HW4

Xinhao Luo

Tuesday 12^{th} May, 2020

1 DPV Problem 5.15

a) Possible frequency example: $f_a = 0.5$, $f_b = 0.25$, $f_c = 0.25$

b) Not possible: only leaf nodes should be used to encode

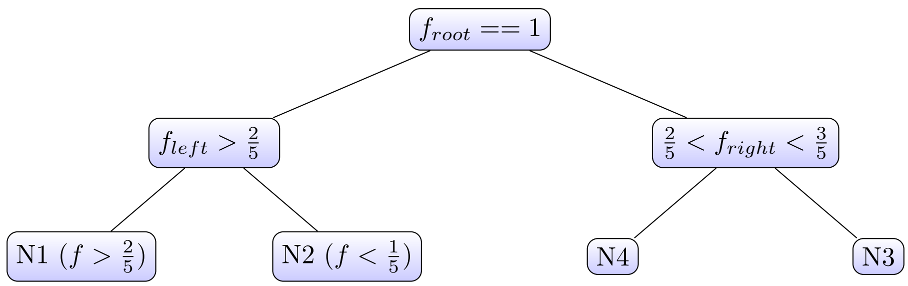
c) Not Possible: It cannot form a full binary tree

d) Code: {0, 100, 1010, 1011, 11}
Possible frequency example: $f_a = 0.5$, $f_b = 0.25$, $f_c = 0.125$, $f_d = 0.0625$, $f_e = 0.0625$

2 DPV Problem 5.16

a) Prove by contradiction

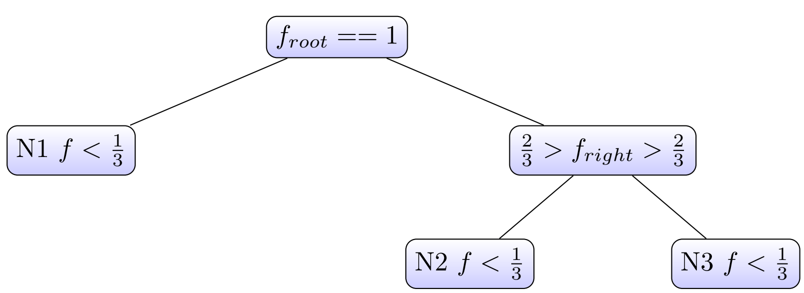
- Assume there is no codeword of length 1
- If there is a no codeword of length 1, the minimum tree we have should like this:



- Assume N1 is the node with more than $\frac{2}{5}$ frequency, then its parent would also larger than $\frac{2}{5}$ as it is the sum of its children.
- On the other half of the tree, we may have in total $\frac{2}{5} < f < \frac{3}{5}$ since we need to ensure $f_{root} == 1$
- However, if the other half of the tree has already larger than $\frac{2}{5}$, and its neighbour, at the same time, occupy at least $\frac{2}{5}$, N2 must be $f < (1 - \frac{2}{5} - \frac{2}{5} = \frac{1}{5})$
- At the same time, the biggest frequency N3, N4 can get is $\frac{3}{5} \times \frac{1}{2} = \frac{3}{10}$, which is smaller than N1
- The rules defined that each time, two smallest nodes will be combined. N2 should be bind with either N3, N4 rather than N1, which is fishy
- This graph does not exist and the assumption is false

b) Prove by contradiction

- Assume there is codeword of length 1
- If there is a no codeword of length 1, the minimum tree we have should like this:



- Assume N1 is the node with codelength 1, then the other half of the tree will be $f_{right} > \frac{2}{3}$
- However, since each leaf can only be $\frac{1}{3}$, we will also have $f_{right} < \frac{2}{3}$
- f_{right} does not exist, so the graph does not exist, which means the assumption is false

3 DPV Problem 2.17

General idea: Doing binary search over the array

Algorithm
Assume index start from 1

Step 1 Take the length of A as l

Step 2 Take $half = l \ // \ 2$

Step 3 Compare $half$ and $A[half]$

- If $half > A[half]$, start from step 1 with $A = A[1 : (half - 1)]$
- If $half < A[half]$, start from step 1 with $A = A[(half + 1) : l]$
- If $half == A[half]$, return $half$

Runtime

Using master theorem, we have: $T(n) = 1 * T(n/2) + O(1)$

- The merge part does nothing, so we have $O(1)$, $1 = n^0$, so $d = 0$
- Each time, we have split the array into half and discard one of them, so we have $b = 2$
- In each iteration, we only start over once, so $a = 1$
- $a = 1$, $b = 2$, $log_b a = log_2 1 = 0$
- $d = 0 = log_b a$, so we have $O(n) = n^d logn = n^0 logn = logn$

4 DPV Problem 2.19

a)

$$\begin{aligned} T(k,n) &= O(n,n) + O(2n,n) + ... + O((k-1)n,n) \\ &= O(2n + 3n + ... + kn) \\ &= O(n \times \sum_{i=2}^{k} i) \\ &= O(n\frac{(2+k)(k-1)}{2}) \\ &= O(k^2 n) \end{aligned} \tag{1}$$

b) **Algorithm** Define $merge(a1, a2)$ that takes two sorted array and return one sorted array. The whole process should take $O(2n) = O(n)$

```
def sort(arrays):
    if len(arrays) < 2:
        return arrays[0]
    elif len(arrays) < 3:
        return merge(array[0], array[1])
    else:
        size = (len(arrays) - 1) // 2
        return merge(sort(arrays[0:size]), sort(arrays[size:]))
```

Runtime Using master theorem, we have:
$T(kn) = 2 * T(k/2) + O(kn)$
$T(k) = O(1)$ when $n = 1$

- The merging process will take $O(kn^1)$ as it will need to go through each element once. $d = 1$
- Each time, the method will recur twice, $a = 2$
- Each time, the length of the array is cut by half, $b = 2$
- $log_b a = log_2 2 = 1 = d$,
- $T = O(knlogn)$

5 DPV Problem 2.23

a) **Algorithm** Takes two a, b as parameters

   (a) Recursively split the array into half and look for each sublist's majority, and merge two sublist's result
   (b) If two sublist has the same majority, then return the majority
   (c) If two sublist has different majority, go through the merged array and find these two majority's frequency and compare
   (d) If one sublist has a majority while the other don't have, count if this element is the majority after merged
   (e) If two sublist both don't have the majority, no majority appears

   Runtime

   (a) Based on master's theorem, we may have $T(n) = 2 \times T(n/2) + O(n)$
   (b) each time, we called ourselves twice, cut the list by half, and each time we need to count the elements on necessary
   (c) $a = 2, b = 2, d = 1$, $log_b a = log_2 2 = 1 = d$
   (d) $O(n) = nlogn$

b) **Algorithm** Takes an array of elements

   Step 1 Split the array into pairs, we now have $n/2$ pairs
   Step 2 for each pairs, discard if the pairs are not the same, else keep one of the elements in the pair. Now we at most will have $n_{new} \leq n_{old}/2$ elements
   Step 3 Return to step 1 if more than 1 elements remains
   Step 4 If there is one elements left, this element will be the majority, otherwise no majority
   Step 5 If there is odd number of element appears, get its frequency after pairs, if it is the majority, return that element, or discard it

   Runtime

   (a) Based on master's theorem, we may have $T(n) = 1 \times T(n/2) + O(n)$
   (b) each time, we called ourselves once each time; cut the list by half, and each time we need to compare the result at the end
   (c) if there is odd number of element, after $O(n)$ pairs, then do an $O(n)$ search. $O(2n) = O(n)$
   (d) $a = 1, b = 2, d = 1$, $log_b a = log_1 2 = 0 < d$
   (e) $O(n) = n$