DAA Lab 5

Aim → Read about debugging tools in programming language and apply debugging techniques. Application of Knapsack problem and huffman coding.

Experiment 12
~~Fractional Knapsack problem~~
Huffman Encoding / Compression

- Pseudo Code

1] Class huffmanNode
    char, freq, left, right

2] func build_huffman_tree (text)
    freq ← count_freq (text)
    heap ← [huffmanNode (char, freq[char]) for char in freq]
    heapify (heap)
    while len(heap) > 1:
        node1 = heappop (heap)
        node2 = heappop (heap)
        merged = huffmanNode (None, node1.freq + node2.freq, node1, node2)
        heappush (heap, merged)

    return heap [0]

fun __lt__ (self, others)
    return self.freq < others.freq

Page No.
Date

```
fun generate_code(node, code, codes)
    if node <- none
        return
    if node.char <- ~~none~~ not none
        codes[node.char] = code
    generate_code(node.left, code+"0", codes)
    generate_code(node.right, code+"1", codes)

fun compress(text)
    root = build_huffman_tree(text)
    codes = {}
    generate_code(root "", codes)
    compressed = "".join(['codes[char] for
                          char in text'])
    return compressed, codes

fun calculate_compression_ratio(og text,
                                compressed text)
    size = len(og text)*8
    sizec = len(compressed_text)*8
    ratio = ~~compressed~~ sizec / size

    return size, sizec, ratio
```

- Time Complexity
  1. Counting char freq → O(n)
  2. Building heap with len(freq) → O(k log k)
  3. Generating huffman code → O(k)
  4. Compressing text by iterating → O(n)
  5. Calculate Ratio → O(1)

∴ Total time taken = O(n + k + k log k)

o Test Cases

1] Input : file1.html
Expected Output:
Og text size : 1688 bits
Compressed text size = ~~899~~ bits
Compression Ratio : 0.532

2] Input: file~~2~~1.pdf
Expected Output
Og size = 1688 bits
Compressed size = 909 bits
Compression Ratio = 0.538

3] Input: file1.docx
Og size : 1688 bits
Compressed size = 899 bits
Compression Ratio : 0.532

4] Input file: file1. ~~html~~ txt
Og size : 1464 bits
Compressed size : 806 bits
Compression Ratio : 0.55

Experiment 1
Fractional Knapsack problem

```
class item (w, v, sl)
    weight = w
    value = v
    shelf_life = sl

func w_vratio (self)
    return ~~w/v~~ self.weight / self.value

func priority (self)
    return (self.w_vratio (), - self.shelflife)

func fractional_knapsack (capacity, items)
    if capacity <= 0 or items <- empty
        raise error of invalid capacity

    items.sort (key = lamda x: x.priority (),
                                  reverse = true)
    total_~~v~~v = 0
    current_w = 0

    for item in items:
        if current_w + item.weight <= capacity
            current_w += item.weight
            total_v += item.value
    else
        remaining_capacity = capacity - current_w
        total_v += item.value * (remaining_
                                  capacity /
                                   item.wt)
        break
return total_v
```

Page No. | Date

- Time Complexity
1] w_ratio → O(1)
2] Priority → O(1)
3] Sorting items → O(n log n)
4] Adding items in knapsack → O(n)

∴ ~~Total~~

∴ Total time = O(n) + O(n log n) + 2O(1)

- Test Cases

1] Input → [item(18,47,3), item(14,30,9) .... item(5,87,3)]
(100 items)
Output → Knapsack Value = 2878.33

2] Input → [item1(17,41,6), item2(8,29,2), ... item100(19,93,1)]
Output → Knapsack Value = 3018.02

3] Input → [item1(0,0,8) item2(7,9,0) .... item100(19,93,1)]
Output → Error → Weight and Value cannot be 0

4] Input → [item1(-10,0,8) item2(7,-9,10) .... item100(20,21,3)]
Output → Error → Weight cannot be -ve

Conclusion → Hence we have seen the application of Knapsack problem in ~~O(n + k log k) and~~ O(n + n log n) and Huffman coding in O(n + k log k).