Name: Kaustubh Dhongade
Reg no.: 231070018
Batch: A

DAA-Assigment-5

Exp_1: *Fraction Kanpsack*

Aim: Consider a XYZ courier company. They receive different goods to transport to different cities. Company needs to ship thegoods based on their life and value. Goods having less shelf life and high cost shall be shipped earlier. Consider list of 100 such items and capacity of transport vehical is 200 tones. Implement Algorithm for fractional knapsack problem.

Code:

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

struct Item {
    int product_id;
    double value;
    double weight;
    int shelf_life;
    double ratio;

    Item(int id, double v, double w, int s)
        : product_id(id), value(v), weight(w), shelf_life(s) {
        ratio = (weight > 0 && shelf_life > 0) ? value / weight * (1.0 /
shelf_life) : 0;
    }
```

```
};

void generateItemsCSV(const std::string &filePath) {
    std::ofstream file(filePath);

    if (!file.is_open()) {
        std::cout << "Error: Could not create the CSV file!" << std::endl;
        return;
    }

    srand(42); // Set seed for reproducibility

    file << "Item_ID,Shelf_Life_Days,Weight_tonnes,Value_INR\n";

    for (int i = 1; i <= 100; ++i) {
        int shelf_life = rand() % 99 + 1; // Random shelf life between 1 and
99
        double weight = 5 + static_cast<double>(rand()) / RAND_MAX * (50 - 5);
// Random weight between 5 and 50
        double value = 10000 + static_cast<double>(rand()) / RAND_MAX *
(1000000 - 10000); // Random value

        file << i << "," << shelf_life << "," << weight << "," << value <<
"\n";
    }

    file.close();
}

double fractionalKnapsack(std::vector<Item> &items, double capacity,
std::vector<Item> &selected_items) {
    std::sort(items.begin(), items.end(), [](const Item &a, const Item &b) {
        return a.ratio > b.ratio;
    });

    double total_value = 0.0;

    for (const auto &item : items) {
        if (capacity <= 0) break;

        if (item.weight <= capacity) {
            total_value += item.value;
            selected_items.push_back(item);
            capacity -= item.weight;
        } else {
            double fraction = capacity / item.weight;
            total_value += item.value * fraction;
```

```
            selected_items.push_back(Item(item.product_id, item.value *
fraction, capacity, item.shelf_life));
            break;
        }
    }

    return total_value;
}

int main() {
    const std::string csvFilePath = "fractional_knapsack_shipping.csv";

    // Generate the items CSV
    generateItemsCSV(csvFilePath);

    // Read the CSV file and create Item objects
    std::ifstream file(csvFilePath);
    std::vector<Item> items;

    if (file.is_open()) {
        std::string line;
        std::getline(file, line); // Skip header

        while (std::getline(file, line)) {
            int id, shelf_life;
            double weight, value;
            sscanf(line.c_str(), "%d,%d,%lf,%lf", &id, &shelf_life, &weight,
&value);
            items.emplace_back(id, value, weight, shelf_life);
        }

        file.close();
    }

    double capacity = 200.0;
    std::vector<Item> selected_items;

    // Fractional knapsack algorithm
    double max_value = fractionalKnapsack(items, capacity, selected_items);

    // Print results
    std::cout << "\nMaximum value that can be accommodated in the knapsack: "
<< max_value << "\n";
    std::cout << "Selected items:\n";
    for (const auto &item : selected_items) {
        std::cout << "Product ID: " << item.product_id
                  << ", Value: " << item.value
                  << ", Weight: " << item.weight
```

```
                  << ", Shelf Life: " << item.shelf_life << "\n";
    }

    return 0;
}
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\dsa> cd 'd:\dsa\output\DAA-Assign-5\output'
PS D:\dsa\output\DAA-Assign-5\output> & .\'knapsack.exe'

Maximum value that can be accommodated in the knapsack: 8.71924e+006
Selected items:
Product ID: 8, Value: 783673, Weight: 45.7688, Shelf Life: 1
Product ID: 28, Value: 175086, Weight: 8.65719, Shelf Life: 2
Product ID: 87, Value: 993565, Weight: 11.4039, Shelf Life: 12
Product ID: 96, Value: 941688, Weight: 13.685, Shelf Life: 11
Product ID: 30, Value: 790501, Weight: 8.28501, Shelf Life: 23
Product ID: 24, Value: 979123, Weight: 29.297, Shelf Life: 9
Product ID: 21, Value: 751375, Weight: 14.1121, Shelf Life: 15
Product ID: 94, Value: 680132, Weight: 29.0841, Shelf Life: 7
Product ID: 38, Value: 925887, Weight: 7.19733, Shelf Life: 46
Product ID: 12, Value: 951054, Weight: 15.8576, Shelf Life: 22
Product ID: 83, Value: 747152, Weight: 16.652, Shelf Life: 18
PS D:\dsa\output\DAA-Assign-5\output>
```

Exp_2: *Huffman Coding*

Aim: Download books from the website in html, text, doc, and pdf format. Compress these books using Hoffman coding technique. Find the compression ratio.

Code :

```
#include <iostream>
#include <fstream>
 #include <string>
 #include <unordered_map>
 #include <queue>
 #include <vector>
 #include <bitset>
 using namespace std;
```

```
// Define a node for the Huffman Tree
struct HuffmanNode
{
char character;
int frequency;
HuffmanNode *left;
HuffmanNode *right;
HuffmanNode(char ch, int freq) : character(ch), frequency(freq),
left(nullptr), right(nullptr) {}
};
// Comparator for the priority queue (min-heap)
struct Compare
{
   bool operator()(HuffmanNode* a, HuffmanNode* b)
{
   return a->frequency > b->frequency;
}
};
// Function to build the Huffman Tree
HuffmanNode* buildHuffmanTree(const string& text)
{
unordered_map<char, int> frequencyMap;
// Calculate frequency of each character
for (char ch : text)
{
if (isalpha(ch))
{ // Only letters
frequencyMap[ch]++;
}
}
// Priority queue to build the tree based on frequency
priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> minHeap;
// Insert each character and its frequency as a node into the minHeap
for (const auto& pair : frequencyMap)
{
minHeap.push(new HuffmanNode(pair.first, pair.second));
}
// Build the Huffman Tree
while (minHeap.size() > 1)
{
HuffmanNode* left = minHeap.top(); minHeap.pop();
HuffmanNode* right = minHeap.top(); minHeap.pop();
HuffmanNode* merged = new HuffmanNode('\0', left->frequency +
right->frequency);
merged->left = left;
merged->right = right;
minHeap.push(merged);
}
return minHeap.top();
}
// Function to generate Huffman Codes
void generateCodes(HuffmanNode* root, const string& code,
unordered_map<char, string>& huffmanCodes)
```

```
 {
 if (!root) return;
 if (root->character != '\0')
 { // Leaf node
 huffmanCodes[root->character] = code;
 }
 generateCodes(root->left, code + "0", huffmanCodes);
 generateCodes(root->right, code + "1", huffmanCodes);
 }
 // Function to compress text using Huffman coding
 string compress(const string& text, unordered_map<char, string>&huffmanCodes)
 {
 string compressedText;
 for (char ch : text)
 {
 if (huffmanCodes.find(ch) != huffmanCodes.end())
{
 compressedText += huffmanCodes[ch];
}
 }
 return compressedText;
}
 // Function to calculate compression ratio
 void calculateCompressionRatio(const string& originalText, const
string&compressedText)
 {
 int originalSize = originalText.length() * 8; // Each character is 8bits
 int compressedSize = compressedText.length(); // Already in bits
 cout << "Original size (in bits): " << originalSize << endl;
 cout << "Compressed size (in bits): " << compressedSize << endl;
 cout << "Compression ratio: " << (double)compressedSize / originalSize
 << endl;
 }
 int main() {
 string text;
 cout << "Enter the text to compress: ";
 getline(cin, text);
 // Build Huffman Tree and generate codes
 HuffmanNode* root = buildHuffmanTree(text);
 unordered_map<char, string> huffmanCodes;
 generateCodes(root, "", huffmanCodes);
 // Display Huffman Codes for each letter
 cout << "Huffman Codes:\n";
 for (const auto& pair : huffmanCodes) {
 cout << "'" << pair.first << "': " << pair.second << endl;
 }
 // Compress the text
 string compressedText = compress(text, huffmanCodes);
 // Display compression ratio
 calculateCompressionRatio(text, compressedText);
return 0;
}
```

Output:

```
PS D:\dsa\output\DAA-Assign-5\output> & .\'huffman.exe'
Enter the text to compress: qqweqwewewqeweqweweqweqweqweqweqqqqeeeeewwwwww
Huffman Codes:
'e': 11
'q': 10
'w': 0
Original size (in bits): 392
Compressed size (in bits): 80
Compression ratio: 0.204082
PS D:\dsa\output\DAA-Assign-5\output>
```

Conclusion: In the first experiment, we implemented an algorithm to solve the fractional knapsack problem for XYZ courier company. By prioritizing goods based on shelf life and value, we optimized the selection of items for transport within a 200-ton capacity. The approach ensured that items with shorter shelf lives and higher costs were shipped first, enhancing logistics efficiency and reducing waste • In the second experiment, we downloaded books in various formats (HTML, text, DOC, and PDF) and applied Huffman coding for compression. This technique successfully reduced file sizes, leading to efficient storage and faster download speeds. The compression ratios indicated significant space savings, demonstrating the effectiveness of Huffman coding.