**Name :- Nilanshu Ranjan**
**Reg.No.:- 231070035**
**Class :- S.Y.B.Tech**

# *DAA LaB-05*

Debugging is a key part of programming, and using the right tools and techniques can make it much more efficient. For C++ (and similar languages), here are some common debugging tools and techniques that could help you in your process:

### 1. Debugging Tools

- **GDB (GNU Debugger)**: A powerful debugger for C and C++ programs. GDB allows you to run your code line by line, inspect variable values at different execution points, and set breakpoints to stop execution at specific points.
- **Integrated Debuggers in IDEs**: Many IDEs like **Visual Studio**, **CLion**, or **Code::Blocks** have built-in debuggers. These tools offer features like setting breakpoints, watching variables, and stepping through code, making the debugging process more visual and intuitive.
- **LLDB**: Part of the LLVM project, it's an alternative debugger to GDB, especially popular on macOS and with modern C++ standards.

### 2. Debugging Techniques

- **Step Execution (Step Over, Step Into, Step Out)**:
  - **Step Over**: Runs the current line of code and moves to the next line in the same function, skipping over function calls without entering them.
  - **Step Into**: Enters a function call, allowing you to debug the code inside it.
  - **Step Out**: Completes execution of the current function and returns to the calling function.
  - These commands help you control the pace of execution to investigate where the code may be behaving unexpectedly.
- **Variable Inspection**:
  - Inspecting variables at each step lets you see their current values and track any unexpected changes. IDE debuggers and GDB offer a "watch" feature, which allows you to monitor specific variables throughout your code's execution.

    - You can also print variable values at checkpoints using print statements if you aren't using a debugger.
- **Conditional Breakpoints**:
    - If you suspect an issue with a specific condition or loop, setting conditional breakpoints can be helpful. This means the program will only pause when certain conditions are met, saving you time by skipping over iterations where the issue isn't present.
- **Memory Analysis**:
    - C++ programs often have issues with memory, like segmentation faults. Tools like **Valgrind** (for Linux) can help detect memory leaks or invalid memory accesses, helping you identify issues with dynamic memory usage.

### 3. Error Identification

- By stepping through each part of the code and inspecting variables, you can catch:
    - Incorrect variable values or unexpected state changes.
    - Misbehaving loops, especially infinite loops or ones that exit prematurely.
    - Segmentation faults, often due to accessing invalid memory or null pointers.
    - Logic errors, where values do not match expected results.

### Example Debugging Process

Let's say you're debugging a simple C++ function that performs arithmetic operations. Here's how you could proceed:

1. **Set Breakpoints**: Set a breakpoint at the beginning of the function to pause and begin debugging from that point.
2. **Step Through the Code**: Use **step over** for non-essential lines and **step into** for any function calls to observe their behavior.
3. **Inspect Variables**: At each step, check the values of variables and see if they match expectations.
4. **Apply Conditional Breakpoints**: If the function contains a loop, set a conditional breakpoint for specific cases to catch problematic iterations.
5. **Detect Memory Issues**: Run tools like Valgrind if you suspect memory issues like invalid accesses or leaks.

PAGE No.
DATE / /

* Algorithm for fractional knapsack

```
// Computes Fractional Knapsack (W, items)
// input  W i.e. maximum weight a knapsack can
          hold
// output:  maximum achiavable value

for each item in items :
      value-to-weight-ratio = (item-value)
                               item. weight)

Sort items in descending order by value to weight
   ratio

total value =0

for each item in sorted items
  If item. when weight <=W :
        W= W - item. weight
        total-value = total-value + item. value
else
      fraction = W/ item. weight
      total-value = total-value + (item value x fraction)
        W=0
      break

return total-value
```

PAGE No.
DATE / /

* Time Complexity Analysis

① Brute Force Approach : Test all combinations

① ~~Greedy~~ for n items , $2^n$ subsets are possible

② for each ~~element~~ subset ~~&~~ $O(n)$ time is needed for value calculation

$\therefore$ Time Complexity $= O(n \cdot 2^n)$

This is not feasible for larger n

② Greedy approach:

In optimal approach, we first sort ~~the~~ in order of decreasing value to shelf life ratios

So, sorting takes $O(n \log n)$

After sorting $O(n)$ is needed for calculation of total value

So, time complexity $= O(n \log n) + O(n)$

$TC \simeq O(n \log n)$

* Test Cases

Considering knapsack capacity as const. i.e. 20

| | Item no. | Item-value | Item-weight | Item-shelf | Output |
|---|---|---|---|---|---|
| 1) | 1 | 10 | 60 | 5 | 53.33 |
| | 2 | 20 | 80 | 10 | |
| | 3 | 30 | 100 | 3 | |
| 2) | 1 | 5 | 50 | 1 | 40 |
| | 2 | 20 | 90 | 2 | |
| | 3 | 15 | 50 | 4 | |
| 3) | 1 | 5 | 30 | 8 | 50 |
| | 2 | 10 | 10 | 6 | |
| | 3 | 20 | 100 | 5 | |
| | 4 | 15 | 60 | 4 | |
| 4) | 1 | 10 | 100 | 3 | 20 |
| | 2 | 10 | 100 | 2 | |
| | 3 | 10 | 100 | 1 | |
| 5) | 1 | 5 | 50 | 3 | 25 |
| | 2 | 8 | 60 | 1 | |
| | 3 | 12 | 90 | 2 | |
| 6) | 0 | 0 | 0 | 0 | list empty |
| 7) | 1 | 5 | 30 | 3 | capacity cannot be zero |
| | 2 | 9 | 50 | 2 | |
| | 3 | 8 | 40 | 3 | |

taking knapsack capacity to zero

| | | | | | |
|---|---|---|---|---|---|
| 8Y | 1 | 10 | 0 | 5 | 0 |
| | 2 | 20 | 0 | 3 | |
| 9Y | 1 | 0 | 10 | 5 | weight cannot be zero |
| | 2 | 0 | 20 | 3 | |
| 10Y | 1 | 300 | 1000 | 5 | 60 |
| 11Y | 1 | -100 | 10 | 6 | weight can't be -ve |
| 12Y | 1 | +10 | -10 | 1 | value can't be -ve |

PAGE No.
DATE / /

* Algorithm - Huffman Coding

1) Calculate Frequencies

For each character in input
~~If ch~~
If character is in frequency - table
increment frequency of characters in frequency-table

else
add character to frequency-table with frequency = 1

2) Initialize Priority queue

For each character & frequency in frequency-table
create a node with character & frequency
Add node ~~in~~ to priority-queue

3) Build the huffman tree

while (priority-queue has more than one node)
left_node = remove node with lowest frequency
from priority-queue
right-node = remove node with next lowest frequency
from priority queue

Create new-node with:
frequency = left-node. frequency + right-node. frequency
left = left-node
right = right-node

Add new-node to priority-queue
END while

Generate-code (node, current-code)

If node is NULL
    Return
If node is a leaf (node.character is not NULL)
SET huffman-code [node.character] = current-cu

else
    generate-code (node.left, current-code + "0")
    generate-code (node.right, current-code + "1")

PAGE No. DATE / /

* Time Complexity

1. Brute force

1) Generate binary strings of lengths $k = 2^m$

2) for each character 'm' total no. of ~~subsets~~ will be
$$\sum_{k=1}^{m} 2^k = 2^{m+1} - 2 \approx O(2^m)$$

Comparing one code with another will involve nested loops adding $n^2$ cost

3) also, for each subset, we have $O(k^2)$ Checks thus
$$T.C. = O\left(\sum_{k=1}^{m} \binom{2^m}{k} \cdot k^2\right)$$

largest check will ocur at $k = m$
thus, $TC = O(2^m, m^2)$

2. Greedy Approach

1) Counting frequency of every letter using has $O(n)$

2) To build priority queue from each unique character k, we will have
$$O(k \log k)$$

PAGE No.
DATE / /

for merging, to the loop runs (k-1) times, $O(k)$

Overall

$$T.C = O(k \log k)$$

* Test Cases :

| File type | Original text size | Compressed | Compression ratio |
|---|---|---|---|
| TxT | 1016 | 441 | 0.43 |
| DOCX | 50216 | 21390 | 0.43 |
| HTML | 50480 | 21433 | 0.42 |
| PDF | 2680 | 526 | 0.43 |
| md | error-type not supported | | |
| Pdf (empty) | no-content extracted | | |
| Txt (non existing) | no file | | |

Program :-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
#include <random>

using namespace std;

// Class to represent an item in the knapsack
class Item {
public:
    int weight;
    int value;
    int shelf_life;

    // Constructor
    Item(int value, int weight, int shelf_life) {
        if (weight <= 0) {
            throw invalid_argument("Invalid weight: " + to_string(weight)
+ ". Weight of an item cannot be zero or negative.");
        }
        if (value < 0) {
            throw invalid_argument("Invalid value: " + to_string(value) +
". Value of an item cannot be negative.");
        }
        if (shelf_life <= 0) {
            throw invalid_argument("Invalid shelf life: " +
to_string(shelf_life) + ". Shelf life must be greater than zero.");
        }
        this->weight = weight;
        this->value = value;
        this->shelf_life = shelf_life;
    }

    // Utility function to calculate the value to weight ratio
    double value_weight_ratio() const {
        return static_cast<double>(value) / weight;
    }
```

```cpp
    // Utility function to combine shelf life and value-weight ratio in
comparison
    pair<double, int> priority() const {
        return { value_weight_ratio(), -shelf_life }; // Higher
value-weight, lower shelf life prioritized
    }
};

// Function to implement the fractional knapsack problem
double fractional_knapsack(int capacity, vector<Item>& items) {
    if (capacity <= 0) {
        throw invalid_argument("Capacity of the knapsack must be greater
than zero.");
    }
    if (items.empty()) {
        throw invalid_argument("Item list cannot be empty.");
    }

    // Sort items by priority: value-weight ratio (desc) and shelf life
(asc)
    sort(items.begin(), items.end(), [](const Item& a, const Item& b) {
        return a.priority() > b.priority();
    });

    double total_value = 0.0; // Total value accumulated
    int current_weight = 0;   // Current weight of the knapsack

    for (const Item& item : items) {
        if (current_weight + item.weight <= capacity) {
            // Take the whole item
            current_weight += item.weight;
            total_value += item.value;
        } else {
            // Take fraction of the item
            int remaining_capacity = capacity - current_weight;
            double fraction = static_cast<double>(remaining_capacity) /
item.weight;
            total_value += item.value * fraction;
            break; // Knapsack is full
        }
```

```
    }

    return total_value;
}

// Test case runner with error handling
void run_test_case(const vector<Item>& items, int capacity) {
    try {
        // Print the items with their weights, values, and shelf lives
        cout << "\nItems (Weight, Value, Shelf Life):\n";
        for (size_t i = 0; i < items.size(); ++i) {
            cout << "Item " << i + 1 << ": Weight = " << items[i].weight
                 << ", Value = " << items[i].value
                 << ", Shelf Life = " << items[i].shelf_life << '\n';
        }

        // Calculate and print the total knapsack value
        double result = fractional_knapsack(capacity,
const_cast<vector<Item>&>(items));
        cout << "Knapsack Value for " << items.size() << " items: " <<
result << "\n\n";

    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << '\n';
    }
}

int main() {
    // Test cases
    vector<Item> items_1 = { Item(10, 60, 5), Item(20, 80, 10), Item(30,
100, 3) };
    run_test_case(items_1, 200);

    vector<Item> items_2 = { Item(5, 50, 1), Item(20, 90, 2), Item(15, 50,
4) };
    run_test_case(items_2, 200);

    vector<Item> items_3 = { Item(5, 30, 8), Item(10, 10, 6), Item(20,
100, 2), Item(15, 60, 5) };
    run_test_case(items_3, 200);
```

```
    // Negative Test Cases with proper error handling
    cout << "\nNegative Test Cases\n";

    // Empty item list
    vector<Item> items_6;
    run_test_case(items_6, 200);  // Expected: Error for empty item list

    // Knapsack capacity is zero
    vector<Item> items_7 = { Item(5, 30, 3), Item(10, 50, 5) };
    run_test_case(items_7, 0);  // Expected: Error for zero capacity

    // Item with zero weight
    try {
        vector<Item> items_8 = { Item(10, 0, 5) };  // Invalid item
        run_test_case(items_8, 200);
    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << '\n'; // Expected: Error for zero
weight
    }

    // Item with negative value
    try {
        vector<Item> items_9 = { Item(-10, 5, 5) };  // Invalid item
        run_test_case(items_9, 200);
    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << '\n'; // Expected: Error for
negative value
    }

    // Item with zero shelf life
    try {
        vector<Item> items_10 = { Item(10, 5, 0) };  // Invalid item
        run_test_case(items_10, 200);
    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << '\n'; // Expected: Error for zero
shelf life
    }

    return 0;
```

```
}
```

Output :-

```
PS C:\Users\Vedant\OneDrive\Desktop\CPP> cd 'c:\Users\Vedant\OneDrive\Desktop\CPP\output'
PS C:\Users\Vedant\OneDrive\Desktop\CPP\output> & .\'Knapsack.exe'

Items (Weight, Value, Shelf Life):
Item 1: Weight = 60, Value = 10, Shelf Life = 5
Item 2: Weight = 80, Value = 20, Shelf Life = 10
Item 3: Weight = 100, Value = 30, Shelf Life = 3
Knapsack Value for 3 items: 53.3333


Items (Weight, Value, Shelf Life):
Item 1: Weight = 50, Value = 5, Shelf Life = 1
Item 2: Weight = 90, Value = 20, Shelf Life = 2
Item 3: Weight = 50, Value = 15, Shelf Life = 4
Knapsack Value for 3 items: 40


Items (Weight, Value, Shelf Life):
Item 1: Weight = 30, Value = 5, Shelf Life = 8
Item 2: Weight = 10, Value = 10, Shelf Life = 6
Item 3: Weight = 100, Value = 20, Shelf Life = 2
Item 4: Weight = 60, Value = 15, Shelf Life = 5
Knapsack Value for 4 items: 50
```

```
Negative Test Cases

Items (Weight, Value, Shelf Life):
Error: Item list cannot be empty.

Items (Weight, Value, Shelf Life):
Item 1: Weight = 30, Value = 5, Shelf Life = 3
Item 2: Weight = 50, Value = 10, Shelf Life = 5
Error: Capacity of the knapsack must be greater than zero.
Error: Invalid weight: 0. Weight of an item cannot be zero or negative.
Error: Invalid value: -10. Value of an item cannot be negative.
Error: Invalid shelf life: 0. Shelf life must be greater than zero.
```

```
#include <iostream>
```

```cpp
#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;
}

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;
}
```

```
PS C:\Users\Vedant\OneDrive\Desktop\CPP> cd 'c:\Users\Vedant\OneDrive\Desktop\CPP\output'
PS C:\Users\Vedant\OneDrive\Desktop\CPP\output> & .\'Huffman.exe'
Enter the text to compress: book.txt
Huffman Codes:
'k': 111
'x': 110
't': 01
'o': 10
'b': 00
Original size (in bits): 64
Compressed size (in bits): 16
Compression ratio: 0.25
PS C:\Users\Vedant\OneDrive\Desktop\CPP\output>
```