Intro to Algo HW #4

Saturday, March 7, 2020 3:04 PM

Problem 5.15

a)

a	2/3
b	1/6
c	1/6

From the root A is to the left, and B/C are to the right. Thus A could have a frequency of 2/3 and B and C are equally likely to happen with this encoding. So they each have 1/6.

b)

a	
b	
c	

This encoding is not possible. A cannot be 0 and then lead to C at 00 because each letter must be a leaf (end node).

c)

a	
b	
c	

This encoding is not optimal. This Huffman encoding does not form the full binary tree.

d)

a	2/3
b	1/12
c	1/24
d	1/24
e	1/6

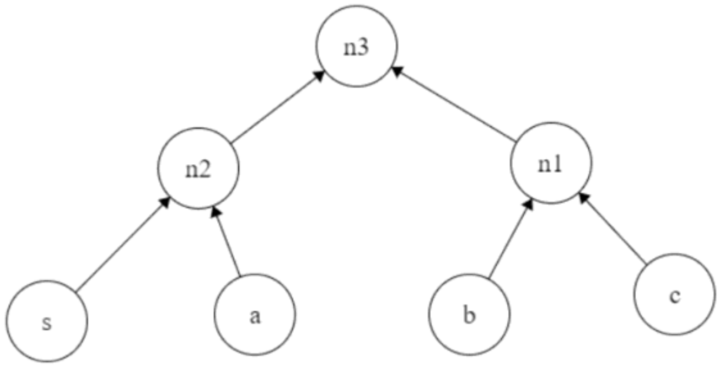
Problem 5.16

a)

Proof By Contradiction:

Assume that there is a tree where the node s has a frequency of $>2/5$. Without a length of 1, then the node s cannot be a direct child of the root node.

Huffman trees are full trees so we show a tree like this:



The frequency of node *n1* should be greater than that of s otherwise the merging would not work properly. Thus $p(n1) + p(s) > 4/5$. The sum of the root and it's immediate children should be 1.

$p(n2) + p(n1) = 1 \ = p(s) + p(a) + p(n1)$.
Thus $p(a) < 1 - \left(p(s) + p(n1)\right) = \frac{1}{5}$

$p(n1) > 2/5$ thus either b or c should be $> 1/5$. This is a contradiction because the tree would then be incorrect. That smaller node should have been merged with a first.

Thus the original claim is true.

b)

Proof By Contradiction:

Assume that there is a tree where some node a is length 1 and has a frequency of 1/3. Then we have two leaf nodes b, c that are length 2 from the root.

Because $p(a) + p(b) + p(c) = 1$, and $p(a) < 1/3$, either b or c must be greater than 1/3. This is a contradiction. Not only must all the nodes have frequency $< 1/3$ but this would break the Huffman structure.

Thus the original statement is true.

Problem 2.17

The algorithm is as follows:

Input the array.
Check the middle index, if the value at the middle index is greater than middle index then repeat with the sub array $A[1: middle\_index - 1]$.

If the value at the middle index is less than middle index then repeat with the sub array $A[middle\_index + 1: len(A) - 1]$

If the value at the middle index is equal to middle index then return true.

Analysis: We are dividing the array by half every time and are only doing a constant lookup if each iteration. Therefore we are only doing $\log n$ operations thus the algorithm is $O(\log n)$

Problem 2.19

a)

We show the runtime of merging. A merge between 2 arrays takes O(2n). With 3 it takes O(3n). Thus if we have to repeatedly merge them with k number of arrays we have
$O(2n + 3n + 4n + 5n \ldots + kn)$.
$O(n(2 + 3 + 4 + 5 + 6 + \cdots + k))$

We see that is it $O(n \, * \sum_{i=0}^{k} i)$ which in big O-notation is $O(k^2 * n)$

b)

The algorithm is as follows. A brief description is we use k as it's own array, and each array of size n is an index in the array k

We group the arrays by merging them with their neighbor. So if we have arrays a, b, c, d, e, f then we want to end up with ab, cd, ef.
We repeat this until we have one completely sorted array.

Analysis: We are essentially running a merge sort on k elements. This already makes it $O(k \log k)$. But each element is a size n array. Because we are subdividing the arrays each time, we end up with some constant times n which is still $O(n)$. Thus the algorithm is $O(nk \log k)$

Problem 2.23

a)

If A has a majority element then at least 1 of the sub arrays should have that as a majority element.
If both sub arrays have the same majority element then that element is the majority element of the main array.

The algorithm is as follows:

Split the array into 2 equal size sub-arrays.
First we find majority elements by splitting and searching. We split each array in half and if an element meets the requirement for majority element, we return up the recursion.
If there is a majority element count the number of repetitions in the array.
If both arrays have the same majority element then return that element. Otherwise if the number of majority elements in one of the arrays is greater than half the size of the main array return that element.
Otherwise there are no majority elements.

Analysis: Searching and splitting is a $O(\log n)$ operation, as we cut the array in half everytime. Then verifying if a majority element is valid for the main array is a $O(n)$ operation. Because we do that for each of the split arrays the total runtime is $O(n \log n)$

b)

The algorithm is as follows.

Input some array A.
Randomly assign pairs within the array.
If $|A|$ is odd then have some tiebreaker set equal to the last element in the list.
Delete the pairs that don't have matching elements.
Repeat this recursively with the pairs array.
Return prospective majority elements
Calculate if they are valid majority elements. Use tiebreaker if necessary
If it is then return that element.

Analysis: This works because if A has a majority elements, then upon random matching, it will be the last element standing. This is shown by the pigeon hole principle, if it is in the majority then it exists in at least one pair that doesn't contain itself. Thus eliminating the other non-major elements.

This algorithm is $O(n)$ because you must go through the array and assign pairs and check frequency. The other main part is $O(\log n)$ but because this is big O-notation then the main runtime is $O(n)$.