Date
Page

DAA LAB - 5

Task 1:

Algorithm:

CSV file format

Sr.No, ShelfLife, Cost, Capacity

// Input: Taken from csv file with above format

file_path = dataset.csv

dataset = readcsv(file_path)

Assumption: Max ~~weigh~~ capacity = 200 tonnes

W = 200

~~Fractional Knapsack~~

ReadCSV (~~file~~ dataset)

// Input: CSV file dataset

// Output: Creates a list of dictionaries of the csv file

return dataset.todict(orient="records")

Fractional Knapsack (items, W)

// Input: List of dictionaries of csv file (items) and max capacity (W)

// Output: ~~Maximum cost of the~~ Total cost of items based on given conditions

for i ∈ items:

i[ValueIndex] = i[cost] / (i[ShelfLife] * i[capacity])

$ Sort items in descending order of ~~value~~ ValueIndex

totalcost = 0

c = 0

```
for i ∈ items:
    if c ≥ W:
        break
    if i['capacity'] + c <= W:
        totalcost += i[cost]
        c += i[capacity]
    else:
        fraction = (W - c)/i[capacity]
        totalcost += fraction*i[cost]
        c = W
return totalcost
```

Testcases:

- Negative Testcases:
  - Any of the columns have non positive number
    Output: INVALID INPUT
  - If dataset is empty
    Output: EMPTY FILE
  - If any column is unfilled:
    Output: INVALID SIZE OF COLUMNS

- Positive Test cases:
  - Input: knapsack_data_1.csv
    Output: The Maximum cost in Transport vehicles such that Items have max Cost, Min capacity and min Shelf Life is 14402.86
  - Input: Knapsack_data_2.csv
    Output: The maximum cost in Transport vehicles such that Items have max Cost, Min Capacity and min Shelf Life is 14729

Task - 2

Algorithm:

~~// Input: Taken from a textfile~~

// Input: Taken from a textfile

filename = file.txt

data = filename.read()

~~COMPRESSFILE(data)~~

~~// Input: Contents of txt file~~

~~// Output: Compression Ratio of file.~~

~~originalLength = len(data) * 8~~

COMPRATIO (data)

// Input: Contents of txt file

// Output: Compression Ratio of file

originalLength = len(data) * 8

NewLength = len(Huffman(data))

return originalLength / NewLength

HUFFMANTREE(data)

// Input: contents of txt file

// Output: Root node of Huffmann tree

n = len(data)

Put all unique characters with their frequency into ~~the~~ a minimum priority queue (Q)

Each tree node has a left child, right child and a frequency.

for i = 1 to n - 1

Allocate a new node z

z -> left = x = dequeue (Q)

z -> right = y = dequeue (Q)

z -> freq = x -> freq + y -> freq

enqueue (Q, z)

Date
Page

```
return dequeue(Q)

HUFFMANCODE ~~(data)~~ ~~(root)~~
~~// Input: Root node of huffmann tree~~
~~//output~~:
HUFFMANCODE (root, code)
// Input: Initially root node of Huffmann tree and
        ~~a~~ Empty string (code) initially which becomes the
        code generated so far
//Output: Dictionary with huffman codes for all
        characters (leaf nodes)
if ~~a~~ root is leaf node:
    ~~Store root->character~~
    Dict+= [root->character: code]
    return Dict
if root->left exists:
    HUFFMANCODE(root->left, code+'0')
if root->right exists:
    HUFFMANCODE(root->right, code+'1')

HUFFMAN (data~~base~~)
// Input: Contents of file
// Output: Huffman encoded contents of file
filename = file2.txt
data2 = filename.write()
for all characters i in data:
    ~~data2 = replace = D~~
Dict = HUFFMANCODE (HUFFMANTREE(data), "")
for all characters i in data:
    ~~data2 = D~~ data2.write() = Dict[i]
return data 2
```

Testcases:

- Negative test cases:

• If input file is not txt, html, pdf or doc file:

Output: NOT PROPER FORMAT OF FILE

• If input file is empty:

Output: EMPTY FILE

- Positive Testcases:

1. Input = file1.txt

   file1.txt = " abcdaaaaa"

   Output = Compression Ratio is 24.00

2. Input = file2.txt

   File2.txt = "the quick brown fox jumps over the lazy dog"

   Output = compression ratio is 13.23

3. Input = file3.txt

   file3.txt = "A wonderful serenity has taken possession of my entire ~~world~~ soul, like these sweet mornings of spring which I enjoy with my whole heart. I am alone, and feel the charm of existence in this spot."

Output = compression ratio is 15.13

Time Complexity:

~~1~~. Task 1:

1. Brute force:

- It involves checking of every single subset of items
- But for fractional we would also have to consider ~~fractio~~ subsets with fractional amounts
- So we would have to consider infinitely many subsets, so traversing through them would be basically $O(\infty)$, so not possible
- while for non fractional it is $O(2^n)$ as ~~for~~ there are a finite number of subsets of count 2^n for a set of n elements.

2. Greedy Approach:

- The greedy approach takes $O(n\log n)$ as:
  - we calculate value by weight ratio which traverses array once so, it takes $O(n)$
  - Then we sort the array according to it which takes $O(n\log n)$ time as that is the most efficient sorting time

- $\therefore$, Total Time $= O(n) + O(n\log n)$

$$= O(n(1+\log n))$$

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

Date
Page

Task 2:

1. Brute force:

- Brute force method is extremely inefficient.
- steps:
  - Generate all codes:
    we would have to generate all possiblge binary codes for set of characters which would take $O(2^n)$ where n is number of characters

  - Frequency calculation:
    For each generated code, we would then find frequency of each code based on input frequencies

  - Encoding:
    Encoding would involve checking each character with every generated code, making worse complexity

- Thus, it would take a bit more than 2^n steps to complete, so it is $O(2^n)$

2. Greedy Approach:

- ~~It is~~ It's time complexity is $O(n \log n)$ as:
  - Frequency Calculation:
    This involves single traversal of file, taking $O(n)$ time, n = No. of characters in file.
  - Building Priority Queue / Min Heap:
    Building appropriate min heap takes $O(n)$ time.
  - Making Tree:
    Extraction & Insertion in min heap takes $O(\log n)$ time which we conduct $n-1$ times until 1 tree is made.
    So total time is $O(n \log n)$
  - Traversal of Tree:
    We traverse the tree & assign left edge '0' & right edge '1' which takes $O(n)$ time as we visit all nodes
  - Generating code:
    If we map each code to each character this only takes another $O(n)$ time as lookup in map takes $O(1)$ time for 1 character so for n characters it takes $O(n)$
- $\therefore$ ~~Of~~ Total time $= O(n) + O(n) + O(n \log n) + O(n) + O(n)$
  $= O(n(4 + \log n))$
  $= O(n \log n)$