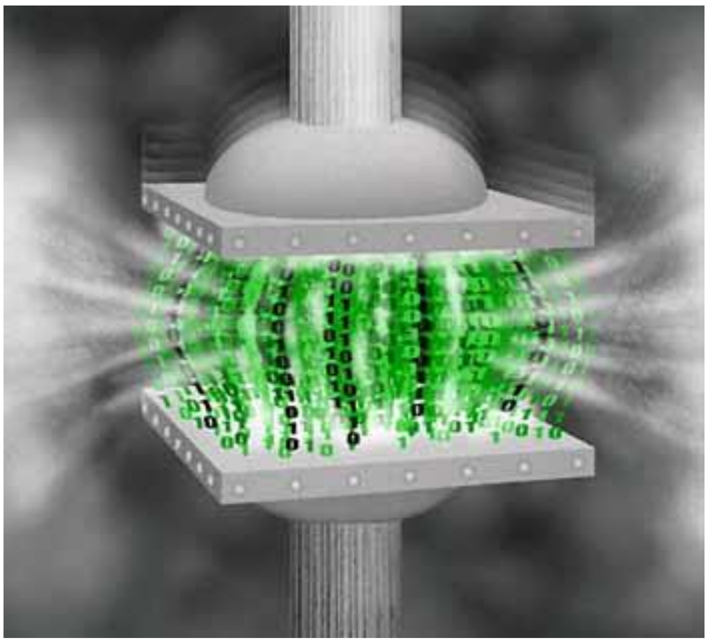


Entropy Encoding and Run length Coding

Data compression

-JIGAR GADA

jigargada90@gmail.com
Graduate Student, EE Dept.
University of Southern California

LAST UPDATED ON 2/12/2014
Image Reference:
https://www.ibm.com/developerworks/community/blogs/ibmnas/entry/v7ku_compression?lang=en

2: Entropy Encoding

2.1 Motivation:

Compression plays a very important role in multimedia and other applications. Without compression, an average 1 hour video size could easily exceed 100's of GB. Apart from video, sending messages and images without compression would increase the Bandwidth and memory requirements. This explains the importance of compression.

Entropy encoding is a lossless compression scheme which makes use of variable lengths for codewords to reduce the coding redundancy. They are also called prefix codes as none of the codewords are the prefix of other codewords. There are two basic type of Entropy encoding methods:

- Shannon Fano code
- Huffman code (with Adaptive Huffman)

2.2 Approach for Shannon Fano and Huffman coding

Following 4 binary files have been given for compression:

- Text.dat
- Image.dat
- Audio.dat
- Binary.dat

2.2.1. Approach for Shannon Fano and Huffman coding

Theory of Shannon Fano code and Huffman code can be understood from the following link:
http://www.binaryessence.com/dct/en000046.htm
http://www.binaryessence.com/dct/en000042.htm

Implementation/ Algorithm (Understanding the code)

The code for entropy encoding can be split into different sections:

1. File Read
   The data files are read 1 byte at a time. C++ I/O (*ifstream, ofstream*) routines have been used for file reading/writing. All the file I/O routines can be found in the folder *FileIO* and well commented. Refer the C++ documentations for details.

2. Statistics of the file
   This section is responsible for calculating and storing the local statistics relevant to the file:
   - Total # of symbols
   - Count of each symbol
   - HashMap which converts the letters to integers. E.g. LetterMap['a'] = 0, LetterMap['h'] = 8 and so on. This makes the task of coding pretty simple.
     The letters are numbered in the way they are read from the file. E.g. if the file content is 'Hello' then LetterMap['H'] = 0, LetterMap['e'] = 1, LetterMap['l'] = 2, LetterMap['o'] = 4.

All functions related to statistics of the file are stored in the folder *FrequencyChart.*

3. Building the Tree

a) Huffman Tree:

Data Structure

Linked List is used to store the Huffman tree. For details of the function, refer to the files in the folder *Huffman.*

Node Class is used which maintains the Linked List.

```
class NodeH {
      int symbol;   // stores the symbol of the node. If intermediate node,
      the symbol = -1
      float prob;   // stores the probability of that node
      bool bit;   // stores the bit value 0 or 1
      NodeH *next;  //pointe to its parent
}
```

Memory Management

For the Huffman tree, there will be a total of N = 2*n-1 nodes where n: Total # of symbols in the file. Therefore Memory is assigned for the N nodes as follows:

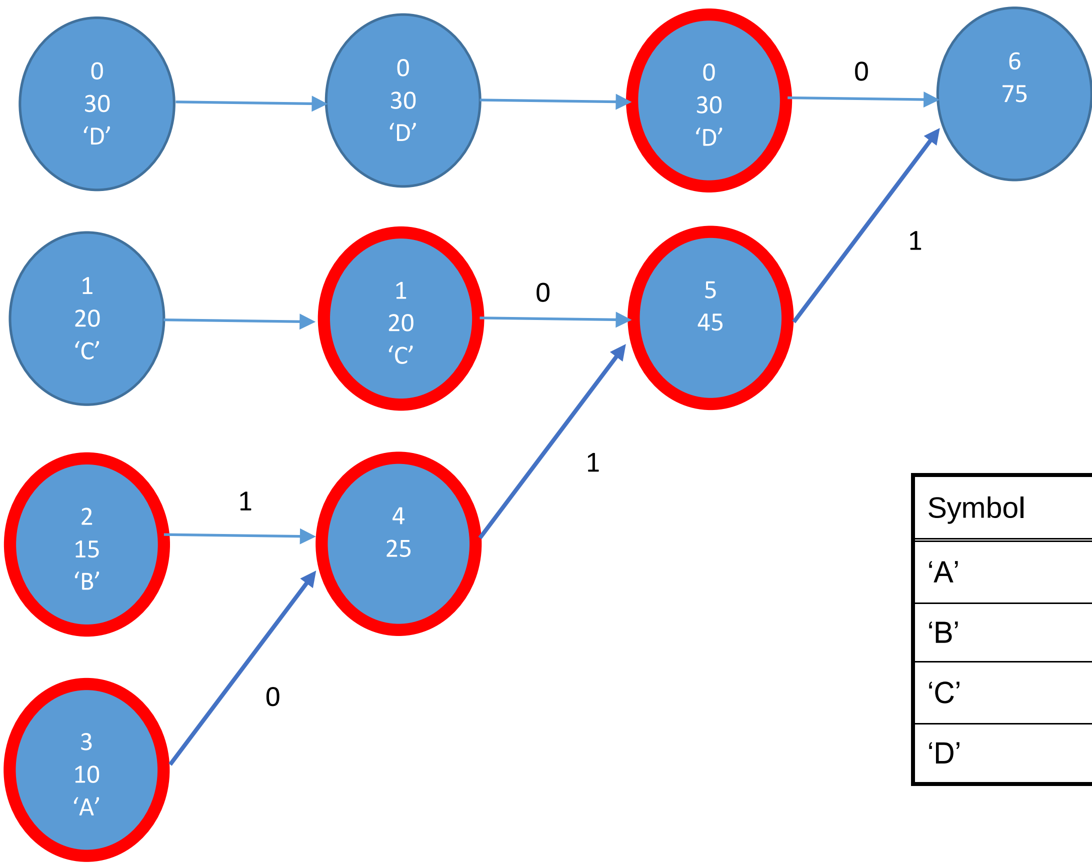
```
int size = 2*no_of_symbols-1;
NodeH **LLArray = new NodeH*[size];
```

Main Logic

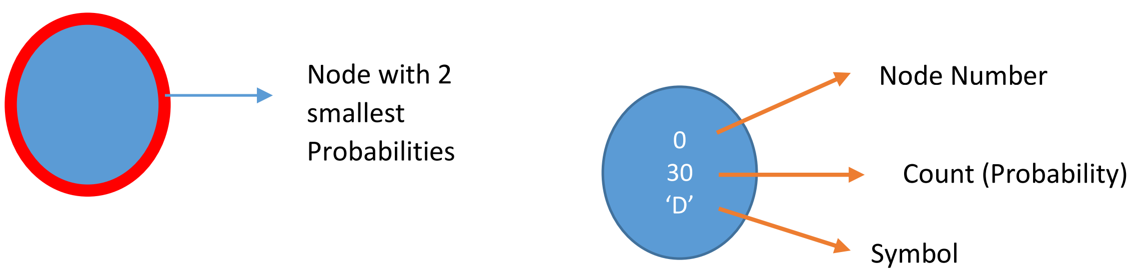
```
Initialize only the n (symbol) nodes with their symbol and corresponding
count value.
k = no_of_symbols
while(k > 1) //loop will run for k-1 iterations
      - Find 2 nodes A & B with the smallest probabilities.
      - Assign smallest node bit as 0 and second smallest bit as 1.
      - Create a new Node X.
      - Initialize the probability of X with sum of probabilities of A & B.
      - Assign -1 to the probability of A & B so that these nodes are not
      considered again for computing the smallest probabilities.
      - Make X as the parent of A & B.
End while
```

To get the codewords for the symbols, start from the symbol node and trace it till the last node. Remember the codewords are stored in the *__reverse__* fashion as this is simple and will be taken care of when writing the codes in the file.

The following demonstration of Huffman code will explain the working of the algorithm.



Symbol	Code Word
‘A’	110
‘B’	111
‘C’	10
‘D’	0



b) Shannon Fano code:

Data Structure

Tree data structure is used to maintain the Shannon Fano Tree and is comparatively easy to implement as compared to Huffman tree. For details of the function, refer to the files in the folder *ShannonFano.*

```
class Node {
      Node *parent; //pointer to the parent
      int start;  //start value in array
      int end; // end value in array
      bool bit; //either 0 or 1 depending on left or right child
}
```

Memory Management

For this, memory is allocated only to the n number of nodes where n: # of symbols unlike previous case where 2*n-1 nodes are allocated memory.

```
Node **list = new Node*[no_of_symbols];
```

Main Logic

Sort the symbols in the descending order of probability. REMEMBER to keep a track of the index of symbols when sorting it. This will be required while decoding the symbol.
Initialize the root node as follows:

```
Node *rootNode = new Node;
rootNode->setStart(0); //start value as beginning
rootNode->setEnd(no_of_symbols-1); //end value as the end
```

Call the recursive build function:

```
void buildShannonTree(Node *node){

if(start value != end value){
      -  Get the split point in variable *end* (Point where probability is
         greater than 0.5).
      -  Create the left and right child.
      -  Assign bit 0 to left child and bit 1 to right child.
      -  Set parent of left and right child as current node.


      //recurse the procedure for left and right nodes
      buildShannonTree(leftNode);
      buildShannonTree(rightNode);
}
else{
      //maintain a list of all leaf nodes.
      list[count++] = node;
```

```
}
}
```

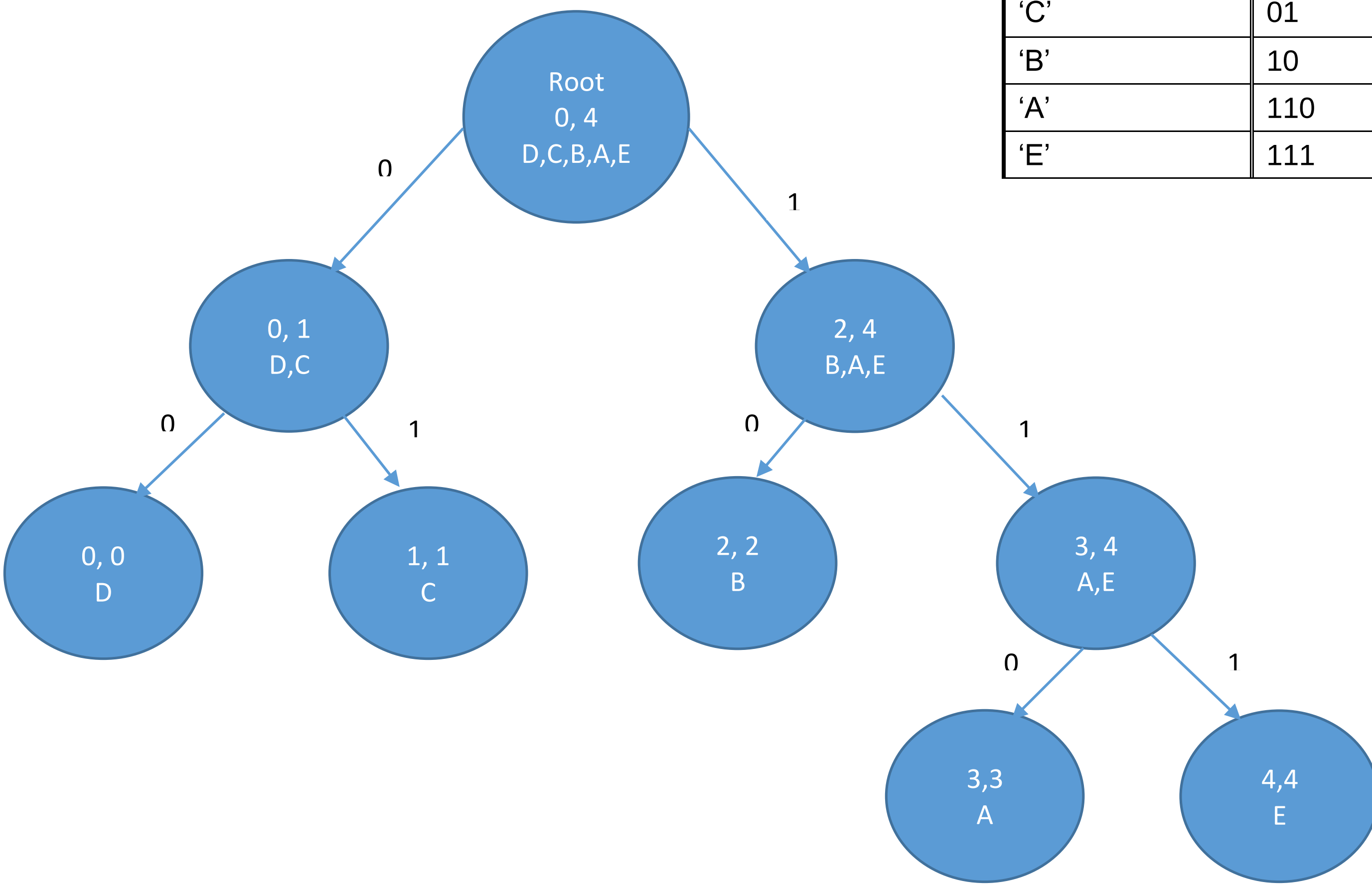
To get the codewords for the symbols, start from the symbol node and trace it till the last node. Remember the codewords are stored in the *reverse* fashion as this is simple (Going from child to parent is simple but not vice-versa) and will be taken care of when writing the codes in the file.

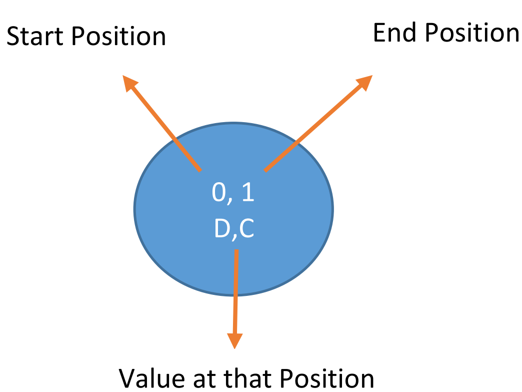
The following demonstration of Shannon Fano code will explain the working of the algorithm.

Consider the following nodes sorted in the descending order:

Symbol	'D'	'C'	'B'	'A'	'E'
Position	0	1	2	3	4
Probability	30	20	15	10	5

Symbol	Codewords
'D'	00
'C'	01
'B'	10
'A'	110
'E'	111





4. File Write

Once the tree is built we can traverse through the tree and a block code is generated for each of the symbols as shown in the example 2.2.3 for Huffman and Shannon Fano. The file is then read symbol by symbol and corresponding codewords for each of the symbol are written in the *encodedData.dat* file.

Header

The initial section of the output file contains the header which stores the information of the count for each symbol so that the corresponding tree can be generated by the decoder. The header information is stored as follows:

Original File size	# of symbols (1byte)	Symbol (1 byte)	Count (2 bytes)	Symbol (1 byte)	Count (2 bytes)	…

NOTE: *2 bytes* have been allocated for the count so that the maximum count can be 65535. Original File size will help us

If we consider the example shown in the section 2.2.3 for Huffman code, we get the following header.

fileSize	4	A	10	B	15	C	20	D	30

Codewords

The information of the codewords for each symbol are stored using 2 variables.

1. Long unsigned code (64 bits)
2. Unsigned char size (0 – 255)

For e.g. if the codeword for A is 0111 then the last 4 bits of *code* will have 0111 and *size* variable will store the size as 4.

Let us consider a simple example to get a feel of how the codewords are stored.
Consider a file containing the following content “Hello” then by doing Huffman coding, we get the codewords as follows:

h	10
e	111
l	0
o	110

Codewords are stored byte by byte. For Huffman/Shannon Fano coding we need to keep a buffer and as the size of the buffer becomes 8, we write it to the file.
If the last buffer contains less than 8 elements, ***we simply append 0’s*** and write it to the file.
The **original file** size in our header will help us to scrap the extra 0’s at the end of the file.

The string “hello” will be written to the file as:

157	1
10011101	000000001

h	10
e	111
ll	00
o	110

5. Decoding
   Following steps will be taken by a Shannon Fano/ Huffman decoder:
   i. Read the file Size and store it in N.
   ii. Read the symbols and the count and build the Huffman/ Shannon Fano tree.
   iii. Read the bytes and start decoding them from LSB as shown in the above table.
   iv. Repeat step iii until the N symbols read.

2.2.2. Approach for Adaptive Huffman

Adaptive Huffman is based on Huffman coding used for real time applications. Huffman coding requires a prior information about the statistics of the file which is not the case with Adaptive Huffman. Apart from that, Adaptive Huffman technique adapts to the local statistics of the file thereby compressing the file efficiently.

Details of the Adaptive Huffman technique can be understood from the following link:

http://www.binaryessence.com/dct/en000097.htm

1. File Read
   The data files are read 1 byte at a time. C++ I/O (*ifstream, ofstream*) routines have been used for file reading/writing. All the file I/O routines can be found in the folder *FileIO*.

2. Algorithm
   Vitter Algorithm has been used for building the tree. Demonstration of the Vitter algorithm has been described in this link.
   http://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html

   Data Structure
   Tree data structure has been used for building the Huffman tree with following entities:

```
class Tree {
      int value;
      int order;
      int weight;
      Tree *parent;
      Tree *leftChild;
```

```
    Tree *rightChild;
}
```

Memory Allocation

Memory is allocated dynamically as and when new symbols are encountered. Two vectors are used to keep a track of the symbols and the leaf nodes.

```
std::vector<Tree*> treeElements; //Tree elements are stored in descending
//order of *ORDER*
std::vector<Tree*> leafNodes; //This makes it simple to extract the symbol
```

Main Logic

There are two important tree manipulation function that form the crux of the logic.

```
/**
 * This function gives birth to two new nodes
* when a new symbol occurs.
* the left child --> NYT node and
* the right right --> new Symbol
* @param:
* 1. int value - value of the symbol
 */
void giveBirth(int value)

/**
 * This function checks the sibling property if the symbol
* is already present in the tree or the NYT's weight.
* Note the vectors of tree elements are arranged
* in descending order of 'Order'.
* If Any of the vector having a higher weight has a lower order
 * than the vector with lower weight, then those two branches
* are swapped.
* @param:
* 1. int pos - position in the vector of tree elements
 */
void checkNodeForSwapping(int pos)

Pseudo code for encoding.

Initialize the root node with order of 512
While(symbol occurs)
    Check if symbol present in the Vector of leaf nodes.
    If not present
        Encode the NYT node followed by 9 bit symbol.
        Give birth to the symbol
        Check Node for swapping
    Else
        Encode the symbol Value
```

```
            Increment the count for that symbol and check the node for
            swapping.
      End If
End While
Write the EOF (write any character) at the end with MSB as 1 in the 9th bit.
```

3. File Writing

   Consider the example from the following link to encode the string “aardv”
   http://www.cs.duke.edu/csed/curious/compression/eg1.html

Color	Tree code
Color	ASCII symbol code
Color	EOF

   “a”
   Initially an 8 bit code for ‘a’ with EOF bit as 0 will be sent:
   0 01100001

   “a”
   Code of a in the tree sent
   1

   “r”
   Code for NYT with 8 bit code for ‘r’ with EOF sent
   0 01110010 0

   “d”
   Code for NYT with 8 bit code for ‘d’ with EOF sent
   0 011001000 00

   “v”
   Code for NYT with 8 bit code for ‘v’ with EOF sent
   0 01110110 000

   “EOF”
   Code for NYT with any codeword with EOF = 1 sent
   1 01100001 0011

4. Decoding

   Decoding procedure explained in the Vitter algorithm link will work.

2.3. Output

Run the file ***runThisFile.sh*** and select the appropriate options:

1. Enter the file name

```
jigar@jigar-VPCEA16FG: ~/Desktop/EE669_Hw1/EntropyCoding
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/Ahuffman/encodeSymbol.d
" -MT"src/Ahuffman/encodeSymbol.d" -o "src/Ahuffman/encodeSymbol.o" "../src/Ahuf
fman/encodeSymbol.cpp"
Finished building: ../src/Ahuffman/encodeSymbol.cpp

Building file: ../src/main.cpp
Invoking: Cross G++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/main.d" -MT"src/main.d"
 -o "src/main.o" "../src/main.cpp"
Finished building: ../src/main.cpp

Building target: EntropyCoding
Invoking: Cross G++ Linker
g++  -o "EntropyCoding"  ./src/ShannonFano/Node.o ./src/ShannonFano/SFTable.o ./
src/ShannonFano/shannonFano.o  ./src/Huffman/Node/NodeH.o  ./src/Huffman/Huffman
Table.o ./src/Huffman/huffman.o  ./src/FrequencyChart/LetterFrequency.o  ./src/F
ileIO/fileIO.o  ./src/CodeTable/codetable.o  ./src/CodeAnalyse/analyseCode.o  ./
src/Ahuffman/Tree.o ./src/Ahuffman/TreeManipulations.o ./src/Ahuffman/encodeSymb
ol.o  ./src/main.o
Finished building target: EntropyCoding

-----------------------------
Enter the *full* file Path with name:
/home/jigar/Desktop/EE669_Hw1/EntropyCoding/resources/text.dat
```

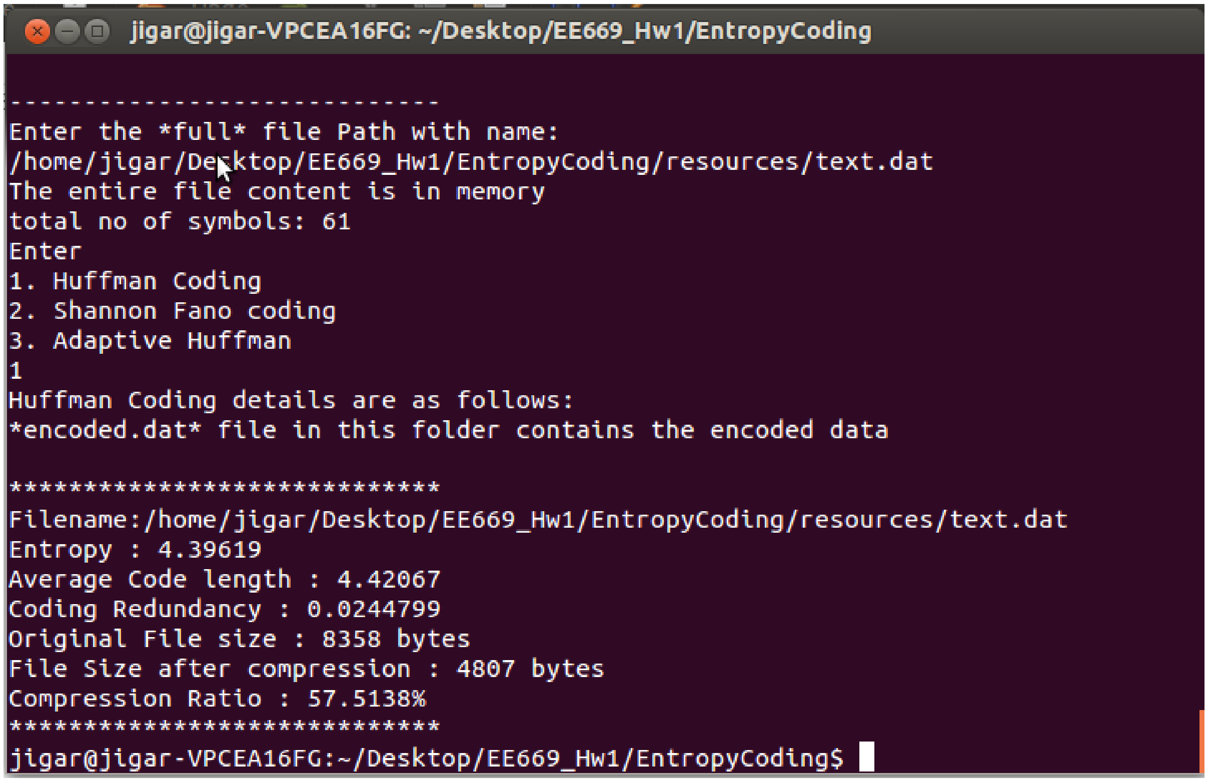
2. Enter the appropriate option

```
jigar@jigar-VPCEA16FG: ~/Desktop/EE669_Hw1/EntropyCoding
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/main.d" -MT"src/main.d"
 -o "src/main.o" "../src/main.cpp"
Finished building: ../src/main.cpp

Building target: EntropyCoding
Invoking: Cross G++ Linker
g++  -o "EntropyCoding"  ./src/ShannonFano/Node.o ./src/ShannonFano/SFTable.o ./
src/ShannonFano/shannonFano.o  ./src/Huffman/Node/NodeH.o  ./src/Huffman/Huffman
Table.o ./src/Huffman/huffman.o  ./src/FrequencyChart/LetterFrequency.o  ./src/F
ileIO/fileIO.o  ./src/CodeTable/codetable.o  ./src/CodeAnalyse/analyseCode.o  ./
src/Ahuffman/Tree.o ./src/Ahuffman/TreeManipulations.o ./src/Ahuffman/encodeSymb
ol.o  ./src/main.o
Finished building target: EntropyCoding

-----------------------------
Enter the *full* file Path with name:
/home/jigar/Desktop/EE669_Hw1/EntropyCoding/resources/text.dat
The entire file content is in memory
total no of symbols: 61
Enter
1. Huffman Coding
2. Shannon Fano coding
3. Adaptive Huffman
```

You get the following details:



```
jigar@jigar-VPCEA16FG: ~/Desktop/EE669_Hw1/EntropyCoding

-----------------------------
Enter the *full* file Path with name:
/home/jigar/Desktop/EE669_Hw1/EntropyCoding/resources/text.dat
The entire file content is in memory
total no of symbols: 61
Enter
1. Huffman Coding
2. Shannon Fano coding
3. Adaptive Huffman
1
Huffman Coding details are as follows:
*encoded.dat* file in this folder contains the encoded data

*****************************
Filename:/home/jigar/Desktop/EE669_Hw1/EntropyCoding/resources/text.dat
Entropy : 4.39619
Average Code length : 4.42067
Coding Redundancy : 0.0244799
Original File size : 8358 bytes
File Size after compression : 4807 bytes
Compression Ratio : 57.5138%
*****************************
jigar@jigar-VPCEA16FG:~/Desktop/EE669_Hw1/EntropyCoding$
```

Compression Ratio = File Size After Compression/ Original File size *100

2.4. Results

				Huffman			Shannon-Fano		
	Original File Size (bytes)	# of Symbols present	Entropy	Average Length	o/p File Size (bytes)	C.R (%)	Average Length Code	o/p File Size (bytes)	C.R (%)
Text.dat	8358	61	4.40	4.42	4807	57.59	5.63	4872	58.29
Image.dat	65536	230	7.59	7.62	63128	96.32	7.86	63397	96.73
Audio.dat	65536	249	6.46	6.49	53918	82.27	7.53	54447	83.07
Binary.dat	65536	2	0.18	1.00	8204	12.52	1.00	8204	12.32

	Original File Size (bytes)	Adaptive Huffman	
		o/p File Size	C.R
Text.dat	8358	4706	56.35
Image.dat	65536	62827	95.86
Audio.dat	65536	53559	81.72
Binary.dat	65536	8424	12.85

2.5. Discussion

Shannon Fano	Huffman code
• Uses a top down approach	• Uses a bottom down approach
• High average code word length	• Low average code word length
• High coding redundancy	• Low coding redundancy
• Easy to implement	• Bit difficult to implement
• Data Structures: Tree, Linked List	• Data Structures: Priority Queues, Linked List
• Not used in any application	• Used for entropy encoding in multimedia codecs like JPEG, MP3.

<table>
<tr><th>Huffman code</th><th>Adaptive Huffman code</th></tr>
<tr><td>• Not Real time, needs statistical information of the file prior to encoding</td><td>• Real time, Adapts to the local statistics of the file.</td></tr>
<tr><td colspan="2">• For the above files, Compression ratio is better for Adaptive Huffman compared to Huffman</td></tr>
<tr><td>• Data structure: Priority Queues, Linked List</td><td>• Data structure: Trees and Vectors</td></tr>
</table>

2.6. Assumptions

- Max Count value for a particular symbol: 65535
- Max file size: 4.2 GB
- Max code length: 64 bits

3: Run Length Coding

3.1. Motivation

Run length coding is a suitable option when multiple symbols are repeated. Even if the symbols are not repeated, some preprocessing can be done so that symbols get repeated and run length coding used to achieve compression. For lossy compressions where quantization is used, many of the bytes below the threshold are boiled to 0. RLE can then be efficiently used to achieve high compression. It is very simple to implement and computationally efficient.

Run Length Coding can be modified to get better compression also called as modified run length coding.

3.2. Approach

We will analyze both the Run length and modified run length encoding. Also some preprocessing will be done like BWT transform before applying modified run length coding.

a) Run Length Coding

Details of RLC can be found on the following link:
http://www.binaryessence.com/dct/en000050.htm

File Read
File read operations are same as discussed in Section 2.3.

Main Logic
Pseudo code

Encoding

```
FOR all characters in the file starting from second character
      IF current symbol not equal to previous symbol or count >= 255
            Write the count
            Write the *previous* symbol to the encoded file
            Make count = 0
      ELSE
            Increment count by 1
      END IF
END FOR
Write last symbol with the count.
```

File Write
Counts and symbols are written byte by byte and alternately. Thus the maximum count value is 255. If the count exceeds 255, we simply consider it as a new symbol.

Decoding

Counts are stored first and then the symbol

```
N = 0
```

```
WHILE N < # of bytes in the file
      FOR count number of time
            Write the next symbol Value
      END FOR
      Increment N by 2.
END WHILE
```

Input/Decoded file	aaabcddd
Encoded File	3a1b1c3d

Run Length encoding will not always compress the input file. If the symbols in the file are random then there will be extra overhead of the count and instead of compressing the file, size will exceed the original file size.

b) Modified Run Length Encoding

Main Logic

Pseudo Code

Encoding

```
FOR all symbols in the file
      IF(current symbol not equal to previous symbol or count >= 126)
            IF count > 0
                  Write(128+count+1)
                  Write(previous symbol)
                  Make count = 0
            ELSE
                  IF MSB of previous symbol = 1
                        Write(128+1)
                        Write(previous symbol)
                  ELSE
                        Write just the symbol
                  END IF
            END IF
      END IF
      Increment count by 1
END FOR
```

Decoding

```
WHILE (n < size of file)
      IF MSB is 1
            For (count – 128 number of time)
                  Write next symbol
            END FOR
            Increment n by 2
      ELSE
            Write current symbol once
      END IF
END WHILE
```

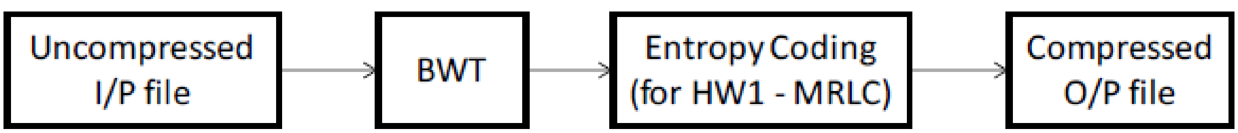
Input/Decoded file	aaabcddd
Encoded File	131abc131d

Modified Run length encoding will also not always compress the input file. For symbols whose value is greater than 128, we need to encode 2 symbols for their occurrence. However Modified RLE has a better Compression ratio than RLE.

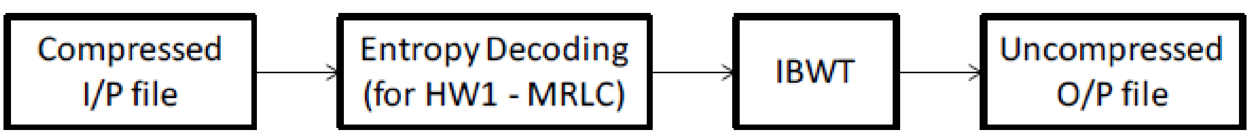
c) BWT

Burrows-Wheeler Transform is a good pre-processing step before using modified run length coding. It sorts the data in some manner so that symbols occur in repetitive fashion which is most suitable for run length coding.
Details of BWT can be found on the following link:
https://sites.google.com/site/compgt/bwt

Coding Process



Decoding Process



This block diagram explains the preprocessing by BWT.

<u>Main Logic</u>

Pseudo code
Encoding

```
FOR all symbols in the file
      Consider block size of N
      Store the data in an array of size N
      Create an NxN matrix containing circular shifts of original array
      Sort the matrix along the rows
      Get the index of the original string in the sorted matrix
```

```
        Send the last column of the sorted matrix along with the index to the
file
END FOR

DO the Modified RLE
```

Decoding

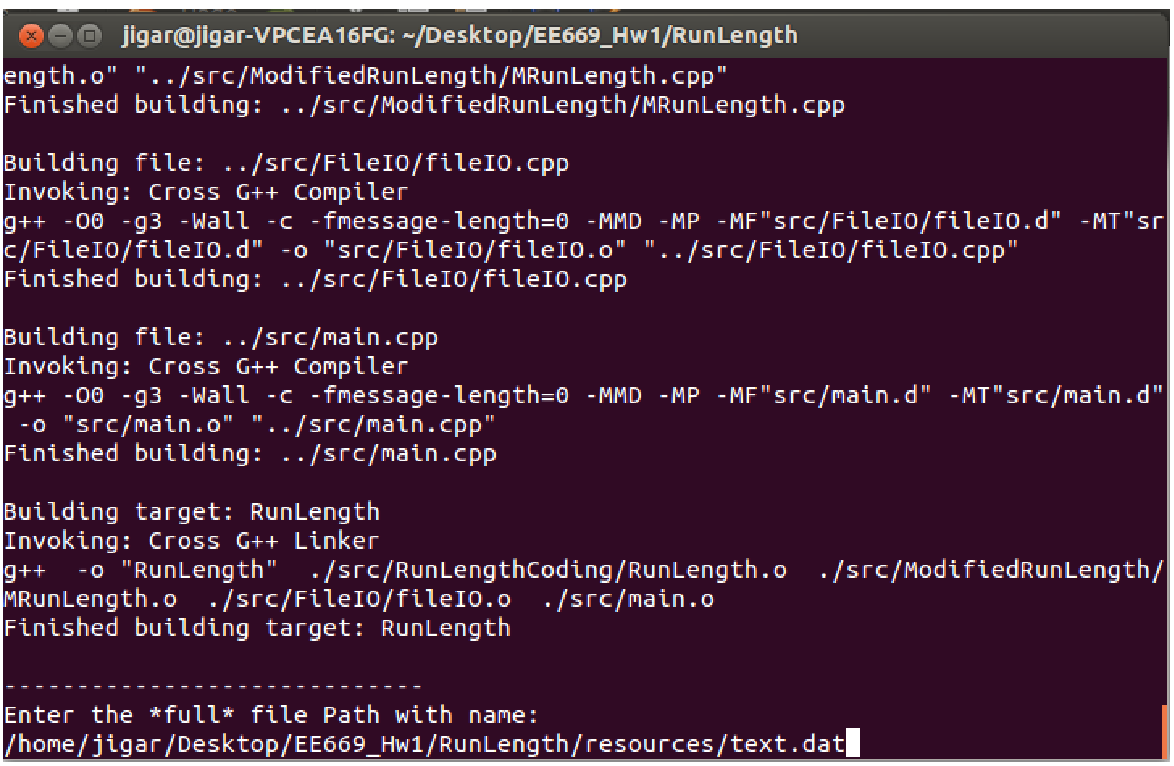
```
Do the Modified RLE decoding

FOR each block size of size N
        Sort the array to get the first column of the matrix
        From the index, go to the corresponding position in the sorted array
and fetch symbol X
        Find the position of symbol X in the received array, fetch its
        position and store it in index.
        Repeat steps 2 and 3 until all N characters decoded
END FOR
```

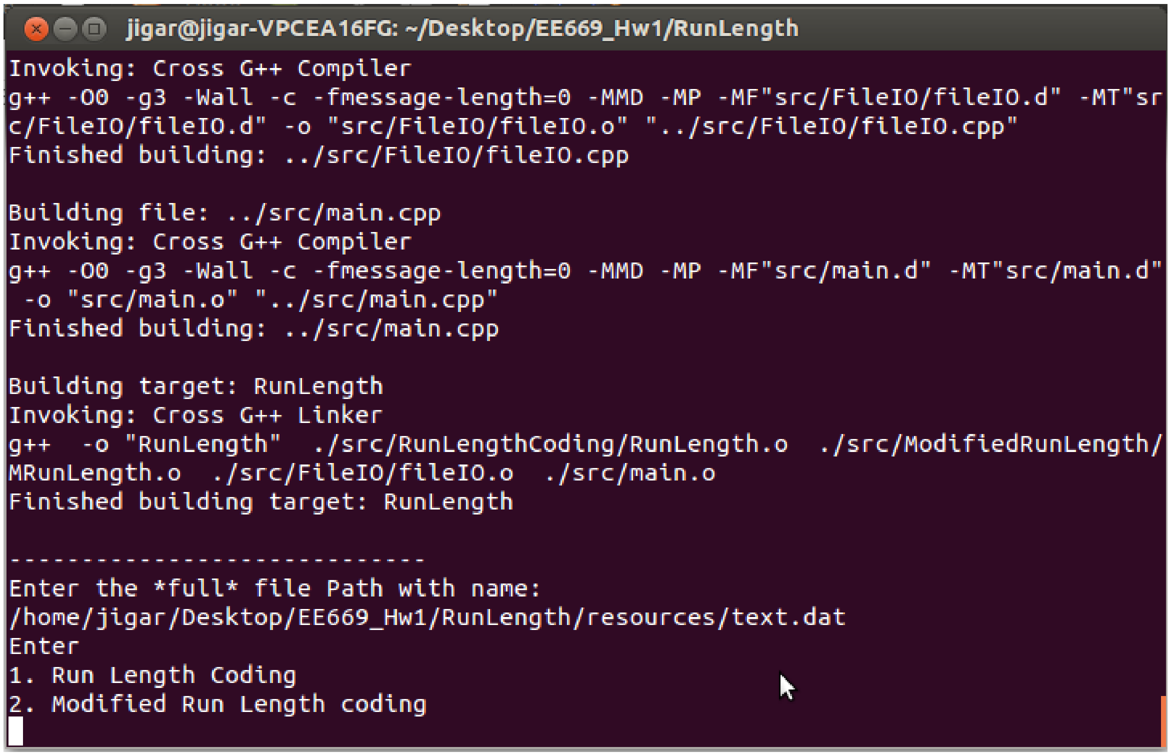
3.3. Output

Run the file **runThisFile.sh** in the runLength folder.

1. Enter the file name



2. Select the appropriate option



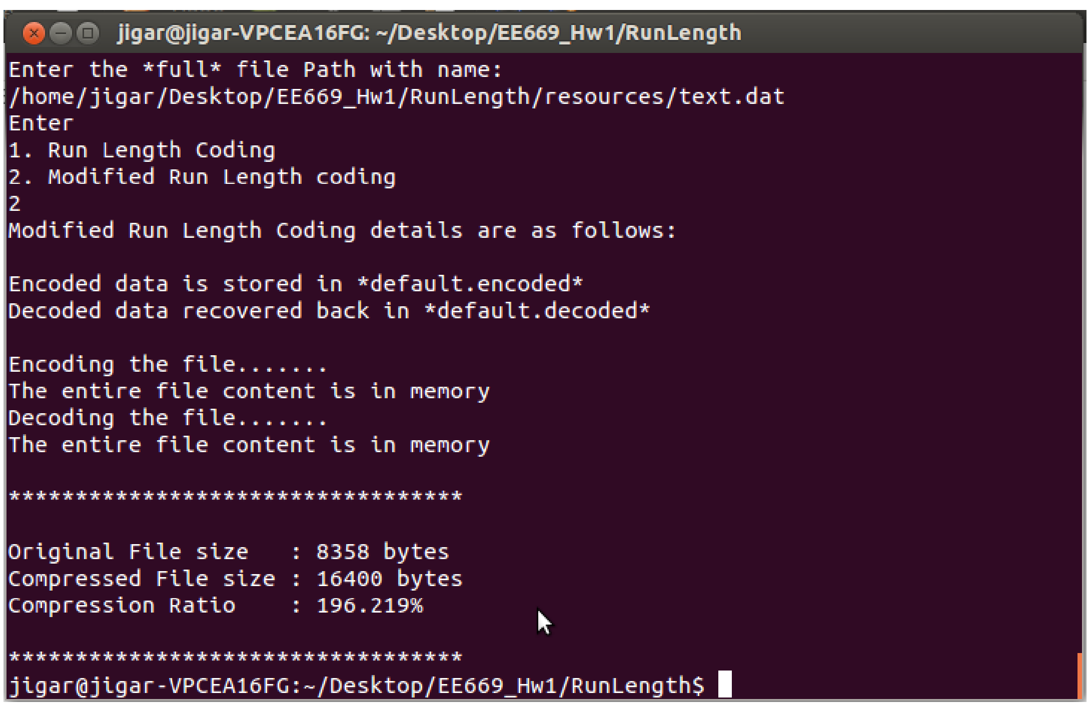
```
jigar@jigar-VPCEA16FG: ~/Desktop/EE669_Hw1/RunLength
Invoking: Cross G++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/FileIO/fileIO.d" -MT"sr
c/FileIO/fileIO.d" -o "src/FileIO/fileIO.o" "../src/FileIO/fileIO.cpp"
Finished building: ../src/FileIO/fileIO.cpp

Building file: ../src/main.cpp
Invoking: Cross G++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/main.d" -MT"src/main.d"
 -o "src/main.o" "../src/main.cpp"
Finished building: ../src/main.cpp

Building target: RunLength
Invoking: Cross G++ Linker
g++  -o "RunLength"  ./src/RunLengthCoding/RunLength.o  ./src/ModifiedRunLength/
MRunLength.o  ./src/FileIO/fileIO.o  ./src/main.o
Finished building target: RunLength

-----------------------------
Enter the *full* file Path with name:
/home/jigar/Desktop/EE669_Hw1/RunLength/resources/text.dat
Enter
1. Run Length Coding
2. Modified Run Length coding
```

3. Results displayed as follows:



```
jigar@jigar-VPCEA16FG: ~/Desktop/EE669_Hw1/RunLength
Enter the *full* file Path with name:
/home/jigar/Desktop/EE669_Hw1/RunLength/resources/text.dat
Enter
1. Run Length Coding
2. Modified Run Length coding
2
Modified Run Length Coding details are as follows:

Encoded data is stored in *default.encoded*
Decoded data recovered back in *default.decoded*

Encoding the file.......
The entire file content is in memory
Decoding the file.......
The entire file content is in memory

**********************************

Original File size   : 8358 bytes
Compressed File size : 16400 bytes
Compression Ratio    : 196.219%

**********************************
jigar@jigar-VPCEA16FG:~/Desktop/EE669_Hw1/RunLength$
```

3.3. Results

		Run Length		Modified Run Length	
	Original File Size (bytes)	**o/p File Size (bytes)**	**C.R (%)**	**o/p File Size (bytes)**	**C.R (%)**
Binary.dat	65536	4780	7.29	4438	6.77
Audio.dat	65536	108534	165.61	83205	130.01
Image.dat	65536	124320	189.69	82766	126.29
Text.dat	8358	16400	196.29	8343	99.82

	BWT with Block Size=20		BWT with Block Size=20	
	o/p File Size (bytes)	**C.R (%)**	**o/p File Size (bytes)**	**C.R (%)**
Binary.dat	12762	19.47	9759	14.89
Audio.dat	93080	142.02	91897	140.22
Image.dat	86749	132.37	85622	130.65
Text.dat	8720	104.30	8550	102.97

3.4. Discussion

- For the above files we can clearly see that Modified RLE performs better than RLE.
- BTW preprocessing didn’t improve the compression ratio as we are processing it block by block. By doing this, we are breaking the patterns in the original file.
- If we consider a bigger block size then RLE can get us better compression ratios. However it comes with the cost of higher computations.