

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 $z.\text{left} = x = \text{EXTRACT-MIN}(Q)$

6 $z.\text{right} = y = \text{EXTRACT-MIN}(Q)$

7 $z.\text{freq} = x.\text{freq} + y.\text{freq}$

8 $\text{INSERT}(Q, z)$

9 return $\text{EXTRACT-MIN}(Q)$ // return the root of the tree

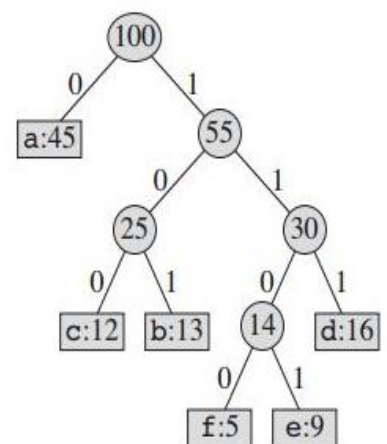
--C++ code:

[Huffman Coding\huff.cpp](#)

--Result:

```
E:\IITI 2nd Year\4th Semester\DAA LAB\Project 1049,1051\Huffman Coding\huff.exe
Enter the number of characters : 6
Enter character, frequency :
a 45
b 13
c 12
d 16
e 9
f 5

Huffman Codes :
a: 0
c: 100
b: 101
f: 1100
e: 1101
d: 111
```



--Complexity Analysis:

Huffman Build complexity: $O(n \log n)$ 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 $O(\log n)$ time as it calls `minHeapify()`, the loop contributes $O(n \log n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \log n)$. We can reduce the running time to **$O(n \log \log n)$** 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.
9. procedure SF-Split(S)
10. begin
11. if ($|S| > 1$) then
12. begin
13. divide S to S1 and S2 with about same count of units
14. add 1 to codes in S1
15. add 0 to codes in S2
16. SF-Split(S1)
17. SF-Split(S2)
18. end
19. end

--C++ code:

[Shannon-Fano\shannon_fano.cpp](#)

--Complexity Analysis :

$$O(l + n \log n)$$

where l : length of the input text

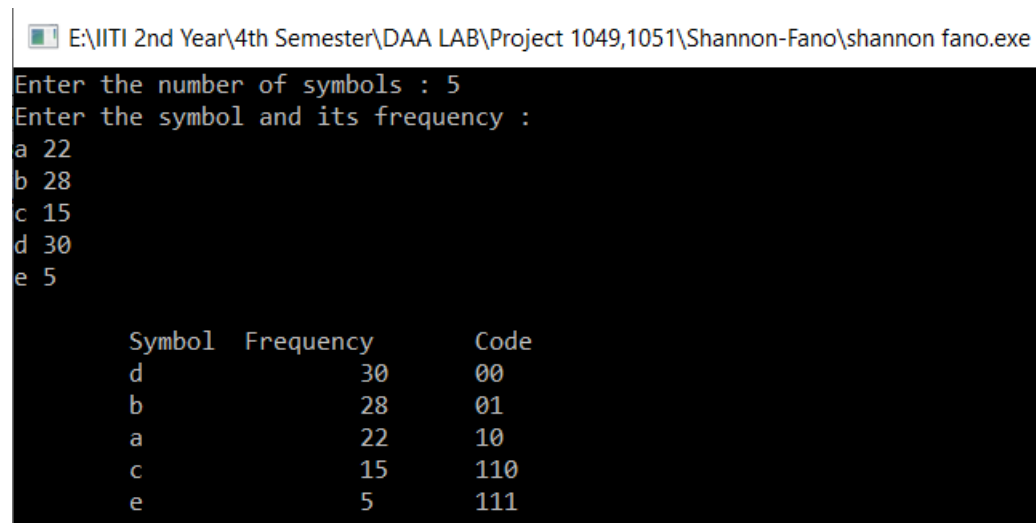
n : distinct number of symbols in the input text

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:

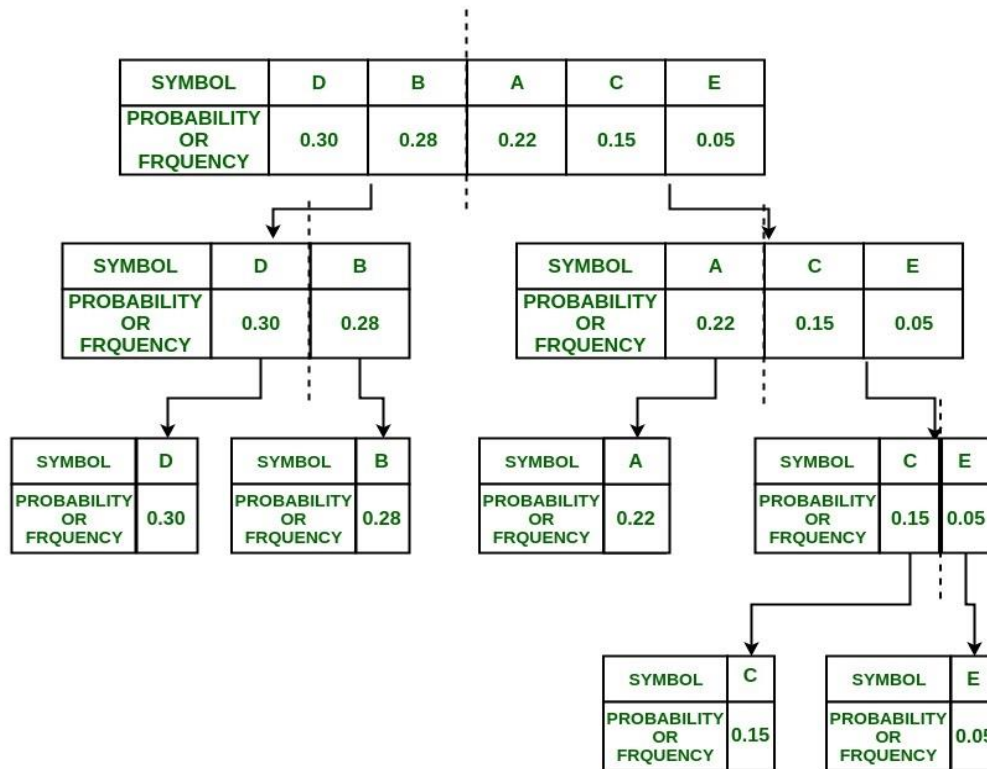
In this algorithm, we DIVIDE the set S into two subsets on the basis of minimum difference of sum of probabilities. Hence, it follows the divide and conquer approach.

--Result:



```
E:\IITI 2nd Year\4th Semester\DAA LAB\Project 1049,1051\Shannon-Fano\shannon fano.exe
Enter the number of symbols : 5
Enter the symbol and its frequency :
a 22
b 28
c 15
d 30
e 5

Symbol  Frequency  Code
d        30        00
b        28        01
a        22        10
c        15        110
e         5        111
```



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](#)

--Pseudocode:

```
int num1=0,num2=1,de1=1,de2=1;
int total = 1;
for(int i=0;i<l;i++){
    int j=0;
    int k;
    for(k=0;k<num;k++){
        if(arr[k].ch == st[i])
            break;
        j += arr[k].oc;
    }
    num1 *= total;
    num2 *= total;
    de1 *= total;
    de2 *= total;
    int f = (num2-num1)/total;
    num1 += j*f;
    num2 = num1 + f*arr[k].oc;
    for(int m=0;m<=i;m++){
        cout<<st[m];
    }
    cout<<"\t\t";
    cout<<num1<<"/"<<de1<<"\t\t"<<num2<<"/"<<de2<<endl;
}
```

--Complexity Analysis:

Best Case: $O(1)$

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:

E:\IITI 2nd Year\4th Semester\DAA LAB\Project\Arithmetic Coding\arco.exe

```
BE_A_BEE
B          3/8          5/8
BE         24/64         30/64
BE_        222/512        234/512
BE_A       1860/4096        1872/4096
BE_A_      14940/32768      14964/32768
BE_A_B     119592/262144    119640/262144
BE_A_BE    956736/2097152    956880/2097152
BE_A_BEE   7653888/16777216    7654320/16777216
```

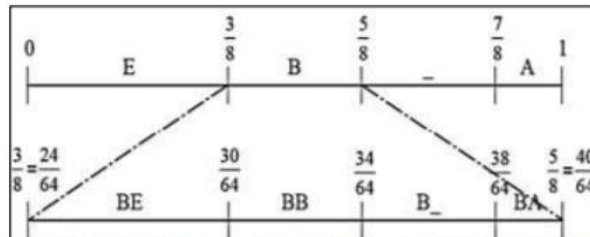
BE_A_BEE

And we now compress it using arithmetic coding.

- 1) Step 1: in the first step we do is look at the frequency count for the different letters:

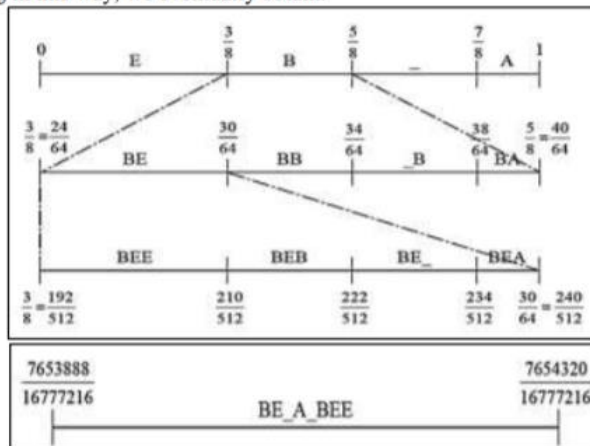
E	B	-	A
3	2	2	1

- 2) Step 2: In second step we encode the string by dividing up the interval $[0, 1]$ and allocate each letter an interval whose size depends on how often it count in the string. Our string start with a 'B', so we take the 'B' interval and divide it up again in the same way:



The boundary between 'BE' and 'BB' is $3/8$ of the way along the interval, which is itself $2/3$ long and starts at $3/8$. So boundary is $3/8 + (2/8) * (3/8) = 30/64$. Similarly the boundary between 'BB' and 'B' is $3/8 + (2/8) * (5/8) = 34/64$, and so on.

- 3) Step 3: In third step we see next letter is now 'E', so now we subdivide the 'E' interval in the same way. We carry on through the message and, continuing in this way, we eventually obtain:



So we represent the message as any number in the interval $[7653888/16777216, 7654320/16777216]$

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:

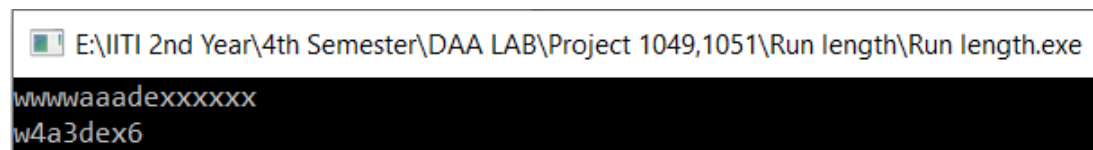
$O(n)$

where n: length of the input text

--C++ code:

[*Run length\Run length.cpp*](#)

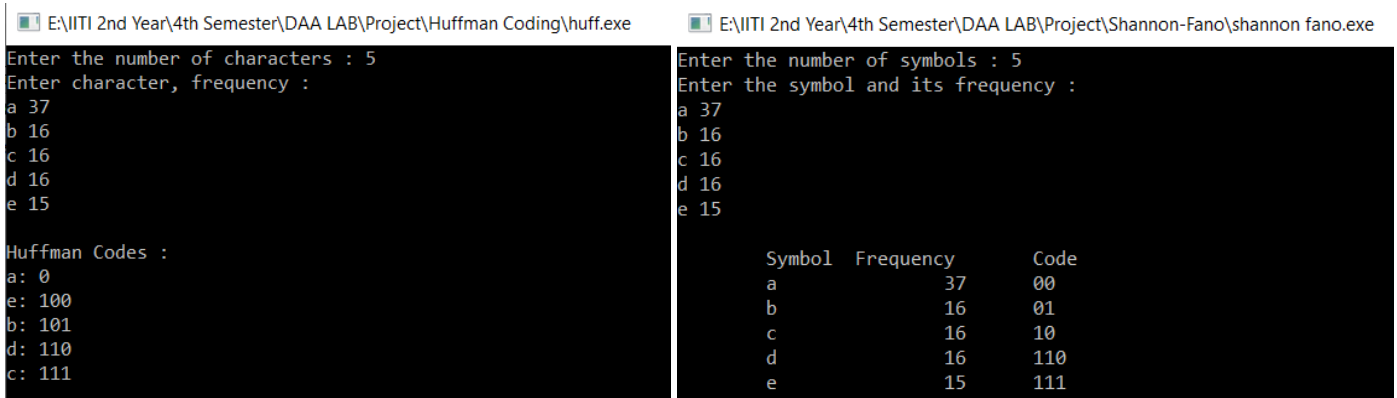
--Result:



```
E:\IITI 2nd Year\4th Semester\DAA LAB\Project 1049,1051\Run length\Run length.exe
wwwaaaadexxxxxx
w4a3dex6
```

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.



```
E:\IITI 2nd Year\4th Semester\DAA LAB\Project\Huffman Coding\huff.exe
Enter the number of characters : 5
Enter character, frequency :
a 37
b 16
c 16
d 16
e 15

Huffman Codes :
a: 0
e: 100
b: 101
d: 110
c: 111

E:\IITI 2nd Year\4th Semester\DAA LAB\Project\Shannon-Fano\shannon fano.exe
Enter the number of symbols : 5
Enter the symbol and its frequency :
a 37
b 16
c 16
d 16
e 15

Symbol  Frequency      Code
a       37             00
b       16             01
c       16             10
d       16             110
e       15             111
```

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 $O(n \log n)$. We can reduce the running time to **$O(n \log \log n)$** 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