Name :- Ashutosh Kumar
Reg.No.:- 231070006
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.

DAA-Lab 5

Code: For fractional Knapsack

```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) {
    // 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++) {
        double ratio = static_cast<double>(shelfLife[i]) / values[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;
    vector<int> values;
    vector<int> shelfLife;
    weights = {10, 20, 30};     // Example weights
    values = {60, 100, 120};    // Example values
    shelfLife = {2, 3, 5};      // Example shelf lives
    int capacity;
    capacity = 50;                      // Example capacity
    double maxValue = fractionalKnapsack(weights, values, shelfLife,
capacity);
    cout << "Maximum value in knapsack: " << maxValue << endl;

    weights = {60, 80, 100};     // Example weights
    values = {10, 20, 30};    // Example values
    shelfLife = {5, 10, 3};      // Example shelf lives
    capacity = 50;                      // Example capacity
     maxValue = fractionalKnapsack(weights, values, shelfLife, capacity);
    cout << "Maximum value in knapsack: " << maxValue << endl;

    weights = {50, 90, 50};     // Example weights
    values = {5, 20, 15};    // Example values
    shelfLife = {1, 2, 4};      // Example shelf lives
    capacity = 50;                      // Example capacity
     maxValue = fractionalKnapsack(weights, values, shelfLife, capacity);
    cout << "Maximum value in knapsack: " << maxValue << endl;


    return 0;
}
```

Output:

```
Maximum value in knapsack: 230

Maximum value in knapsack: 8.33333
Maximum value in knapsack: 15
```

DAA Lab 5

Name :Ashutosh Kumar
Batch : A
Reg Num: 231070006

EXPERIMENTAL TASK 1

AIM: Consider a XYZ courier company. They receive different goods to transport to different cities. Company needs to ship the goods 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 vehicle is 200 tones. Implement Algorithm for fractional knapsack problem.

ALGORITHM:

EXPERIMENT : _____ No. Date

* Algorithm :-

Fractional knapsack Brute - force

Shelf life[]

Function Fractionalknapsack (weights[], values[], capacity):

~~A = Weight~~

// Input :- 1) Array of Weights of each item
2) Array of values of each item
3) Array of shelf life
4) Total capacity

n = number of items
max_value = 0

// Generate all subsets of items
for each subset of items :
subset_weight = 0
subset_value = 0

// calculate total weight and value for this subset
for each item in subset

if subset_weight + weight[item] <= capacity
subset_weight += weight[item]
subset_value += values[item]
else
remaining_capacity = capacity - subset_weight
subset_value += values[item] * (remaining_capacity / weights[item])

subset_weight = capacity
break

SIGN _____

EXPERIMENT : _____ No. Date

if subset_value > max_value
max_value = subset_value

* Time Complexity :-

1) As we first generate all possible subsets of given items
∴ Generation of subsets = $O(2^n)$

2) Subset Evaluation :-
For each subset the algorithm runs for the subsets size times ∴ $O(n)$

3) Total Time Complexity :-
$O(n.2^n)$

SIGN

EXPERIMENT : _____ No. Date

Fractional Knapsack - Greedy Approach

```
function FractionalKnapsack (Weights[], Values[], shelf life[],
                             capacity )

// Input :- 1> Array of weight of each item
            2> Array of value of each item
            3> Array of shelflife of each item
            4> Total capacity
// output :- Maximum value
// sort according to shelf life / value ratio
  vector < pair <double, pair <int, int >>> V
    for each item in weights
    for ( i ← 0 ; i < Weights.size() ; i++ ) :
        double ratio = shelflife [i] / value [i]
        V ← { ratio , { Weights[i], Values [i] } }

// Sort vector V
  sort €
    current_weight = 0
    current_value  = 0

  For ( i ← 0 ; i < v.size() ; i++ ) :
        W = V [i]. second . first
    If V = v [i]. second . second
      if ( W + current_weight >/< = capacity
            current_value + = V
```

SIGN. _____

EXPERIMENT : _____ No. Date

```
else :
    remaining_capacity = capacity - current_weight
    currentvalue += v * (remaining_capacity / W)

    current_weight = capacity
    Break.

return current value
```

* Time complexity :-

1) creating a vector pair of ratio of shelflife / value and Weight and value of each item
   = O(n)
2) Sorting This created vector
   = O(n log n)
3) Calculating answer using greedy approach
   = O(n)

Total time complexity :-
= O(2n + n log n)
= O(n log(n))

SIGN. _____

CONCLUSION:

In conclusion, the implementation of the fractional knapsack algorithm for the XYZ Courier Company effectively optimized the shipping process by prioritizing goods based on their shelf life and value. The developed code was robust, handling various edge cases such as negative values, missing data, and empty files gracefully, by providing clear error messages and halting execution when necessary. The program efficiently sorted items to maximize value within the vehicle's capacity of 200 tons, ensuring that high-value, short-shelf-life items were shipped first. Overall, this solution not only streamlined operations but also enhanced the company's ability to meet delivery deadlines, ultimately improving customer satisfaction and profitability in a competitive logistics landscape.

Name : Ashutosh Kumar

Reg. Number: 231070006

Batch : A

DAA Lab 5 Hoffman

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;
}
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:

```
Enter the text to compress: ashutosh.txt
Huffman Codes:
'o': 1111
'x': 1110
Huffman Codes:
'o': 1111
'x': 1110
'x': 1110
'u': 000
'a': 001
's': 110
'h': 01
't': 10
Original size (in bits): 96
Compressed size (in bits): 30
Compression ratio: 0.3125

ashutosh kumar@Ashutosh-PC MINGW64 /e/programs/code forces/output
$
```

Algorithm : Huffman Coding

(1) Calculate frequency for each Character in input

If Character is in frequency-table - increment frequency of Characters in frequency-table.

else Add Character to frequency-table with frequency as 1.

(2) Initialize Priority queue
For each character & frequency in frequency-table Create a node with character & frequency Add node 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.

Creak 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-Code

else
generate-Code (node.left, Current-code + "0")
Generate-Code (node.right, Current-Code + "1")

* Time Complexity:

1) Brute force:
Generate binary String of length $K = 2^n$

2) for each character 'm' total no. of Subsets

$$\sum_{K=1}^{m} 2^K = 2^{m+1} - 2 \approx O(2^m)$$

Comparing one Code with another will involve nested loops adding n^2 also.

(3) Also for each subset we have $O(K^2)$ checks thus;

$$T.C = O\left(\sum_{K=1}^{m} (2^K) \cdot K^2\right)$$

largest check will occur at $K=m$
thus, $TC = O(2^m, m^2)$

(2) Greedy Approach

1) Counting frequency : $O(n)$

2) To build priority queue from each unique charcter K, we will have:

$$O(K \log K)$$

3) for merging the loops run $(K-1)$ times $O(K)$.

Overall $O(n) + O(K \log K) + O(K)$

$$TC = O(n + K + K \log K)$$

$$TC \approx O(K \log K)$$

* Test Case :

File type	Orignal text Size	Compressed	Comp. 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		