॥ श्री ॥

Mani S. Patinkar
231071036.

Laboratory - 5

Task-1

FRACTIONAL KNAPSACK.

① Greedy approach

NAME: function fractionalKnapsack(weights, values, capacity)
// i/p: The weights and values of the items, the capacity of sack.
// o/p: Max value of knapsack.

```
function fractionalKnapsack(weights, values,
                            capacity) {

    n = length(weights)
ratio = array of (values[i] / weights[i])

// create an array of items (index,
weight, value, value-to-weight
ratio

items = array of tuples (index, weights[i],
                         values[i],
                         ratio[i]) for
        i in 0 to n-1.

Sort items by ratio in descending
order.
```

```
total_value = 0
total_weight = 0

for item in items:
    if total_weight + item.weight <= capacity:
        // Take the whole item

        total_weight += item.weight
        total_value += item.value

    else:
        // Take fraction (max) of item
        that fits.

        remaining_capacity = capacity - total_weight
        total_value += item.value * (remaining_
                                     capacity/item.
                                     weight)

        break // knapsack full.

return total_value
```

TIME COMPLEXITY

- Calculating value to weight ratios :- O(n) because there are n items.
- Sorting items by value to weight ratio O(nlogn) ⟶ most efficient sort

- Filling the knapsack
Once sorted, we iterate through the items, adding them to knapsack until capacity is reached
→ O(n)

∴ Total time complexity = O(nlogn)

(2) Brute-force (works only for 0-1 knapsack as it is exhaustive search).

NAME: function knapsackBruteForce (weights, values, capacity)

i/p : weights, values of items, capacity of sack

o/p : Max value of knapsack

n = length(weights)
max_value = 0

Generate 2^n subsets of items

for subset in 0 to $(2^n - 1)$:
total_weight = 0
total_value = 0

```
// Check each item whether its in the subset
for i in 0 to n-1:
    if subset & (1 << i) != 0
// the ith item is included in subset

        total_weight += weights[i]
        total_value += values[i]

if total_weight <= capacity:
    max_value = max(max_value, total_value)

return max_value
```

Time complexity

We are generating 2^n subsets, so this has exponential time complexity of $O(2^n)$.

Test-cases

tw

1) Items: 3
V = 60 w = 10
V = 100 w = 20
V = 120 w = 30
Capacity = 50
Max-val = 240

2) Items: 3
V = 20 w = 5
V = 30 w = 10
V = 40 w = 15
capacity = 30
Max-val = 90.

3) Items: 3
V = 100 w = 30
V = 60 w = 20
V = 120 w = 50
capacity = 15
Max-val = 50

4) Items = 3
V = 70 w = 10
V = 80 w = 20
V = 40 w = 5
capacity = 0.
Max-val = 0.

5) Items = 3
V = 500 w = 30
V = 300 w = 10
V = 200 w = 20
capacity: 20
max_val = 466.67

-ve

1) Items = 0
Max_val = 0

3) Items = 2
Capacity = -10
error

4) Items = 2
V = 100 w = -20
:
error.

Conclusion :- Greedy algo takes nlogn whereas brute force takes expenential a^n time. Program written using ullman style.

-ve

1) Items = 0
Max_val = 0

2) Items = 2
capacity = 0
max_val = 0

3) Items = 2
capacity = -10
error

4) Items = 2
V = 60 w = 10
:
Error

5) Items = 2
V = 100 w = -20
:
Error.

Conclusion :- Greedy algo takes nlogn whereas brute-force takes exponential a^n time. Program written using allman style.

Task-2

Huffmann coding

① Greedy approach

Huffman encoding has 3 main functions:

① ALGORITHM huffman coding (text)

// i/p : text to be encoded
// o/p :- the root of huffman tree & generated nodes.

→ Initialize a freq dictionary to count occurence of each character in text.

→ create a min-heap priority queue with (weight, node) for each character based on freq.

→ // Building tree
while size of heap > 1 :
→ pop 2 nodes with lowest freq (left, right).
→ create a new node
merged node's freq = left freq + right freq
→ assign left, right as children at this node.

→ push new node back in heap.

→ return root node, call generate_nodes function for root

(2) ALGORITHM generate_nodes (node, codes, current_code)

// I/P : a node of the huffman tree, empty dictionary codes, current code.
// O/P : codes dictionary containing huffman codes of each character.

→ if node is not none:
  → if it has a symbol
    → add its symbol and corresponding current_code to codes_dictionary
  → Recursively call generate_codes for left child with current_code appended with "0".
  → Recursively call generate_codes for right child with current_code appended with "1"

→ return codes { }

(3) ALGORITHM compress_and_calc_ratio(text)

// i/p : text file (any format)
// o/p : compressed string and ratio

→ call the fn huffman coding to get generated codes.
→ create compressed string by concatenating codes for each character in text.
→ original size = length of text * 8 (in bits)
→ compressed size = length of compressed text
→ compressed_ratio = $\frac{\text{compressed\_size}}{\text{original\_size}}$
→ return compressed_text and ratio

TIME complexity

→ Counting frequencies tokens O(K), K is len of text.
→ constructing & initializing min heap takes O(n) n = no. of unique ~~text~~ symbols in text.

→ building tree:
loop executes (n-1) times
in each iteration pop 2 insert 1, each
push, pop takes O(log n)

entire is O(n log n)

O(n log n) is time complexity.

② ALGORITHM huffman_bf (text)

// i/p :- input document
// o/p :- compression ratio, original text size, compressed text size

→ create a list of unique_chars in text.

→ num_bits = Smallest integer that can be represent all unique characters in binary
- ($\log_2$ (len(unique_chars))).

→ codes = { }
→ for each char in unique_chars assign a unique binary code of length num_bits using format (i, '0{num_bits}b')
(i is index of char).

→ create a string compressed by joining codes at all characters of text

→ compressed_size = len(compressed)
→ original_size = len(original_text) * 8
→ compression_ratio = compressed_size / original_size

→ return compressed_size, original_size, ~~compress~~ compression_ratio

TIME COMPLEXITY

→ Assigning binary code to every character = log n.

→ n characters ∴ O(n log n).

Conclusion :- Time complexity of brute force is n log n. Program written using allman style.