Name: Arpita Dinesh Singh
2nd-year B.tech
Computer engineering

Roll No:231071005

DESIGN ANALYSIS OF ALGORITHM

LABORATORY 05

EXPERIMENT TASK- 1
(Transportation of goods to different cities)

4.CODE
Code for CSV file for 100 items ID,value,weight and shelf Life choice:

```
import csv
import random
# Create CSV file with 100 random items
with open('items.csv', mode='w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['ID', 'Value', 'Weight', 'ShelfLife'])  # Add Shelf Life header

    for i in range(1, 101):
        value = random.randint(1, 100)  # Value between 1 and 100
        weight = random.randint(1, 10)   # Weight between 1 and 10
        shelf_life = random.randint(1, 30)  # Shelf life between 1 and 30 days
        writer.writerow([i, value, weight, shelf_life])  # Write Shelf Life

print("CSV file 'items.csv' has been created with 100 items.")
```

CODE USING FRACTIONAL KNAPSACK(Greedy Method):

```
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <algorithm>

using namespace std;

struct Item {
    int id;
    double value;
    double weight;
    double shelfLife;
    double ratio;
};

// Comparator function to sort items by their value-to-weight ratio,
// and by shelf life if the ratios are the same
bool compare(Item a, Item b) {
    if (a.ratio == b.ratio) {
        return a.shelfLife < b.shelfLife; // Prioritize lesser shelf life
    }
    return a.ratio > b.ratio; // Sort in descending order
}

double fractionalKnapsack(vector<Item>& items, double capacity) {
    double totalValue = 0.0;

    // Sort items by their value-to-weight ratio
    sort(items.begin(), items.end(), compare);

    // Print the value, weight, ratio, and shelf life for the first 10 items
```

```
    cout << "Sorted Items (ID, Value, Weight, Shelf Life, Ratio):\n";
    cout << "-----------------------------------------------------\n";
    cout << "ID\tValue\tWeight\tShelf Life\tRatio\n";
    for (size_t i = 0; i < min(items.size(), size_t(10)); ++i) {
        cout << items[i].id << "\t"
             << items[i].value << "\t"
             << items[i].weight << "\t"
             << items[i].shelfLife << "\t\t"
             << items[i].ratio << "\n";
    }
    cout << "-----------------------------------------------------\n";

    for (const auto& item : items) {
        if (capacity <= 0) {
            break; // If the capacity is full, stop
        }

        if (item.weight <= capacity) {
            // If the item can fit entirely
            totalValue += item.value;
            capacity -= item.weight;
        } else {
            // If only a fraction can fit
            totalValue += item.value * (capacity / item.weight);
            capacity = 0; // Knapsack is now full
        }
    }

    return totalValue;
}

int main() {
    vector<Item> items;
    ifstream csvFile("items.csv");
```

```
if (!csvFile.is_open()) {
    cerr << "Error opening file!" << endl;
    return 1;
}

string line;
// Skip the header line
getline(csvFile, line);

// Read items from the CSV file
while (getline(csvFile, line)) {
    stringstream ss(line);
    Item item;
    string temp;

    getline(ss, temp, ','); // Read ID
    item.id = stoi(temp);
    getline(ss, temp, ','); // Read Value
    item.value = stod(temp);
    getline(ss, temp, ','); // Read Weight
    item.weight = stod(temp);
    getline(ss, temp, ','); // Read Shelf Life
    item.shelfLife = stod(temp);
    item.ratio = item.value / item.weight; // Calculate value-to-weight ratio

    items.push_back(item);
}
csvFile.close(); // Close the file

double capacity = 200; // Capacity of the knapsack
double maxValue = fractionalKnapsack(items, capacity);
cout << "Maximum value in the knapsack: " << maxValue << endl;

return 0;
```

5.OUTPUT:

```
Sorted Items (ID, Value, Weight, Shelf Life, Ratio):
------------------------------------------------------------
ID      Value   Weight  Shelf Life      Ratio
10      90      1       12              90
30      74      1       29              74
48      62      1       10              62
36      57      1       24              57
82      99      2       26              49.5
23      98      2       9               49
2       95      2       29              47.5
44      92      2       18              46
52      91      2       17              45.5
43      89      2       20              44.5
------------------------------------------------------------
Maximum value in the knapsack: 3645.75
```

6.CONCLUSION:

In this assignment, we explored the application of the fractional knapsack problem to a real-world scenario involving a courier company that needs to efficiently transport goods with varying values and weights. The primary objective was to prioritize items based on their shelf life and value, ensuring that those with shorter shelf lives and higher values are shipped first.

<u>EXPERIMENT TASK- 2</u>

<u>(Compress books using huffman coding)</u>

4.CODE:

CODE for Document reading:

```
from collections import defaultdict

def count_character_frequencies(input_filename, output_filename):
```

```
    frequency_map = defaultdict(int)

    # Try reading with different encodings
    encodings = ['utf-8', 'latin-1', 'windows-1252']  # Add more if needed

    for encoding in encodings:
        try:
            with open(input_filename, 'r', encoding=encoding) as file:
                for line in file:
                    for char in line:
                        frequency_map[char] += 1

            # Write the frequency map to the output file
            with open(output_filename, 'w', encoding='utf-8') as output_file:
                for char, freq in frequency_map.items():
                    output_file.write(f"{char}:{freq}\n")

            print(f"Frequencies written to {output_filename}.")
            return  # Exit if successful

        except UnicodeDecodeError:
            print(f"Failed to decode with encoding: {encoding}")
        except FileNotFoundError:
            print(f"Error: The file '{input_filename}' was not found.")
            return
        except Exception as e:
            print(f"An error occurred: {e}")

    print("Failed to read the file with available encodings.")

# Example usage
if __name__ == "__main__":
    input_file_path = "C:\\Users\\dines\\Desktop\\sampleBook.txt"  # Update
with your actual file path
    output_file_path = "freq.txt"  # Output frequency file
```

```
    count_character_frequencies(input_file_path, output_file_path)
```

CODE for HUFFMAN CODING:

```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>

using namespace std;

// Node structure for the Huffman tree
struct Node {
    char ch;
    int freq;
    Node* left;
    Node* right;

    Node(char character, int frequency) : ch(character), freq(frequency),
left(nullptr), right(nullptr) {}
};

// Comparator for the priority queue
struct Compare {
    bool operator()(Node* l, Node* r) {
        return l->freq > r->freq;
    }
};

// Function to generate the Huffman codes
void generateCodes(Node* root, const string& str, unordered_map<char,
string>& codes) {
```

```
    if (!root) return;
    if (root->left == nullptr && root->right == nullptr) {
        codes[root->ch] = str;
    }
    generateCodes(root->left, str + "0", codes);
    generateCodes(root->right, str + "1", codes);
}

// Function to build the Huffman tree
Node* buildHuffmanTree(const unordered_map<char, int>& freqMap) {
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Create a node for each character and push it to the priority queue
    for (const auto& pair : freqMap) {
        pq.push(new Node(pair.first, pair.second));
    }

    // Merge nodes until there's only one node left
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        Node* combined = new Node('\0', left->freq + right->freq);
        combined->left = left;
        combined->right = right;
        pq.push(combined);
    }

    return pq.top(); // Root of the Huffman tree
}

// Function to calculate the original and compressed sizes and the
compression ratio
void calculateSizes(const unordered_map<char, int>& freqMap, const
unordered_map<char, string>& codes,
                    int& originalSize, int& compressedSize) {
```

```
    originalSize = 0;
    compressedSize = 0;

    for (const auto& pair : freqMap) {
        originalSize += pair.second * 8; // Assuming each character is 8 bits
        compressedSize += pair.second * codes.at(pair.first).length();
    }
}
// Function to read frequency map from a specified file
unordered_map<char, int> readFrequencyMap(const string& filename) {
    unordered_map<char, int> freqMap;
    ifstream infile(filename);

    if (!infile.is_open()) {
        cerr << "Error: Could not open the file " << filename << endl;
        return freqMap;
    }

    string line;
    while (getline(infile, line)) {
        if (!line.empty()) {
            char ch = line[0];
            size_t colon_pos = line.find(':');
            if (colon_pos != string::npos && colon_pos + 1 < line.size()) {
                string freq_str = line.substr(colon_pos + 1);
                try {
                    int freq = stoi(freq_str);
                    freqMap[ch] = freq;
                } catch (const std::invalid_argument&) {
                    cerr << "Invalid frequency for character '" << ch << "': " <<
freq_str << "\n";
                }
            }
        }
    }
```

```
    return freqMap;
}

// Function to clean up the Huffman tree
void deleteTree(Node* root) {
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
}

int main() {
    // Specify the full path to your local frequency map file here
    string local_file_path = "C:\\Users\\dines\\Desktop\\freq.txt"; // Update
this line with your actual path
    // Read frequency map from file
    unordered_map<char, int> freqMap =
readFrequencyMap(local_file_path);

    if (freqMap.empty()) {
        cerr << "Error: Frequency map is empty. Check the file format." <<
endl;
        return 1;
    }

    // Build the Huffman tree
    Node* root = buildHuffmanTree(freqMap);
    // Generate Huffman codes
    unordered_map<char, string> codes;
    generateCodes(root, "", codes);

    // Output the codes
    cout << "Huffman Codes:\n";
    for (const auto& pair : codes) {
```

```
        cout << pair.first << ": " << pair.second << "\n";
    }
    // Calculate original and compressed sizes
    int originalSize, compressedSize;
    calculateSizes(freqMap, codes, originalSize, compressedSize);
    // Output the sizes and compression ratio
    cout << "Original Size: " << originalSize << " bits\n";
    cout << "Compressed Size: " << compressedSize << " bits\n";
    double compressionRatio = static_cast<double>(originalSize) /
compressedSize;
    cout << "Compression Ratio: " << compressionRatio << "\n";
    // Clean up the Huffman tree
    deleteTree(root);
    return 0;
}
```

5.OUTPUT:

```
Huffman Codes:
h: 111111
g: 1111101
f: 1111100
d: 111101
c: 1111001
,: 11110001
v: 111100001
I: 1111000000
 : 1110
a: 11011
A: 1111000001
p: 1101011
b: 1101010
T: 1000101110
k: 10001110
j: 1000101101
q: 1000101111
├: 1000101010
M: 1000101100
: 0
.: 10001111
s: 10110
m: 1000100
i: 10000
:: 100010100
```

```
w: 1000110
o: 10010
C: 1000101011
r: 10011
e: 1010
l: 10111
t: 11000
n: 11001
u: 1101000
y: 1101001
Original Size: 10488 bits
Compressed Size: 4124 bits
Compression Ratio: 0.393211
```

6.CONCLUSION:

In conclusion, the provided code effectively combines character frequency analysis and Huffman coding to optimize data compression. The Python script counts character occurrences across different file encodings, storing the results in a structured format. The C++ implementation then uses this frequency map to build a Huffman tree, generating unique codes that significantly reduce data size.