CSE 490GZ Final Exam Solutions

1) True or false questions

a) True – Blocks of bit-planes are encoded separately, providing the opportunity to spend different amounts of bits on different blocks.

b) False – JPEG uses the DCT transform.
c) True – This is due to auditory masking.
d) False – There are no dependencies across a group of frames.
e) False – The luminance component is more important for visual quality, so it is sampled at a higher rate.

f) False – Using k-d trees, it requires time O(log n) on average.
g) False – Blocking artifacts are smaller, but are not eliminated.
h) True – Motion vectors can be predicted from other motion vectors.
i) True – The lowest subband is transmitted first yielding a small image with high fidelity. Successive subbands yield larger images with high fidelity.

j) True – GT-DCT uses bit plane-coding of DCT coefficients and was shown to outperform JPEG.

k) True – They all use bit-plane encoding which leads to natural embedded codes.

l) False – Decoding each frame requires the previous one, so one lost frame affects the remaining frames in the video sequence.

m) True – Predictive coding using differences is used to code the DC coefficient and the AC coefficients are zig-zag coded.

n) True – PSNR does not accurately reflect audio quality.
o) True – B-frames use both forward and backward prediction.

2) Examining differences in encoding/decoding speed

a) LZ77 – The encoder must search for the longest match to determine the triple that encodes the next portion of the input string, while the decoder is given the triple and does not need to perform the search.

b) Burrows-Wheeler Transform – The encoder must sort the cyclic shifts of the input string while the decoder does not.

c) MPEG-1 – The encoder needs to perform motion compensation which involves searching for the block in the previous frame that leads to the least distortion. The decoder is given the motion vector and does not need to perform the search.

3) Decode: 0010011100010 using LZW with doubling

Dictionary, size = 4

Symbol	Code
a	00
b	01
c	10
	11

Input: 00<u>10011100010</u>
Output: a
Next entry: a?

Dictionary, size = 4

Symbol	Code
a	00
b	01
c	10
ac	11

Input: 0010011100010

Output: a c

Next entry: c?

Dictionary, size = 8

Symbol	Code
a	000
b	001
c	010
ac	011
ca	100
	101
	110
	111

Input: 0010011100010

Output: a c ac

Next entry: ac?

Dictionary, size = 8

Symbol	Code
a	000
b	001
c	010
ac	011
ca	100
acc	101
	110
	111

Input: 0010011100010

Output: a c ac ca

Next entry: ca?

Dictionary, size = 8

Symbol	Code
a	000
b	001
c	010
ac	011
ca	100
acc	101
cac	110
	111

Input: 0010011100010

Output: a c ac ca c

Next entry: c?

Decoded string: a c ac ca c

4) Using the Burrows-Wheeler Transform decode the following:
L = bbabbaa
X = 2

- Compute the mapping T by using the following rule:
If F[i] is the k-th x in F, then T[i] is the index for the k-th x in L. In other words, F[i] = L[T[i]].

index: 0 1 2 3 4 5 6
F: a a a b b b b
L: b b a b b a a
T: 2 5 6 0 1 3 4

- Decode the original string by using T[i] as the next index to read from F (that is, i' = T[i]) where initially i = X.

```
index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a

index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a b

index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a b b

index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a b b a

index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a b b a b

index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a b b a b b

index:  0 1 2 3 4 5 6
F:      a a a b b b b
T:      2 5 6 0 1 3 4
output: a b b a b b a
```

Decoded string: a b b a b b a

5) P(a) = 1/4, P(b) = 3/4
C(a) = 0, C(b) = 1/4
Decode "011" using arithmetic coding with scaling (assume that we are decoding 4 symbols)

Tag = .011 = 3/8

W	L	R	Tag	Output	
	0	1	3/8		
1	1/4	1	3/8	b	
3/4	4/16	7/16	3/8	a	scale by 2x - 0.5
	0	3/8	2/8		scale by 2x
	0	6/8	4/8		
6/8	6/32	24/32	4/8	b	
18/32	21/64	48/64	4/8	b	

Decoded string: babb

6) Do one iteration of the Lloyd algorithm with initial codewords c(0)=2 and c(1)=3 on the data:

pixel value	0	1	2	3	4	5	6	7
frequency	200	100	100	50	50	100	200	200

Initialization:

$X(0) = [0,1,2]$

$X(1) = [3,4,5,6,7]$

$D(0) = 100*1^2 + 200*2^2 = 900$

$D(1) = 50*1^2 + 100*2^2 + 200*3^2 + 200*4^2 = 5450$

$D = D(0) + D(1) = 6350$

First iteration:

$c'(0) = \text{round}((200*0 + 100*1 + 100*2) / 400) = 1$

$c'(1) = \text{round}((50*3 + 50*4 + 100*5 + 200*6 + 200*7) / 600) = 6$

$X'(0) = [0,1,2,3]$

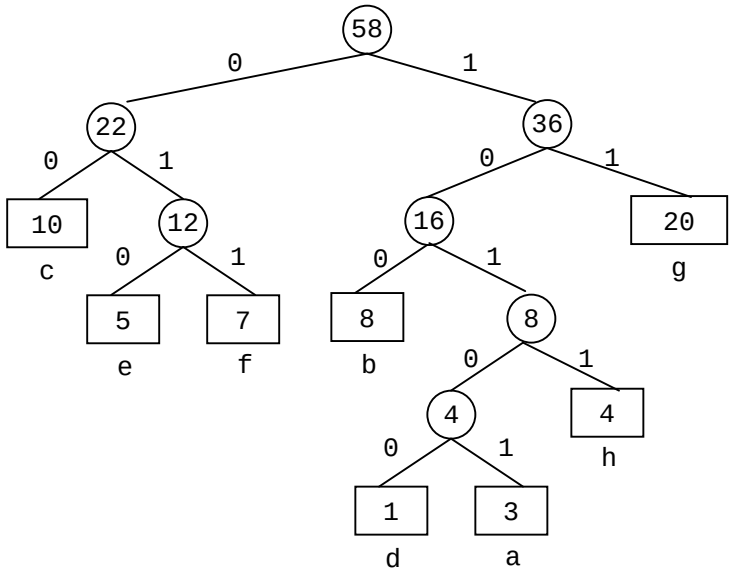
$X'(1) = [4,5,6,7]$

$D'(0) = 300*1^2 + 50*2^2 = 500$

$D'(1) = 300*1^2 + 50*2^2 = 500$

$D' = D'(0) + D'(1) = 1000$

7a) Optimal Huffman tree



b) ABR = (5*3 + 3*8 + 2*10 + 5*1 + 3*5 + 3*7 + 2*20 + 4*4) / 58 = 2.69

a b c d e f g h

8) Show the remaining steps in the Sequitur algorithm.

abaab aaba

S -> AaA

A -> ab

abaaba aba

S -> AaAa Enforce diagram uniqueness.

A -> ab Aa occurs twice. Create new rule B -> Aa

abaaba aba

S -> BB Enforce rule utility.
A -> ab A only occurs once.
B -> Aa Remove A -> ab

abaaba aba

S -> BB
B -> aba

abaabaa ba

S -> BBa
B -> aba

abaabaab a

S -> BBab
B -> aba

abaabaaba

S -> BBaba Use rule B -> aba
B -> aba

a b a a b a a b a

S -> BBB
B -> aba

9) Show how the class version of SPIHT processes the following 6 bits (assuming the signs of the coefficients in S have already been processed from the encoded bit stream)

Encoded bit stream: 101100
S = (0,0),(0,1),(1,0),(1,1)
Z = (R,0,1),(R,1,0),(R,1,1)
Z' =

Encoded bit stream: 101100
(R,0,1) is significant
S = (0,0),(0,1),(1,0),(1,1)
    (0,2),(0,3),(1,2),(1,3)

Encoded bit stream: 101100
(0,2) has negative sign
(0,3) has positive sign
(1,2) has positive sign
(1,3) has negative sign

Z = (RC,0,1),(R,1,0),(R,1,1)
Z' =

Encoded bit stream: 101100
(RC,0,1) is insignificant
Z = (R,1,0),(R,1,1)
Z′ = (RC,0,1)

10) Move-to-front coding followed by arithmetic coding uses the fewest number of bits. The move-to-front coding will produce at least 2n-2 zeros. Only the first “a” and first “b” may not have an index of 0. This highly skewed alphabet will be coded very efficiently using arithmetic coding. Indeed it can be shown that the arithmetic code would have O(log n) bits.

The two other methods use about the same number of bits.

Arithmetic coding using the frequencies of the symbols as a first-order model uses about 2n bits. Since a’s and b’s are equally likely then the arithmetic coding interval has size $(1/2)^{2n}$.

Adaptive arithmetic coding yields an arithmetic coding interval of

1	2	3	…	n	1	2	3	…	n
-	-	-	…	-	-	-	-	…	-
2	3	4	…	n+1	n+2	n+3	n+4	…	2n+1

which equals n!n!/(2n+1)!.

Taking the $\log_2$ of the reciprocal yields $\log_2((2n+1)!/(n!)^2)$ bits approximately. It can be shown by induction on n that $\log_2((2n+1)!/(n!)^2) > 2n$ so that the adaptive arithmetic coding is no better than arithmetic coding using the frequencies. A little more work shows that in the limit $\log_2((2n+1)!/(n!)^2)/n = 2$ so that in the limit the two methods use about 2n bits (no real compression).