-JIGAR GADA

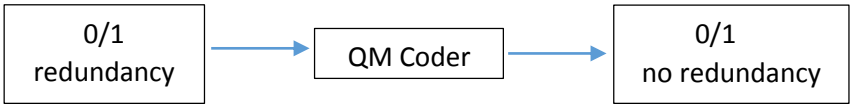
USC ID: 8979-1025-58
jgada@usc.edu
Graduate Student, EE Dept.
University of Southern California

QM Coder

Problem 1. QM Coder

1.1 Motivation

QM coders are arithmetic coders, an entropy coding technique used with binary symbols, 0 and 1. Since all the data finally boils down to 0's and 1's,and most compression techniques use entropy encoding as a final step, this technique is quite useful as it achieves good compression for data in particular format. It removes the redundancy in the data to make the data identically independently distributed (IID).

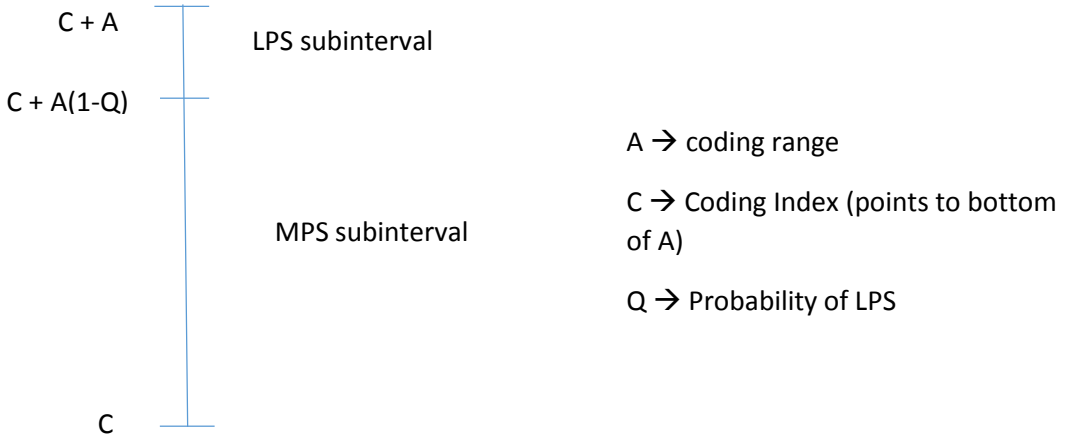


Arithmetic coder in general has less coding redundancy compared to Huffman encoding at the cost of complexity (complexity is hardly an issue now). So it's a more natural choice for entropy encoding compared to Huffman. However due to patent issues for the arithmetic coder, Huffman coding was a preferred option for most of the compression techniques. Good news is that the patent have expired and the QM coders have been used in JPEG 2000, JBIG, H.264/AVC and other data compression techniques.

1.2 Approach

There are just two symbols for a binary arithmetic coder. They are classified as *More Probable Symbol* (MPS) and *Less Probable Symbol* (LPS). Basic idea is the same as arithmetic coder.

Basic QM Coder:



Receiving MPS	**Receiving LPS**
C' := C	C' := C + A(1-Q)
A' := A*(1-Q)	A' := A*Q

More details on QM coder: http://www.cs.ucf.edu/courses/cap5015/QM_coder.pdf

1.2.1 Problems in QM coder and its solution

1. Potentially unbounded resolution of A

   If we use the process described above as it is, we need the resolution of A to be very high. This involves dealing with floating point numbers which can slow down the process.

   Solution:

   We deal only with integers with total range of A as 0x0000 to 0x10000 and if the value of A becomes less than 0x8000, we do *renormalization* as follows:

   C' := 2C

   A' := 2A

   Re-estimate the value of Q from the look up table given. A glance of look up table is as follows:

Appendix A: The state transition table for the QM coder

State	Qe (Hex)	Qe (Dec)	Increase state by	Decrease state by
0	59EB	0.49582	1	S
1	5522	0.46944	1	1
2	504F	0.44283	1	1
3	4B85	0.41643	1	1
4	4639	0.38722	1	1
5	415E	0.36044	1	1
6	3C3D	0.33216	1	1
7	375E	0.30530	1	1
8	32B4	0.27958	1	2
9	2E17	0.25415	1	1
10	299A	0.22940	1	2
11	2516	0.20450	1	1
12	1EDF	0.17023	1	1

2. Multiplication

   Multiplication of A*Q and A*(1-Q) can slow down the process.

   Solution:

   Eliminate multiplication by approximating A as:

   $$\frac{3}{4} \leq A \leq \frac{3}{2}$$

   $$A \approx 1$$

   $$A * Q \approx Q$$

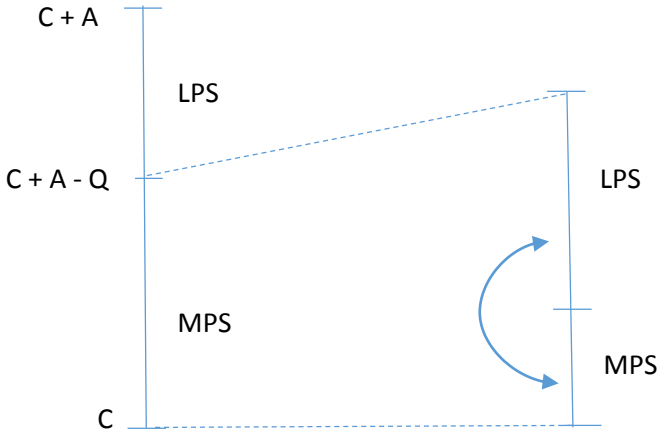
New Values of A' are:
A - A*Q → A – Q
A*Q → Q

   This approximation eliminates the multiplication and speeds up the performance.

3. A may become less than Q

   At this stage if we receive an MPS as the next bit, then the value of A':= A – Q will become negative and coding range cannot be negative. This problem occurs because of the approximation of A.

Solution:
In such cases we do a conditional exchange, i.e. invert the symbol order. We exchange the interval for that particular iteration (MPS acts like LPS and vice-versa).
NOTE: We do not do swapping of MPS and LPS.



1.2.2. Pseudo code for QM coder encoding

```
if encoder receives MPS
{
      A=A-Q_e;
if A<0x8000
{
if A<Q_e {
 if C+A > 0xFFFF {carry = true;}
      C+=A; A=Q_e;
}
Q_e changes its state and then the value according to
Appendix A column 4;
renormalize();

 }
}
else if encoder receives LPS
{
A=A-Q_e;
if A>=Q_e {
 if C+A > 0xFFFF {carry = true;}
      C+=A; A=Q_e;
}
Q_e changes its state and then the value according to Appendix A
column 5;
renormalize();
}
```

```
renormalize()
{
 if carry == true {encoder outputs 1; carry = false;}
while (A < 0x8000)
{
      A<<=1;
      encoder outputs MSB of C;
      C<<=1;
}

}
```

NOTE: This is not the actual way in which the QM encoding is done. The carry bit and the MSB of C bit are handled properly to reduce the number of bits sent.

1.2.3. Context Adaptive QM coder (CABAC)

This is the form of the entropy coding which gives better compression than ordinary QM coder and is used in H.264/MPEG-4 encoding. It basically adapts to the local context and relies on conditional probability rather than just dealing with the raw probability values.

The context can be previous 1, 2, 3 or n bits. In general for a context of n bits, there will be 2^n QM models.
E.g. If we consider 1-bit context and an input sequence as '100100'

1st bit	**Previous bit**	**QM coder to use**
1	0	QM 0
0	1	QM 1
0	0	QM 0
…		

For more details on CABAC, refer
http://en.wikipedia.org/wiki/Context-adaptive_binary_arithmetic_coding
http://www.cs.sfu.ca/CourseCentral/820/li/material/source/papers/CABAC.pdf

1.3. **Part B → Q.1**

1.3.1 Following 4 binary files have been given for compression:

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

1.3.2. Implementation of code

1. FileRead
   The data files are read 1 byte at a time and converted into bit stream. 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.
The QM table is read from the file QM_state_transition.txt present in the same folder and is loaded in the Qmtable. Details relating to the table storage can be found in the folder *Table*.

2. Main logic
   Pseudo code provided in the section 1.2.2 is followed as it is without any modifications.
   Following Initial conditions have been taken:

A	0x10000
C	0x0000
Current state	10
MPS	0
LPS	1

3. File Write
   The file is written bit by bit dynamically as and when renormalization takes place and/or the carry flag is true.
   NOTE: File IO routines in C++ don't allow bit writing so we buffer the data until one complete byte is full and then write the data to the file. If the last buffer contains less than 8 elements, *we simply append 0's* and write it to the file.

1.3.3. Results

The files have been encoded using 2 different mappings:

1. The original files are read directly bit by bit and the QM encoder is used.
   Compression Ratio (C.R) = (File Size after compression)/ (Original File Size) * 100

	Original File Size (bytes)	**QM encoding**	
		o/p File Size (bytes)	**C.R (%)**
Text.dat	8358	10090	120.72
Image.dat	65536	79201	120.85
Audio.dat	65536	79862	121.86
Binary.dat	65536	6512	9.94

2. The original files are encoded using Huffman encoding and are then read bit by bit before feeding it to the QM Encoder.

	File Size After Huffman Encoding (bytes)	**QM encoding**	
		o/p File Size (bytes)	**C.R (%)**
Text.dat	4807	5819	121.05
Image.dat	53918	65016	120.58
Audio.dat	63128	76865	121.76
Binary.dat	8204	1150	14.02

1.4. **Part B → Q.2**

1.4.1. Implementation of code

1. File Read
   File read operations are same as explained in section 1.3.2.

2. Main code
   The source code for CABAC is used which does the encoding given the 'context' and the 'current symbol'.
   Following routines (in red) of the source code provided are called:

```
FILE *fp;
fp=fopen("CABACencoded.dat", "w+"); //Stores the encoded data in this
//file
//calling the parameterized constructor for CABAC with the file pointer
QM obj(fp);

//Initialize the encoder parameters
obj.StartQM("encode");

//Encoding begins
WHILE Last Symbol read
      IF current symbol = 0
            gc = get Context based on previous n bits
            obj.encode(0,gc)
      ELSE IF current symbol = 1
            gc = get Context based on previous n bits
            obj.encode(0,gc)
      END IF
END WHILE

//Flush the remaining contents
obj.Flush();
```

3. File Write
   File Writing is done by the provided source code itself.

1.4.2. Preprocessing on Files

The preprocessing is done in the following way:

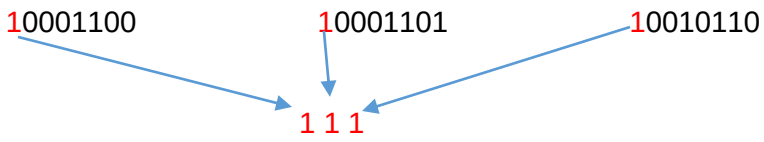
Step 1. The whole files is read and stored as bytes.
Step 2. First the MSB of each byte is read and stored, then the 2nd MSB is read and stored and eventually the LSB is read and stored.

E.g. If the file contains 3 bytes '140, 141, 150' then the preprocessing is done as follows:

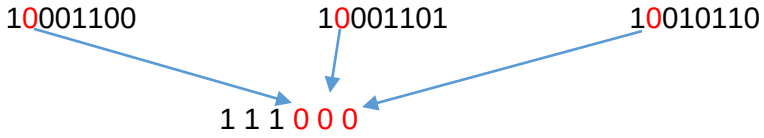
Bytes	140	141	150

Bit pattern	10001100	10001101	10010110

Step 1. Read the MSB

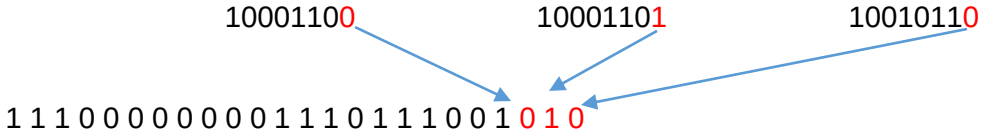


Step 2. Read the 2nd MSB



.
.

Step 8. Read the LSB



Final bit stream is '111000000011101110010 10'

This algorithm is chosen because:

- There is no drastic change between neighboring pixels/data in image/audio file. So the bytes are more or less same in neighboring areas.
- QM coder gives best compression when it gets a long stream of MPS.
- The above algorithm gives long stream of 0's and make the data amenable for compression using QM coding.

1.4.2. Results

The files have been encoded using 2 different mappings:

1. The original files are read directly bit by bit and the CABAC encoder is used.
   Compression Ratio (C.R) = (File Size after compression)/ (Original File Size) * 100

		N = 1			
		Without Preprocessing		**With Preprocessing**	
	Original File Size (bytes)	**Output File Size (bytes)**	**C.R (%)**	**Output File Size (bytes)**	**C.R (%)**

Text.dat	8358	8635	103.31	6151	73.59
Image.dat	65536	66253	101.09	45142	68.88
Audio.dat	65536	66962	102.18	57218	87.31
Binary.dat	65536	1914	2.92	7598	11.59

		N = 2			
		Without Preprocessing		**With Preprocessing**	
	Original File Size (bytes)	**Output File Size (bytes)**	**C.R (%)**	**Output File Size (bytes)**	**C.R (%)**
Text.dat	8358	8562	102.44	6072	72.65
Image.dat	65536	66139	100.92	45098	68.81
Audio.dat	65536	66553	101.55	56934	86.87
Binary.dat	65536	1887	2.88	7533	11.49

		N = 3			
		Without Preprocessing		**With Preprocessing**	
	Original File Size (bytes)	**Output File Size (bytes)**	**C.R (%)**	**Output File Size (bytes)**	**C.R (%)**
Text.dat	8358	8483	101.50	6078	72.72
Image.dat	65536	65930	100.60	44956	68.60
Audio.dat	65536	66524	101.51	54780	83.59
Binary.dat	65536	1864	2.84	7481	11.42

2. The original files are encoded using Huffman encoding and are then read bit by bit before feeding it to the CABAC Encoder.

		N = 1			
		Without Preprocessing		**With Preprocessing**	
	File Size After Huffman Encoding (bytes)	**Output File Size (bytes)**	**C.R (%)**	**Output File Size (bytes)**	**C.R (%)**
Text.dat	4807	4965	103.29	4959	103.16
Image.dat	53918	55298	102.56	55708	103.32
Audio.dat	63128	65270	103.39	65487	103.74
Binary.dat	8204	981	11.96	1233	15.03

		N = 2			
		Without Preprocessing		With Preprocessing	
	File Size After Huffman Encoding (bytes)	Output File Size (bytes)	C.R (%)	Output File Size (bytes)	C.R (%)
Text.dat	4807	4968	103.35	4959	103.16
Image.dat	53918	55493	102.92	55771	103.44
Audio.dat	63128	65319	103.47	65591	103.90
Binary.dat	8204	987	12.03	1224	14.92

		N = 3			
		Without Preprocessing		With Preprocessing	
	File Size After Huffman Encoding (bytes)	Output File Size (bytes)	C.R (%)	Output File Size (bytes)	C.R (%)
Text.dat	4807	4949	102.95	4969	103.37
Image.dat	53918	55451	102.84	55735	103.37
Audio.dat	63128	65381	103.57	65568	103.87
Binary.dat	8204	973	11.86	1214	14.80

1.5. Discussion

- QM coder is more efficient form of entropy encoding compared to Huffman encoding and is used in most of the compression techniques. In a way, QM coder has superseded Huffman coding.

- QM coder used in section 1.3. is just for understanding purpose. The carry flags and MSB of C flag can be encoded more efficiently. Hence we don't get any notable compression for the media files except for the 'binary.dat' file which has only 2 symbols and a long streams of 0's or 1's.

- CABAC adapts locally to the context and uses the QM coder efficiently and gives better compression ratios compared to context independent BAC. So, instead of using BAC directly, CABAC is used in most of the known compression techniques.

- BAC/ CABAC works best when data is presented as long streams of 1's or 0's. Hence when raw data file is given as input to the BAC, we don't get enough compression. This

can be verified from the results in section 1.4.2. When data is preprocessed properly producing long streams of 0's or 1's Compression ratio is improved by huge amount.

- The number of previous bits used in the context also changes the CR. However the CR is not directly proportional to the number of context bits used. In some cases, increasing the context bits can decrease the CR as observed from the table in section 1.4.2. For e.g. for audio.dat file in table 2 and 3 of section 1.4., we get a CR of 86.87% for N = 2 and CR of 86.59% for N = 3. Thus the number of context bits have to be chosen wisely depending on the type of media file