 = strassen(c + d, e)
    p4 = strassen(d, g - e)
    p5 = strassen(a + d, e + h)
    p6 = strassen(b - d, g + h)
    p7 = strassen(a - c, e + f)

    # Computing the values of the 4 quadrants of the final matrix c
    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7

    # Combining the 4 quadrants into a single matrix by stacking horizontally and vertically.
    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))

    return c
```

p1 = a(f-h)
p2 = (a+b)h
p3 = (c+d)e
p4 = d(g-e)
p5 = (a+d)(e+h)
p6 = (b-d)(g+h)
p7 = (a-c)(e+f)

$$\underset{A}{\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]} \times \underset{B}{\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]} = \underset{C}{\left[\begin{array}{c|c} p5+p4-p2+p6 & p1+p2 \\ \hline p3+p4 & p1+p5-p3p7 \end{array}\right]}$$

A.) Divide matrices A and B in 4 sub-matrices of size **N/2 x N/2** as shown in the above diagram.
B.) Calculate the **7 matrix multiplications** recursively.
C.) Compute the submatrices of C.
D.) Combine these submatrices into our new matrix C

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

```
T(N) = 7T(N/2) +  O(N^2)

From Master's Theorem, time complexity of above method is
O(N^Log7) which is approximately O(N^2.8074)
```

2.4. (L) Write a program using the divide-and-conquer approach to multiply the given matrices.

```cpp
#include <bits/stdc++.h>

using namespace std;

// Matrix Operations
// To Intialise the Matrix
int **init_matrix(int n);

// To Take Input to Matrix
void input(int **M, int n);

// To Print the Matrix
void print_matrix(int **M, int n);

// To Add Two Matrices of Size( n X n )
int **add(int **M1, int **M2, int n);

// To Subtract Two Matrices of Size( n X n )
int **subtract(int **M1, int **M2, int n);

// Strassen Multiplication Function
int **Strassen_Multiply(int **A, int **B, int n);

// Checks if n is Power of 2 or Not
```

```cpp
bool check(int x)
{
    return x && (!(x & (x - 1)));
}

int main()
{
    cout << "Enter Size of the Matrix (Power of 2): ";

    int n;
    cin >> n;

    if (check(n))
    {
        int **A = init_matrix(n);
        input(A, n);

        int **B = init_matrix(n);
        input(B, n);

        cout << "Matrix A:" << endl;
        print_matrix(A, n);

        cout << "Matrix B:" << endl;
        print_matrix(B, n);

        int **C = init_matrix(n);
        C = Strassen_Multiply(A, B, n);

        cout << "MATRIX C [AXB]:" << endl;
        print_matrix(C, n);
    }
    else
    {
        cout << "Matrix Can't Be Multiplied!" << endl;
        cout << "Strassian Multiplication => Only Works on Square Matrices whose Dimension is
 a Power of 2!\n";
    }

    return 0;
}

// --------------------------------MATRIX_OPERATIONS------------------------------------

// To Intialise the Matrix
int **init_matrix(int n)
{
    int **temp = new int *[n];
    for (int i = 0; i < n; i++)
        temp[i] = new int[n];
```

```
    return temp;
}

// To Take Input to Matrix
void input(int **M, int n)
{
    cout << "Enter Matrix: " << endl;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> M[i][j];
    cout << endl;
}

// To Print the Matrix
void print_matrix(int **M, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            cout << M[i][j] << " ";
        cout << endl;
    }
    cout << endl;
}

// To Add Two Matrices of Size( n X n )
int **add(int **M1, int **M2, int n)
{
    int **temp = init_matrix(n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            temp[i][j] = M1[i][j] + M2[i][j];
    return temp;
}

// To Subtract Two Matrices of Size( n X n )
int **subtract(int **M1, int **M2, int n)
{
    int **temp = init_matrix(n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            temp[i][j] = M1[i][j] - M2[i][j];
    return temp;
}

// Strassen Multiplication Function
int **Strassen_Multiply(int **A, int **B, int n)
{
    // Base Case
    if (n == 1)
```

```
    {
        int **C = init_matrix(1);
        C[0][0] = A[0][0] * B[0][0];
        return C;
    }

    int **C = init_matrix(n);
    int k = n / 2;

    int **A11 = init_matrix(k);
    int **A12 = init_matrix(k);
    int **A21 = init_matrix(k);
    int **A22 = init_matrix(k);
    int **B11 = init_matrix(k);
    int **B12 = init_matrix(k);
    int **B21 = init_matrix(k);
    int **B22 = init_matrix(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][k + j];
            A21[i][j] = A[k + i][j];
            A22[i][j] = A[k + i][k + j];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][k + j];
            B21[i][j] = B[k + i][j];
            B22[i][j] = B[k + i][k + j];
        }
    }

    int **P1 = Strassen_Multiply(A11, subtract(B12, B22, k), k);
    int **P2 = Strassen_Multiply(add(A11, A12, k), B22, k);
    int **P3 = Strassen_Multiply(add(A21, A22, k), B11, k);
    int **P4 = Strassen_Multiply(A22, subtract(B21, B11, k), k);
    int **P5 = Strassen_Multiply(add(A11, A22, k), add(B11, B22, k), k);
    int **P6 = Strassen_Multiply(subtract(A12, A22, k), add(B21, B22, k), k);
    int **P7 = Strassen_Multiply(subtract(A11, A21, k), add(B11, B12, k), k);

    int **C11 = subtract(add(add(P5, P4, k), P6, k), P2, k);
    int **C12 = add(P1, P2, k);
    int **C21 = add(P3, P4, k);
    int **C22 = subtract(subtract(add(P5, P1, k), P3, k), P7, k);

    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[k + i][j] = C21[i][j];
            C[k + i][k + j] = C22[i][j];
        }
    }

    return C;
}
```

SAMPLE TEST CASE:

```
Enter Size of the Matrix (Power of 2): 4
Enter Matrix:
1 2 1 2
2 3 1 4
4 1 2 1
3 3 4 2

Enter Matrix:
1 2 4 3
2 1 2 4
4 2 1 3
4 3 2 1

Matrix A:
1 2 1 2
2 3 1 4
4 1 2 1
3 3 4 2

Matrix B:
1 2 4 3
2 1 2 4
4 2 1 3
4 3 2 1

MATRIX C [AXB]:
17 12 13 16
28 21 23 25
18 16 22 23
33 23 26 35
```


SUBMITTED BY:

U19CS012

BHAGYA VINOD RANA