classmate

Date

Page

★ ALGORITHM FRACTIONAL KNAPSACK

// Compute the max value that can be shipped within capacity of 200 tonnes such that goods with less shelf life and higher cost are shipped first.
// Input: List of goods (each having weight, shelf life, cost), max weight capacity (200).
// Output: Max value that can be shipped.

for each item in goods:

find value = cost / weight

push value in heap.

while heap is not empty and totalweight < capacity:

a] pop item with smallest shelf life and highest value density from heap
b] weight_to_ship = min(weight, capacity - total weight)
c] total_value += value ~~dens~~ density x weight_to ship ~~take~~

d] total weight += weight_to_ship
e] if total_weighth == capacity then exit with knapsack is full.

return total_value.

★ TIME COMPLEXITY

building heap takes $O(n)$
insertion into heap of n items = $O(n\log n)$.

extracting all items takes $O(n\log n)$.

Best Case: when knapsack capacity is
greater than all weights combined

Worst Case: iterate through all items and fills
with fraction of one.

Avg case: fills after several items (but not all)

→ In either of them building the heap
takes $O(n\log n)$.

∴ Time complexity is $O(n\log n)$.

* Time complexity (Brute Force)

If we take the weight steps ~~or~~ to be m and there are n items then all possible combination amount to m^n.

Therefore, time complexity will be $O(m^n)$.

This is computationally very expensive and is not feasible to calculate for large values of 'n'.

Test Case:

a] Randomly generate 100 items with random value, random weight and random shelflife. (All values in acceptable ranges).

b] weights are negative or equal to zero

c] values of items are zero

d] shelf life are negative or ~~less~~ zero

e] vehicle capacity less than 0.

★ Huffman coding:

// ~~Compute~~ Use heap to build huffman tree
// Input: text
// Output: huffman tree.

→ initialise a freq dictionary to count occurrence of each char in text.

→ Create a min heap priority queue with (weight, node) for each char.

→ while size of heap > 1:

a] pop 2 nodes with lowest freq (x, y)
b] new_freq = x.freq + y.freq.
c] create new node and assign x, y as children nodes of new node
d] push new node in heap.

return root node.

classmate

Date

Page

generate_codes(node, codes, currentcode)

// compute the huffman codes for each character from the tree.

// Input: node of huffan tree, empty dictionary, current code.

// Output: codes dictionary containing huffman codes of each character.

→ if node is not none:

a] if it has a symbol

i] add its symbol and corresponding current code to code dictionary

b] recursively call generate codes for left ~~and~~ child with current code appended with "0".

c] recursively call generate codes for right child with current node appended with "1".

return codes{}

classmate
Date
Page

ALGO for compression ratio (text)

//Compute : compressed code and find the compression ratio
// Input : the text file (extracted from any format)
//Output : compressed string and ratio.

→ call the function huffan coding to get generated codes

create compressed string by concatenting codes for each char in text.

original size = length of text in bits

compressed size = length of compressed text

compression_ratio = compressed size / original size

return compressed text and ratio.

classmate
Date
Page

★ Time Complexity.

counting frequencies takes O(k)
k = length of text

contracting and initialising min heap takes O(n), n = no of unique symbols.

building tree:

loop executes (n-1) times.
in each we pop 2, insert 1, each push and pop takes O(logn).

∴ entire is O(nlogn).

∴ O(nlogn) is time complexity.

here, there is nothing specifically different between best average and worst since building tree, generating codes, finding ratio works same in all.

∴ Time complexity is O(nlogn).

Test Cases:

a] pdf file
og size: 1138 char
compressed size: 5368 bits
ratio : 1.70

b] docx file
og size: 704 char
compressed size: 3390 bits
ratio :- 1.66

c] txt file
og size : 718 char
compressed size : 1218 bits
ratio : 4.72

d] html file

og size : 2221 char
compressed size : 9150 bits
ratio : 1.94.

e] jpg file
unsupported format

f] docx file
incorrect path

g] pdf file
empty file

classmate
Date
Page

h] txt file
empty file

i] mp4 file
unsupported format.

The code follows PEP 8 guidelines.

Output:

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Huffmancoding.py
Reading book from: C:\Users\Krish\Downloads\Pdf_File.pdf
Original size: 1138 characters
Compressed size: 5368 bits
Compression Ratio: 1.70
Reading book from: C:\Users\Krish\Downloads\Docx_File.docx
Original size: 704 characters
Compressed size: 3390 bits
Compression Ratio: 1.66
Reading book from: C:\Users\Krish\Downloads\Text_File.txt
Original size: 718 characters
Compressed size: 1218 bits
Compression Ratio: 4.72
Reading book from: C:\Users\Krish\Downloads\Html_file.html
Original size: 2221 characters
Compressed size: 9150 bits
Compression Ratio: 1.94
Reading book from: C:\Users\Krish\Downloads\Html_file2.html
Original size: 10521 characters
Compressed size: 39889 bits
Compression Ratio: 2.11
```

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Huffmancoding.py
Reading book from: C:\Users\Krish\Downloads\Image.jpg
Unsupported file format: C:\Users\Krish\Downloads\Image.jpg
Reading book from: C:\Users\Krish\Downloads\Doc.docx
Error reading C:\Users\Krish\Downloads\Doc.docx: Package not found at 'C:\Users\Krish\Downloads\Doc.docx'
Reading book from: C:\Users\Krish\Downloads\Zero_letter_File.pdf
Error reading C:\Users\Krish\Downloads\Zero_letter_File.pdf: Cannot read an empty file
Reading book from: C:\Users\Krish\Downloads\Single_Letter_File.txt
Compression failed or resulted in an empty string.
Reading book from: C:\Users\Krish\Downloads\Video_File.mp4
Unsupported file format: C:\Users\Krish\Downloads\Video_File.mp4

Process finished with exit code 0
```

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Fractional_Knapsack.py
Maximum profit achievable: 1171.49 units
Total weight used: 200.00 tons
Remaining capacity: 0.00 tons

Items selected:
Item       Fraction (%)    Weight (Selected/Total)   Value (Selected/Total)    Shelf Life (days)
----------------------------------------------------------------------------------------------------
Item_12    100.0           7.00/7                         91.00/91                       12
Item_80    100.0           5.00/5                         85.00/85                       19
Item_76    100.0           10.00/10                        90.00/90                       12
Item_49    100.0           7.00/7                         89.00/89                       18
Item_81    100.0           11.00/11                        83.00/83                       12
Item_41    100.0           8.00/8                         49.00/49                       10
Item_99    100.0           9.00/9                         64.00/64                       14
Item_82    100.0           10.00/10                        87.00/87                       19
Item_62    100.0           12.00/12                        63.00/63                       12
Item_55    100.0           15.00/15                        92.00/92                       15
Item_53    100.0           19.00/19                        82.00/82                       14
Item_93    100.0           19.00/19                        91.00/91                       17
Item_19    100.0           13.00/13                        44.00/44                       13
Item_86    100.0           8.00/8                         30.00/30                       15
Item_4     100.0           15.00/15                        66.00/66                       18
Item_44    74.4            32.00/43                        65.49/88                       10
```

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Fractional_Knapsack.py
Error: Total weight of items is less than or equal to zero.
```

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Fractional_Knapsack.py
Error: Shelf life of items must be greater than 0.
```

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Fractional_Knapsack.py
Error: No value could be obtained from the items.
```

```
C:\Users\Krish\AppData\Local\Programs\Python\Python312\python.exe C:\Users\Krish\Downloads\Fractional_Knapsack.py
Error: Vehicle capacity must be greater than 0.
```

Conclusion:

The **fractional knapsack problem** involves selecting items with given weights and values to maximize the total value within a fixed capacity, allowing fractional parts of items to be taken. **Huffman coding** is a data compression technique that assigns variable-length binary codes to characters based on their frequency, minimizing the total encoded size while ensuring prefix-free encoding.

Fractional knapsack using brute force has time complexity of O(m^n) and for greedy approach is O(nlogn) which is due the use of min heap priority queue.
Huffman coding gives the time complexity of O(nlogn) for greedy approaches.

Greedy algorithms offer optimal and practical solutions to both problems by reducing time complexity compared to brute force.