Arpita Singh
231071005

Experiment Task - 1

1) Algorithm -

→ Brute force Algorithm

1. Input // List of items (value, weight), capacity of the knapsack
2. Initialize : max-value to 0 and best_combination to empty list
3. For each combination of items :
   - Calculate total weight and total value of combination
   - If total weight ≤ capacity
     - If total value > max_value ; update max-value and best_combination
4. Output : max-value and best_combination

→ Greedy Algorithm

1. Input // List of items (value, weight), capacity of knapsack
2. Calculate the value-to-weight ratio for each item
3. Sort items in descending order based on value-to-weight ratio
4. Initialize total_value to 0
5. For each item :
   - If the item can fit entirely (weight ≤ capacity)
     - Add its value to total_value
     - Subtract its weight from capacity

   Else :
   - Add the fractional value (ratio × remaining capacity) to total_value
   - Break the loop
6. Output total_value

2) positive: Test cases

① ID weight, value
1, 100, 20
2, 200, 30
3, 150, 25
4, 300, 50

Expected output: ratio

ID	weight	value	ratio	
4	50	300	6	maximum
2	30	200	6.67	value in
3	25	150	6	knapsack: 640

② 1, 60, 100
2, 100, 20
3, 120, 30

3	30	120	4	
2	20	100	5	max value in
1	10	60	6	knapsack: 280

③ 1, 500, 30
2, 400, 50
3, 300, 20

1	30	500	16.67	
2	50	400	8	max value in
3	20	300	15	knapsack: 830

④ 1 100, 200

1	200	100	0.5	max val : 100

⑤ 1, 50, 10

1	10	50	5	max val: 50

Negative Test Cases:

① Empty CSV file
Expected output: Error - opening file

② Invalid data Format → ID, value, weight
Expected output: 1, 100, 20
Error: Invalid stoi conversion 2, twohundred, 30

③ Negative weight -
Expected output: Error: weight cannot be negative.

④ Non, numeric Data -

ID	weight	value
2	abc	100

Expected output - Error: invalid stoi conversion

⑤ Overcapacity items:

ID	weight	value	ratio	
1	250	100	0.4	max val is 0
2	300	200	0.67	
3	400	150	0.375	

Expected output:

231071005

3 Time complexity

A Brute-force Algorithm :

1. No of subsets : For n items, each item can be included or excluded in knapsack, so 2^n subsets.
2. Check each Subset : For each subsets, we need to calculate total weight and total value :
   - This takes O(n) time in worst case, as we examine all n items.

$$T(n) = O(n) \times O(2^n) = O(n.2^n)$$

B Greedy Algorithm :

1. Calculate Ratios : For each item, compute value-to-weight:
   - This takes O(n) time for n items
2. Sorting : We need to sort item
   - Sorting will take O(nlogn)
3. Iterate Through items : After sorting, we iterate through list it will take O(n)

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

∴ time complexity using knapsack algorithm is O(nlogn).

Arpita Singh

231071005

Experiment Task 2

Algorithm (DS (Greedy method)

1] → Download Books: Identify, access the website, download the book in desired formats, store files in desired directory.

→ Read and Process: Open and read content, Extract text from documents, in format like PDFs and DOC.

→ Frequency count: count frequency of character in text

→ Build a priority Queue: ✓

- Create a priority queue (or min heap) to hold nodes, each node contains character and frequency
- Insert Nodes: For each character and its frequency, create a node and insert it into priority queue.

→ ~~Generate Huffman Codes~~: Build Huffman tree:

- ~~Traverse the Huffmann~~
- while there is more than one node in priority queue:
  - Remove the two nodes with lowest frequency (high priority)
  - Create a new internal node with these two nodes as children and a frequency equal to combined frequency.
  - Insert new node back into priority queue.

→ Generate huffman codes: Traverse Huffman tree recursively:

- Assign a binary code (0 for left branch and 1 for right branch)
- Store the codes ~~xx~~ where key is character and value - Huffman code.

→ Encode: Replace each character with original text with corresponding Huffman code to form encoded string.

→ Calculate Compression Ratio: Determine size of original text and encoded string. compute compression ratio = size of

Test Cases

Positive :

2] ① Input : {'a': 5, 'b': 9, 'c': 12, 'd': 35}
Expected output : e : 000 a : 001 b : 010 c : 011
d : 100 f : 101 compress ratio : 1.71

② Input : {'a': 3, 'b': 3, 'c': 3, 'd': 3}
Expected output : a : 00 b : 01, c : 10 d : 11
compress ratio : 2.56

③ Input : {'a': 10}
Expected output : a : Compress ratio : 1.0

④ Input : {'a' : 5, 'b' : 15}
Expected output : a : 0 b : 1 compress ratio : 1.0

⑤ Input : {'b' : 15}
Expected output : b : Compress ratio : 1.0

Negative Test Cases :

① Input : {}
Expected output : error : Frequency map is empty

② Input : {'a': 0}
Expected output : Error : Character frquency cannot be zero

③ Input : {'a': -5, 'b': 3}
Expected output : Error : Frequency cannot be negative

④ Input : { 1: 5, 2: 3}
Expected output : Error : Invalid Character type

⑤ Input : {'a': 10000, 'b': 50651}
Expected output : Error : Large input cannot be handled.

231071005

Time Complexity

3] A] Brute force Algorithm:

In worst case, you might need to perform n-1 comparison for first pass, n-2 for second and so on to 1.

so comparison →

$$(n-1)+(n-2)+\ldots 1 = \frac{n(n-1)}{2}$$

so time complexity = $O(n^2)$

but this may be exhaustive and not run code properly so it is not used

B] Greedy Method :

1] 1. Creating node for each character and insert in priority queue. This takes O(n) time for loop n characters

2. Inserting each node in priority queue takes O(log n) and for each n nodes will take O(nlogn)

3. To remove two nodes with smallest frequency and merge them will take O(logn)
since happening n-1 times so O(n-1 logn) = O(nlogn)
for tree building → O(n) + O(nlogn) + O(nlogn) = O(nlogn)

2] Generating huffman codes : Each node visited onces and n nodes in total so T(n) = O(n)

3] Calculating compression ratio : iterate through frequency map of size n, each lookup O(1) so O(1) x n = O(n)
final complexity = O(nlogn) + O(n) + O(n) = O(nlogn)