Homework 5

Due: Oct. 23, 2018 at 6:00 PM (early)
Oct. 26, 2018 at 6:00 PM (on time)
Oct 28, 2018 at 6:00 PM (late)

This is a partner homework as usual: work on all problems together; come to all office hours together; you are responsible for everything you and your partner submit and it is an academic code violation to submit something that is not yours. Check for your partner in the Google doc that was sent out. **If you don't have a partner, please let us know.**

1-credit track: do problems 1, 4.3, 4.6 (difficult; skim 4.5 first for context), 4.7, and 5 *and indicate "***1-credit***" on each problem.*

Greedy algorithms

Recall the generic strategy for proving the correctness of a greedy algorithm:

1. Our algorithm makes a series of choices, $c_1, c_2, ..., c_k$.
2. We will show by induction on i from 0 to k that there exists an optimal solution OPT_i that contains our first i choices.
3. To show the induction step from i to $i+1$, we need to find OPT_{i+1} that contains *all* of the first $i+1$ choices. There are two cases: a) if OPT_i already contains our choice c_{i+1}, then we can simply use OPT_i as OPT_{i+1}. Otherwise, b) we show (this is the meat of the proof!) that there is a way to "swap" choice c_{i+1} into OPT_i without sacrificing optimality—showing that this is now an optimal solution, OPT_{i+1}, that uses all our first $i+1$ choices $c_1, \ldots, c_{i+1}$, and proving our induction hypothesis.
4. Thus at the end of the induction we have shown that there is an optimum solution that makes *all* the same choices $c_1, \ldots, c_k$ that our greedy algorithm did, showing that our greedy choices are optimal.

Codr would have worked perfectly, but we matched coders so well that they fell in love and forgot about coding. As a result, the company decided to pivot and focus on matchmaking, to create more love in the world. Cue: Trainder. Trainder is a new app that helps single commuters make the most of their time on transportation, by sending them on dates (think Tinder for Trains).

Problem 1

Help Jessica find the one!

1. (12 points) Trainder is a new app that helps single commuters maximize their time on transportation by going on dates (think Tinder for Trains). Jessica is an avid user of Trainder and has already recieved a number of invitations to go on dates during her cross-country trip from California to NYC. Unfortunately some of the invitations may be at overlapping times. Jessica wants to attend as many dates as possible (without being rude and arriving late or leaving early). If the i^{th} date goes from time s_i to time t_i, evaluate each of the three

scheduling strategies below, and either *prove* it is correct or find a *counterexample* to show that it is not optimal.

(a) Greedily pick dates in increasing order of start time. (Sort dates in increasing order of start time s_i i.e. earliest to latest, and repeatedly run the following process: pick the first date in the list, then delete all overlapping dates).

(b) Greedily pick dates in decreasing order of start time. (Sort dates in decreasing order of start time s_i i.e. latest to earliest and repeatedly run the following process: pick the first date in the list, and delete all overlapping dates).

(c) Greedily pick dates in increasing order of length $(t_i - s_i)$.

2. (8 points) Jack would rather work on his new start-up idea, Tinder for Pets, than go on dates. His mother however is much keener for him to go dating. Knowing he is currently on a train to California, his mother plans and invites other single people to a party on his train using Trainder. To appease his mother, Jack agrees to meet all the people she invited, but wants to minimize the time spent at the party. He knows that the ith guest at this party will arrive at time s_i and will leave at time t_i . He wants to make as few appearances as possible—each time leaving immediately—but wants to ensure that everyone at the party sees him at least once in case they report back to his mom. Design a greedy algorithm for Jack to figure out what times he should appear at the party, and prove its correctness.

Problem 2

Trainder 2.0 is now live

Trainder recently decided to pivot and expand its product to not only trains, but all forms of transportation. Their new mission is to help users find love on any type of commute. If you put in your planned route, Trainder will now find you dates along the way. Logan is planning to drive from Minneapolis to San Francisco and has several matches for her planned route. Though she's not really interested in dating, her Tesla can travel a maximum distance of 200 miles before she must recharge, so she figures she can recharge during dates. She knows the location of each date beforehand so all she has to do is pick which dates she wants to go on to recharge her car (you may assume each date location has a Tesla charging station).

1. (8 points) Find a greedy algorithm for Logan to compute how to finish her trip in the fewest stops possible. Points on this problem will only be given for the *proof* that your algorithm is optimal. (Note that the "swapping" step of the standard greedy proof approach is a bit unwieldy here, so you may want to prove the optimality of your algorithm more directly: what does your strategy maximize at each step? Prove this by induction.)

2. (3 points) After studying your algorithm for the previous problem, Logan realizes that each date will cost a different amount of money (each invitation has a specific activity for the date, and Logan likes to pay for herself), and what she actually wants to do is minimize the total cost of her trip. Fortunately, each invitation already comes with an estimate of the cost, including the cost of charging her Tesla at that spot. She thinks of the following greedy algorithm: after each stop, for future dates within 200 miles' drive of her current location, she computes the "cost per mile" of going to a date there, dividing its cost by the amount of progress she would make by stopping there; given this list of costs, she then chooses the dates

with the best cost per mile. Demonstrate to Logan that being greedy can be costly, that is, describe an example where her algorithm gives suboptimal performance.

3. (4 points) Find a dynamic programming algorithm for Logan's cost-saving problem. Include an explanation of the meaning of any tables you ask her to construct, and how to fill them out. Prove the correctness of your algorithm (briefly—you should already have practice at getting to the heart of dynamic programming proofs of correctness in few words.)

Problem 3

Disrupting the encoding industry

(5 points) As opposed to Huffman encoding, here is a proposal for a slightly different algorithm to arrange a tree to encode a probability distribution (review the problems in the next section, or Dasgupta et al. to remind yourself of the framework). Construct a "very imbalanced" tree by repeating the following process: choose the domain element $p(i)$ of smallest probability that hasn't been chosen so far, and join it to the tree T constructed so far by making $p(i)$ and T the two children of a new node. This will lead to a tree where the ith–most–frequent element will end up at depth i (though the two least frequent elements end up at the same depth).

As precisely as possible, describe why the following sketch of the proof of correctness of Huffman encoding breaks down for this modified (and bad!) algorithm. This involves going over the original proof until you understand how it works.

> We show by induction on i that there is an optimal coding tree that contains as a subtree the tree merged from the i least frequent elements as specified by our algorithm. To prove the i^{th} step of the induction, assume by the induction hypothesis that there is an optimal coding tree OPT_{i-1} that contains as a subtree T the tree formed by the algorithm consisting of the $i-1$ least frequent elements; and assume for the sake of contradiction that the i^{th} least frequent element, which we denote $p(i)$, is *not* a sibling of T in OPT_{i-1}.
>
> We will show that we can modify OPT_{i-1} to have $p(i)$ and T appear as siblings without making the *expected bits per symbol* (expected depth in the tree of an element chosen from our probability distribution) of our encoding scheme worse. Move $p(i)$ and T in OPT_{i-1} by swapping them with two siblings at the lowest level of the tree. This can only improve the *expected bits per symbol* because $p(i)$ and T together contain all the i^{th} smallest elements from the probability distribution, and moving them down by swapping them with larger elements can only help. Thus there *is* an optimal tree containing "$\widehat{p(i)\ T}$", violating our assumption, and letting us conclude that our algorithm indeed finds an optimal tree.

Information compression

Problem 4

I can handle the business side, but you need to handle information compression

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 examine the lossless compression scheme known as *Huffman coding*.

As covered in class, we will be encoding symbols from an *alphabet* Σ, from which symbols are drawn according to a probability distribution p, where $p(i)$ denotes the probability of drawing the ith symbol. We represent an encoding scheme for the alphabet Σ by a binary tree, where each leaf is labeled by a distinct element of the alphabet, and the symbol σ at a given leaf is represented in the encoding by a sequence of 0's and 1's describing the path from the root to the leaf, where 0 means "go left from the parent" and 1 means "go right from the parent". (See chapter 5.2 of Dasgupta et al. for details.)

1. (2 points) Under the ASCII encoding of text, changing a single bit will end up changing only one letter of the decoded output. Find a binary encoding tree 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.)

2. (4 points) Given a probability distribution p each of whose probabilities is a (negative) power of 2 (and which sum to 1 since p is a probability distribution), show that there exists a binary encoding tree where each node i is at depth $|\log_2 p(i)|$.

3. 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) (1 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) What is the expected number of bits–per–symbol of the Huffman code, for sequences produced by rolling your die? (Recall the general expression for expected bits–per–symbol of a code represented as a binary tree.)

(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 the universal measure of information, and is our benchmark for whether a compression algorithm is successful.

4. (2 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.)

5. (4 points) We have seen in class that Huffman coding is optimal; but now 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.

   Hints: 1) Since you are trying to get a symbol p for which its depth d is greater than $1 + |\log_2 p(\sigma)|$, it seems easiest to aim as close to the threshold as possible—aim for, say, $d = 1.01 + |\log_2 p(\sigma)|$, which, since d is an integer means that p should be very close to an inverse power of 2 (check: should p be a tiny bit *bigger* or *smaller* than an inverse power of 2 for this equation to be true?); pick the simplest $p(i)$ and d like this that might work out, and aim to somehow arrange for an element of probability $p(i)$ to end up at depth d in the tree. 2) Check the problem below to see a *correction* to Wikipedia's Shannon-Fano encoding, where, since the below fix is correct, your counterexample should be treated rather differently by the two different variants of Shannon-Fano; perhaps look for a probability distribution where the two algorithms diverge as early as possible. 3) You might need to try a few variations; there is some "hacking" involved here; talk to a TA if you are worried you are not looking in a productive direction.

6. (10 points) To make the Wikipedia article true, we need to modify the Shannon-Fano algorithm. Specifically, replace step 3 of the algorithm with "Let s be the smallest element of the list, and T be the total. Divide the list into two parts at the *first* point which makes the

total of the left part greater than $(T-s)/2$." (Remember that the list is sorted in decreasing order, in step 2.)

Prove that this correction makes the second sentence of the Wikipedia article true. This can be stated equivalently as: if a symbol σ ends up at depth d then its probability must be at most $2/2^d$.

We strongly suggest your solution take the following approach: 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 v is a child of p then $f(v) \leq \frac{1}{2} f(p)$, namely, "each child has f value at most half its parent's," which implies that nodes deep in the tree have exponentially small f values. Then argue that if a symbol σ ends up at depth d in the tree then its probability must be at most $2/2^d$, by showing how to bound its probability in terms of the f value of its parent (which you already know is small by the previous sentence).

For reference, we have copied the steps of the algorithm from the Wikipedia article below:

1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.
2. Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the left and the least common at the right.
3. Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible.
4. The left part of the list is assigned the binary digit 0, and the right part is assigned the digit 1. This means that the codes for the symbols in the first part will all start with 0, and the codes in the second part will all start with 1.
5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

7. 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.

Problem 5

An All-In-One Compression Solution in the Cloud, for Your BIG Data

This problem presents several scenarios where you must design a good scheme to encode data. (**Note:** you do not need to describe a decoding algorithm for your coding scheme, but please think of how to decode each scheme before you start writing it up, to ensure that it makes sense and is unambiguously decodable.)

1. (5 points) You want to encode integers between 1 and $2^{64} - 1$ ("64 bit integers"), where you want to encode smaller numbers with fewer bits. Can you think of a scheme to encode a sequence of such integers so that when encoding a number i, if $i < 2^k$ for some integer k, then you use at most $k + 5$ bits? (Hint: try $k + 6$ first.)

2. (5 points) Consider the probability distribution $P(A) = 0.99$, $P(B) = .001$, $P(C) = .001$, $P(D) = .005$, $P(E) = .002$, $P(F) = .001$. Construct an explicit (simple) scheme for compressing long sequences drawn from this distribution, that is within 10% of optimal: compute the bits per symbol of your scheme, and compare with the entropy of P. (Hint: divide the sequence into sections of length 100 first.)

3. (5 points) You want to compress a sequence of 1,000,000 numbers each between 1 and 10,000, with the additional guarantee that: consecutive numbers in the sequence differ by no more than 15. How could you achieve within 10% of the optimal compression? (Do not try to compute the entropy of a random sequence like this, but you should be able to estimate it, by ignoring the bounds 1 and 10,000, and design a scheme accordingly.)

4. (3 points) The Onion is trying to compress its archives, and notices that each of its articles can be divided into sections that are either sarcastic, or sad; these two types of sections use rather different words. Let P be the probability distribution of words in a sarcastic section, and let Q be the distribution of words in a sad section. Suppose both P and Q have entropy 8 bits per word. Suppose you want to compress an article with 500 words, and with 4 sarcastic and 4 sad sections. Describe a compression scheme for such articles, evaluate its bits per symbol, and argue why it is close to optimal.

5. (3 points) A sequence of 10,000 people are asked which is their favorite Clickhole Quiz. Their answers are stored in a file, with each line having either the form:

   Person 1,592's favorite quiz is "Do you Deserve Hair?"

   or the form

   Person 9,787 thinks quizzes are no fun at all.

   Half of the respondents like the quizzes, and half do not. The lines are sorted by person number (1 through 10,000). Find a compression scheme for this file. You may assume that

each of the 243 Clickhole quizzes are equally popular. Describe your compression scheme, the total number of bits it will take, and explain why it should be close to optimal.

6. (4 points) You are given a list of addition problems "$a+b=c$" where a, b, and c are all integers between 0 and 1,000, except half the addition problems are wrong, for example "800+789=2". Describe a compression scheme for this list, keeping in mind that correct addition problems should take somewhat less space to store than incorrect problems. Evaluate the average number of bits-per-problem for your scheme and explain (briefly, intuitively) why it is close to optimal. (Do not try to compute the entropy of a random addition problem, but your explanation should make clear why correct addition problems should be treated differently from wrong ones.)