Problem 1 (10 Points). Prove the following two properties of the Huffman encoding scheme:

(a) If some character occurs with frequency more than 2/5, then there is guaranteed to be a codeword of length 1.

(b) If all characters occur with frequency less than 1/3, then there if guaranteed to be no codeword of length 1.

Solution:

(a) Suppose some character, say A, occurs with frequency more than 2/5, but that A does not have a codeword of length 1. Consider what the alphabet looks like right before A is merged with another letter, say B (that is, after we've performed possibly many merges, we have a modified alphabet, and now we are merging A with some other character). This means A and B have the lowest frequencies in the alphabet, and there is some other character C that has a higher frequency. But then $f_C > f_A > 2/5$, so $f_B < 1/5$.

Now, if we assume towards contradiction that no codewords have length 1, C must have been the result of a merge of two characters, say D and E that had minimum frequencies. Since C had frequency over 2/5, at least 1 of D and E, say D, has frequency over 1/5. But then $f_D > 1/5 > f_B$, meaning that D and E did not have the minimum frequencies since B has smaller frequency that D.

(b) Suppose all frequencies are less than 1/3, and there is some character A that has a codeword of length 1. Since all characters have frequency below 1/3, there must be more than 3 characters. Consider what happens when we've reduced the alphabet to just three characters. Since A has a codeword of length 1, it must have not merged, so it is one of the characters. The next step must merge the other two characters B and C. But since $f_A < 1/3$, $f_B + f_C > 2/3$, so one of B or C, say B, must have frequency greater than 1/3. This means $f_A < 1/3 < f_B$, so B and C could not have been the minimal-frequency characters.

Problem 2 (5 Points). Under a Huffman encoding of n symbols with frequencies $f_1, f_2, ..., f_n$, what is the longest a codeword could possible be? Give an example set of frequencies that would produce this case.

Solution: Say for $i < n$, $f_i = 1/2^i$, and $f_n = 1/2^{n-1}$. Then in each step of the algorithm, we will merge the last two characters. The result will be that for $i < n$, character i has a codeword consisting of $i-1$ 1s followed by a zero, while character n consists of $n-1$ 1s. Therefore, the longest codeword will have length $n-1$.

Problem 3 (15 Points). Show that for any integer n that is a power of 2, there is an instance of the set cover problem with the following properties:

i. There are n elements in the base set B.

ii. The optimal cover uses just 2 sets.

iii. The greedy algorithm picks at least $\log_2 n$ sets.

Solution: Suppose $n = 2^{k+1}$. Let B consist of pairs (X, Y) where X is between an integer in $[0, 2^k)$, and Y is 0 or 1.

Let S_1 be the pairs with $X \in [2^{k-1} - 1, 2^k)$. Let S_2 be the pairs with $X \in [2^{k-2} - 1, 2^{k-1} - 1)$. In general, S_i is the pairs with $X \in [2^{k-i} - 1, 2^{k-i+1} - 1)$ for $i = 2, ..., k-1$. Let T_b be the sets with $Y = b$.

We have that $|S_1| = 2^k + 2$, $|S_i| = 2^{k+1-i}$ for $i = 2, ..., k-1$, and $|T_b| = 2^k$. Further, the S_i are all distinct, and the union of S_1 through S_ℓ is the set of pairs where $X \in [2^{k-i} - 1, 2^k)$. Thus, if we've picked S_1 through S_ℓ, each Y_b contains $2^{k-i} - 1$ of the remaining elements, which is less than the number of elements in $S_{\ell+1}$. Therefore, the greedy algorithm will pick the elements S_1 through S_{k-1} in order. What is left are two elements, each belonging to a different Y_b, so the greedy algorithm will choose them next. In all, the greedy algorithm will choose $k + 1 = \log n$ sets.

However, the optimal solution simply uses Y_0 and Y_1, giving a total of two sets.

Problem 4 (15 Points). Solve the following recurrences, giving the tightest O bound possible:

(a) $T(n) = T(n-1) + n$

(b) $T(n) = T(\sqrt{n}) + 1$

(c) $T(n) = T(n-1) + 2^n$

Solution:

(a) $T(n) = T(0) + \sum_{i=1}^{n} i = T(0) + \frac{n(n+1)}{2} = \Theta(n^2)$

(b) Let $m = \log n$, and $T'(m) = T(2^m)$. Then

$$T'(m) = T(2^m) = T(2^{m/2}) + 1 = T'(m/2) + 1$$

We already know that T' is solved by $T'(m) \in \Theta(\log m)$. Then $T(n) = T'(\log n) = \Theta(\log \log n)$

(c) $T(n) = T(0) + \sum_{i=1}^{n} 2^n$, and we know geometric sums where the base is larger than 1 are just big theta of the last element, or $T(n) \in \Theta(2^n)$

Problem 5 (10 Points). Solve the following recurrences using the master method, giving the tightest O bound possible:

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 10T(n/3) + n^2$

Solution:

(a) $a = 2$, $b = 3$, $d = 0$, and $a > b^d$, so $T(n) = O(n^{\log_3 2})$.

(b) $a = 5$, $b = 4$, $d = 1$, and $a > b^d$, so $T(n) = O(n^{\log_4 5})$.

(c) $a = 7$, $b = 7$, $d = 1$, and $a = b^d$, so $T(n) = O(n \log n)$.

(d) $a = 9$, $b = 3$, $d = 2$, and $a = b^d$, so $T(n) = O(n^2 \log n)$.

(e) $a = 10$,$b = 3$, $d = 2$, and $a > b^d$, so $T(n) = O(n^{\log_3 10})$

Problem 6 (10 Points).

(a) Prove that it cannot be asymptotically faster to square an integer than it is to multiply two integers. That is, show that if we can square an n digit integer in time $O(n^d)$, that we can also multiply two n digit integers in time $O(n^d)$.

(b) Prove that it cannot be asymptotically faster to square a matrix than it is to multiply two matrices.

Solution:

(a) Suppose we can square two integers in time $O(n^d)$. To multiply two integers x and y, we add x and y, and square the result. We are left with $(x + y)^2 = x^2 + 2xy + y^2$. Then we just compute x^2 and y^2, and subtract the results, and finally divide by two (which is just a bit shift if x and y are binary numbers. So we have 3 squaring operations of $O(n^d)$, and some additions and subtractions, taking time $O(n)$. Thus the total running time is $O(n^d)$ (notice $d \geq 1$ since squaring must take at least linear time).

(b) Suppose we want to multiply 2 n by n matrices A and B. We construct a new matrix C containing A and B as blocks:

$$C = \begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}$$

Using the matrix multiplication formula, squaring C gives

$$C \times C = \begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}$$

Thus, we can just read off the upper n by n block of the solution.

To construct C takes $O(n^2)$ times, and squaring C takes $O((2n)^d) = O(n^d)$ time. $d \geq 2$ since we must at least read the entire matrix to square it. Thus, we can multiply in time $O(n^d)$.

Problem 7 (5 Points). You are given a list (in either array form or linked list form) of n elements from some ordered domain, and you notice that some of the elements are duplicates; that is, they appear more than once in the list. Show how to remove all duplicates from the list in time $O(n \log n)$.

Solution: We will modify merge sort. Basically, when we merge two lists, we repeatedly look at the head of the lists. If the heads are equal, we delete one of them. Otherwise, we add the smaller one to the end of the merged list.

We prove that if the two lists we are merging are sorted and have no duplicated, then at any step of the merge, the merged list will be in order, have no duplicated, and have all of its elements strictly smaller than all of the elements left in either of the two lists we are merging. This is true up front, and when we are going to add an element to the merged list, it is the smallest element of the list it belongs to, and is strictly smaller than the head of the other list, and so smaller than all of the elements in that list. Thus, adding it does not violate this property.

Conversely, all elements are present in the final list, since we only delete an element when we see a duplicate. Therefore, this algorithm is correct, and satisfies the same recurrence as Merge Sort, so its running time is $O(n \log n)$.

Problem 8 (10 Points). You are given two sorted arrays of with m and n elements, respectively. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

Solution: Let X be the first list, and Y be the second. Let's define an algorithm `Find`$(k, [a, b], [c, d])$, which finds the kth smallest element in the union of the elements of X with index $i \in [a, b]$, and the elements of Y with index $i \in [c, d]$. We will implement `Find` using a divide-and-conquer approach.

Let s be the midpoint of the interval $[a, b]$, and t be the midpoint of the interval $[c, d]$. Let x_s and y_t be the elements corresponding to these indexes in the two lists. We have several cases:

- $x_s < y_t$ and $k \leq s - a$. Since x_s is greater than $s - a$ elements of X, any element in Y greater than y_t is greater than $s - a \geq k$ elements of the union. Thus, we can ignore the upper half of the elements of Y, so we call `Find`$(k, [a, b], [c, t])$
- $x_s < y_t$ and $k > s - a$. Now, any element in X smaller than x_s will be smaller than $s - a < k$ elements, so we can safely ignore the lower half of X. We've removed $s - a$ elements below s in X, so the recursive call looks like `Find`$(k - (s - a), [s + 1, b], [c, d])$
- $x_s > y_t$ and $k \leq t - c$. Similar to the first case, we can make the call `Find`$(k, [a, s], [c, d])$.
- $x_s > y_t$ and $k > t - c$. Similar to the second case, we can make the call `Find`$(k - (t - c), [a, b], [t + 1, d])$.

As a base case, if one of the ranges has no values, we just output the kth value in the other list.

Now, to find the kth smallest element, simply call `Find`$(k, [0, m - 1], [0, n - 1])$.

For running time, each recursive call halves the size of the range we are looking at. We can only halve X $\log m$ times and Y $logn$ times, so we make $O(\log m + \log n)$ calls before we terminate. Each call consists of one a couple comparisons and arithmetic operations, so the running time is $O(\log m + \log n)$.

Problem 9 (20 Points). A list A of n elements $a_i, ..., a_n$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $a > b$?". However, you *can* answer questions of the form: "is $a = b$?" in constant time.

(a) Show how to solve this problem in $O(n \log n)$ times. (Hint: Split the list into two lists A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (Hint: Here's another divide-and-conquer approach:

- Pair up the elements of A arbitrarily, to get $n/2$ pairs
- Look at each par: if the two elements are different, discard both of them; if they are the same, keep just one of the,

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if A does.)

Solution:

(a) If an list has a majority element x, then if we split the list into two collections S and T each of $n/2$ elements, one must have more than $n/4$ xs: Otherwise, if both have at most $n/4$ xs, then there are at most $n/2$ xs overall, so x cannot be majority. This gives us the following algorithm:

- Split the list arbitrarily into two lists S and T, and recursively compute the majority elements x_S and x_T of these lists, if they exist.
- If x_S and x_T exist and $x_S = x_T$, then between the two lists there must be at least $2(n/4 + 2) > n/2 + 1$ elements equal to x_S, so output x_S.
- If x_S exists and x_T does not, then x_S is the only possible majority element. Just go through the whole input list, counting the number of x_Ss, and return x_S if there are more than $n/2$ of them. Otherwise, report that there is no majority element.
- Similarly, if x_T exists, and x_S does not, count the number of x_Ts, and output x_T if there are more than $n/2$ of them.
- If x_S and x_T both exist, but $x_S \neq x_T$, scan the whole list, counting the number of x_Ts and the number of x_Ss, and report the one that has more than $n/2$.
- Lastly, if neither exist, there is no majority element, so report so.

There are two recursive calls of size $n/2$, plus $O(n)$ outside work (scanning through the list at most twice), so $T(n) = 2T(n/2) + O(n)$, which is solved with $T(n) = O(n \log n)$.

(b) Following the hint, pair up elements of A arbitrarily, getting $\lfloor n/2 \rfloor$ pairs. For each pair, if the elements are different, discard both, if they are the same keep just one of them. We are left with a list L, and we recursively compute the majority element of L. If n is odd, one element will not be paired up: keep it as well.

First, we keep at most element from each pair, plus the last element, so $|L| \leq \lceil n/2 \rceil$.

Now, define a *weak majority element* as an element that occurs at least $n/2$ times (as apposed to strictly more than $n/2$ times). There are clearly at most 2 weak majority elements.

Suppose out input has a weak majority element x, meaning there are at least $n/2$ xs. Consider the following process: first throw out all the pairs where the elements don't match. Afterward, each of the remaining elements are paired up in matching pairs, so throw out one element of each remaining pair. This is clearly equivalent to the method presented.

Whenever we throw out a pair, we throw out at most one x and at least one element not equal to x. Thus, x remains a weak majority element after just throwing out mismatched pairs. Now we throw away exactly half of the occurrences of each element (assuming n is even), so x is still a majority element.

The only corner case is when n is odd, and the unpaired element is not x. Let n' be the number of elements left after throwing out the mismatched pairs, and let k be the number of pairs of xs (so $2k \geq (n'+1)/2$). After throwing away one element from each pair, we are left with $(n'+1)/2$ elements. Thus, x is a weak majority element of L.

This gives the following algorithm for finding weak majority elements:

- Construct L
- Find at most 2 elements x and y that are weak majority elements of L. If our input has any weak majority inputs, they will be in the set $\{x, y\}$.
- Check if x or y are weak majority elements of the input (by scanning the whole list). If so, output the ones that are.
- Otherwise, report no weak majority elements.

Now, any majority element is also a weak majority element, and we have an algorithm that outputs weak majority elements. Thus, we just need to test if either of the two possible weak majority elements are in fact majority elements.

Since L has size at most $\lceil n/2 \rceil$, we get $T(n) = T(n/2) + O(n)$, which is solved by $T(n) = O(n)$.

Total points: 100