Homework 5

Due: Mar 2, 2013 2:00 PM (early)
Mar 4, 2013 11:59 PM (on time)
Mar 6, 2013 11:59 PM (late)

The written portion of the homework must be handed in to the CS157 hand-in bin located on the CIT 2nd floor between the Fishbowl and the locker. The programming portion of homeworks (when applicable) must be handed in electronically via the hand-in script.

Each problem must be handed in separately with your name on the top and the time of hand-in. You may turn in different problems for different deadlines. For more information, please review the assignment and due date policy in the course missive on the course webpage.

Please ensure that your solutions are complete and communicated clearly: use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

You are allowed, and often encouraged, to use Wikipedia to help with your homework. See the collaboration policy on the course webpage for more details.

In this homework you will be working in pairs. These pairs are for this assignment only, and any future pair assignments must be done with a different partner. Please email the HTA list if you want us to find you a partner. You and your partner should write and hand in one solution per problem and put both your names on it.

Working in pairs will give you an opportunity to improve your thinking, communication, and writing skills. If something you write requires a verbal explanation for your partner to understand it, consider this a valuable sign that this explanation should be included in your writeup. In particular, you are responsible for everything you and your partner submit. It is an academic code violation to sign your name to something that is not yours. Further, however, the material covered on the homework will help prepare you for the exam, so aim for mastery of all of it.

Problem 1

Bilbo has a long journey ahead of him and wants to stay in comfortable hotels along his route each night, but can only walk 20 miles in a day. Fortunately, he has a guidebook charting the locations of all the hotels along his route. (Bilbo already knows which route he is taking; he only has to choose where along his route to stay each night.)

1. (8 points) Find a greedy algorithm for Bilbo to compute how to finish his journey in the least number of days. Points on this problem will only be given for the *proof* that your algorithm is optimal; more points will be given for simpler and clearer proofs.
2. (3 points) After studying your algorithm for the previous problem, Bilbo realizes that, actually, the different hotels cost different amounts, and what he actually wants to do is minimize the total cost of his journey. (Luckily, his guidebook also lists the cost of each hotel.) He thinks of the following greedy algorithm: wherever he is, for each hotel within a day's walk

of him (20 miles), he computes the "cost per mile" of staying there, dividing its cost by the amount of progress he would make by staying there; given this list of costs, he then chooses to spend his next night at the hotel with the best cost per mile. Demonstrate for Bilbo that being greedy can be costly, that is, describe an example where Bilbo's algorithm gives suboptimal performance.

3. (4 points) Find a dynamic programming algorithm for Bilbo's problem. Make it clear to Bilbo why it works, including an explanation of the meaning of any tables you ask Bilbo to construct. (Your solution for this part should look like an explanation, not a proof.)

Problem 2

(10 points) You and your group of friends spend all your time hanging out in your dorm room unless there is a computer science event happening, in which you attend the event, and then immediately return to your dorm room. However, there may be overlapping events, in which case you aim to have at least one of you attend each event. Of course, you must arrive to computer science events on time, and it is rude to leave early.

Design a greedy algorithm so that you and your friends, between you, can attend every computer science event, if this is possible at all. As above, points on this problem will only be given for the *proof* that your algorithm is optimal; more points will be given for simpler and clearer proofs. Remember, if you are having trouble proving the correctness of your algorithm, consider a different algorithm; if your proof seems unwieldy and awkward, consider a different proof approach—your first idea might be correct, but might not be the best.

Problem 3

Because so many computational processes are limited by memory or transfer costs, one of the key computational tasks is *information compression*, where data is *encoded* into a more compact form. In some sense, compression is a way of spending processing resources to save memory or bandwidth resources. Compression algorithms are usually categorized as either *lossless* or *lossy*, where lossless algorithms let one exactly recover the original data (for example "zipping" a file), and therefore are used invisibly in many different hardware and software components (some hard drives, network protocols, document file formats, operating system services); lossy algorithms on the other hand can compress to much smaller sizes because they *lose* information (examples include most image, audio, and video compression schemes); there is no generic lossy analog of "zipping" a file, because which parts of the information one is willing to lose entirely depend on the meaning of the information.

In this problem, we will derive an *optimal* lossless compression scheme known as *Huffman coding.* (Of course, "optimal" is not entirely the same thing as "practical".)

1. (2 points) Look up and write down the ASCII encoding of "Hobbit", interpreted in binary. How many bits-per-letter have you used? (There are a few different notions of "ASCII"; we do not care which one you use.)

2. (3 points) Suppose you have an *alphabet* Σ of n symbols that we wish to encode in binary—the result of such an encoding is called a "codeword". "Alphabet" is a technical term, more general than just letters: Σ could consist of, for example, 26 lowercase letters, or all the 256

ASCII symbols, or all English words plus punctuation, or all possible three-tuples of letters, etc.

If you want to represent each element of Σ in binary, how many bits-per-symbol is this encoding, as a function of n? (Assume all codewords must be of the same length.)

3. (5 points) Suppose, for whatever reasons, we now want to consider *variable-length codewords*. We now have the problem of ambiguity: if "01" is a codeword, then "0101" cannot be a codeword, because it could be confused with two copies of "01". Such codes are called "prefix-free" because no codeword is allowed to be a prefix (initial segment) of another codeword. We will represent a prefix-free code as a binary tree with each leaf labeled by a symbol in Σ, and where decoding a binary string happens as follows: start at the top of the tree and read bits, where each 0 means you descend to the left child and each 1 means you descend to the right child; the algorithm stops when it hits a leaf and returns the symbol there. Prove that every binary tree whose leaves contain exactly one copy of each symbol in Σ corresponds to a prefix-free binary encoding of Σ.

4. (2 points) For an alphabet Σ with n symbols, what is the smallest depth binary tree we could construct containing the symbols Σ at the leaves? (This is the same as "what is the prefix-free encoding whose worst-case encoding length is best?")

5. (3 points) Under the ASCII encoding, changing a single bit will end up changing only one letter of the decoded output. Find a prefix-free encoding and a message of length at least 5 where changing a single bit in the encoded message will change *all* the decoded symbols. (This is a general phenomenon: compressed information is more "brittle"; perhaps you have seen compressed video where several seconds have the wrong color and strange ghost-like artifacts—this might have been caused by the corruption of a single bit.)

6. (3 points) So far, we have not seen why variable-length encoding is helpful. The crux is that clever encoding is only helpful when there is some pattern in the data to be exploited. Here we work under the following assumption: there is a *probability distribution* p over the alphabet Σ, where each symbol $\sigma \in \Sigma$ occurs with probability $p(\sigma)$. Namely, to construct a message $\{x_1, \ldots, x_k\}$ of length k, each of x_1 through x_k is drawn independently at random from p. This models the phenomenon that the letter "e" occurs significantly more often in English than the letter "x". (Of course, however, letters in an English sentence are of course not distributed *independently*; there is significant correlation or anti-correlation.) We aim to design a code so that the most common symbols have the shortest codewords, and the least common symbols have the longest.

   Letting $\ell(\sigma)$ denote the length of the encoding of a symbol σ under a certain prefix-free encoding (which equals its depth in the corresponding binary tree), what is the *expected* (that is, the "average") number of bits-per-symbol that this encoding uses, assuming that the symbols are chosen according to the probability distribution $p(\sigma)$?

7. (6 points) Show that a length function ℓ defines a valid prefix-free code **if and only if**

$$\sum_{\sigma \in \Sigma} 2^{-\ell(\sigma)} \leq 1.$$

   Thus the problem of finding the best prefix-free code consists exactly of finding the function ℓ mapping the alphabet Σ to nonnegative integers such that it minimizes the expression you found in part 6 subject to this constraint. Fortunately, there is a better way to look at this challenge!

8. Huffman codes are constructed as binary trees via the following method. For each symbol σ, construct the trivial binary tree with just one node σ, labeled by the probability $p(\sigma)$. Make a list storing each of these objects (binary tree labeled by a probability). Until the list has just one element, repeat the following: find the two elements in the list with the smallest probabilities and "merge" them into one element by 1) replacing their probabilities with the sum of the probabilities, and 2) merging the two binary trees into one by creating a root node that has these two trees as children. The resulting tree represents the Huffman code for the pair (Σ, p).

   (a) (2 points) Suppose I have a lucky die that, instead of returning $\{1, 2, 3, 4, 5, 6\}$ with equal probability, returns them with probabilities $\{0.01, 0.03, 0.06, 0.3, 0.1, 0.5\}$. What is the Huffman encoding scheme? (As usual, feel free to include a diagram if it would help, and feel free to draw your diagram by hand.)

   (b) (2 points) Encode the lucky sequence 665666366466 using this code. How many bits-per-symbol does this use?

   (c) (1 point) Encode the unlucky sequence 1112153 using this code. How many bits-per-symbol does this use?

   (d) (2 points) Using the formula from part 6, what is the expected number of bits-per-symbol?

   (e) (2 points) The *entropy* of a distribution p is defined as $\sum_{\sigma \in \Sigma} p(\sigma)|\log p(\sigma)|$, where the base of the logarithm is 2 when we are working with *bits*. The entropy is a *lower bound* on the efficiency of any coding scheme. What is the entropy of p and how much short of this goal is Huffman coding? (Entropy is a deep concept, with a lot of material online about it. In our context, one way to think about the definition of entropy is the following: $|\log p(\sigma)|$ can be thought of as "the number of bits we could hope to store σ with"—see part 4; the entropy is the average value of this, in fact, the expected value of $|\log p(\sigma)|$, over the distribution p.)

9. (4 points) How would you describe Huffman encoding as a "greedy algorithm"? (Your explanation should relate the choices made by the greedy algorithm to the goals of the greedy algorithm, and make clear why these choices are a natural way to seek to achieve the goals.)

10. (15 points) As promised, Huffman coding is *optimal*, in the following sense: it has the least expected bits-per-symbol (see part 6) of any prefix-free encoding scheme. That is, this particular greedy way of assembling the binary tree is at least as good as any other. Prove it! (Follow the steps below.)

    (a) Show that there is an optimal encoding scheme where the two least frequent symbols are siblings and at the deepest level of the tree.

    (b) Consider a tree T which has, as above, the two least frequent symbols as siblings at the deepest level of the tree (though do not assume here that T is optimal!). Consider *merging* these two symbols by replacing them with a new symbol (not from the alphabet) having probability the total probability of the two original symbols, and modifying T to make a tree T' by removing these two siblings and labeling their parent by the new symbol. What is the *difference* between the expected bits-per-symbol of T versus T'?

(c) Argue that the previous two parts imply that, if T' is an optimal encoding of the merged alphabet, then T is an optimal encoding of the original alphabet.

(d) Using the previous part, analyze the Huffman encoding algorithm (backwards!) to show that it yields an optimal encoding.

11. (6 points) Now that we know Huffman coding is optimal, we would like to know how good it is, how the expected bits-per-symbol compares with the entropy lower bound of $\sum_{\sigma\in\Sigma} p(\sigma)|\log p(\sigma)|$. This is hard to analyze for Huffman coding, but since Huffman coding is the optimal scheme, we can instead analyze *any* other prefix-free coding scheme, and know that Huffman coding will perform at least as well. Wikipedia suggests `http://en.wikipedia.org/wiki/Shannon-Fano_coding`, which is a different attempt at a greedy algorithm for this problem. (Shannon-Fano coding is *not* a successful greedy algorithm; it often produces codes that are suboptimal; the fact that Claude Shannon, the founder of information theory, is credited with this algorithm is a sign of how hard designing correct greedy algorithms can be). The second sentence of this article is the crucial point: for *each* symbol σ, its depth in the Shannon-Fano coding tree is at most 1 more than $|\log p(\sigma)|$; thus the average length of codewords is at most 1 more than the (weighted) average of this, which is the entropy $\sum_{\sigma\in\Sigma} p(\sigma)|\log p(\sigma)|$. This implies that Shannon-Fano encoding has expected bits-per-symbol within 1 bit of the entropy bound. And Huffman encoding is at least as good, proving that Huffman encoding guarantees expected bits-per-symbol at most $1+\sum_{\sigma\in\Sigma} p(\sigma)|\log p(\sigma)|$, which is exactly the kind of bound we wanted.

    Unfortunately, Wikipedia is wrong! (Or, at least, inconsistent.) Show that the second sentence of the Wikipedia article is *not* true for Shannon-Fano encoding, as described in the article. Namely, find an alphabet Σ and a probability distribution p over the alphabet that provides a counterexample.

12. (12 points) To make the Wikipedia article true, we need to modify the Shannon-Fano algorithm. Specifically, replace step 3 of the algorithm with "Divide the list into two parts, with the total frequency counts of the left part being *greater than*, but as close to the total of the right part as possible." Prove the claim, which can be stated equivalently as: if a symbol σ ends up at depth d then its probability must be at most $2/2^d$.

    Crucial hint: for any internal node v (not a leaf), define the function $f(v)$ to be the total of all the probabilities of all descendants of v *except* the descendant with smallest probability. Show that if p is a parent of v then $f(p) \geq 2f(v)$, and finish the argument by induction.

13. We now know that the expected number of bits-per-symbol for the optimum prefix-free encoding is between the entropy $\sum_{\sigma\in\Sigma} p(\sigma)|\log p(\sigma)|$ and the entropy plus 1. But the gap is still a bit disturbing. In fact, Shannon's source coding theorem provides the answer: one can encode with bits-per-symbol *arbitrarily* close to the entropy if instead of encoding each symbol separately, we are allowed to encode in larger blocks.

    In some sense the worst case for Huffman encoding is with a heavily biased coin: consider the alphabet $\{a, b\}$ where the probability of a is 0.9 and the probability of b is 0.1.

    (a) (1 point) What is the entropy of this distribution?

    (b) (1 point) What is the expected bits-per-symbol for Huffman encoding here?

    (c) (3 points) Instead, try considering longer blocks of a's and b's at a time: redefine the alphabet to consist of all *triples* of a's and b's, namely an alphabet of size $2^3 = 8$. Write down the probabilities of each of these 8 possibilities (using the probabilities for a and b from above), and derive the Huffman encoding for these "supersymbols". Compare the performance here to the previous case and to the entropy bound.

Comments: extending the approach of the last part is infeasible because the size of the Huffman encoding tree will grow exponentially. For the particular case of the last part, *run length encoding* is an easy and effective scheme (which is used successfully in JPEGs, among many other places). More generally, variants of the *Lempel-Ziv* algorithm are very popular in practice, which aggregate "supersymbols" on the fly in a way that naturally adapts to the data being compressed. Look these up on Wikipedia for more information.