CIS 121

Fei Liu

Homework 05

Recitation 201

Collaborators:

Q1.

(a) The longest codeword possible is of length $n-1$. A tree created with the Huffman algorithm is always full, so every node must have two children or be a leaf. However, we want to create the tree so that the codeword for the symbol with the lowest frequency should have the greatest depth possible in the tree. To do this, one of the children of each node must be a leaf, so that the tree can reach the greatest depth possible.

For the Huffman algorithm to create a tree of this fashion, there must be certain restrictions on the frequencies of set S.

For $n = 1$, we only have one possibility for encoding that symbol with the Huffman algorithm, and it will be of length one.

For $n = 2$, we also only have one possibility for encoding that symbol with the Huffman algorithm, and it will be a root node, that has two leaves. In this case, we see that the longest encoding of a symbol is also just 1.

For $n = 3$, there's also only one possibility for the tree created from the Huffman algorithm. One child of the root node will be a leaf that represents the symbol with the greatest frequency, while the other child will be a sub-tree that splits into two more leaves, those leaves representing the remaining two symbols.

For an an arbitrary number of symbols $n > 3$, run through the for loop in the Huffman algorithm once. Now, we have one "merged" node. Let's run another iteration of for loop and see what happens. The two symbols with the smallest frequencies will be selected. Notice that the frequency of the "merged" symbol must be amongst the two smallest. If it is not, a completely new node will be created, with its own two leaves. This breaks the tree invariant that I described in the first paragraph for a tree that will give you the longest possible codeword: One of the children of each node must be a leaf. Thus, the condition that we must impose upon these three frequencies is that the frequency of the "merged" symbol cannot be the greatest of all three.

To generalize the condition from the previous paragraph, in a min-heap of $n > 3$ nodes, the "merged" node must be amongst the two smallest frequencies so that it can be used in every iteration of the for loop in the algorithm (I guess another way to say this is there can only be at most one merged-node in the min-heap at all times).

Here's how an example would play out for a set of symbol frequencies of arbitrary size n. After sorting the frequencies from greatest to smallest, W.L.O.G., $f_1 > f_2 > f_3 > ... > f_n$. Our frequencies would exhibit this condition: $f_n + f_{n-1} < f_{n-3}$. After one iteration of the for loop in Huffman's algorithm, the remaining number of frequencies decrease by one, and we impose this condition on a new set of f_n, f_{n-1}, f_{n-3} until we reach $n = 3$. At this point, the frequency of the "merged" node can not be the greatest amongst the three remaining nodes, and after this condition is met, we will be able to create a tree that will have the longest encoding possible for one of the symbols.

(b) For a set of characters of size $n = 1, 2, 3$, it is obvious that for the resulting tree created by the Huffman algorithm, the root node of that tree will always have a child that is a leaf or is the leaf itself (for $n = 1$). That leaf child IS the codeword of length 1.

For a set of characters of size $n \geq 4$, let's denote the character with the frequency greater than 3/7 f_G, and W.L.O.G., sort out the frequencies of the characters such that $f_1 > f_2 > ... > f_n$.

First, I will prove that f_G has to be either f_1 or f_2. If $f_3 > 3/7$, then f_1 and f_2 must be greater than f_3, but then that would mean that $f_1 + f_2 + f_3 > 1$, which is not possible. This same argument holds for

$f_4 > 3/7, f_5 > 3/7, ..., f_n > 3/7$. Now, that I've proven f_3 to f_n can't be f_G, now I have to prove that f_2 can be f_G. If $f_2 > 3/7$, then f_1 can be a value that is infinitesimally greater than f_2, and the remaining frequencies will all be very very small as $f_1 + f_2$ is already very close to one. Notice that if f_2 is to be f_G, then $f_2 < 1/2$, because if $f_2 \geq 1/2$, then $f_1 + f_2 \geq 1$. As for f_1 being f_G, that's pretty obvious.

Now let's run the for loop in the Huffman algorithm until the number of nodes in our min-heap is 3. We now split into two cases: $f_2 = f_G$, or $f_1 = f_G$.

If f_2 is f_G, then the three remaining nodes are f_1, f_2, and the "merged" node. It is very intuitive that f_2 has not been merged in some previous iteration, because I have stated before that if f_2 were to be f_G, then $f_3, ..., f_n$ are all very very small (less than 1/7 to be exact), and thus f_2 will never be amongst the two smallest in any previous iteration of the for loop. In this case, we know that the frequency of the "merged node" is less than both f_1 and f_2, and thus when we apply the for loop one more time, the min-heap will consist of f_1 and the "merged node". Running the for loop one last time, the symbol with the frequency f_1 will be the symbol with a codeword of length 1. We have proved that in this case, there is guaranteed to be a codeword of length one.

If f_1 is f_G, then the min-heap will consist of f_1, and two other nodes (it doesn't matter whether these nodes represent a "merged" frequency or a single frequency). Let's denote these other two nodes f_2 and f_3. First, I prove that f_1 has not merged with another node in a previous iteration of the for loop, and this is easy because if f_1 was merged with another node at some previous iteration, then there has to be another frequency that is greater than f_1, but this is impossible because f_1 IS the greatest frequency in our set. Now, let me assume towards a contradiction that f_1 is not the greatest frequency out of f_1, f_2, f_3. This assumption tells me that either f_2 or f_3 must be greater than f_1 (otherwise f_1 is the greatest frequency), but not both (because the sum of the frequencies would go over 1). Note that whichever of f_2 or f_3 that is greater than f_1 has to be a "merged" node because I've already established that f_1 is the greatest frequency. W.L.O.G., let's say $f_2 > f_1$. This means that $f_2 > 3/7$, and thus $f_3 < 1/7$. f_2 must be a "merged node", let's denote its children as f_{2a}, f_{2b}. $max(f_{2a}, f_{2b}) > 3/14$. Notice, that 3/14 is greater than $f_3 = 1/7$. Here, we have reached a contradiction because in some previous iteration of the for loop, the smaller of f_{2a}, f_{2b} would have been merged with f_3 (notice that it doesn't matter whether f_3 is a "merged node" or a single frequency). With a contradiction, I have proved that f_1 is indeed the largest frequency of f_1, f_2, f_3, and on the next iteration of the for loop f_2 and f_3 will be merged, then, on the last iteration, f_1 will be merged and the symbol with the frequency f_1 will have a codeword of length 1.

I have proved both cases for $n \geq 4$ end up guaranteeing a codeword of length one, and together with the cases for $n = 1, 2, 3$, I have proved that if there exists some character that has a frequency greater than 3/7, then it is guaranteed that there is a codeword of length 1.

Q2. My algorithm will first sort book 1 into position 1, then book 2 into position 2, and so on, until book n is sorted into position n. Let's call the input array A. For my explanation, A[1] is the first element in the array. Here's what my algorithm does:

1. Initialize a boolean variable called isOutOfBounds that checks if $k + 1 > n$.

2. If isOutOfBounds is false, then build a min-heap (H) of size k, using elements of the input array from the second position to the $(k+1)$th position, else build a min-heap of size $k - 1$ using elements of the input array from the second position to the nth position.

3. Call extractMin(H), store this value in a variable called min.

4. Initialize another variable called temp.

5. Compare A[1] to min. If A[1] is bigger, set temp equal to A[1], set A[1] = min.

6. Pass the min-heap (H), the input array (A), the integer k, the variable temp, and an index counter (i) starting at 2 into a helper function. Also, return whatever this helper function returns.

My helper function takes in 5 arguments: the min-heap, the input array, an integer variable (k), the variable temp, and an index counter. Here's what one iteration of my helper function will do:

1. If i is equal to A.length, A[i] = temp, and then return A.
2. Initialize a boolean variable called isOutOfBounds that checks if $k + i >$A.length.
3. If isOutOfBounds is false, then insert the $(k + i)$th element into H. The insert function from lecture notes on 09/29 also uses swim for a max-heap, but flipping a sign in the conditional of the while loop will make it work for a min-heap.
4. Initialize a variable called min that stores whatever extractMin(H) returns.
5. Check if temp is smaller than min. If it is, then A[i] = temp, and temp = min. If not, then A[i] = min.
6. Do a recursive call on this helper function with the five arguments: the min-heap (H), the input array (A), the integer k, the variable temp, and the index counter $i++$.

Now I will do a run-time analysis on my algorithm. From the lecture notes on 09/29, we have proven that for a heap of size a, building a min or max heap takes O($\frac{a}{2}lg(a)$), extractMin O($lg(a)$) takes, and insert takes O($lg(a)$) time. In the original function, steps 1, 4, and 5 all take constant time. Step 2 builds a min-heap of size k, so it takes O($\frac{a}{2}lg(k)$) time, and step 3 uses extractMin, so it takes O($lg(k)$) time. Step 6 takes however long the recursive helper function will take, let's denote this T_H. My algorithm will take O($\frac{k}{2}lg(k)$) + O($lg(k)$) + c + T_H time to complete. To simplify, c and $lg(k)$ are both O($\frac{k}{2}lg(k)$), so it becomes O($\frac{k}{2}lg(k)$) + T_H.
The helper function will go through $(n - 1)$ iterations, but on the last iteration, it will only take a constant time c. Steps 1, 2, 5, and 6, take constant time to perform. Step 3 calls on insert, which takes O($lg(k)$), and step 4 calls on extractMin, which takes O($lg(k)$). Thus, one iteration of my helper function takes O($lg(k)$) + c time, which simplifies to O($lg(k)$). With $n-2$ iterations, the total running time of my helper function is $(n-2)$O($lg(k)$), which is O($(n-2)lg(k)$).

With T_H =O($(n - 2)lg(k)$), the total running time of my algorithm is equal to O($(n - 2)lg(k)$) + O($\frac{k}{2}lg(k)$). If $k = n$, then for $n \geq 4$, $n - 2 > k/2$, and for $k < n$, then for $n \geq 3$, $n - 2 > k/2$, so $\frac{k}{2}lg(k) < (n - 2)lg(k)$ for all $n \geq 4$, so $\frac{k}{2}lg(k)$ is O($(n - 2)lg(k)$), and the running time of my algorithm simplifies to O($(n - 2)lg(k)$). This running time is smaller than the limit imposed in the question for all $n > 2$, so my algorithm's running time is O($n * lg(k)$).