Project 2

Aniket Konkar

1. Problem Statement

Option 9: 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.

2. Theoretical Analysis

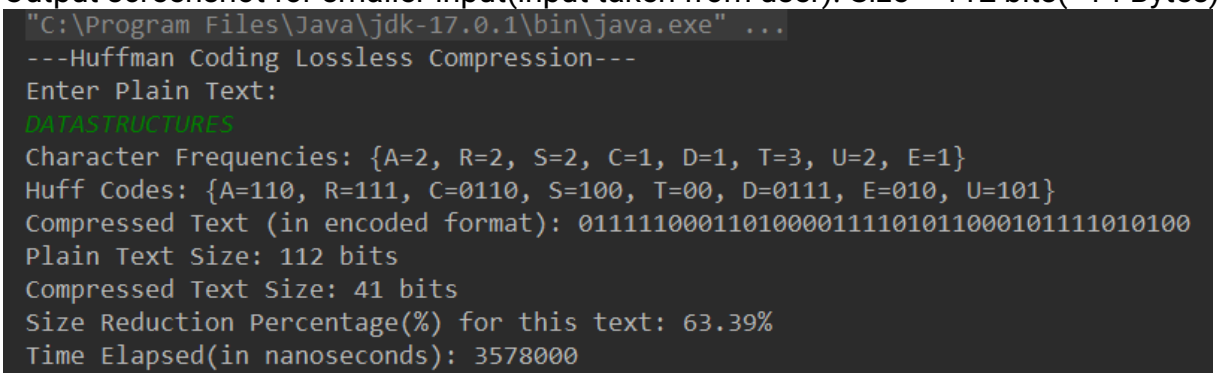
Huffman coding is a lossless data compression algorithm. Below linked code implementation compresses input data in the following way – For a given input text:

1. Count frequencies of individual unique characters.(This is already given in the problem statement, however, have implemented code to count unique symbols and their frequencies)
2. Create a leaf node for each unique character which consists of that character and its frequency and build a min heap(aka Priority Queue) of all these leaf nodes.
3. Extract two nodes with the minimum frequencies from the min heap.
4. Create a new node with its frequency equal to the sum of the two extracted nodes' frequencies and insert this node back into the min heap.
5. Repeat steps 3 and 4 until the min heap contains only one node. This remaining node is the root node, and the Huffman tree is complete.
6. Traverse the Huffman tree and assign a value of 0 when moving in the left direction and a value of 1 when moving in the right direction. The path from the root node to a leaf node represents the binary Huffman code for that symbol. All these codes are prefix-free i.e., no code is a prefix of another code.
7. Replace every plain text character with its respective binary Huffman code to form the compressed encoded text.

Time complexity:
Inserting nodes into the min heap tree – requires O(*log n*) time for every insertion. We have *n* leaf nodes – where n represents the total number of unique symbols/characters in the input data. We know that a tree with *n* leaves has *2n-1* nodes. Hence, **this algorithm runs in O(*n log n*) time**, where n is the number of unique symbols/characters in the input data.

3. Experimental Analysis - 3.1 Program Listing

Code:https://github.com/insp7/CSCI_6212/blob/master/Project2HuffmanCoding/src/HuffmanCoding.java
To execute the code, run the HuffmanCoding.java file. **Note: `HuffmanCoding.java` depends on the files - `FreqComparator.java` and `FreqMapHeapTreeNode.java` present in the *src* folder to run successfully.**
Output screenshot for smaller input(input taken from user). Size = 112 bits(=14 Bytes)



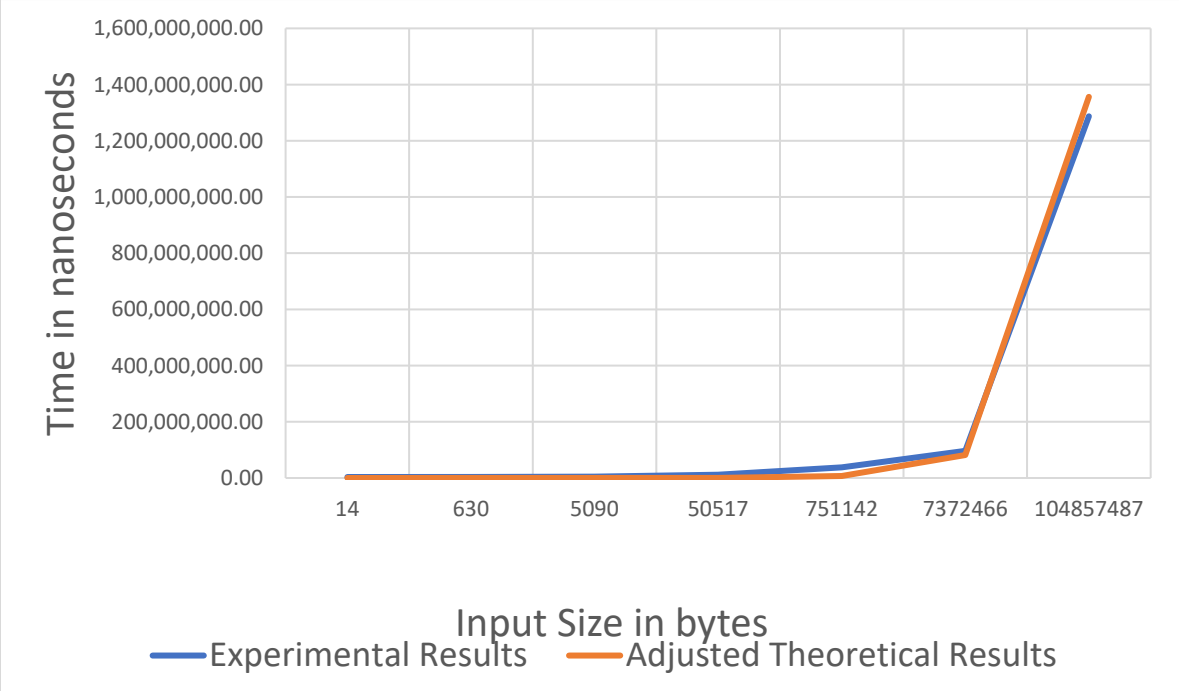
3.2 Data Normalization Notes

The Scaling Constant used here is 0.485480735. The formula to calculate scaling constant:
Scaling Constant = Average of Experimental Results (in ns) / Average of Theoretical Results.
Also, *Adjusted Theoretical Result = Scaling Constant * Theoretical Result*

3.3 Output Numerical Data

Input Values (Size in Bytes)	Experimental Results (in ns)	Theoretical Results	Adjusted Theoretical Results
14	3506100	53.30	25.88
630	3950301	5858.50	2844.19
5090	4917599	62675.46	30427.73
50517	11591400	789301.92	383190.88
751142	37917501	14661334.99	7117795.69
7372466	96533399	168193344.31	81654628.49
104857487	1287112401	2793807641.02	1356339788.14
Average =>	206504100.1	425360029.9	

3.4 Graph



3.5 Graph Observations

For smaller values of n, the theoretical time taken is smaller than the time portrayed by experimental results. However, for large values of n, theoretical values are greater than the experimental values as expected. Hence, the time taken for the algorithm will be no longer than the theoretical values which is the upper bound.

4. Conclusions

The time complexity for this algorithm is **O(*n * log n*)**. Overall, it is observed that the theoretical results are similar to the experimental results.