DAA LAB 5

Amit Rajkumar Ingle

S.Y.B.Tech Computer

231070020

Task-1:

Read information about debugging tools for your programing language. Try to debug your code. Apply step execution. Inspect variables and find any errors through the debugging technique.

->

DEBUGGING AND ITS IMPORATANCE:

Debugging is the process of identifying, analyzing, and removing errors or bugs from software to ensure it operates as intended. It plays a crucial role in software development by helping detect and fix mistakes, which improves code quality and enhances reliability. Effective debugging not only prevents crashes and incorrect outputs but also aids developers in understanding their code better, leading to optimized algorithms and improved performance. By employing systematic debugging techniques and tools, developers can streamline the process of finding and resolving issues, ultimately resulting in more stable and efficient software applications.

Debugging Process :

1. Identify the Bug : First, you need to notice something wrong with your program. This could be an incorrect output, a crash, or unexpected behavior.

2. Reproduce the Bug : Try to recreate the issue consistently. This ensures that you understand the conditions under which the bug occurs.

3. Isolate the Bug : Narrow down where the bug is happening by inspecting the code or using a debugger to step through the program.

4. Analyze : Examine the relevant part of the code and determine why it’s not behaving as expected. You can inspect variable values, the flow of execution, and other aspects of the program.

5. Fix the Bug : Once identified, modify the code to fix the problem.

6. Test the Fix : After the change, ensure the bug is fixed by running tests to confirm that the program works correctly without introducing new issues.

7. Document the Fix : It's good practice to document what was wrong and how you fixed it. This helps in understanding future issues that may arise.

Basic debugging techniques :

Step Execution (Stepping Through Code) :

Step execution allows the developer to run a program one line or instruction at a time, which helps to closely observe how the program behaves and isolates where an issue might be occurring. The key types of stepping commands are:

Step Into : This command allows the debugger to move into the next function call. If the current line contains a function, it will go inside that function and execute it line by line. This is useful for examining how individual functions behave.

Step Over : The debugger moves to the next line in the current function without going inside any function calls on that line. This is used when you trust a function to work correctly but want to observe how it integrates with the rest of the code.

Step Out : This continues execution until the current function finishes and returns control to the calling function. This is helpful when you've stepped into a function but want to exit and see how the program behaves at a higher level.

Why Use Step Execution ?

Pinpoint errors : Step execution helps you monitor the flow of control in your program, making it easier to identify exactly where things go wrong.

Understand logic : By stepping through code, you can verify that each part of the program is executing as expected, confirming that the logic flows correctly.

Variable Inspection (Monitoring Variables) :

Inspecting variables during runtime allows you to see the values held by different variables at specific points in the program, which is critical for catching bugs caused by incorrect or unexpected values.

Types of Variable Inspection :

Watch Variables : Some debuggers allow you to "watch" certain variables. This means you monitor the variable's value throughout the program's execution and are alerted if the value changes. This can help track down bugs related to unexpected changes in state.

Hover Inspection (In IDEs) : Many modern IDEs allow you to hover over variables during debugging to see their current values instantly. This provides quick insights without needing to run specific commands.

Print/Display Variables : Debuggers like GDB allow you to manually print or display the value of variables at any point using commands like `print variable_name` or `display variable_name` in GDB. This is particularly useful when you want to check a variable's value at specific points.

Why Use Variable Inspection ?

Trace logical errors : Incorrect variable values are often the root cause of logical errors. Monitoring these values helps you verify the program's state at various execution points.

Memory management : C++ programs often deal with manual memory management. Inspecting pointers and dynamically allocated memory can help avoid memory leaks, null pointer dereferences, or invalid accesses.

list c++ tools for debugging

1. GDB (GNU Debugger)
2. LLDB
3. Visual Studio Debugger
4. Valgrind
5. Coverity
6. Clang Static Analyzer
7. CLion Debugger
8. rr (Time Travel Debugging)
9. LiveRecorder

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.

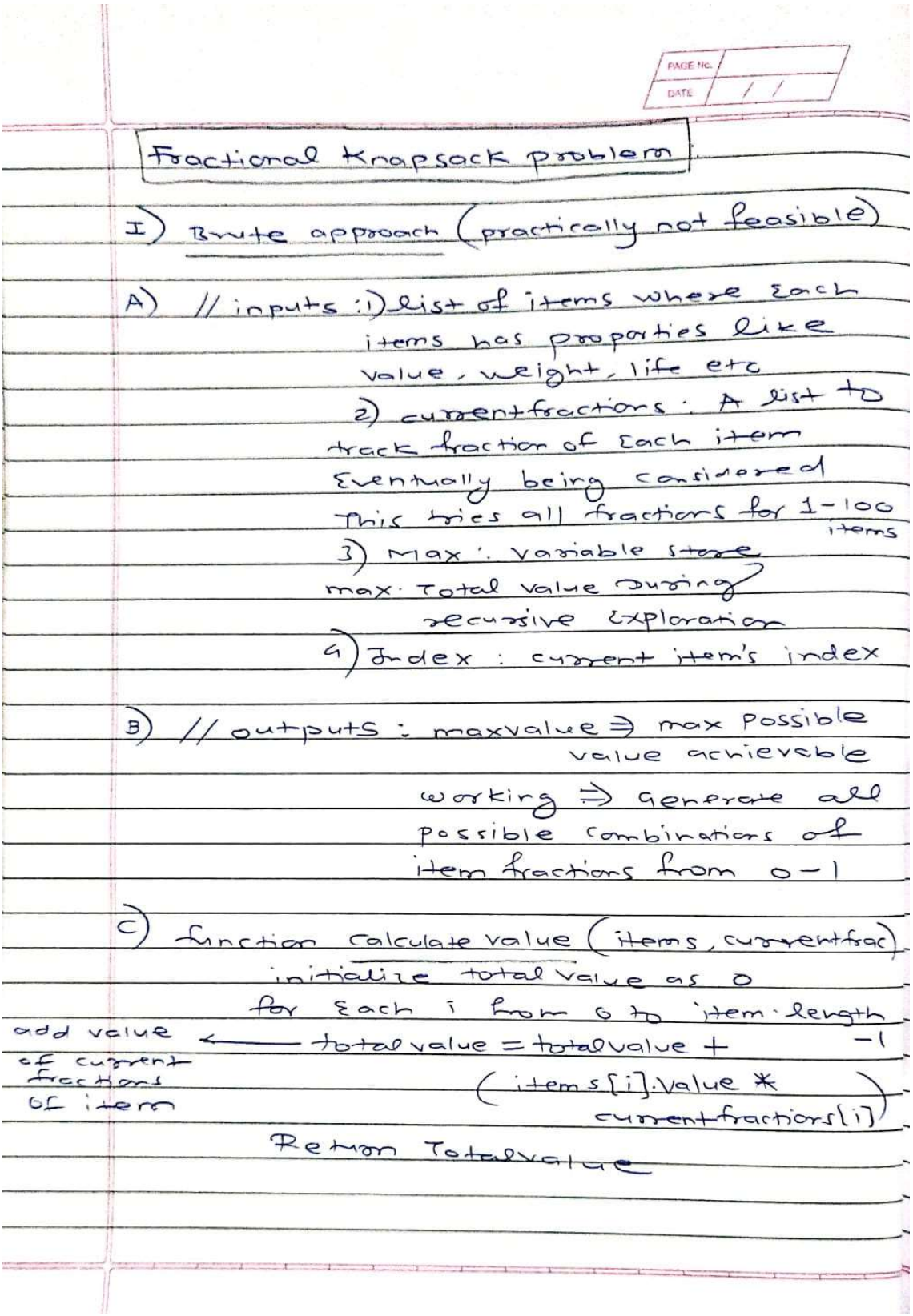
THEORY:

The Knapsack Problem is a well-known optimization problem where you have a set of items, each with a weight and value, and a knapsack with a weight capacity. The goal is to select items to maximize the total value while ensuring the total weight doesn't exceed the knapsack's capacity. It's a classic example of combinatorial optimization and has many real-world applications like resource allocation and budgeting.

- **0/1 Knapsack Problem** : You either take an item entirely or leave it, no fractions allowed. The solution is typically found using dynamic programming techniques.
- **Bounded/Unbounded Knapsack** : Variants where items can be selected a limited number of times (bounded) or an unlimited number of times (unbounded).
- **Fractional Knapsack Problem** :

The Fractional Knapsack Problem is a variation where you can take fractions of an item instead of whole items. The approach is greedy; you pick the item with the highest value-to-weight ratio and take as much of it as possible until the knapsack is full.

ALGORITHM:



PAGE No.
DATE / /

Fractional Knapsack problem:

I) Brute approach (practically not feasible)

A) // inputs : 1) list of items where each items has properties like value, weight, life etc
2) currentfractions : A list to track fraction of each item Eventually being considered This tries all fractions for 1-100 items
3) max : variable store max. Total value during recursive exploration
4) Index : current item's index

B) // outputs : maxvalue ⇒ max possible value achievable
working ⇒ generate all possible combinations of item fractions from 0-1

C) function calculate value (items, currentfrac)
initialize total value as 0
for each i from 0 to item.length - 1
total value = totalvalue + (items[i].value * currentfractions[i])
add value of current fractions of item
Return Totalvalue

PAGE No.
DATE / /

function fractionalKnapsackBrute (items, currentFractions, maxValue, index)

if index >= items.length() // Base case

currentValue = calculateValue (items, currentFractions)

if (currentValue > maxValue)

maxValue = currentValue

// update max value

// consider each possible fractions of current item

for fraction from 0.0 to 1.0 with 0.1 increment.

weightUsed = items[index].weight * fraction

If weightUsed <= capacity

currentFractions[index] = fraction

fractionalKnapsackBrute (items, currentFrac, maxValue, index+1)

// End of if

// End of for loop

currentFractions[index] = 0.0

// taking current item at all in this case currentFrac = 0

fractionalKnapsackBrute (items, currentFrac, maxValue, index+1)

// End of function

II) Greedy approach (optimal & practical)

// inputs :

① item : list of item with properties weight, life, value etc

② capacity : capacity of knapsack

// outputs : ① list of items included in knapsack with max profit

② max possible profit

- function fractionalknapsack (items, capacity)

set total value to 0

set knapsackIds as empty list

set remainingcap to capacity

for Each item in items :

set id to item[0]

set value to item[1]

set weight to item[3]

if weight ≤ remainingcap

knapsack.add(id)

total value += value

remainingcap -= weight

Else // get fractions of item

set fraction to remainingcap / weight

if fraction > 0

total value += value * fraction

knapsack.add(Id)

set remaining cap to 0 // filled knapsack

break // exit loop

return (knapsackIds, total value)

- calculateRatios (items)

for eachitem in items :

value = item[1]

life = item[2]

weight = item[3]

ratio = value / (life * weight)

item.add(ratio)

- Sort By ratios // sorting in non-dec order

function sortbyratios (items)

// use any sorting technique like merge sort, quick sort with custom logic below

sort (items, customlogic)

// item[4] hold ratio

function customlogic (item1, item2)

2 adj items in item list

if (item1[4] > item2[4])

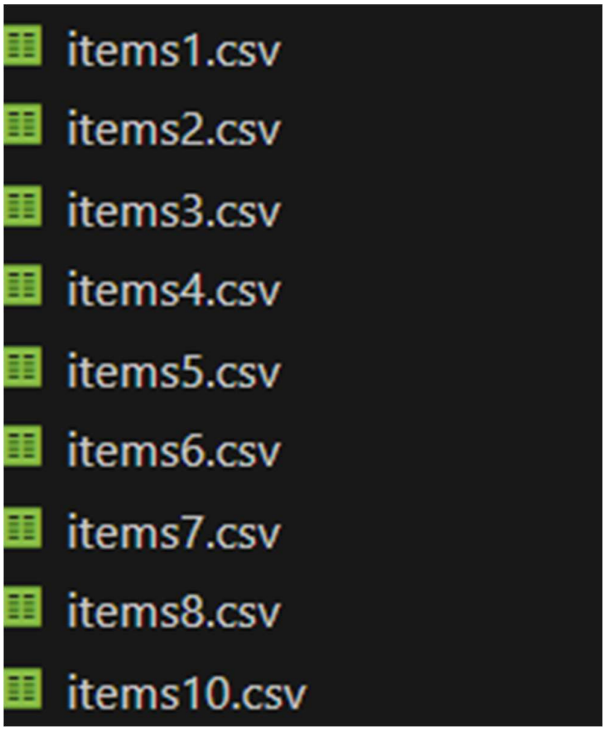
return true // item1 before item2

else

return false // item2 before item1

TEST CASES:

```
TEST CASES
1-5 : NORMAL
6   : OUT OF RANGE DATA
7   : NEGATIVE DATA
8   : MISSING DATA
9   : FILE NOT EXIST
10  : EMPTY FILE
```



TIME COMPLEXITY:

1) Brute approach :

The main idea around subset generation

let, n be no. of items

Each item can be → Included
→ Excluded

Combinally there are 2 possibility for any item it can be either included or Excluded

1 → 2 ways

n → 2^n ways

we do this for every item & final sack contains combination of these item in total so by product rule of combinations.

$= 2 \cdot 2 \cdot 2 \cdots + n \text{ times} \simeq 2^n$ times

$O(2^n)$ → now after generating subsets we calculate for each subset total weights & value.

We perform linear iteration for this we have $O(n)$ operations for each 2^n subset

So ultimately $O(2^n * n)$

after this we re-iterate across all subset to find one with max value

Because there are 2^n subsets we take (2^n) time for traversal

Hence combining all steps of approach we have,

$$O(2^n) + O(2^n * n) + O(2^n)$$

ignoring lower order terms

& $\boxed{\text{time complexity} = O(2^n \times n)}$

2) greedy approach :

Here main idea is around value/wt ratio & sorting them in non-increasing order.

we perform following operations ⇒

① calculating ratios :

for this purpose, we linearly iterate across the list we fetch values, weights. for each item consisting of n item & we calculate ratio of value/wt calculation takes O(1) time But iteration take O(1+1+.. n times)

$\therefore O(n)$ time

PAGE No.
DATE / /

② Sorting items :
This is heart of greedy approach we use efficient sorting algorithms like mergesort or quicksort Both of them have Best case complexity nlogn. we use merge sort all of its time complexity cases are O(nlogn)

∴ This step have O(nlogn).

③ After sorting how we solve problem by filling knapsack one by one since here we do simple linear iteration over items we have O(n) time complexity for this step O(1+1+ . . . n times)

Combining all these steps

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

After ignoring constants & lower order terms

⇒ nlogn is dominant

Time complexity = O(nlogn)

PROGRAM:

CSV GENERATOR

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <string>

// Utility function to generate a random value within a specified range
int getRandomValue(int min, int max) {
    return min + (std::rand() % (max - min + 1));
}

// Function to generate a CSV file with normal values
void generateCSV(const std::string& filename, int numItems) {
    std::ofstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    // Write CSV header
    file << "Item ID,Item Value(1000),Item Life(days),Item Weight(tons)\n";

    for (int i = 1; i <= numItems; ++i) {
        std::string itemID = "item" + std::to_string(i);

        // Generate normal values (no negatives or zeros)
        int weight = getRandomValue(1, 10);  // Weight (in tons)
        int value = getRandomValue(20, 500);  // Item value (in thousands)
        int shelfLife = getRandomValue(1, 31);  // Shelf life (in days)
```

```
        // Write data to CSV
        file << itemID << "," << value << "," << shelfLife << "," << weight << "\n";
    }

    file.close();
}

// Function to generate a unique filename
std::string generateUniqueFilename(const std::string& baseFilename) {
    int index = 1;
    std::string filename;

    while (true) {
        filename = baseFilename + std::to_string(index) + ".csv";
        std::ifstream f(filename);
        if (!f.good()) { // If file does not exist, break the loop
            break;
        }
        index++;
    }

    return filename;
}

// Main function
int main() {
    std::srand(static_cast<unsigned>(std::time(nullptr)));

    int numItems = 100; // Number of items to generate
    std::string baseFilename = "items";
    std::string file_path = generateUniqueFilename(baseFilename);

    generateCSV(file_path, numItems);
```

```
    std::cout << "CSV file " << file_path << " generated successfully.\n";
    return 0;
}
```

KNAPSACK SOLVER

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <cstdlib>
#include <iomanip>
#include <algorithm>

const double CAPACITY = 200.0; // Define capacity as a constant

// Function to read and validate CSV data
std::vector<std::vector<std::string>> readCSV(const std::string& filename) {
    std::vector<std::vector<std::string>> items;
    std::ifstream file(filename);

    if (!file.is_open()) {
        std::cerr << "Error: Could not open the file." << std::endl;
        exit(EXIT_FAILURE);
    }

    std::string line;
    std::getline(file, line); // Skip header

    // Check if the file is empty after reading the header
    if (file.peek() == std::ifstream::traits_type::eof()) {
```

```
        std::cerr << "Error: The file is empty." << std::endl;
        exit(EXIT_FAILURE);
    }

    while (std::getline(file, line)) {
        std::vector<std::string> itemData;
        std::stringstream ss(line);
        std::string value;

        while (std::getline(ss, value, ',')) {
            itemData.push_back(value);
        }

        if (itemData.size() == 4) {
            items.push_back(itemData);
        } else {
            std::cout << "Invalid data format in CSV, each row must have four values." << std::endl;
            exit(EXIT_FAILURE);
        }
    }

    file.close();
    return items;
}

void validateItems(const std::vector<std::vector<std::string>>& items) {
    for (const auto& item : items) {
        // Check for missing data
        if (item[1] == "_" || item[2] == "_" || item[3] == "_") {
            std::cout << "Missing data encountered for item " << item[0] << ": One or more fields are missing.
Problem can't be solved." << std::endl;
            exit(EXIT_FAILURE); // Exit the program
        }
```

```
        double value = std::stod(item[1]);
        double life = std::stod(item[2]);
        double weight = std::stod(item[3]);


        if (value < 0) {
            std::cout << "Negative value encountered for item " << item[0] << ": Value cannot be negative. Problem can't be solved." << std::endl;
            exit(EXIT_FAILURE); // Exit the program
        }
        if (life < 0) {
            std::cout << "Negative life encountered for item " << item[0] << ": Life cannot be negative. Problem can't be solved." << std::endl;
            exit(EXIT_FAILURE); // Exit the program
        }
        if (weight < 0) {
            std::cout << "Negative weight encountered for item " << item[0] << ": Weight cannot be negative. Problem can't be solved." << std::endl;
            exit(EXIT_FAILURE); // Exit the program
        }
        if (value < 20 || value > 500) {
            std::cout << "Invalid value for item " << item[0] << ": Value should be in range [20, 500]. Skipping this item." << std::endl;
            continue; // Skip this item
        }
        if (life < 1 || life > 31) {
            std::cout << "Invalid life for item " << item[0] << ": Life should be in range [1, 31]. Skipping this item." << std::endl;
            continue; // Skip this item
        }
        if (weight < 1 || weight > 10) {
            std::cout << "Invalid weight for item " << item[0] << ": Weight should be in range [1, 10]. Skipping this item." << std::endl;
            continue; // Skip this item
        }
```

```
    }
}

void calculateRatios(std::vector<std::vector<std::string>>& items) {
    for (auto& item : items) {
        double value = std::stod(item[1]);
        double life = std::stod(item[2]);
        double weight = std::stod(item[3]);
        double ratio = value / (life * weight);
        item.push_back(std::to_string(ratio));
    }
}

void sortByRatios(std::vector<std::vector<std::string>>& items) {
    std::sort(items.begin(), items.end(), [](const std::vector<std::string>& a, const std::vector<std::string>& b) {
        return std::stod(a[4]) > std::stod(b[4]);
    });
}

void processItems(std::vector<std::vector<std::string>>& items) {
    validateItems(items);
    calculateRatios(items);
    sortByRatios(items);
}

std::pair<std::vector<std::string>, double> fractionalKnapsack(std::vector<std::vector<std::string>>& items,
double capacity) {
    double totalValue = 0.0;
    std::vector<std::string> knapsackIds;
    double remainingCapacity = capacity;

    for (const auto& item : items) {
        std::string id = item[0];
```

```
        double value = std::stod(item[1]);
        double weight = std::stod(item[3]);

        if (weight <= remainingCapacity) {
            knapsackIds.push_back(id);
            totalValue += value;
            remainingCapacity -= weight;
        } else {
            double fraction = remainingCapacity / weight;
            if (fraction > 0) {
                totalValue += value * fraction;
                knapsackIds.push_back(id);
                std::cout << "Taking " << fraction * 100 << "% of item " << id << ". Total Value: " << totalValue << "\n";
            }
            remainingCapacity = 0;
            break;
        }
    }

    return {knapsackIds, totalValue};
}

void displayAndStore(const std::vector<std::vector<std::string>>& items, const std::vector<std::string>&
knapsackIds, const std::string& outputFilename, double initialCapacity) {
    std::ofstream outputFile(outputFilename);

    if (!outputFile.is_open()) {
        std::cerr << "Error: Could not open the output file." << std::endl;
        return;
    }

    std::cout << "Items in Knapsack:" << std::endl;
    std::cout << std::setw(10) << "Item ID"
```

```
        << std::setw(15) << "Value"
        << std::setw(15) << "Life"
        << std::setw(15) << "Weight"
        << std::setw(20) << "Remaining Capacity" << std::endl;
outputFile << "Item ID,Value,Life,Weight,Remaining Capacity" << std::endl;

double remainingCapacity = initialCapacity;

for (const auto& id : knapsackIds) {
   for (const auto& item : items) {
      if (item[0] == id) {
         std::cout << std::setw(10) << item[0]
                   << std::setw(15) << item[1]
                   << std::setw(15) << item[2]
                   << std::setw(15) << item[3]
                   << std::setw(20) << remainingCapacity << std::endl;

         outputFile << item[0] << ","  << item[1] << "," << item[2] << "," << item[3]
                    << "," << remainingCapacity << std::endl;

         // Update remaining capacity based on item weight
         remainingCapacity -= std::stod(item[3]); // Subtract current item's weight

         // If the current item weight exceeds the remaining capacity, show the updated value
         if (remainingCapacity < 0) {
            remainingCapacity = 0;
         }

         break;
      }
   }
}
```

```
    outputFile.close();
}

int main() {
    std::string filename;
    std::cout << "Enter the input CSV file path: ";
    std::getline(std::cin, filename);

    std::string outputFilename = "output_" + filename;

    std::cout << "Solving the problem for " << filename << " via greedy approach." << std::endl;

    std::vector<std::vector<std::string>> itemData = readCSV(filename);

    // Check if no items were read
    if (itemData.empty()) {
        std::cout << "No items found in the file. Problem can't be solved." << std::endl;
        return 0; // Exit the program
    }

    processItems(itemData);
    std::pair<std::vector<std::string>, double> result = fractionalKnapsack(itemData, CAPACITY);
    std::vector<std::string> knapsackIds = result.first;
    double maxValue = result.second;

    std::cout << "The maximum profit for the knapsack is " << maxValue << std::endl;
    displayAndStore(itemData, knapsackIds, outputFilename, CAPACITY);

    return 0;
```

OUTPUT:

CSV FILE 1

```
Enter the input CSV file path: items1.csv
Solving the problem for items1.csv via greedy approach.
The maximum profit for the knapsack is 17586
Items in Knapsack:
```

Item ID	Value	Life	Weight	Remaining Capacity
item67	248	2	1	200
item82	427	1	5	199
item2	371	5	1	194
item19	353	1	5	193
item73	250	1	4	188
item36	394	4	2	184
item53	488	4	3	182
item76	280	7	1	179
item92	462	6	2	178
item4	236	1	7	176
item8	337	4	3	169
item16	494	6	3	166
item97	462	6	3	163
item46	495	13	2	160
item7	222	6	2	158
item37	484	3	9	156
item57	254	5	3	147
item11	340	7	3	144
item42	434	29	1	141
item20	283	19	1	140
item96	393	4	7	139
item74	323	12	2	132
item75	429	8	4	130
item38	361	27	1	126
item100	226	9	2	125
item22	296	8	3	123
item52	241	7	3	120
item41	466	6	7	117
item28	271	14	2	110
item9	336	5	7	108
item90	318	17	2	101
item1	374	12	4	99
item40	186	8	3	95
item13	464	20	3	92
item43	307	10	4	89
item81	232	31	1	85
item91	187	13	2	84
item14	68	1	10	82
item99	426	7	9	72
item12	481	15	5	63
item98	470	15	5	58
item94	497	10	8	53
item80	235	8	5	45
item3	256	11	4	40
item56	375	22	3	36
item62	414	15	5	33
item88	455	29	3	28
item27	354	17	4	25
item60	154	10	3	21
item68	385	9	9	18
item79	292	7	9	9

CSV FILE 2

```
Enter the input CSV file path: items2.csv
Solving the problem for items2.csv via greedy approach.
The maximum profit for the knapsack is 15227
Items in Knapsack:
```

Item ID	Value	Life	Weight	Remaining Capacity
item95	360	3	1	200
item4	363	5	1	199
item68	132	1	2	198
item100	451	2	4	196
item79	429	3	3	192
item84	458	10	1	189
item48	256	3	2	188
item73	63	1	2	186
item28	372	3	5	184
item26	370	5	3	179
item8	271	11	1	176
item51	184	1	8	175
item60	227	5	2	167
item65	257	4	3	165
item6	495	24	1	162
item77	215	4	3	161
item47	320	18	1	158
item25	309	3	7	157
item13	431	30	1	150
item38	274	3	9	149
item45	174	6	3	140
item33	426	23	2	137
item82	156	17	1	135
item24	491	11	5	134
item92	383	22	2	129
item9	204	6	4	127
item11	96	6	2	123
item58	477	6	10	121
item78	427	11	6	111
item86	135	21	1	105
item18	183	15	2	104
item62	494	28	3	102
item14	231	8	5	99
item74	485	18	5	94
item57	414	16	5	89
item41	476	25	4	84
item30	250	6	9	80
item71	388	28	3	71
item91	314	9	8	68
item89	124	6	5	60
item80	295	25	3	55
item64	492	18	7	52
item94	296	19	4	45
item61	154	10	4	41
item31	197	19	3	37
item56	136	20	2	34
item44	54	2	8	32
item20	201	20	3	24
item53	116	12	3	21
item72	389	16	8	18
item35	332	11	10	10

CSV FILE 3

```
Enter the input CSV file path: items3.csv
Solving the problem for items3.csv via greedy approach.
Taking 25% of item item63. Total Value: 13591.5
The maximum profit for the knapsack is 13591.5
Items in Knapsack:
```

Item ID	Value	Life	Weight	Remaining Capacity
item47	244	1	1	200
item26	278	1	4	199
item79	208	1	4	195
item8	465	1	9	191
item67	87	1	2	182
item87	241	1	6	180
item81	319	2	5	174
item16	93	5	1	169
item60	286	4	4	168
item2	70	4	1	164
item4	232	2	7	163
item25	258	8	2	156
item48	493	31	1	154
item41	415	3	9	153
item88	371	3	9	144
item83	370	4	8	135
item34	491	24	2	127
item5	88	1	9	125
item97	76	4	2	116
item39	412	11	4	114
item59	186	4	5	110
item99	461	13	4	105
item12	68	4	2	101
item9	380	5	10	99
item98	471	7	9	89
item43	343	23	2	80
item82	434	20	3	78
item51	421	10	6	75
item54	252	12	3	69
item78	280	14	3	66
item52	418	21	3	63
item53	194	15	2	60
item21	406	21	3	58
item75	184	29	1	55
item57	483	20	4	54
item49	372	31	2	50
item19	447	25	3	48
item31	118	2	10	45
item92	128	25	1	35
item55	409	12	7	34
item3	258	11	5	27
item58	274	10	6	22
item30	269	15	4	16
item66	398	18	5	12
item13	197	15	3	7
item74	197	23	2	4
item63	186	6	8	2

CSV FILE 4

```
Enter the input CSV file path: items4.csv
Solving the problem for items4.csv via greedy approach.
Taking 37.5% of item item94. Total Value: 13675.9
The maximum profit for the knapsack is 13675.9
Items in Knapsack:
```

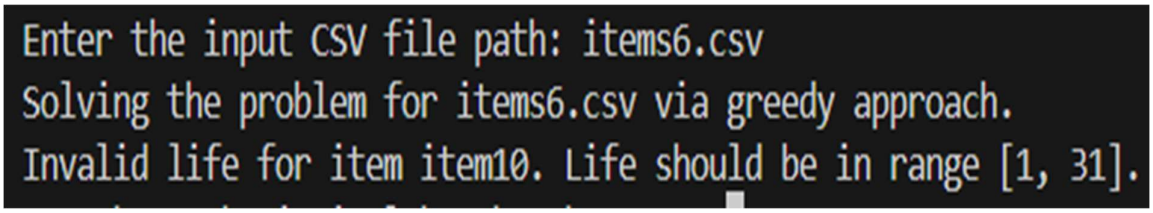
Item ID	Value	Life	Weight	Remaining Capacity
item61	440	1	6	200
item39	362	1	5	194
item67	286	1	4	189
item77	385	6	1	185
item15	455	11	1	184
item51	231	10	1	183
item25	283	13	1	182
item66	491	23	1	181
item18	256	6	2	180
item23	480	3	8	178
item28	438	7	4	170
item78	428	4	7	166
item21	111	9	1	159
item11	353	3	10	158
item16	205	2	9	148
item14	132	12	1	139
item84	319	29	1	138
item87	490	23	2	137
item54	372	5	7	135
item76	179	3	6	128
item49	193	10	2	122
item59	497	11	5	120
item97	346	4	10	115
item56	293	6	6	105
item88	432	9	6	99
item100	438	11	5	93
item72	310	11	4	88
item55	465	9	8	84
item46	308	7	7	76
item38	83	7	2	69
item31	231	22	2	67
item95	224	9	5	65
item41	129	14	2	60
item58	64	2	7	58
item44	308	18	4	51
item24	248	12	5	47
item30	475	15	8	42
item2	407	13	8	34
item65	221	19	3	26
item62	62	16	1	23
item37	357	24	4	22
item82	72	20	1	18
item80	162	15	3	17
item70	332	12	8	14
item63	273	27	3	6
item94	133	5	8	3

CSV FILE 5

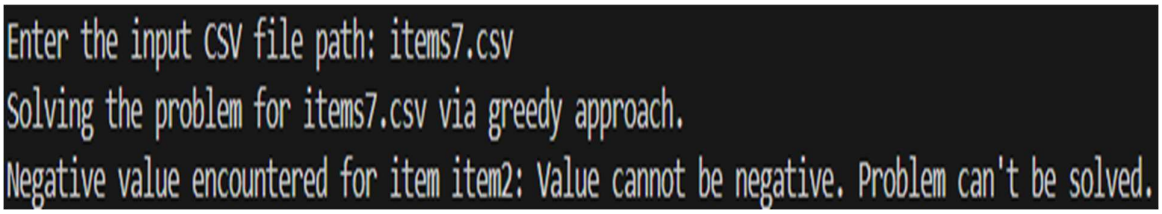
```
Enter the input CSV file path: items5.csv
Solving the problem for items5.csv via greedy approach.
Taking 12.5% of item item63. Total Value: 16542
The maximum profit for the knapsack is 16542
Items in Knapsack:
```

Item ID	Value	Life	Weight	Remaining Capacity
item61	427	1	2	200
item98	406	2	2	198
item82	454	1	5	196
item52	367	1	6	191
item57	311	3	2	185
item47	464	10	1	183
item22	171	4	1	182
item35	418	4	3	181
item49	496	2	8	178
item37	473	4	4	170
item56	413	3	5	166
item41	467	19	1	161
item67	289	6	2	160
item94	347	8	2	158
item75	459	23	1	156
item23	190	1	10	155
item13	500	7	4	145
item90	97	1	6	141
item24	436	27	1	135
item66	339	3	7	134
item51	357	24	1	127
item34	368	11	3	126
item12	236	3	8	123
item20	188	5	4	115
item33	328	7	5	111
item71	151	2	10	106
item11	472	9	7	96
item62	412	28	2	89
item43	365	17	3	87
item64	410	20	3	84
item4	130	10	2	81
item83	477	25	3	79
item40	297	6	8	76
item93	286	9	6	68
item96	409	13	6	62
item78	463	19	5	56
item77	216	25	2	51
item31	279	11	6	49
item3	484	17	7	43
item8	314	26	3	36
item16	280	14	5	33
item80	473	30	4	28
item95	451	23	5	24
item21	64	17	1	19
item84	289	26	3	18
item81	92	25	1	15
item85	277	11	7	14
item28	429	20	6	7
item63	168	6	8	1

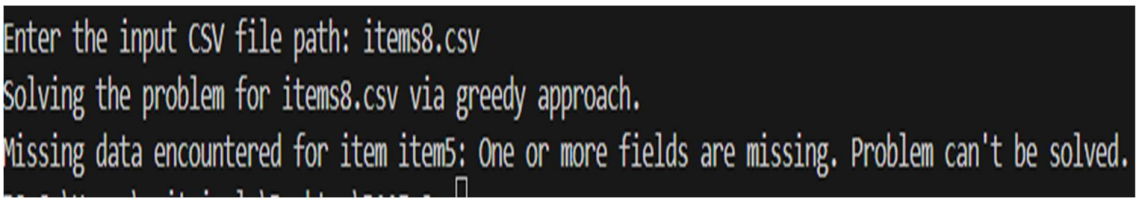
CSV FILE 6



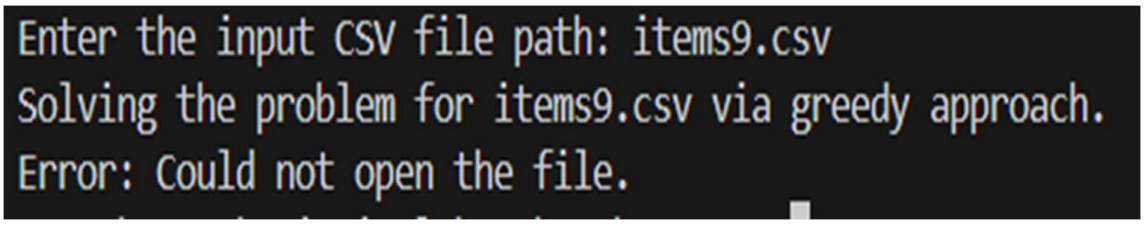
CSV FILE 7



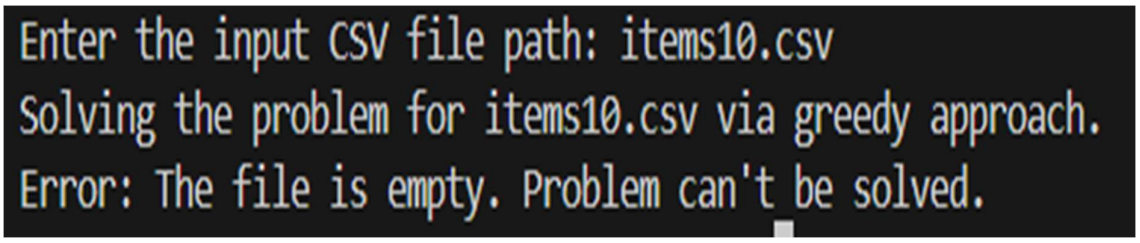
CSV FILE 8



CSV FILE 9



CSV FILE 10



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.

EXPERIMENTAL TASK 2

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

THEORY:

Huffman coding is a method for lossless data compression that assigns variable-length binary codes to input characters based on their frequencies. More frequent characters get shorter codes, while rarer ones receive longer codes. This method is optimal for minimizing the average code length when the frequency of each symbol is known. The algorithm works by repeatedly combining the two least frequent symbols to build a binary tree, which results in a prefix code—ensuring that no code is a prefix of another, allowing for unique decoding. **To evaluate the effectiveness of compression, you can calculate the compression ratio, which compares the size of the original data to the size of the compressed data. This ratio provides insight into the efficiency of the compression method used. Huffman coding is widely employed in various applications, including image and audio formats, making it a valuable technique in data storage and transmission.

ALGORITHM:

BRUTE

```
// Inputs : symbols / characters needed
            to be encoded

// outputs : valid prefix codes
    // list containing valid prefix code
       sets which can represent character

• function prefixcodebrute (symbols):
      minlength = 1
      maxlength = symbol.length()
          // maxlength of codewords
  Set validprefixcodes as Empty list
  // stores all subsets or combination
  // generating all possible code combinatio
       for given lengths

  for length from minlength to maxlength:
      Set result as empty list
    // store binary string of current length
  generateAllBinarystrings (length, "", result)
  // check all combinations of prefix
                                codes
  for Each subset of result with size
                          == symbol length
    if isPrefixfree (subset):
          validprefixcodes.add (subset)

  // End of if
  // End of for
  // End of for
  return validprefixcodes;
```

- function isprefixfree(codeset):

  for Each code1 in codeset:
    for Each code2 in codeset:
      if code1 != code2 And
        code2 start with code1:
        return false
      // End of if
    // End of for
  // End of for

  return True // else its

- function generateAllBinarystrings
  (n, currstr, result)

  If currstr.length() == n // Base case
  result.add(currstr) // for valid binary string
  //add valid binary string
  to result

  return
  // End of if.

  // Recursive calls ⇒

  generateAllBinarystrings(n, currstr+"0", result)
  generateAllBinarystrings(n, currstr+"1", result)

  // End of function

GREEDY

C → set of n characters

c → Each character ∈ C

c.freq → frequency of c

|c| → no. of distinct characters

Q → min priority queue

Algorithm ⇒

HUFFMAN (C)

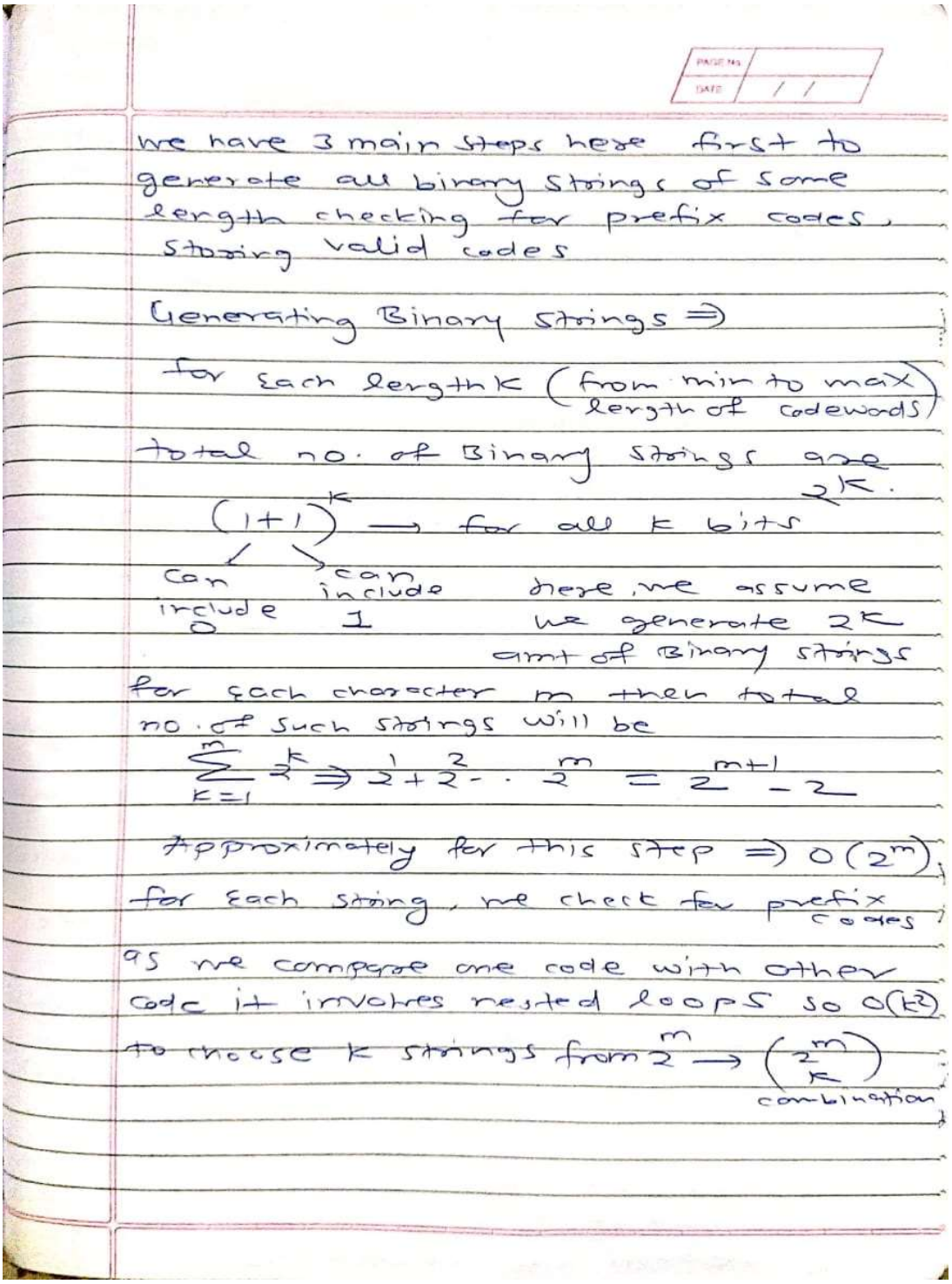
```
1.  n = |C|
2.  Q = C
3.  for i = 1 to n-1
4.      allocate a new node z
5.      z.left = x = EXTRACT-MIN(Q)
6.      z.right = y = EXTRACT-MIN(Q)
7.      z.freq = x.freq + y.freq
8.      INSERT (Q, z)
9.  return EXTRACT-MIN(Q)
                // return root of tree
```

TEST CASES:

1. TEXT FILE
2. DOCX FILE
3. PDF FILE
4. HTML FILE
5. EMPTY FILE

TIME COMPLEXITY:

BRUTE APPROACH



we have 3 main steps here first to generate all binary strings of some length checking for prefix codes, storing valid codes

Generating Binary strings ⇒

for each length K (from min to max length of codewords)

total no. of Binary strings are 2^K.

$(1+1)^K$ → for all K bits

can include 0 | can include 1

here, we assume we generate 2^K amt of Binary strings

for each character m then total no. of such strings will be

$$\sum_{K=1}^{m} 2^K \Rightarrow 2^1 + 2^2 \cdots 2^m = 2^{m+1} - 2$$

Approximately for this step ⇒ $O(2^m)$,

for each string, we check for prefix codes

as we compare one code with other code it involves nested loops so $O(k^2)$

to choose K strings from 2^m → $\binom{2^m}{K}$ combination

for each subset we have $O(k^2)$ checks.

the overall Time complexity thus for checking part

$$O\left(\sum_{k=1}^{m} \binom{2^m}{k} k^2\right)$$

largest check occurs as $k=m$

ignoring lower order terms

$O(2^m \cdot m^2)$ as $\binom{2^m}{m}$ can be proven to be $\neq 2^m$

for storage there isn't much complexity to discuss it take $O(1)$ time to insert to last position of set.

combining all steps we have

$$O(2^m) + O(2^m \cdot m^2)$$

$$\approx O(2^m \cdot m^2)$$

$\therefore$ Time complexity $= O(2^m \cdot m^2)$

GREEDY APPROACH

Huffman coding is greedy algorithm used for data compression.

Two main steps of Huffman coding ⇒

1. Building Hoffman tree (using priority queue)
2. Generating Binary code for Each character

Step 1 ⇒ ① Initial step: Inserting all characters into priority queue

Assume there are n unique characters & Each character has freq.
we need to insert all n nodes into priority queue where Each Insertion take o(logn) time

all char insertion takes → nlogn time

② Merging nodes ⇒

It involves Extracting 2 smallest element of min freq then merging into new node with combined freq.

since n-1 merging operations.
Each merging operation take log n

∴ $O((n-1) * \log n) = n\log n$

Step 2 ⇒

We generate Binary code by performing pre-order / in-order traversal.

In Each traversal, Each node is visited Exactly once.

since (2n-1) nodes

- n leaf nodes
- n-1 internal nodes

∴ It takes O(n) time

Total Complexity ⇒ Building tree + Generating Codes

⇒ nlogn + n

⇒ nlogn

PROGRAM:

MAIN CODE

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <memory>
#include <cstdlib>
#include <array>
#include <map>
#include <vector>
#include <queue>
#include <algorithm>
#include <regex>


// Huffman Node Definition
struct HuffmanNode {
    char character;
    int frequency;
    HuffmanNode *left, *right;


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


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


// Function to read text from a TXT file
```

```
std::string readTxt(const std::string& filePath) {
    std::ifstream file(filePath);
    if (!file.is_open()) {
        std::cerr << "Could not open the file: " << filePath << std::endl;
        return "";
    }
    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}

// Function to read text from a DOCX file using Python
std::string readDocx(const std::string& filePath) {
    std::string command = "python read_docx.py \"" + filePath + "\"";
    std::array<char, 128> buffer;
    std::string result;

    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(command.c_str(), "r"),
pclose);
    if (!pipe) {
        std::cerr << "Failed to run command\n";
        return "";
    }
    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
        result += buffer.data();
    }
    return result;
}

// Function to read text from a PDF file using Python
std::string readPdf(const std::string& filePath) {
    std::string command = "python read_pdf.py \"" + filePath + "\"";
    std::array<char, 128> buffer;
    std::string result;
```

```
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(command.c_str(), "r"),
pclose);
    if (!pipe) {
        std::cerr << "Failed to run command\n";
        return "";
    }
    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
        result += buffer.data();
    }
    return result;
}

// Function to read text from an HTML file
std::string readHtml(const std::string& filePath) {
    std::ifstream file(filePath);
    if (!file.is_open()) {
        std::cerr << "Could not open the file: " << filePath << std::endl;
        return "";
    }

    std::stringstream buffer;
    buffer << file.rdbuf();
    std::string htmlContent = buffer.str();

    // Use regex to remove HTML tags
    std::regex htmlTags("<[^>]*>");
    std::string text = std::regex_replace(htmlContent, htmlTags, " ");

    // Optionally, you can also trim whitespace
    text.erase(std::remove_if(text.begin(), text.end(), ::isspace), text.end());

    return text;
}
```

```
// Function to build Huffman tree
HuffmanNode* buildHuffmanTree(const std::string& text) {
    std::map<char, int> frequency;
    for (char ch : text) {
        frequency[ch]++;
    }

    std::priority_queue<HuffmanNode*, std::vector<HuffmanNode*>, Compare> minHeap;
    for (const auto& pair : frequency) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }

    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 generateHuffmanCodes(HuffmanNode* root, const std::string& code,
std::map<char, std::string>& huffmanCodes) {
    if (!root) return;
    if (root->character != '\0') {
        huffmanCodes[root->character] = code;
    }
    generateHuffmanCodes(root->left, code + "0", huffmanCodes);
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}
```

```
// Function to encode text using Huffman codes
std::string huffmanEncode(const std::string& text, const std::map<char, std::string>& huffmanCodes) {
    std::string encodedText;
    for (char ch : text) {
        encodedText += huffmanCodes.at(ch);
    }
    return encodedText;
}

// Function to save the compressed text to a file
void saveCompressedFile(const std::string& encodedText, const std::string& outputPath) {
    std::ofstream outputFile(outputPath);
    if (!outputFile.is_open()) {
        std::cerr << "Could not open the file for writing: " << outputPath << std::endl;
        return;
    }
    outputFile << encodedText;
    outputFile.close();
    std::cout << "Compressed file saved as: " << outputPath << std::endl;
}

// Function to calculate compression ratio
double calculateCompressionRatio(size_t originalSize, size_t encodedSize) {
    return static_cast<double>(encodedSize) / originalSize; // ratio
}

// Main function
int main() {
    std::string filePath;
    std::string fileType;
```

```
    // Specify the file path and type (txt, docx, pdf, or html)
    std::cout << "Enter the file path: ";
    std::cin >> filePath;
    std::cout << "Enter the file type (txt, docx, pdf, or html): ";
    std::cin >> fileType;

    std::string text;
    if (fileType == "txt") {
        text = readTxt(filePath);
    } else if (fileType == "docx") {
        text = readDocx(filePath);
    } else if (fileType == "pdf") {
        text = readPdf(filePath);
    } else if (fileType == "html") {
        text = readHtml(filePath);
    } else {
        std::cerr << "Unsupported file type." << std::endl;
        return 1;
    }

    // Check if the text is empty
    if (text.empty()) {
        std::cerr << "The file is empty." << std::endl;
        return 1;
    }

    // Calculate original size
    size_t originalSize = text.size() * 8; // Size in bits

    // Build Huffman tree
    HuffmanNode* root = buildHuffmanTree(text);

    // Generate Huffman codes
    std::map<char, std::string> huffmanCodes;
```

```
    generateHuffmanCodes(root, "", huffmanCodes);

    // Encode the text
    std::string encodedText = huffmanEncode(text, huffmanCodes);

    // Specify the output file name
    std::string outputFileName = filePath + "_compressed.txt";
    saveCompressedFile(encodedText, outputFileName);

    // Calculate encoded size
    size_t encodedSize = encodedText.size(); // Size in bits

    // Calculate and display compression ratio
    double compressionRatio = calculateCompressionRatio(originalSize,
encodedSize);

    // Output original and compressed sizes
    std::cout << "Original Size: " << originalSize << " bits" << std::endl;
    std::cout << "Compressed Size: " << encodedSize << " bits" << std::endl;
    std::cout << "Compression Ratio: " << compressionRatio << std::endl;

    return 0;
}
```

DOCX READER

```
import sys
from docx import Document

def read_docx(file_path):
    doc = Document(file_path)
    full_text = []
    for paragraph in doc.paragraphs:
        full_text.append(paragraph.text)
    return '\n'.join(full_text)
```

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python read_docx.py <path_to_docx>")
        sys.exit(1)
    file_path = sys.argv[1]
    text = read_docx(file_path)
    print(text)
```

PDF READER

```
import sys
from PyPDF2 import PdfReader

def read_pdf(file_path):
    with open(file_path, "rb") as file:
        reader = PdfReader(file)
        text = ""
        for page in reader.pages:
            text += page.extract_text() or ""
    return text

if __name__ == "__main__":
    pdf_path = sys.argv[1]
    try:
        text = read_pdf(pdf_path)
        # Print using UTF-8 encoding
        print(text.encode('utf-8').decode('utf-8'))

    except Exception as e:
        print(f"Error reading PDF: {e}")
```

OUTPUT:

TEXT FILE :

```
Enter the file path: text_file.txt
Enter the file type (txt, docx, pdf, or html): txt
Compressed file saved as: text_file.txt_compressed.txt
Original Size: 6424 bits
Compressed Size: 3594 bits
Compression Ratio: 0.559465
```

DOC FILE:

```
Enter the file path: doc.docx
Enter the file type (txt, docx, pdf, or html): docx
Compressed file saved as: doc.docx_compressed.txt
Original Size: 5872 bits
Compressed Size: 3255 bits
Compression Ratio: 0.554326
```

PDF FILE :

```
Enter the file path: pdf_file.pdf
Enter the file type (txt, docx, pdf, or html): pdf
Compressed file saved as: pdf_file.pdf_compressed.txt
Original Size: 896 bits
Compressed Size: 495 bits
Compression Ratio: 0.552455
```

HTML FILE:

```
Enter the file path: html_file.html
Enter the file type (txt, docx, pdf, or html): html
Compressed file saved as: html_file.html_compressed.txt
Original Size: 4120 bits
Compressed Size: 2576 bits
Compression Ratio: 0.625243
```

EMPTY FILE:

```
Enter the file path: empty.txt
Enter the file type (txt, docx, pdf, or html): txt
The file is empty.
```

CONCLUSION:

In this Experiment, we developed a system to download books from a website in HTML, TXT, DOCX, and PDF formats, implementing Huffman coding for efficient compression. This technique reduced file sizes while preserving data integrity, allowing us to calculate and analyze compression ratios to evaluate the effectiveness of our approach. We also created a user-friendly interface to facilitate file processing and implemented error handling for empty or unsupported files. The project demonstrated the importance of data compression in managing large text files and highlighted Huffman coding as a robust solution for lossless compression, making it a valuable tool for efficient data storage and transmission.

THE END