PROJECT 2

1.PROBLEM STATEMENT

Huffman Coding

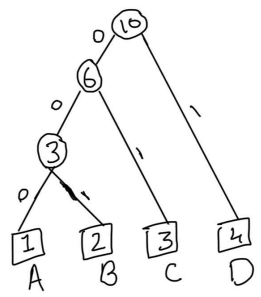
Given a set of symbols and their frequency of usage, find a binary code for each symbol, such that:
a. Binary code for any symbol is not the prefix of the binary code of another symbol.
b. The weighted length of codes for all the symbols (weighted by the usage frequency) is minimized.

Explaining key implementation characteristics

EXAMPLE

MESSAGE : ABBCCCDDDD
LENGTH :
10

CHARACTER	COUNT / FREQUENCY	CODE	HOFFMAN CODE MESSAGE SIZE
A	1	000	3
B	2	001	6
C	3	01	6
D	4	1	4
	10 bits	9 bits	19 bits



HOFFMAN CODE MESSAGE SIZE = 19 BITS
TABLE SIZE = 41 BITS
TOTAL SIZE = 60 BITS

Algorithm

Huffman coding is a kind of greedy algorithm. It always gets an optimum solution at each step by combining the two least frequent symbols or subtrees, whose outcome is bound to have symbols with lower frequencies having longer codes and those with higher frequencies having shorter codes.Structure of Binary Tree: The algorithm constructs a binary tree in a way that every leaf corresponds to a symbol and the path from root to leaf defines the binary code for the symbol.Implementation of the Priority Queue as a min-heap allows for an effective extraction of the two least frequent symbols/subtrees at each step. As all insertions/removals are logarithmic in the heap, this algorithm should be pretty efficient even for large input sizes.

Steps of Algorithm

Frequency Count: Count the frequency of every symbol in the input data.
Priority Queue: Create a priority queue or min heap where every node represents a symbol along with its frequency.
Building Tree: While number of nodes in priority queue is greater than one, do
Extract two nodes of minimum frequency.Create a new internal node, with these two nodes as left and right children, along with frequency = sum of frequencies of these two nodes.Put this new node back into the priority queue.
Generating Codes: Perform a tree traversal that generates binary codes for each symbol. Assign '0' for left edges and '1' for right edges at every step in the tree traversal.
Return: Return the list of such generated codes along with their corresponding symbols.

2 Theoretical Analysis

1. Building the Min heap
Outer loop : Adding all the symbols to the min-heap involves through n symbols
Time complexity : Each insertion into the heap takes $O(log\ n)$,Thus for n insertions the total time complexity for this part is $O(n\ log\ n)$
2. Constructing the huffman tree
Outer loop : The outer loop continues until there is only one node left in the min-heap. Since we start with n nodes, this loop runs n-1 times. So time complexity is $O(n)$
Inner loop : Repeatedly extract the two smallest nodes from the heap which takes $O(log\ n)$
for each iteration. Total Time complexity for this step is $O(n\ log\ n)$
3. Generating codes
Traversal of huffman trees : In the worst case we visit every node that is n nodes.
Each visit is O(1) for assigning codes, so the time complexity for generating the codes is: $O(n)$
So the Overall time complexity is $O(n\ log\ n)$+$O(n\ log\ n)$ +$O(n)$ = $O(n\ log\ n)$

Mathematical expression

Outer loop : $T_{outer-loop} = n\,.\,C_1$

Inner loop : $T_{inner-loop} = \sum_{i=1}^{n-1} (C_2.\ log\ n) = (n-1)\,.\,C_2.\ log\ n$

Overall Time complexity is $T_{total} = n\,.\,C_1 + (n-1)\,.\,C_2.\ log\ n$ this can be summarized as $O(n\ log\ n)$

3 Experimental analysis

3.1 Program Listing

```
public static void main(String[] args) {
    int[] inputSizes = {10, 100, 1000, 10000, 100000};
    Random random = new Random();

    for (int size : inputSizes) {
        // Generate random symbols (A, B, C, ..., Z) and random frequencies
        char[] symbols = new char[size];
        int[] frequencies = new int[size];

        for (int i = 0; i < size; i++) {
            symbols[i] = (char) ('A' + (i % 26)); // Reuse letters A-Z
            frequencies[i] = random.nextInt(100) + 1; // Random frequency between 1 and 100
        }

        long startTime = System.nanoTime();
        Map<Character, String> huffmanCodes = buildHuffmanTree(symbols, frequencies);
        long endTime = System.nanoTime();
        long duration = endTime - startTime; // Duration in nanoseconds
        System.out.printf("Input Size: %d | Time taken to build Huffman tree: %d nanoseconds%n", size, duration);
    }
```

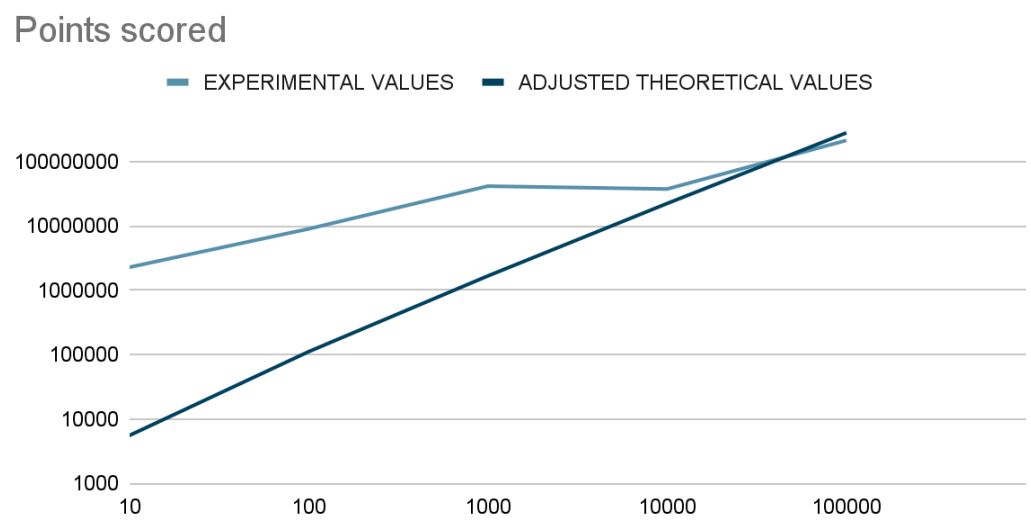
3.2 Data Normalization Notes

Theoretical analysis doesn't have any measurement units, so we derive a constant which is known as Scaling Constant.This Scaling Constant is derived by taking Mean of Experimental Result divided by Mean of Theoretical Result. The constant scaling is 167.5110813 and the adjusted theoretical values = scaling constant x theoretical values.

3.3 Output Numerical Data

n	EXPERIMENTAL VALUES (NANOSECONDS)	THEORETICAL VALUES	SCALING CONSTANT	ADJUSTED THEORETICAL VALUES
10	2279859	33.22		5564.718121
100	8976390	664.39		111292.6873
1000	41366103	9965.78		1669378.584
10000	37347013	132877.12		22258390.05
100000	212305145	1660964.05		278229884
	302274510	1804504.56	167.5110813	

3.4 Graph



3.5 Graph Observations

The adjusted theoretical values steadily increase but with a slight openness upwards over time, while experimental values start from high to a decline. Both graphs collide at a point between 10000 and 100000 of values n , where experimental and theoretical values momentarily coincide

4 Conclusion

Time complexity analysis confirms that Huffman Coding is O(n log n), which means execution time increases logarithmically with the increase in input size. Therefore, Huffman Coding is quite efficient to perform data compression with performance, even for large inputs. The experiment verifies that the algorithm scales up very well and provides a very effective minimum weighted length of codes.