**Assignment 6**

**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Title:**

Assignment Based on Greedy Approach to Implement Huffman Coding Algorithm (Determine Time and space complexity)

**Theory:**

## Objective

To implement the Huffman encoding algorithm using the greedy strategy for lossless data compression and determine its time and space complexity.

## Theory and Explanation

Huffman encoding is a popular greedy algorithm used for lossless data compression. The main idea is to assign variable-length binary codes to characters based on their frequencies such that characters with higher frequency get shorter codes. This reduces the overall size of the encoded data.

The algorithm builds a binary tree called the Huffman tree where each leaf node represents a character with its frequency. It repeatedly merges the two nodes with the smallest frequencies until only one tree remains. The path from root to each leaf gives the unique prefix-free code for each character.

By using prefix-free codes, it ensures that no code is a prefix of another, making decoding unambiguous. Huffman encoding is widely used in compression file formats like ZIP, JPEG, and MP3.

## Key Points

* Huffman encoding assigns shorter codes to more frequent characters and longer codes to less frequent ones.
* Uses a priority queue (min-heap) to efficiently access the nodes with the least frequency.
* Builds a binary tree by merging two smallest nodes repeatedly.
* Produces prefix codes for unambiguous decoding.
* Greedy because it always merges nodes with the smallest frequencies at each step.

## Pseudocode

text

function HuffmanEncoding(char[] chars, int[] freq):

create a priority queue Q to hold nodes

for each character c in chars:

create leaf node for c with its frequency, add to Q

while Q.size > 1:

left = Q.removeMin()

right = Q.removeMin()

merged = new Node(freq=left.freq + right.freq)

merged.left = left

merged.right = right

Q.add(merged)

root = Q.removeMin()

codes = empty map

generateCodes(root, "", codes)

return codes

function generateCodes(node, code, codes):

if node is leaf:

codes[node.char] = code

else:

generateCodes(node.left, code + "0", codes)

generateCodes(node.right, code + "1", codes)

## Example Output

Input:

text

Enter the number of characters:

4

Enter characters:

A B C D

Enter their frequencies:

5 9 12 13

Output:

text

Huffman Codes:

A: 00

B: 01

C: 10

D: 11

**Code:**

import java.util.\*;

class Node implements Comparable<Node> {

    char ch;

    int freq;

    Node left, right;

    Node(char ch, int freq) {

        this.ch = ch;

        this.freq = freq;

    }

    Node(int freq, Node left, Node right) {

        this.ch = '\0';

        this.freq = freq;

        this.left = left;

        this.right = right;

    }

    public int compareTo(Node other) {

        return this.freq - other.freq;

    }

}

public class HuffmanCoding {

    public static void generateCodes(Node root, String code, Map<Character, String> codes) {

        if (root == null)

            return;

        if (root.left == null && root.right == null) {

            codes.put(root.ch, code);

        }

        generateCodes(root.left, code + "0", codes);

        generateCodes(root.right, code + "1", codes);

    }

    public static Map<Character, String> buildHuffmanTree(char[] chars, int[] freq) {

        PriorityQueue<Node> pq = new PriorityQueue<>();

        for (int i = 0; i < chars.length; i++) {

            pq.add(new Node(chars[i], freq[i]));

        }

        while (pq.size() > 1) {

            Node left = pq.poll();

            Node right = pq.poll();

            Node merged = new Node(left.freq + right.freq, left, right);

            pq.add(merged);

        }

        Node root = pq.poll();

        Map<Character, String> codes = new HashMap<>();

        generateCodes(root, "", codes);

        return codes;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of characters:");

        int n = sc.nextInt();

        char[] chars = new char[n];

        int[] freq = new int[n];

        System.out.println("Enter characters:");

        for (int i = 0; i < n; i++) {

            chars[i] = sc.next().charAt(0);

        }

        System.out.println("Enter their frequencies:");

        for (int i = 0; i < n; i++) {

            freq[i] = sc.nextInt();

        }

        Map<Character, String> huffmanCodes = buildHuffmanTree(chars, freq);

        System.out.println("Huffman Codes:");

        for (char c : chars) {

            System.out.println(c + ": " + huffmanCodes.get(c));

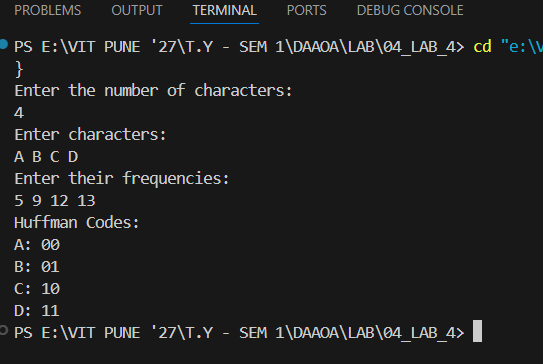
        }

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity

* Building the priority queue takes O(n) for n characters.
* Each merge operation occurs n−1 times, and each operation involves extracting two min nodes and adding one new node, each costing O(logn) because of the priority queue.
* Total time complexity: **O(nlogn)**.

## Space Complexity

* The tree stores all nodes including merged internal nodes; a total of 2n−1 nodes for n characters.
* Codes are stored for each character.
* Priority queue space: O(n)
* Overall space complexity: **O(n)**

## Pseudocode with Complexity Comments

text

CLASS Node

VARIABLE ch, freq

VARIABLE left, right

FUNCTION Node(ch, freq) // Constructor for leaf node

this.ch ← ch // Time: +1

this.freq ← freq // Time: +1

ENDFUNCTION

FUNCTION Node(freq, left, right) // Constructor for internal node

this.ch ← '\0' // Time: +1

this.freq ← freq // Time: +1

this.left ← left // Time: +1

this.right ← right // Time: +1

ENDFUNCTION

FUNCTION compareTo(other) // For priority queue ordering

RETURN this.freq - other.freq // Time: +1

ENDFUNCTION

ENDCLASS

FUNCTION generateCodes(root, code, codes) // Generate Huffman codes by tree traversal

IF root == null // Time: +1 per call

RETURN

IF root.left == null AND root.right == null // Leaf node // Time: +1 per leaf

codes[root.ch] ← code // Store code for character // Time: +1

generateCodes(root.left, code + "0", codes) // Traverse left subtree // Time: O(n) overall

generateCodes(root.right, code + "1", codes) // Traverse right subtree // Time: O(n) overall

ENDFUNCTION

FUNCTION buildHuffmanTree(chars, freq) // chars, freq: arrays of size n

DECLARE priorityQueue pq // Space: +n for storing nodes

FOR i = 0 TO n-1 // Time: +n

pq.add(new Node(chars[i], freq[i])) // Time: O(log n) per insertion

WHILE pq.size() > 1 // Time: O(n log n) for building tree

left ← pq.poll() // Extract min node // Time: O(log n)

right ← pq.poll() // Extract next min // Time: O(log n)

merged ← new Node(left.freq + right.freq, left, right) // Create new node // Time: +1

pq.add(merged) // Insert merged node // Time: O(log n)

root ← pq.poll() // Root of Huffman Tree // Time: +1

DECLARE codes map // Space: +n for character codes

generateCodes(root, "", codes) // Time: O(n) to generate codes

RETURN codes // Map char → code // Time: +1

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter the number of characters:" // Time: +1

INPUT n // Time: +1

DECLARE chars[n], freq[n] // Space: +2n

PRINT "Enter characters:" // Time: +1

FOR i = 0 TO n-1 // Time: +n

INPUT chars[i] // Time: +1 per input

PRINT "Enter their frequencies:" // Time: +1

FOR i = 0 TO n-1 // Time: +n

INPUT freq[i] // Time: +1 per input

huffmanCodes ← buildHuffmanTree(chars, freq) // Time: O(n log n), Space: O(n)

PRINT "Huffman Codes:" // Time: +1

FOR each c IN chars // Time: +n

PRINT c + ": " + huffmanCodes[c] // Time: +1 per output

CLOSE scanner // Time: +1

ENDFUNCTION

**Conclusion**

In this lab exercise, we learned how to implement Huffman Encoding problem using Greedy Strategy.