ASSIGNMENT 2

1 CODE :

```
#include <iostream>

#include <queue>

#include <unordered_map>

#include <vector>

#include <limits.h>


// Node structure for the Huffman Tree

struct Node {

    char move;

    int frequency;

    Node* left;

    Node* right;


    Node(char move, int frequency) : move(move), frequency(frequency), left(nullptr),
right(nullptr) {}

};


// Custom comparator for the priority queue

struct Compare {

    bool operator()(Node* left, Node* right) {

        return left->frequency > right->frequency;

    }

};


// Function to generate the Huffman Codes
```

```
void generateHuffmanCodes(Node* root, const std::string& code, std::unordered_map<char, std::string>& huffmanCode) {
    if (!root)
        return;

    if (!root->left && !root->right) {
        huffmanCode[root->move] = code;
    }

    generateHuffmanCodes(root->left, code + "0", huffmanCode);
    generateHuffmanCodes(root->right, code + "1", huffmanCode);
}

// Function to build the Huffman Tree and generate codes
std::unordered_map<char, std::string> buildHuffmanTree(const std::unordered_map<char, int>& frequencyTable) {
    std::priority_queue<Node*, std::vector<Node*>, Compare> minHeap;

    for (const auto& pair : frequencyTable) {
        minHeap.push(new Node(pair.first, pair.second));
    }

    while (minHeap.size() != 1) {
        Node* left = minHeap.top();
        minHeap.pop();
        Node* right = minHeap.top();
        minHeap.pop();

        int sum = left->frequency + right->frequency;
        Node* newNode = new Node('\0', sum);
```

```
        newNode->left = left;
        newNode->right = right;
        minHeap.push(newNode);
    }

    Node* root = minHeap.top();
    std::unordered_map<char, std::string> huffmanCode;
    generateHuffmanCodes(root, "", huffmanCode);

    // Cleanup the tree to avoid memory leaks
    std::function<void(Node*)> deleteTree = [&](Node* node) {
        if (node) {
            deleteTree(node->left);
            deleteTree(node->right);
            delete node;
        }
    };
    deleteTree(root);

    return huffmanCode;
}

// Function to encode a sequence of moves using Huffman coding
std::string encodeMoves(const std::string& moves, const std::unordered_map<char,
std::string>& huffmanCode) {
    std::string encodedString = "";
    for (char move : moves) {
        encodedString += huffmanCode.at(move);
    }
    return encodedString;
```

```
}

// Function to perform Matrix Chain Multiplication
int matrixChainMultiplication(const std::vector<int>& dims) {
    int n = dims.size() - 1;
    std::vector<std::vector<int>> dp(n, std::vector<int>(n, 0));

    for (int length = 2; length <= n; ++length) {
        for (int i = 0; i <= n - length; ++i) {
            int j = i + length - 1;
            dp[i][j] = INT_MAX;

            for (int k = i; k < j; ++k) {
                int cost = dp[i][k] + dp[k + 1][j] + dims[i] * dims[k + 1] * dims[j + 1];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }
    return dp[0][n - 1];
}

int main() {
    // Huffman Coding Example
    std::unordered_map<char, int> frequencyTable = {
        {'A', 40}, {'B', 30}, {'C', 20}, {'D', 10}, {'E', 5}
    };
```

```
    std::unordered_map<char, std::string> huffmanCode = buildHuffmanTree(frequencyTable);

    std::string moves = "ABBACADAE";
    std::string encodedMoves = encodeMoves(moves, huffmanCode);

    std::cout << "Huffman Codes:\n";
    for (const auto& pair : huffmanCode) {
        std::cout << pair.first << ": " << pair.second << "\n";
    }

    std::cout << "\nEncoded Moves: " << encodedMoves << std::endl;

    // Matrix Chain Multiplication Example
    std::vector<int> matrixDims = {2, 3, 4, 2};  // Example dimensions of matrices M1, M2, M3
    int minCost = matrixChainMultiplication(matrixDims);

    std::cout << "\nMinimum number of multiplications is: " << minCost << std::endl;

    return 0;
}
```

2 CALCULATIONS ON ENCODING AND STORAGE EFFICIENCY :

* Calculations on encoding and storage efficiency

1) Huffman code :

Move	Huffman code
A	0
B	10
C	111
D	1101
E	1100

Encoded data

Move	code length
A	1
B	2
C	3
D	4
E	4

for a sequence 'AABAC'
Total encoded bit length = 2+2+3 = 7 bits

Compression Ratio :-

original size = 5 moves * 36 bits = 180 bits
Encoded size = 7 bits

Compression ratio = $\frac{\text{original size}}{\text{Encoded size}} = \frac{180}{7} \approx 25.71$

2) Matrix chain Multiplication

$M_1 = 2 \times 3$
$M_2 = 3 \times 4$
$M_3 = 4 \times 2$

→ Cost of $M_1 \times M_2 = 2 \times 3 \times 4 = 24$
→ Cost of $(M_1 \times M_2) M_3 = 2 \times 4 \times 2 + 24 = 40$

→ cost of $M_2 \times M_3 = 3 \times 4 \times 2 = 24$
→ cost of $M_1 (M_2 M_3) = 24 + 2 \times 3 \times 2 = 36$

optimal order is $M_1 (M_2 M_3)$ – 36

Storage Efficiency:

1) By huffman encoding, we achieve a highly efficient compression which is 25.71, indicates substantial storage savings.
2) By MCM, it determines minimal computational cost for a sequence. It reduces total no. of operations, enhancing computation performance but it requires efficient storage for matrix dimensions.