*Project Report*

On

Lossless Data Compression Algorithms

Submitted by:

*Sparsh Gupta* || *Tanmay Singh*

170001049 || 170001051

Computer Science and Engineering

2nd year

Under the Guidance of

*Dr. Kapil Ahuja*



Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2019

**Introduction**

Data compression is a process by which a file (Text, Audio, and Video) can be compressed, such that the original file may be fully recovered without any loss of actual information. This process may be useful if one wants to save the storage space. The exchanging of compressed file over internet is very easy as they can be uploaded or downloaded much faster. Data compression has important application in the area of file storage and distributed system.

In short, Data Compression is the process of encoding data to fewer bits than the original representation so that it takes less storage space and less transmission time while communicating over a network. Data Compression is possible because most of the real-world data is very redundant. A compression program is used to convert data from an easy-to-use format to one optimized for compactness. Likewise, an uncompressing program returns the information to its original form.

**Project Objective**

* To analyze the existing data compression algorithms.
* To implement these algorithms.
* To improve the available algorithm for better efficiency and lesser time complexity (if possible).

**Some Data Compression Techniques**

* Huffman Encoding
* Shannon-Fano Encoding
* Run Length Encoding
* Arithmetic Encoding

**Huffman Encoding Technique**

--A binary code tree is generated in Huffman Coding for given input. A code length is constructed for every symbol of input data based on the probability of occurrence. Huffman codes are part of several data formats as ZIP, GZIP and JPEG. It is a sophisticated and efficient lossless data compression technique. The symbol with highest probability has the shortest binary code and the symbol with lowest probability has the longest binary code.

--There are mainly two major parts in Huffman Coding

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

--**Pseudocode**:

HUFFMAN(C)

1 n = |C|

2 Q = C

3 for i = 1 to n - 1

4 allocate a new node z

5 z.left = x = EXTRACT-MIN(Q)

6 z.right = y = EXTRACT-MIN(Q)

7 z.freq = x.freq + y.freq

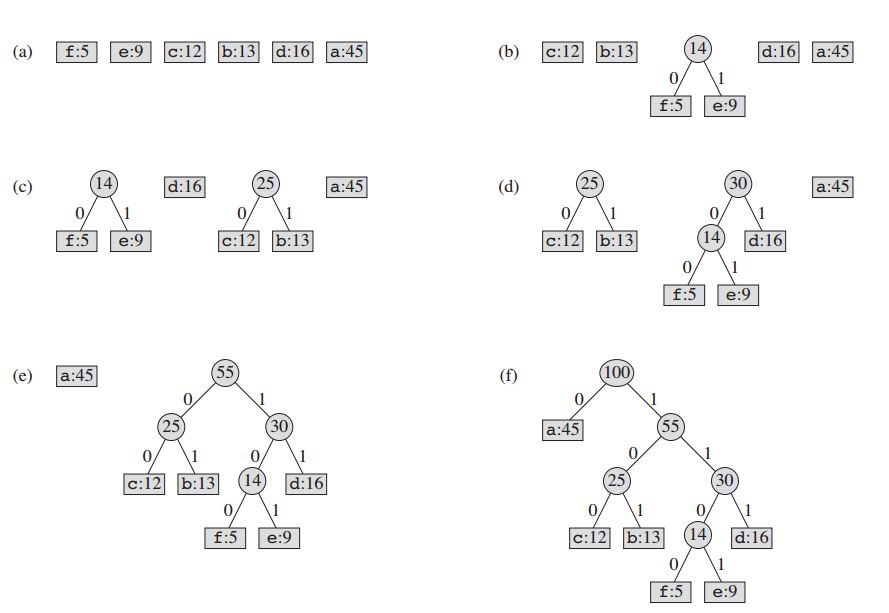
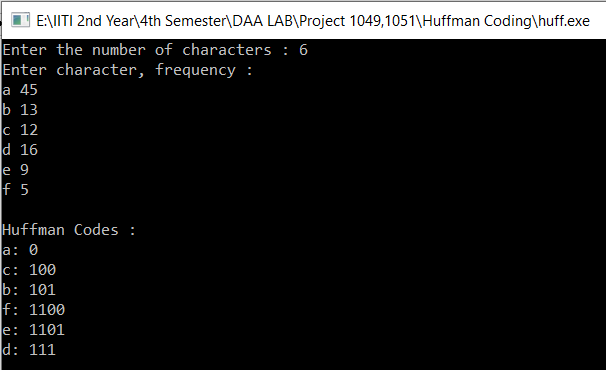
8 INSERT(Q,z)

9 return EXTRACT-MIN(Q) // return the root of the tree

--**C++ code**:

[*Huffman Coding\huff.cpp*](Huffman%20Coding/huff.cpp)

--**Result:**



--**Complexity Analysis:**

**Huffman Build complexity**: **O(n logn)** where n is the number of unique characters, if there are n nodes. The for loop in lines 3–8 executes exactly n - 1 times, and since each extractMin() takes O(logn) time as it calls minHeapify(), the loop contributes O(n logn) to the running time. Thus, the total running time of HUFFMAN on a set of n characters is O(n logn). We can reduce the running time to **O(n log logn)** by replacing the binary min-heap with a \***van Emde Boas** tree.

\* Chapter 20 of Introduction To Algorithms By Thomas H. Cormen

--**Greedy Explanation**:

Huffman coding looks at the occurrence of each character and stores it as a binary string in an optimal way. The idea is to assign variable-length codes to input characters, length of the assigned codes are based on the frequencies of corresponding characters. We create a binary tree and operate on it in bottom-up manner so that the least two frequent characters are as far as possible from the root. In this way, the most frequent character gets the smallest code and the least frequent character gets the largest code.

**Shannon-Fano Encoding Technique**

--This is one of an earliest technique for data compression that was invented by Claude Shannon and Robert Fano in 1949.In this technique, a binary tree is generated that represent the probabilities of each symbol occurring. The symbols are ordered in a way such that the most frequent symbols appear at the top of the tree and the least likely symbols appear at the bottom.

--**Pseudocode**:

1. begin
2. count source units
3. sort source units to non-decreasing order
4. SF-Split(S)
5. output(count of symbols, encoded tree, symbols)
6. write output
7. end
8. procedure SF-Split(S)
9. begin
10. if (|S|>1) then
11. begin
12. divide S to S1 and S2 with about same count of units
13. add 1 to codes in S1
14. add 0 to codes in S2
15. SF-Split(S1)
16. SF-Split(S2)
17. end
18. end

--**C++ code**:

[*Shannon-Fano\shannon fano.cpp*](Shannon-Fano/shannon%20fano.cpp)

--**Complexity Analysis** :

O(l + n logn)

where l: length of the input text

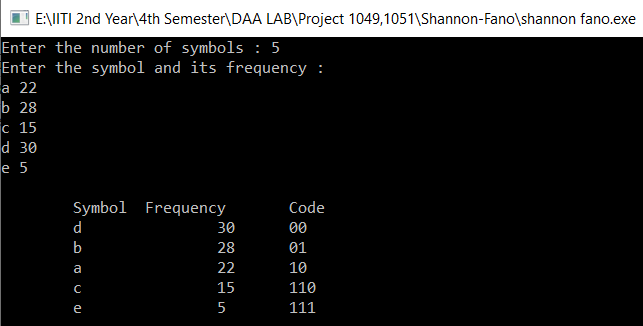
n: distinct number of symbols in the input text

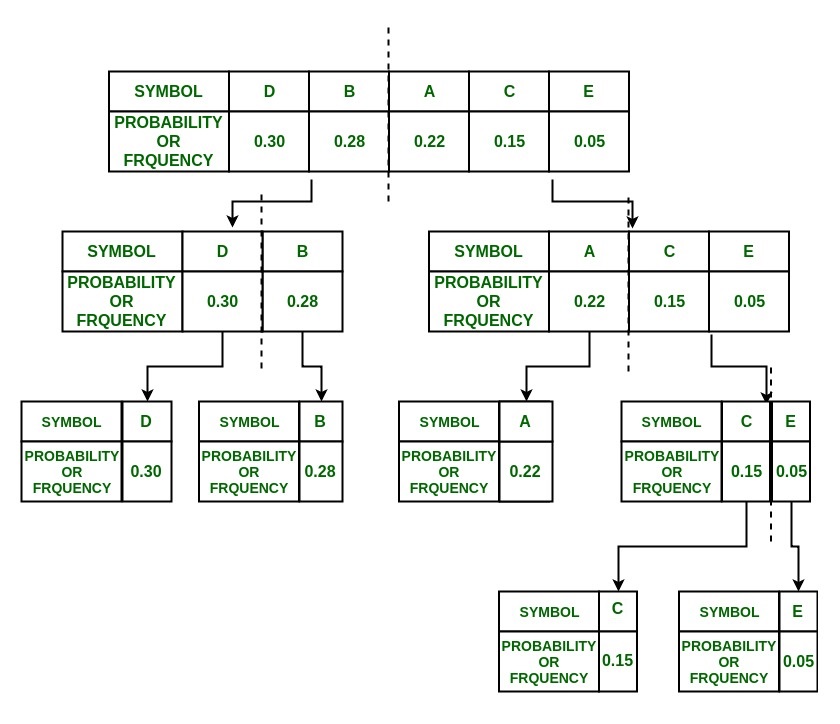
Dividing the problem into three phases. Time complexity of the first phase of computing counts of symbols appearance is linearly dependent on length of the input text. The next phase is ordering symbols, which is implementation dependent. The last phase is generating of the output. This also linearly depends on length of the input text.

--**Explanation**:

In this algorithm, we DIVIDE the set S into two subsets on the basis of minimum difference of sum of probabilities. Hence, it follows the divide and conquer approach.

--**Result:**





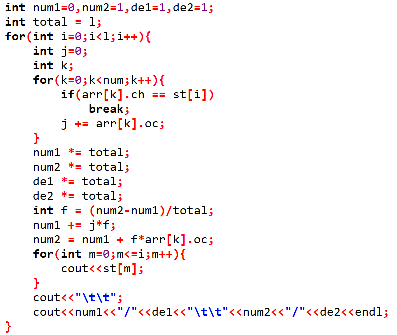
**Arithmetic Encoding Technique**

--Arithmetic encoding is the most powerful compression techniques. This converts the entire input data into a range of floating-point numbers. A floating-point number is similar to a number with a decimal point, like 4.5 instead of 4 1/2. However, in arithmetic coding we are not dealing with decimal number so we call it a floating point instead of decimal point.

--**C++ code :**

[*Arithmetic Coding\arco.cpp*](Arithmetic%20Coding/arco.cpp)

--**Pseudocode:**



--**Complexity Analysis**:

Best Case: O(l)

Worst Case: O(l\*n)

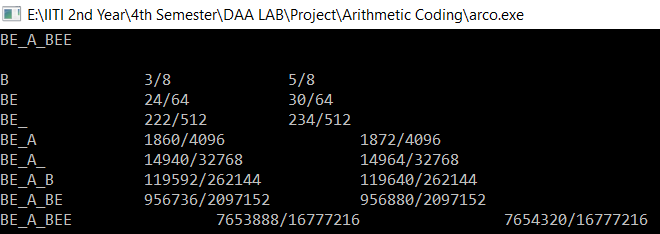
where l: length of the input string

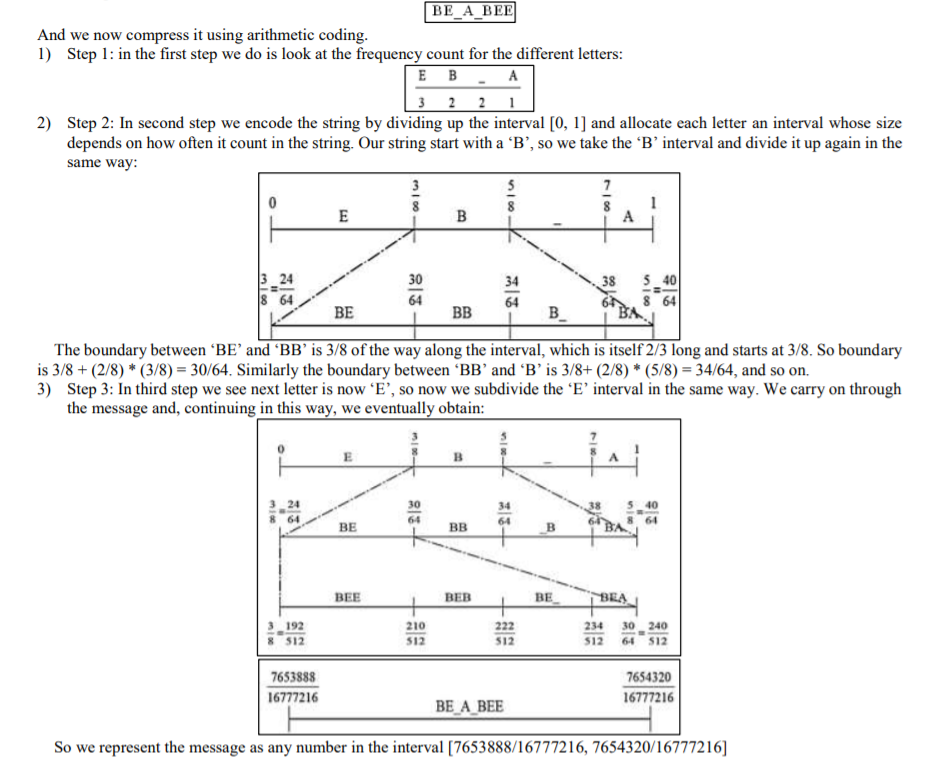
n: distinct number of symbols in the input string

Best case when all the symbols in the string are same or there is only one unique symbol.

Worst case when all the symbols in the string are different.

--**Result**:





**Run-Length Encoding Technique**

--Data often contains sequences of identical bytes. Replacing these repeated byte sequences with the number of occurrences, a reduction of data can be achieved. RLE basically compresses the data by reducing the physical size of a repeating string of characters. It is mainly used when the data contain sequences of identical bytes or data has less entropy.

--**Pseudocode**:

Loop: count = 0

REPEAT

get next symbol

count = count + 1

UNTIL (symbol unequal to next one)

output symbol

IF count > 1

output count

GOTO Loop

--**Complexity Analysis:**

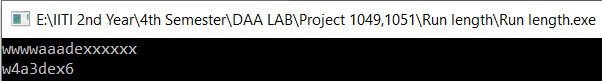
O(n)

where n: length of the input text

--**C++ code**:

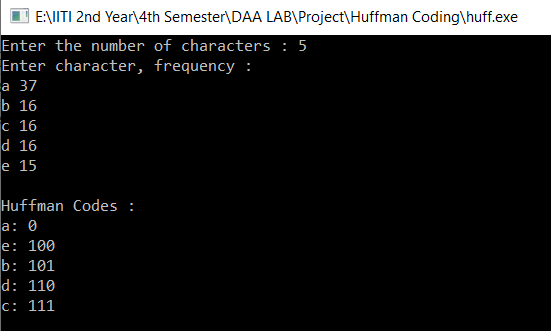
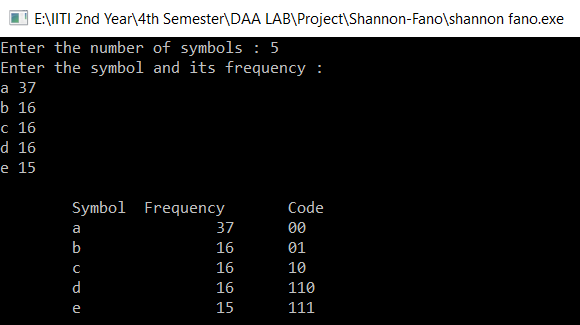
[*Run length\Run length.cpp*](Run%20length/Run%20length.cpp)

--**Result:**



**Huffman Vs Shannon-Fano**

* In Huffman encoding, codes are built bottom up by repeatedly combining the two least common entries in the list of populations until only two are left.
* In Shannon-Fano, the population list is sorted by pop count and then repeatedly (recursively) split in two - with half the population in each half, or as close as one can get - until only two entries are left in a sub-section.
* Huffman has been proven to always produce the (an) optimal prefix encoding whereas Shannon-Fano is (can be) slightly less efficient. Shannon-Fano, on the other hand, is arguably a bit simpler to implement.

For frequency distribution: (37,16,16,16,15),

Huffman code length: 37\*1 + 16\*3 + 16\*3 + 16\*3 + 15\*3 = 226

Shannon-Fano Code Length: 37\*2 + 16\*2 + 16\*2 + 16\*3 + 15\*3 = 231

*“The Shannon-Fano code yields larger average code length relative to the Huffman code.”*

**Conclusion and Future Work**

* The Shannon-Fano code yields larger average code length relative to the Huffman code.
* The total running time of HUFFMAN on a set of n characters is O(n logn). We can reduce the running time to **O(n log logn)** by replacing the binary min-heap with a \***van Emde Boas** tree.

**References**

* *https://ieeexplore.ieee.org/document/6824486*
* *https://www.ripublication.com/irph/ijict\_spl/07\_ijictv3n3spl.pdf*
* *http://www.ijirst.org/articles/IJIRSTV4I1078.pdf*
* *http://www.stringology.org/DataCompression/sf/index\_en.html*
* *https://www.cse.iitk.ac.in/users/satyadev/sp14/chap5.pdf*
* *https://geeksforgeeks.com*
* *https://Wikipedia.com*
* Introduction to Algorithms by Thomas H. Cormen