IIT INDORE

Department of Computer Science Engineering

Course Code – CS254

Design And Analysis Of Algorithms

B.Tech. – 4th Semester

Instructor – "*Dr. Kapil Ahuja*"

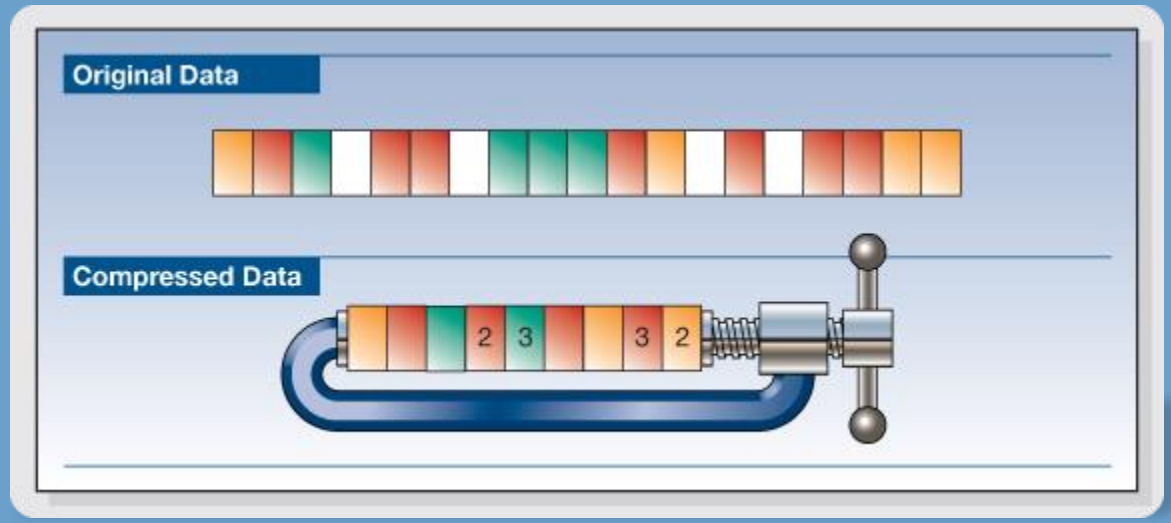
Submitted By :

Sparsh Gupta || Tanmay Singh

170001049 || 170001051

LOSSLESS DATA COMPRESSION ALGORITHM

- *Implement and analyze the Lossless Data Compression Algorithm*



- The project aims at the implementing the existing algorithm and analyze the complexity of the same. Also, we intend to improve the existing algorithms.

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.

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

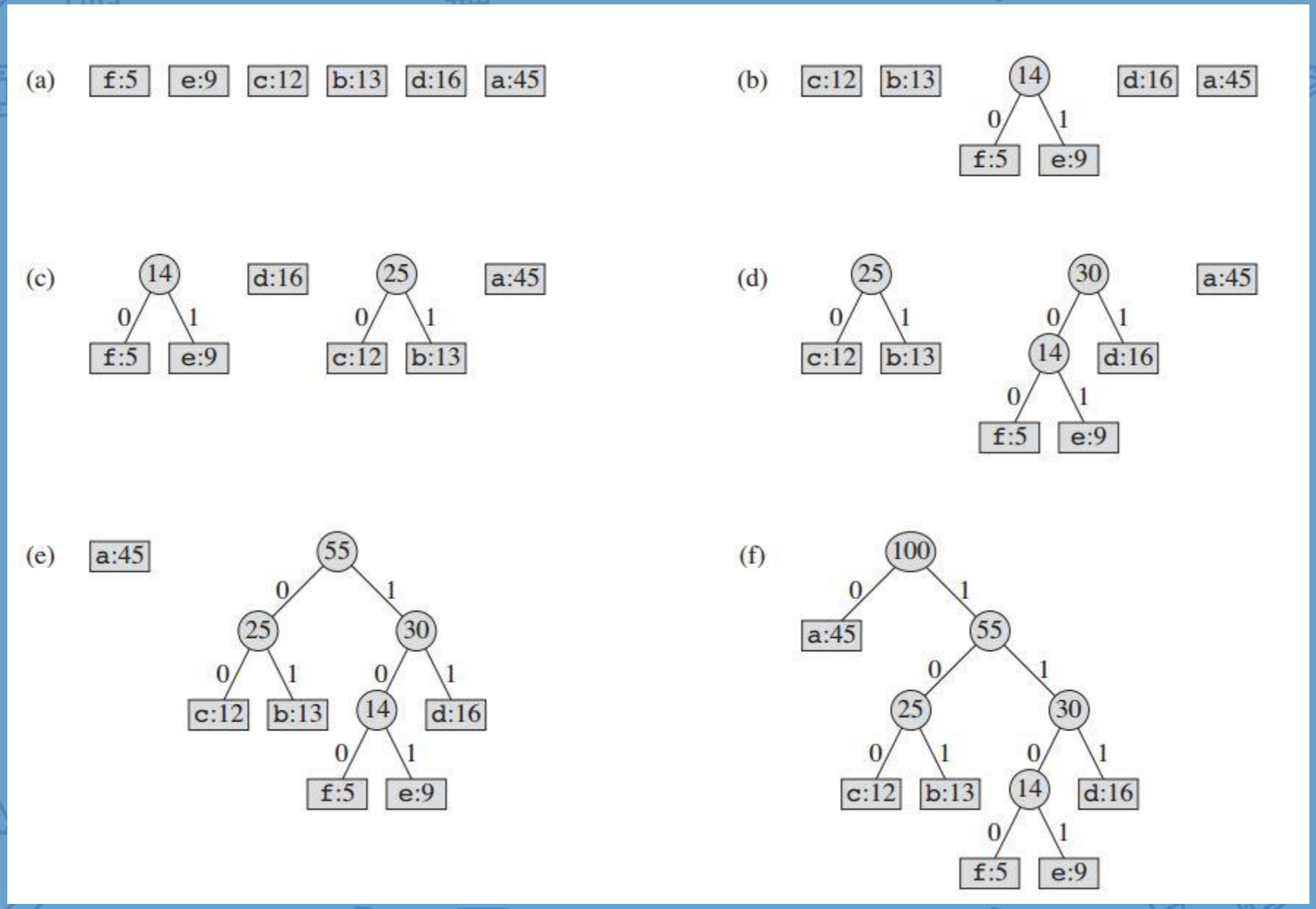
1) Build a Huffman Tree from input characters.

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

--***Steps to build Huffman Tree*** *(GREEDY APPROACH)*

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. *Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)*
2. *Extract two nodes with the minimum frequency from the min heap.*
3. *Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.*
4. *Repeat steps 2 and 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.*


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

--***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

--***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 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.

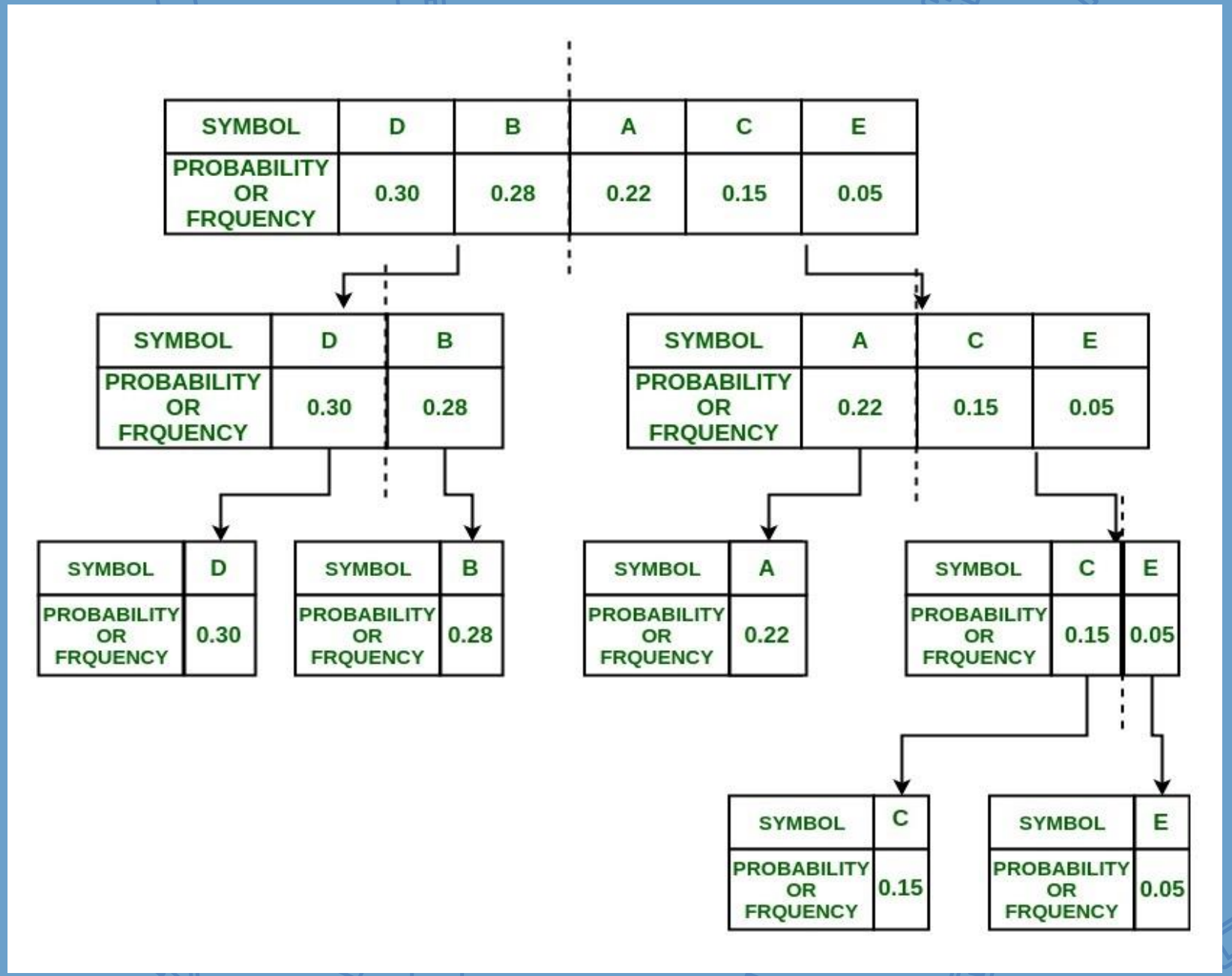
--There are mainly two major parts in Shannon-Fano Coding

1) Build a Binary Tree from input characters.

2) Traverse the tree and assign codes to characters.

--Steps to build Shannon-Fano Binary

1. *Create a list of probabilities or frequency counts for the given set of symbols so that the relative frequency of occurrence of each symbol is known.*
2. *Sort the list of symbols in decreasing order of probability, the most probable ones to the left and least probable to the right.*
3. *Split the list into two parts, with the total probability of both the parts being as close to each other as possible.*
4. *Assign the value 0 to the left part and 1 to the right part.*
5. *Repeat the steps 3 and 4 for each part, until all the symbols are split into individual subgroups.*


SYMBOL
D
B
A
C
E
PROBABILITY OR FRQUENCY
0.30
0.28
0.22
0.15
0.05
SYMBOL
D
B
PROBABILITY OR FRQUENCY
0.30
0.28
SYMBOL
A
C
E
PROBABILITY OR FRQUENCY
0.22
0.15
0.05
SYMBOL
D
PROBABILITY OR FRQUENCY
0.30
SYMBOL
B
PROBABILITY OR FRQUENCY
0.28
SYMBOL
A
PROBABILITY OR FRQUENCY
0.22
SYMBOL
C
E
PROBABILITY OR FRQUENCY
0.15
0.05
SYMBOL
C
PROBABILITY OR FRQUENCY
0.15
SYMBOL
E
PROBABILITY OR FRQUENCY
0.05

--***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
```

```
1.  procedure SF-Split(S)
2.  begin
3.    if (|S|>1) then
4.      begin
5.        divide S to S1 and S2 with about same count of units
6.        add 1 to codes in S1
7.        add 0 to codes in S2
8.        SF-Split(S1)
9.        SF-Split(S2)
10.     end
11. end
```

--***C++ code*** :

Shannon-Fano\shannon fano.cpp

--***Complexity Analysis*** *:*

O(l + n logn)

where *l : length of the input text*

n : distinct number of symbols in the input text

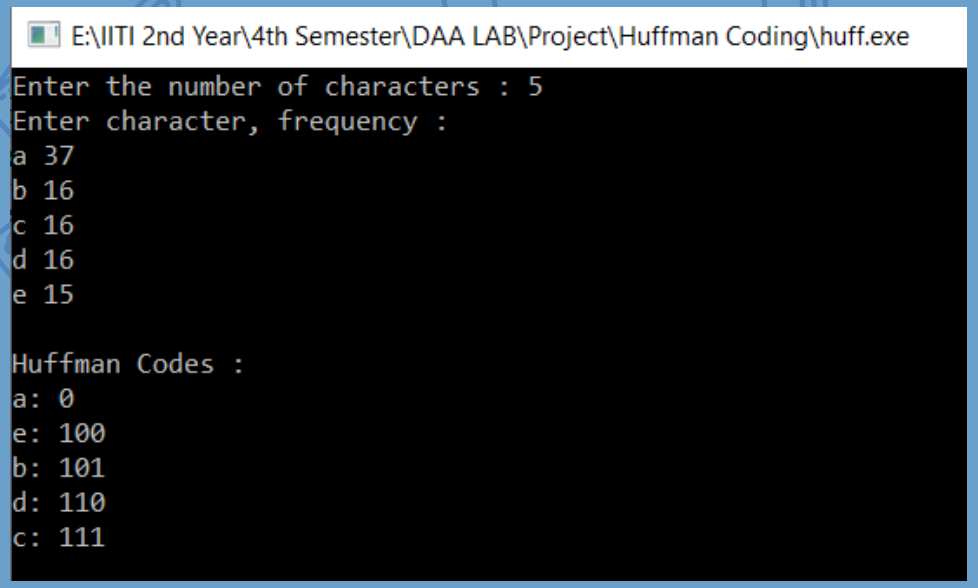
Dividing the problem into three phases. Time complexity of the first phase of computing counts of symbols appearance is linearly dependent on length of the input text. The next phase is ordering symbols, which is implementation dependent. The last phase is generating of the output. This also linearly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

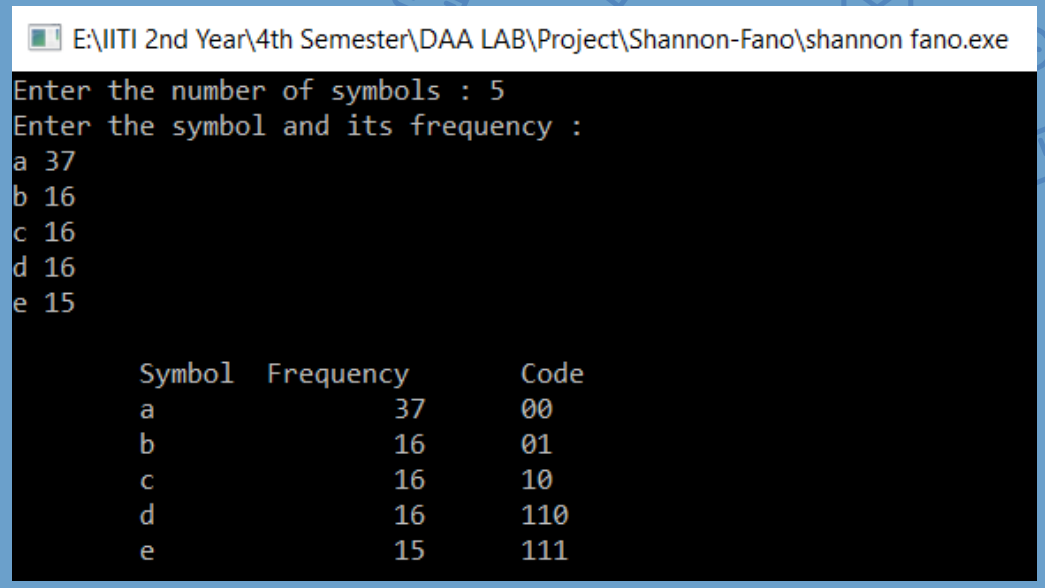
--***Explanation***:

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.

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."

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.

*--**Steps for Arithmetic Encoding***

1. *Take the input as string and find the number of distinct characters with their frequencies and sort & store them in the decreasing order of their frequencies.*
2. *The encoder divides the interval [0,1) into sub-intervals, each representing a fraction of the current interval proportional to the probability of that symbol.*
3. *Whichever symbol corresponds to the actual symbol that is next to be encoded becomes the interval used in the next step.*
4. *When all symbols get encoded the resulting interval unambiguously identifies the sequence of symbols that produced it.*

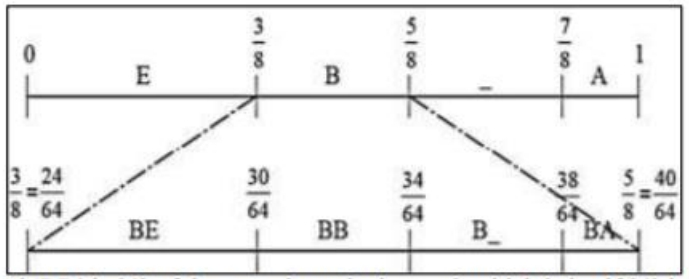
BE_A_BEE

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:

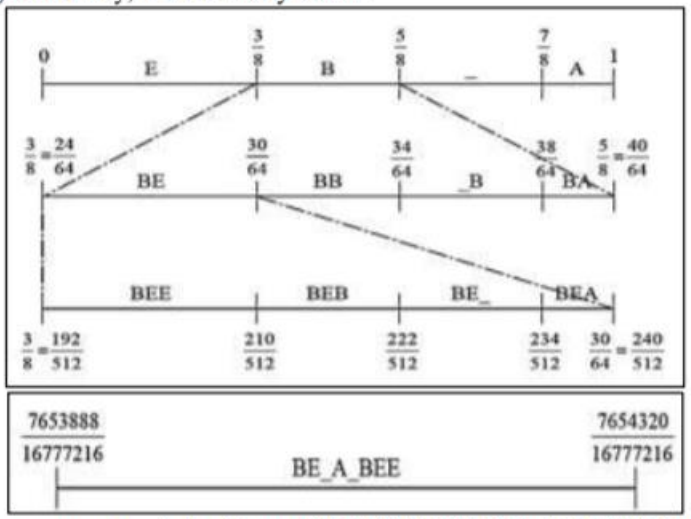
E	B	_	A
3	2	2	1

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]

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
```

--Complexity Analysis of Arithmetic Encoding:

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.

--C++ code :

Arithmetic Coding\arco.cpp

```
int num1=0,num2=1,de1=1,de2=1;
int total = l;
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;
}
```

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.

*--**Steps***

1. *Pick the first character from source string.*
2. *Append the picked character to the destination string.*
3. *Count the number of subsequent occurrences of the picked character and append the count to destination string.*
4. *Pick the next character and repeat steps 2, 3 and 4 if end of string is NOT reached.*

--***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

--***Example*** :

Input : wwwwaaadexxxxxx

Output : w4a3dex6

--***C++ code*** :

Run length\Run length.cpp

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*
- *GeeksForGeeks*
- *Wikipedia*