# Homework 4

Due: Feb 23, 2013 2:00 PM (early)
Feb 25, 2013 11:59 PM (on time)
Feb 27, 2013 11:59 PM (late)

This homework must be handed in electronically via the hand-in script. All code must be your own, though as usual you are welcome to discuss ideas with other students in the class. No documentation or explanation is required on this homework, though it might help you recover partial credit for buggy code.

## JPEG decompression

This homework consists entirely of writing Matlab code. The objective is to decompress our favorite JPEG of Mr. Bilbo Baggins. The JPEG format relies on two tools related to this class: the discrete cosine transform, and Huffman codes. (The cosine transform is covered on the worksheet for class 6, as posted on the website; Huffman codes are the topic of the *next* assignment, covered from a mathematical perspective there, and a practical perspective here—they will not be covered in class.)

Before you start, download "hobbit.jpg", as linked from class 6 on the course homepage. We are providing you with wrapper code for this image; you must fill in the code for the cosine transform, and write 5 simple helper functions, to correctly decode the grayscale component of this image. (While Bilbo is in color, we will be ignoring that aspect of JPEGs here.)

The JPEG file format uses a combination of *lossless* and *lossy* encoding methods to represent images in a very compact form (see the discussion at the start of assignment 5). Lossy image encodings do not exactly store the original image, but instead "lose" parts of the image that are judged to be unlikely to be noticed by the human visual system. The lossy component of JPEG compression is based on the cosine transform, where low-frequency components of the image are judged to be much more important than high-frequency components; these components are separated from each other via the cosine transform; different portions of the cosine transform of the image are then rounded to different precisions.

These rounded cosine transform coefficients are then encoded losslessly via a combination of *Huffman codes*, which have the property (as you will prove in assignment 5) that they store information *as compactly as possible* (among all coding schemes of a certain form). Thus JPEG compression is a two-step process: first figure out (via the cosine tranform) which information we want to discard; next perfectly preserve the information we want to keep, as compactly as possible. This beautiful division of labor looks slightly less beautiful in practice because of the hacks and optimizations needed to take this from theory to practice. But as you work through this assignment, try to keep in mind the overarching goals of this compression scheme, and how they relate to concepts from class.

## Debugging in Matlab

Matlab has a full-featured debugger. Type `help debug` for more information. One of the most useful commands is `dbstop if error` which tells Matlab to pause execution in debug mode whenever an error occurs. The command prompt will change from "`>>`" to "`K>>`" in debug mode. To stop debugging, type `dbquit` or press the appropriate button at the top of the Matlab editor. From debug mode, you can run further Matlab commands, which could bring you two levels deep into debug mode....

You can set breakpoints by clicking immediately to the right of the line numbers in Matlab's editor (a red circle will appear). You can traverse up and down the call stack with `dbup` and `dbdown`. Perhaps the easiest way to transfer variables out of debug mode is by saving them to disk and loading them again. See Matlab's help for the `save` and `load` commands.

## JPEG's fixed length encoding

As a first step, you will need a short function for decoding a certain *fixed-length* code, which is a simple variation on binary. Fill out the stencil `decodeFIXED` that reads a fixed number of bits from a bit string, computes the number these bits represent in binary, and subtracts something from this result depending on an if statement.

## Introduction to variable-length codes

More sophisticated than the above fixed-length codes are *variable-length codes*, where different codewords have different lengths, and you cannot tell how long a codeword will be until you are done decoding it. (Variable-length codes are often colloquially referred to as "Huffman codes", after a particular greedy algorithm for optimally constructing them that you will analyze in assignment 5.)

Consider a binary tree $T$, as in figure 1, with each leaf labeled by a distinct symbol from the set $\Sigma$, which in this case is $\{0, 2, 4, 10, 17\}$. This tree represents a variable-length binary code, when interpreted as follows: given a binary string $b$, start at the root of the tree and read the bits of $b$, where every time a 0 occurs, descend to the left child in the tree and every time a 1 occurs, descend to the right child in the tree; when a leaf is reached, return the leaf's label from $\Sigma$. (Ignore the labels of internal nodes for the moment.)

If this process does not reach the end of the bit sequence $b$, then we can continue decoding the remainder of $b$ by resetting to the root of the tree, and repeating the above process.

As part of decompressing JPEGs, you will write several pieces of code to construct and traverse such trees, to decode variable-length codes that are defined at the top of the "hobbit.jpg" file. We will suggest that you represent variable-length encodings of nonnegative numbers in Matlab by a table $t$ with 2 rows and many columns: each *non-leaf* node in the tree is represented by a column of $t$, where the two rows represent the left and right children respectively; if a value $v$ in the table is *at least 1* then $v$ stores the *column in the table where the corresponding child is stored*; if $v$ is 0 or negative then *the corresponding child is a leaf, with value* $-v$.

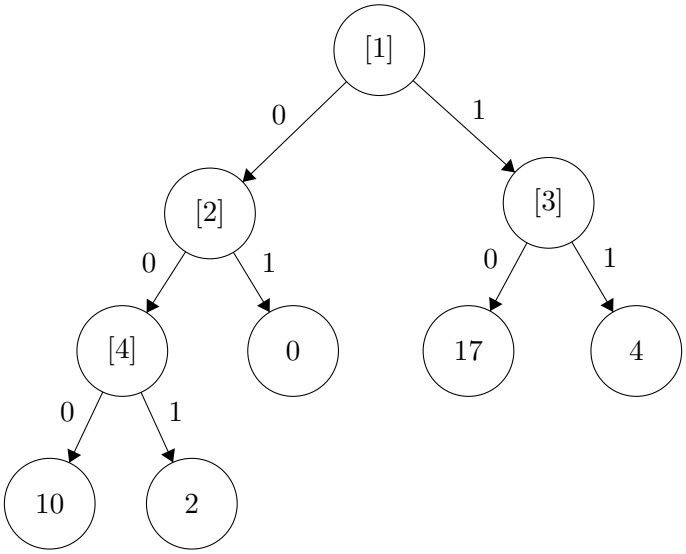


Figure 1: A variable-length coding tree

Thus the table $\begin{bmatrix} 2 & 4 & -17 & -10 \\ 3 & 0 & -4 & -2 \end{bmatrix}$ corresponds to the tree in figure 1, where "000" encodes 10, "001" encodes 2, "01" encodes 0, "10" encodes 17, and "11" encodes 4; each internal node of the tree is labeled by the column of the table which represents it. Further, if we are applying this decoding scheme repeatedly, then the bit string "00101000011110" encodes the sequence $(2, 0, 10, 0, 4, 17)$.

Your next task is to fill out the stencil `decodeVLC` such that, if run 6 times on the above bit string and table, starting from position 1, will return the above decoding. Namely,

```
pos=1;for i=1:6,[out,pos]=decodeVLC([0 0 1 0 1 0 0 0 0 1 1 1 1 0], pos, ...
[2 4 -17 -10;3 0 -4 -2]); out, end
```

should return the 6 elements of the sequence $(2, 0, 10, 0, 4, 17)$ in the 6 iterations of the loop. (In Matlab, ending a line with "..." lets you continue entering the statement on the next line.)

## Defining VLC decoding tables

The efficiency of a variable-length code depends on which codewords get encoded at which lengths—in principal, the most frequently used symbols should be encoded with the shortest codewords. The JPEG specification provides a certain unambiguous way of building variable-length code trees based on how many leafs appear at each depth.

In the JPEG standard, and in the function `createVLC` which you must write, variable-length codes are described by two arrays, the first of which, `nums`, stores the number of codewords of each length (equivalently, the number of leaves in the tree of each depth); the second array, `vals`, stores the values to be encoded. Thus `length(vals)` should equal `sum(nums)`. The array `nums` describes the entire structure of the encoding tree, under the rule: each time a new codeword (leaf in the tree) must be created, it is created at the prescribed depth and at the leftmost possible position; `vals` then describes the values to be placed at the leaves, in order.

This process can be described by the following pseudocode (where, as in Matlab, each array is indexed starting from 1):

CREATEVLC($nums, vals$)
    Initialize empty *tree*
    **for** $i \leftarrow 1$ **to** LENGTH($nums$)
        **do for** $j \leftarrow 1$ **to** $nums(i)$
            **do** Create a new leaf at depth $i$ in *tree* at the leftmost possible position
                Label this leaf with the next value from *vals*
    return *tree*

The JPEG standard goes as far as suggesting how to implement the above pseudocode, advising that you keep a variable `code` that, when written as an $i$ bit binary number, will represent the next available codeword of the desired length $i$: `code` is initially 0, and at the end of each iteration of the $i$ loop it doubles, at the end of each iteration of the inner $j$ loop it increases by 1.

For example, `createVLC([0 1 3],[9 0 14 7])` defines a variable-length code where there are 0

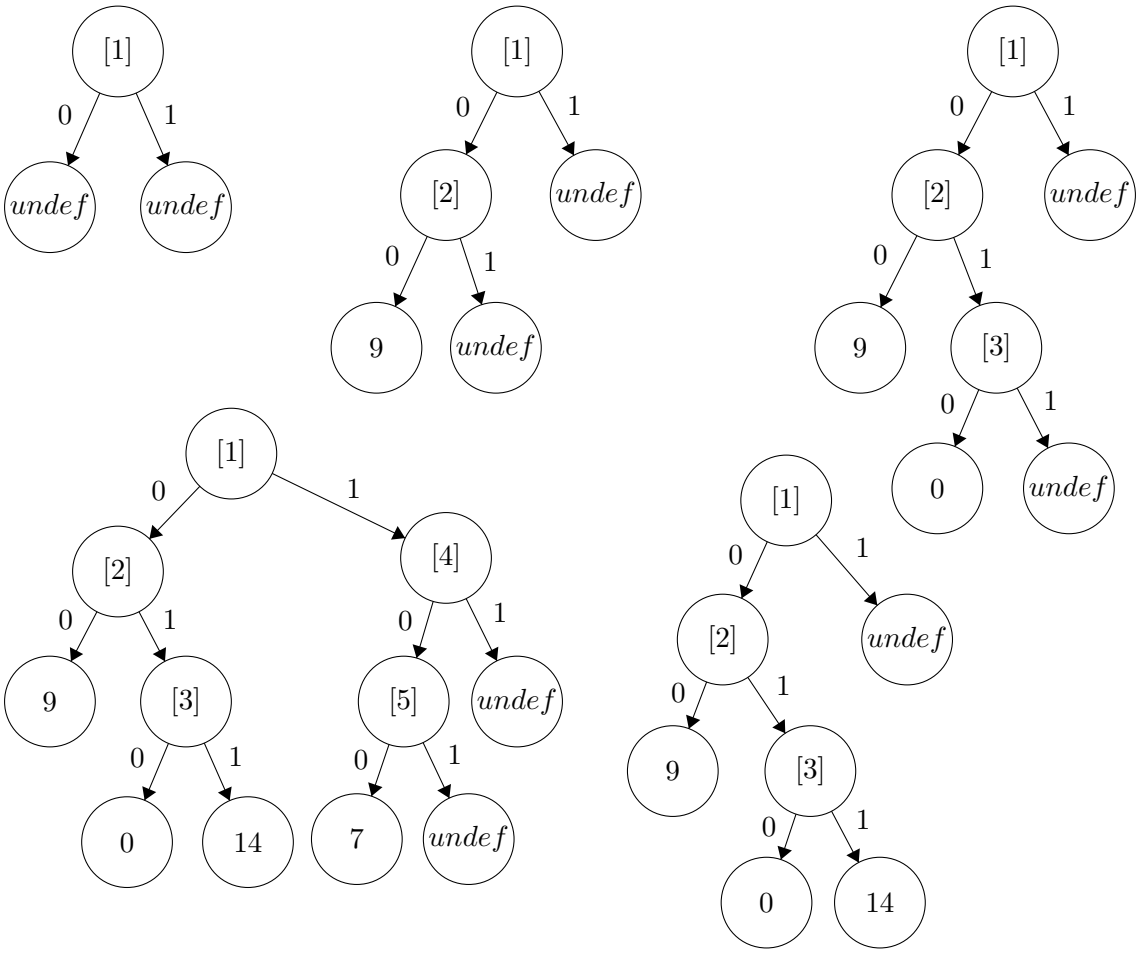


Figure 2: Stages in the construction of a variable-length encoding tree, clockwise from top left

codewords of length 1, 1 codeword of length 2, and 3 codewords of length 3, and further, that "9" should be encoded with a codeword of length 2, and each of "0", "14", and "7" should be encoded with a codeword of length 3. The stages of constructing the tree corresponding to this code are depicted in figure 2. As above, we advise that you represent the final tree in Matlab as $\begin{bmatrix} 2 & -9 & 0 & 5 & -7 \\ 4 & 3 & -14 & 0 & 0 \end{bmatrix}$, where the last two entries correspond to the "*undef*" nodes, but in the most natural Matlab implementation, end up as 0. To trace the role of the variable `code` as suggested above and implemented in the stencil code: it starts out as `code=0` when the first node "9" is inserted at depth 2, where 0 decoded in binary to length 2 is "00" which describes the location of "9" in the tree; after this, `code` is incremented to 1, and because 9 is the last (and only) thing to be encoded at length 2, `code` is doubled to now be 2, which when decoded as a length-3 binary number yield "010" which describes the location of the next entry, "0" in the tree; after this, code is incremented to 3, corresponding to "011" which is the location of the next entry, "14"; and after this code is incremented to 4, corresponding to "100", the location of "7" in the tree.

## Reading a block of 64 cosine transform coefficients

You are now ready to run the code we provided in `bilbo.m`. Try this now. Provided your `createVLC` code has not crashed, execution will now be paused via the `keyboard` command, which will put you at the debugging command prompt ("`K>>`"). Your `createVLC` code has been run four times, creating four variable-length code tables, `t1, t2, t3, t4`. Also, the variable `bits` now stores the raw bits representing the image, via the codes `t1, t2, t3, t4`. Your task in this section is to write a function to decode a block of 64 cosine transform coefficients from `bits` using these coding tables.

The bits representing the image of Bilbo are divided into blocks that encode 64 cosine transform coefficients, representing an $8 \times 8$ block of the image. The next function you must write is `readcoeffs`, which takes as input the array of `bits` loaded from the image (which you will never need to modify), a starting position in this array, `pos`, and two variable-length code tables, the first of which codes the first element in the block, the second of which codes all the remainder (see details below). The output of `readcoeffs` should be a length-64 array, and the position reached in the bitstream. The first cosine transform coefficient represents a function of frequency *zero* in the $x$ and $y$ directions, and, in analogy with electric current, is referred to as "DC", while the other coefficients are "AC".

The input bit string consists of bits encoded with the DC and AC variable-length codes (which you can now decode with your function `decodeVLC`), along with bits that can be decoded via the `decodeFIXED` function that you wrote first. The codes are used in the following pattern:

$$(DC, fixed), (AC, fixed), (AC, fixed), (AC, fixed), \ldots, (AC, fixed), AC_0$$

The last entry denotes that your task for this section is done when you decode "0" with the AC code. The rest of the input is divided into pairs, where the first pair is different from the others.

The first pair, $(DC, fixed)$ represents the following process: the DC code (as decoded with your `decodeVLC` function) encodes the first bits in the sequence; the value decoded will be a *length*. This length should be input to your `decodeFIXED` function, telling it how many bits to decode. The result of this `decodeFIXED` function will be the first out of 64 entries that you will compute in this section, to be saved in `coeffs(1)`.

Each of the following pairs, $(AC, fixed)$ is decoded similarly, though with a few differences. Decoding the initial segment of bits encoding this pair (via `decodeVLC` on the AC code) will return a number $x$ that encodes *two* things simultaneously: `mod(x,16)` will store a length that, as above, should be input to `decodeFIXED` to decode the next segment of bits to a value `val`. However, `val` is not necessarily placed in the next location of the length-64 array `coeffs`: it is placed in the next location of `coeffs` *after* `floor(x/16)` zeros have been inserted into `coeffs`. (Thus, `floor(x/16)` encodes the number of zero coefficients between each pair of nonzero coefficients.)

Test your code by running it on the bits from `bilbo.jpg`, as `[coeffs, pos]=readcoeffs(bits,1,t1,t2)`, making use of the first two variable-length code tables you have created (the first of which will be interpreted as the DC code table, the second as the AC code table). The returned array `coeffs` should start with `[-234 51 -116 -2 -53 \ldots]` and all the values from the 29th position to the 64th should be 0; `pos` should be 171. Decoding the *next* block should yield coefficient that start with -88.

## Assembling the pieces

You are now ready to run the section of code starting after the `keyboard` command. Try it. After 3 to 30 seconds, depending on how fast your decoding is, you should see an image appear. The outline of Bilbo and his sword should be vaguely discernable. Your task in this section is to finish the decoding process, which involves a cosine transform on each $8 \times 8$ block, and some cleanup.

The image of Bilbo is divided into $16 \times 16$ blocks, each of which is represented by *six* subblocks of cosine transform coefficients: the first four blocks are encoded with the tables `t1` and `t2` and encode the intensity of the image in the top left $8 \times 8$ portion of the $16 \times 16$ block, the top right, the bottom left, and the bottom right respectively. The next two blocks are encoded with the *other* tables, `t3` and `t4`, encoding color information. We will ignore the contents of these color blocks, but we *must* decode them properly—since the image is encoded with variable-length codes, we cannot figure out where these useless blocks end without first decoding them. The subblocks of 64 coefficients that your `readcoeffs` code produces must be mapped to $8 \times 8$ arrays; unlike the row and column orders for storing 2-dimensional arrays that we have seen in class, the JPEG standard uses a "zigzag" order, as represented by the array `zz`. Look at the code after the `keyboard` command, to see that it is implementing the process just described. Two tasks remain.

First: you must apply the 2-dimensional cosine transform to each $8 \times 8$ block of the image. See the handout from class 6 for details. This should involve two lines of code, one to compute the matrix $M$, and a second, inside the loops, to apply the 2-dimensional transform via 2 multiplications using $M$. (There is a chance of confusion between the cosine transform and the *inverse* cosine transform; check the notes, or try both if the results for one do not make sense; also, make sure you are transforming *both* the rows and columns of each block—the 2-dimensional transform).

Second (though this will actually happen first in the code): the cosine coefficients are not what they should be, and you must fill in the function `fixJPEGcoeffs` to fix this. Try adding each of the manipulations below to this function in turn, re-run the code, and try to understand the effect it has on the image.

- Scale each of the coefficients by multiplying it by the corresponding entry of the *quantization matrix* `q`. (Make sure to use Matlab's "`.*`" to do element-by-element multiplication, instead of matrix multiplication with "`*`".) Also divide each coefficient by 4.

- There are many different conventions for what each entry of the cosine transform matrix means, and the convention used by the JPEG standard differs from the one used in class: to convert the coefficients to be ready to be transformed by our cosine transform, each of the entries in the first column must be divided by $\sqrt{2}$, and then each of the entries in the first row must be divided by $\sqrt{2}$. (The $(1,1)$ entry will end up divided by 2.)
- It turns out that, even after the entire above decoding process, the $(1,1)$ entries of the coefficients array are still not stored directly, but rather, only the difference between the current $(1,1)$ entry and the $(1,1)$ entry of the *previous* block is stored, with 128 being the default value before any blocks are read. Thus, reverse this process at the end of your `fixJPEGcoeffs` code using the variable `prev` (which is initialized to 128 in "bilbo.m"): add `prev` to `coeffs(1,1)`, and then update `prev` to this result.

The stencil code then crops the image to the right size (the JPEG format stores the image in $16 \times 16$ blocks, but some of these blocks extend off the image), rounds each entry to an integer, and clamps the values between 0 and 255 with the code

```
im=round(min(255,max(0,im(1:398,1:660))));
```

Your result should be similar (though perhaps off by 1 or 2 in each pixel) to the grayscale component of Bilbo, which we can load in Matlab and display as:

```
im=double(imread('hobbit.jpg'));im=im(:,:,1)*.299+im(:,:,2)*.587+im(:,:,3)*.114;
imagesc(im);colormap('gray');colorbar
```

(The coefficients .299, .587, and .114 are from the definition of the color format that JPEG uses.)

Compare your results and debug as necessary.