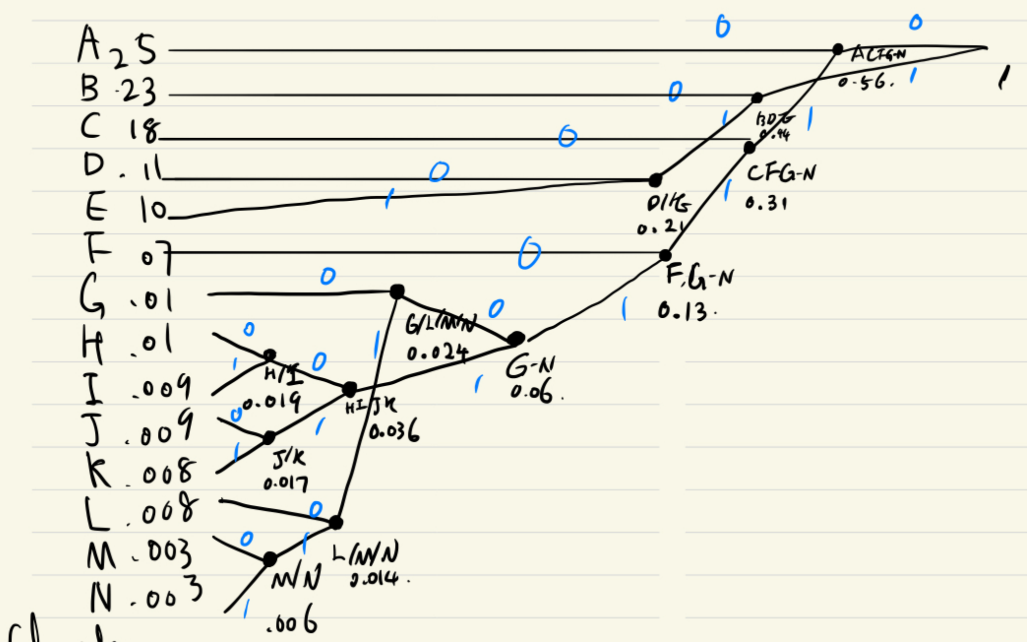
ECE 253, Solutions for Homework 6

1) **Truncated Huffman Coding**



Codewords:

A: 00	G: 0111 00	M: 0111 0110
B: 10	H: 0111 100	N: 0111 0111
C: 010	I: 0111 101	
D: 110	J: 0111 110	
E: 111	K: 0111 111	
F: 0110	L: 0111 010.	

a)

Symbol	Prob.	Codeword (Full Huffman)
A	.25	00
B	.23	10
C	.18	010
D	.11	110
E	.10	111
F	.07	0110
G	.01	0111 00
H	.01	0111 100
I	.009	0111 101
J	.009	0111 110
K	.008	0111 111
L	.008	0111 010
M	.003	0111 0110
N	.003	0111 0111

b) The entropy of this source is computed by:

$$H(X) = \sum_{i=1}^{J} p(s_i) log(\frac{1}{p(s_i)}) = 2.8009$$

c) The expected length of the full Huffman code is:

$$El(C_{full}) = \sum_{x \in A_{full}} p(x)l(x) = 2.826$$

The efficiency of the full Huffman code is $H/L = 2.8009/2.826 = 0.9911$
The expected length of the truncated Huffman code is:

$$El(C_{trunc}) = \sum_{x \in A_{trunc}} p(x)l(x) = 2.83$$

The efficiency of the truncated Huffman code is $H/L = 2.8009/2.83 = 0.9897$
So we can see that the truncated Huffman code comes with a small cost of greater average codeword length. The truncated Huffman code provides however some reduction in the complexity of the code design (since we are doing fewer source reductions). And in the case of JPEG, the use of a truncated Huffman code means that much smaller code tables can be stored and/or transmitted (as the grouped items have a natural numeric ordering and their full codewords do not need to be stored or transmitted).

2) **Arithmetic Coding**

We read in the first symbol b which corresponds to the interval [0.1,0.5). Both endpoints are to the left of 1/2, so we **output 0.**
We rescale to [0.2, 1).
This straddles 0.5, so no further output occurs at this time.

We read in the next symbol, another b. The new sub-interval is [0.28,0.6) which can be calculated as:
$0.2 + 0.1 \times 0.8 = 0.28$
$0.28 + 0.4 \times 0.8 = 0.6$

This straddles 0.5, so no further output occurs at this time.
The sub-interval [0.28, 0.6) has length $\delta = 0.32$.

We read in the last symbol, a. So the new interval is $[0.28, 0.28 + 0.1 \times \delta) = [0.28, 0.312)$.
Both endpoints are to the left of 1/2, so we **output 0.**

We rescale to get [0.56, 0.624).
Both endpoints are the right of 1/2, so we **output 1.**

Rescale to [0.12, 0.248).
Both endpoints are to the left of 1/2, so we **output 0.**

Rescale to [0.24, 0.496).
Both endpoints are to the left of 1/2, so we **output 0.**

Rescale to [0.48, 0.992).
This straddles 0.5, so there is no further output at this time.

3) DCT and DFT

The 2N-point unitary DFT of $g(k)$ is:

$$\begin{aligned}
G(u) &= \frac{1}{\sqrt{2N}} \sum_{k=0}^{2N-1} g(k) e^{-j2\pi uk/2N} && (1)\\
&= \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) e^{-j2\pi uk/2N} + \frac{1}{\sqrt{2N}} \sum_{k=N}^{2N-1} f(2N-1-k) e^{-j2\pi uk/2N} && (2)\\
&= \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) e^{-j2\pi uk/2N} + \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) e^{-j2\pi u(2N-1-k)/2N} && (3)\\
&= \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) (e^{-j2\pi uk/2N} + e^{j2\pi u(k+1)/2N}) && (4)\\
&= \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) (e^{-j2\pi uk/2N} e^{-ju\pi/2N} + e^{j2\pi u(k+1)/2N} e^{-ju\pi/2N}) e^{+ju\pi/2N} && (5)\\
&= \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) (e^{-j\pi u(2k+1)/2N} + e^{j\pi u(2k+1)/2N}) e^{+ju\pi/2N} && (6)\\
&= \frac{1}{\sqrt{2N}} \sum_{k=0}^{N-1} f(k) 2\cos\left[\frac{u\pi(2k+1)}{2N}\right] e^{+ju\pi/2N} && (7)\\
&= \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} f(k) \cos\left[\frac{u\pi(2k+1)}{2N}\right] e^{+ju\pi/2N} && (8)\\
& && (9)
\end{aligned}$$

So, over the N-point range from 0 to $N-1$, the DCT coefficients $C(u)$ and the DFT coefficients $G(u)$ are related by

$$C(u) = \frac{\alpha(u)}{\sqrt{2/N}} G(u) e^{-ju\pi/2N}$$

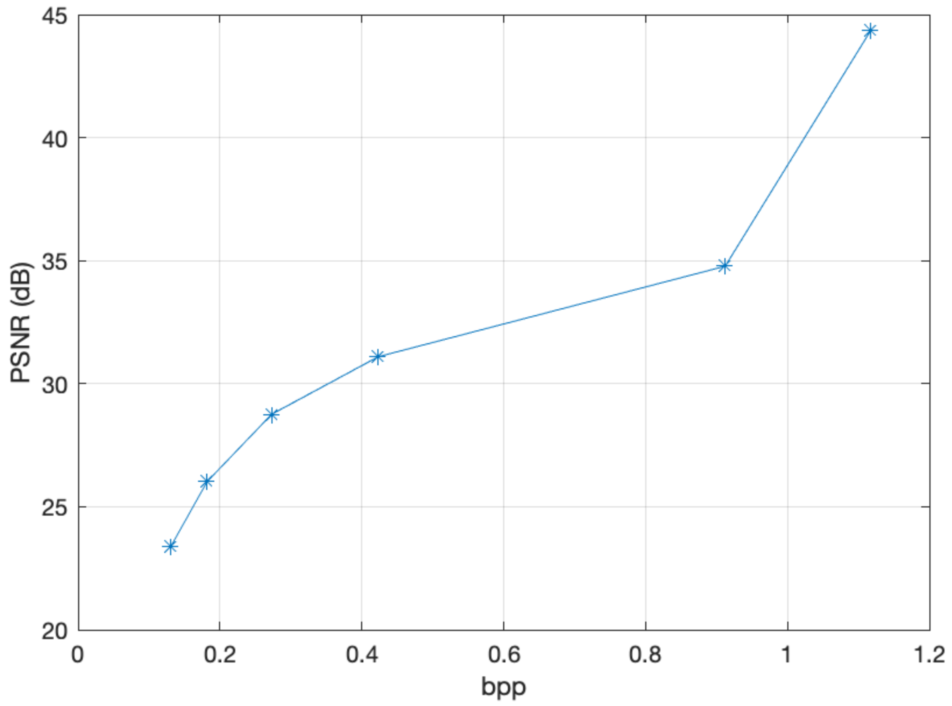
4) **JPEG**

a) The image totem-poles.tif occupies 508,688 bytes which is 4069504 bits. The image is 600 by 800, so 480,000 pixels. So this comes out to 8.478 bits per pixel.

b) Writing out the image with various quality factors, checking the number of bytes, and reading it back in to check the PSNR, we can create a table like this:

Quality Factor	bytes	bpp	PSNR
1	7,829	0.13048	23.370
5	10,920	0.182	26.023
10	16,377	0.2729	28.759
20	25,393	0.4232	31.096
50	54,734	0.9122	34.775
70	67,046	1.117	44.35

At low rates, the blocking artifacts are very obvious, and they look particularly bad in the smooth regions. At high rates, there isn't any noticeable difference between between quality levels 50 and 70. The PSNR does not increase linearly. Constant increments of rate produce diminishing PSNR gains.



c) We can downsample by a factor of 4 in each direction, compress with a specific quality factor, and then upsample and compute the PSNR as follows:

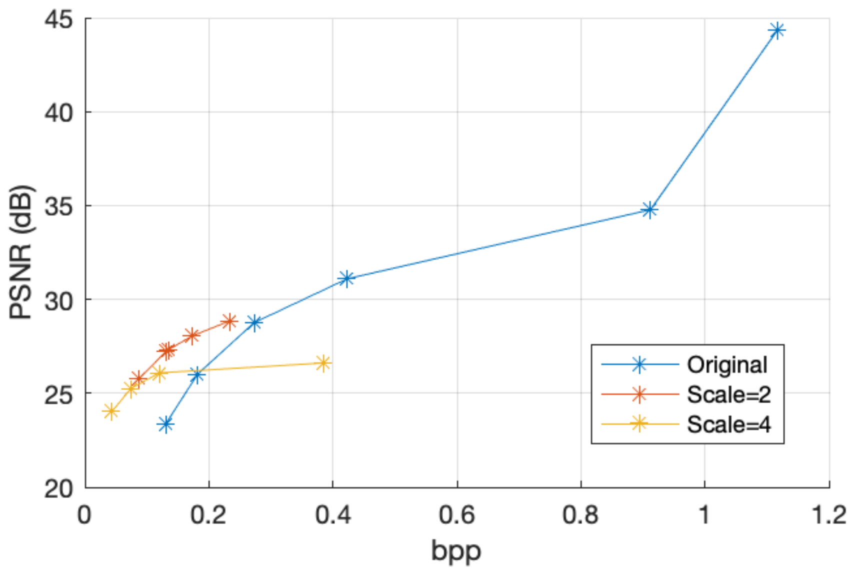
```
>> totemquarter = imresize(totem,0.25);
>> imwrite(totemquarter,'totemS20.jpg','Quality',20)
>> psnr(totem,imresize(imread('totemS20.jpg'),4),255)
ans =
24.0344
```

With this, we can generate a table:

Quality Factor	bytes	bpp	PSNR
20	2,631	0.04385	24.0344
50	4,439	0.07398	25.2438
80	7,192	0.1199	26.0866
100	23,069	0.3845	26.6131

Similarly, we can generate a table for when the downsampling is by a factor of 2 in each direction.

Quality Factor	bytes	bpp	PSNR
10	5,268	0.0878	25.7864
19	7,854	0.1309	27.221
20	8,072	0.1345	27.3181
30	10,375	0.1729	28.0484
50	14,043	0.2340	28.8552



Here, the specific quality factor of 19 was chosen because it makes the number of bytes comes out to 7854 which is very close to the rate obtained by the lowest quality coding of the full size image. The downsampled versions perform better at low bit rates. But at higher rates, they quickly saturate to relatively low quality images.

d) The totem1 image has a very low PSNR of 23.37. It is very blocky, and doing some filtering at the block boundaries can improve the quality. For example, we can modify the pixels at the boundary of each block to be the mean value of the pixels on the two sides of the block edge. This trivial adjustment increases the PSNR from 23.73 to 24.1258. Better results can come from smarter smoothing, or by estimating missing AC coefficients.