DAA LAB ASSIGNMENT 05

Name : Atharva Dhage

Branch : S.Y.B.Tech Computer

Reg. No : 23107017

Task-1:

1. **Learn About Debugging Tools**
   Research debugging tools available for your programming language. Practice debugging your code by applying basic debugging techniques. Experiment with step-by-step execution, inspect variables, and identify any errors using these techniques.

2. **Basic Debugging Techniques**
   Step Execution (Stepping Through Code):
   Step execution is a debugging technique that lets the developer run code line by line. This method allows you to observe how the program behaves in detail, which can help pinpoint where issues occur. Key types of step execution commands include:

   - **Step Into:** Executes the current line and moves into the next function call if there is one. This lets you examine the details of a function's behavior line by line.
   - **Step Over:** Executes the next line in the current function without stepping into any functions called on that line. Use this when you trust the function works correctly and want to see how it integrates with the rest of the code.
   - **Step Out:** Runs the remainder of the current function until it completes, then returns control to the calling function. This is useful when you've entered a function but want to skip further inspection and resume higher-level execution.

Why Use Step Execution?

- **Pinpoint Errors:** Step execution allows you to monitor the flow of your program, making it easier to find the exact location of issues.
- **Understand Logic:** Stepping through the code helps verify that each part of the program behaves as expected, ensuring correct logical flow.

3. **Variable Inspection (Monitoring Variables):**
   Monitoring variables during program execution allows you to observe their values at specific points, which is essential for detecting bugs caused by unexpected or incorrect values.

Types of Variable Inspection:

- **Watch Variables:** Many debuggers offer the ability to "watch" specific variables, so you’re alerted whenever a variable’s value changes. This can help trace bugs caused by unexpected state changes.
- **Hover Inspection (In IDEs):** Some IDEs allow you to hover over variables while debugging to instantly view their current values, providing quick insights.
- **Print/Display Variables:** Debugging tools like GDB let you manually print or display variable values at specific points with commands like print variable_name or display variable_name. This is useful for examining variable values at specific moments.

Why Use Variable Inspection?

- **Trace Logical Errors:** Logical errors are often caused by incorrect variable values. Monitoring these values lets you check the program's state at various points.
- **Memory Management:** In C++, managing memory manually is common. Inspecting pointers and dynamically allocated memory helps avoid issues like memory leaks, null pointer dereferences, or invalid accesses.

4. **C++ Debugging Tools**
   - **GDB (GNU Debugger)**
   - **LLDB**
   - **Visual Studio Debugger**
   - **Valgrind**
   - **Coverity**
   - **Clang Static Analyzer**

Code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Function to calculate the maximum value for the fractional knapsack problem
double fractionalKnapsack(vector<int>& weights, vector<int>& values,
vector<int>& shelfLife, int capacity) {
    // Error handling for invalid inputs
    if (weights.size() != values.size() || values.size() != shelfLife.size()) {
        cout << "Error: All input vectors (weights, values, shelfLife) must have
the same size." << endl;
        return -1;  // Return an error code
    }
    if (capacity < 0) {
        cout << "Error: Capacity cannot be negative." << endl;
        return -1;  // Return an error code
    }

    for (int i = 0; i < weights.size(); i++) {
        if (weights[i] < 0 || values[i] < 0 || shelfLife[i] < 0) {
            cout << "Error: Weights, values, and shelf life must be non-
negative." << endl;
            return -1;  // Return an error code
        }
    }

    // Vector to store items with (shelfLife/value, weight, value)
    vector<pair<double, pair<int, int>>> items;
    int n = weights.size();
```

```
    // Calculate shelfLife/value ratio and store it with corresponding weight
and value
    for (int i = 0; i < n; i++) {
        // Prevent division by zero for value
        if (values[i] == 0) {
            cout << "Error: Values cannot be zero as it results in undefined
ratio." << endl;
            return -1;  // Return an error code
        }
        double ratio = static_cast<double>(values[i]/(shelfLife[i]*weights[i]));
        items.push_back({ratio, {weights[i], values[i]}});
    }

    // Sort items based on the shelfLife/value ratio in descending order
    sort(items.begin(), items.end(), [](const auto& a, const auto& b) {
        return a.first > b.first;
    });

    double currentWeight = 0;
    double currentValue = 0;

    // Iterate over the sorted items
    for (int i = 0; i < items.size(); i++) {
        int w = items[i].second.first;
        int v = items[i].second.second;

        // If adding the whole item does not exceed capacity, add it
        if (currentWeight + w <= capacity) {
            currentWeight += w;
            currentValue += v;
        }
        // Else add the fraction of the remaining capacity and break
        else {
            double remain = capacity - currentWeight;
            currentValue += v * (remain / w);
            break;
        }
    }

    return currentValue;
}

int main() {
    vector<int> weights = {10, 20, 30};      // Example weights
    vector<int> values = {60, 100, 120};     // Example values
    vector<int> shelfLife = {2, 1, 4};       // Example shelf lives
    int capacity = 50;                       // Example capacity

    double maxValue = fractionalKnapsack(weights, values, shelfLife, capacity);
    if (maxValue != -1) {  // Check if an error code was returned
        cout << "Maximum value in knapsack: " << maxValue << endl;
```

```
    } else {
        cout << "Failed to calculate the maximum value due to input errors." <<
endl;
    }

    return 0;
}
```

Output

```
PS D:\daa lab> cd "d:\daa lab\" ; if ($?) { g++ knapsack.cpp -o knapsack } ;
if ($?) { .\knapsack }
Maximum value in knapsack: 230
PS D:\daa lab> cd "d:\daa lab\" ; if ($?) { g++ knapsack.cpp -o knapsack } ;
if ($?) { .\knapsack }
Maximum value in knapsack: 145
PS D:\daa lab> cd "d:\daa lab\" ; if ($?) { g++ knapsack.cpp -o knapsack } ;
if ($?) { .\knapsack }
Maximum value in knapsack: 25.2857
PS D:\daa lab> cd "d:\daa lab\" ; if ($?) { g++ knapsack.cpp -o knapsack } ;
if ($?) { .\knapsack }
Error: Weights, values, and shelf life must be non-negative.
Failed to calculate the maximum value due to input errors.
PS D:\daa lab> cd "d:\daa lab\" ; if ($?) { g++ knapsack.cpp -o knapsack } ;
if ($?) { .\knapsack }
Error: Capacity cannot be negative.
Failed to calculate the maximum value due to input errors.
PS D:\daa lab>
```

Huffman Tree

Code:

```cpp
#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 8 bits
    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;
}
```

```cpp
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:\Atharva\New folder> cd "d:\Atharva\New folder\" ; if ($?) { g++
Huffman.cpp -o Huffman } ; if ($?) { .\Huffman }
Enter the text to compress: Hello Huffman

Huffman Codes:
'k': 111
'x': 110
‘t’: 01
'o': 10
‘b’: 00
Original size (in bits): 96
Compressed size (in bits): 47
Compression ratio: 0.4896

PS D:\Atharva\New folder>
```

Conclusion:

In this lab assignment, we covered essential debugging techniques and implemented two classic algorithms: Fractional Knapsack and Huffman Coding.

Debugging techniques like step execution and variable inspection helped identify errors and verify logic, enhancing code quality and reliability.

The Fractional Knapsack problem illustrated the use of greedy algorithms to maximize value within constraints, with added error handling for robustness.

The Huffman Coding algorithm demonstrated efficient data compression by assigning shorter codes to frequent characters, reducing data size without loss, a technique widely used in multimedia encoding.

Overall, these exercises underscore the importance of debugging and the effectiveness of classical algorithms in solving optimization and compression challenges.