Problem 1 (DPV 5.15)

(a)

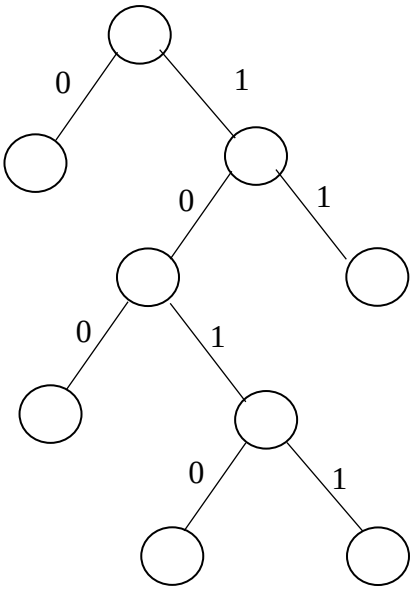
Code	Char	Frequency
0	a	3/5
10	b	1/5
11	c	1/5

b and c first get added to the tree because they have the lowest frequency, and 3/5 > 1/5+1/5. Then, a gets added with the parent node of b & c, and the node goes away.

(b) This one cannot work because one code is a prefix of another. 0 is a prefix of 00.

(c) This one also cannot work because the codes do not form a full binary tree. The 1 branch that has the code 10 as a child is missing a child of code 1 (11) to be a full branch.

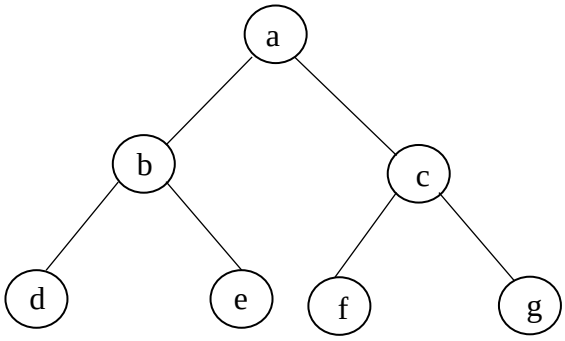
(d) Code: {0, 100, 1010, 1011, 11}



Code	Char	Frequency
0	a	3/5
100	b	1/10
1010	c	1/20
1011	d	1/20
11	e	1/5

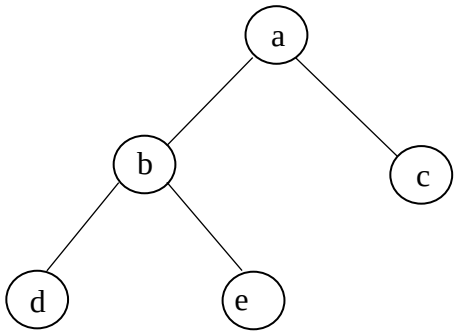
Problem 2 (DPV 5.16)

a) Proof by contradiction. Assume there were no codewords of length 1. Then, this means that the children of the root node must have two children nodes each.



Here, take d as the node with the highest frequency; p(d) > 2/5. The first step of encoding will be merging two nodes with the minimum frequencies after merged. For instance, take f, g and create a parent node c. Next, in order to have node b, p(c) + p(d) > p(d) + p(e) && p(c) + p(e) > p(d) + p(e) should be true. Or otherwise, node c would merge with d or e. Therefore, p(c) must be greater than 2/5. So, p(a) = 1 = p(b) + p(c) = p(d) + p(e) + p(c). And by our assumption, p(d) > 2/5 && p(c) > 2/5. So, p(d) + p(c) > 4/5. This gives us p(e) < 1/5.
Now, looking back at c, p(c) > 2/5, so either p(f) or p(g) should be <1/5. However, this yields a contradiction because whichever one is smaller out of f or g will merge with e instead of creating a parent node c, because that will give the minimum frequency. Therefore, if some character occurs with frequency more than 2/5, then there is guaranteed to be a codeword of length 1.

b)



Assume c is the code with length 1 and has a frequency less than 1/3 (p(c) < 1/3).
By looking at this example, we can see that p(d) + p(e) = p(b) > 2/3. So, either p(d) or p(e) must be greater than 1/3 (p(d) < 1/3 || p(e) < 1/3). However, if this was the case, then whichever one out of d and e that has smaller frequency would merge with c in the first place, since that will yield minimum frequency of the merged parent. Therefore, contradiction.

Problem 3 (DPV 2.17)

<pseudocode>

```
index_value ( a[1 ,,, n])
        if n == 1
                return (a[n] == n) #terminal condition of recursion
        i = n // 2
        if (a[i] == i)
                return true
        else if (a[i] > i)
                return index_value(a[1 ,,, (i-1)]
        else
                return index_value(a[(i+1] ,,, n])
```

First thing to note is that this algorithm only works because array a was sorted. By initially inspecting a[i] == i, we can determine whether the next recursion execution should look at the bottom half or the top half of the array. Since we are dividing and conquering by half for each iteration, the algorithm yields *logn* runtime.

Problem 4 (DPV 2.19)

a) k sorted arrays with n elements.

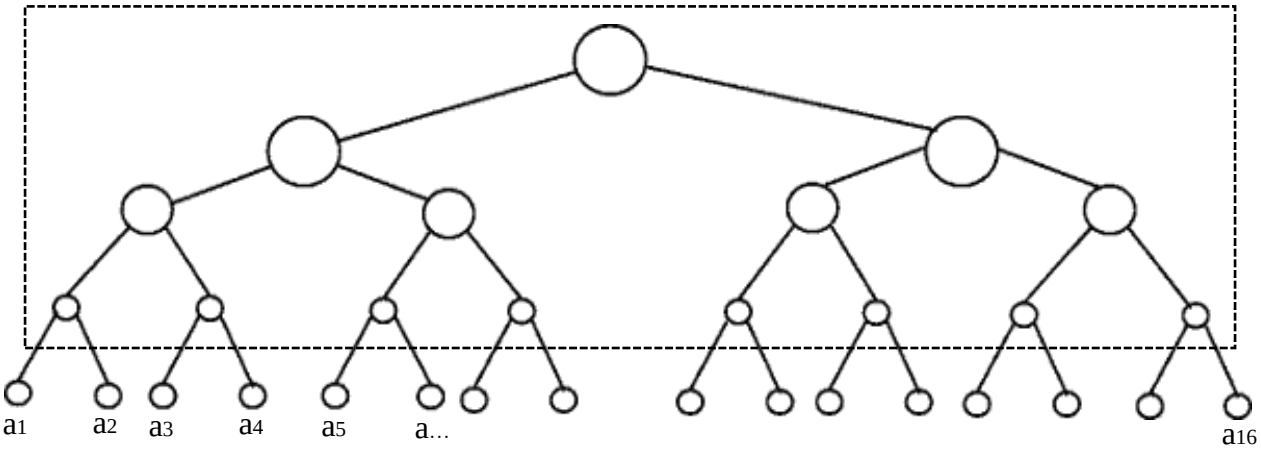
Initially, merging the first and the second array takes *2n* time, since we will have to go through both arrays in linear time. So, the computational time for merging 3 array takes *3n*. Therefore, the total time will be

$\sum_{i=2}^{k-1} i * n \;=\; n \sum_{i=1}^{k-2}(i+1)$ = 2n + 3n + … + kn = n * (2+3+…k) = n * $\frac{(k-1)*(k+2)}{2}$ = $O(k^2 n)$

b) k sorted arrays with n elements.

Say we have our arrays as labeled: a_1, a_2, … a_k.

Merge a_1,a_2, separately merge a_3,a_4, and so on.



The total number of merging operations is going to be k-1, or it is same as the number parent nodes in a full binary tree.

Here, parent nodes refer to such nodes that have child nodes. Say we have a full binary tree like above with n leaf nodes. The total number of nodes in such graph is 2n -1. Therefore, we have n-1 parents to create (or in other words, do that many operations).

Merging a_1 + a_2 takes 2n amount of time. Merging a_3 + a_4 takes 2n amount of time.

Merging (a_1 + a_2) + (a_3 + a_4) is done in 2* 2n = 4n amount of time for $\frac{1}{2}$ k times.

So, iterating the terms, we get

2n * k + 4n * $\frac{1}{2}$ k + 8n * $\frac{1}{4}$ k + …

= *2n* * (*k* * *log k)*

= *O* (*nk log k*)

or expressed in terms of recurrence relation (Dr.G's suggestion)

T(*k*) = 2 * T($\frac{1}{2}k$) + O(*nk*)

Problem 5 (DPV 2.23)

a) In order for A to have a majority element, both subarrays A_1 and A_2 should also have majority elements, which should be the same. Split the arrays into subarrays and repeat the process until we have arrays of size 1. The splitting can happen in a recursive fashion. The base case of the recurrence relation is n=1. By the Master Theorem, the recurrence relations is given by

$$T(n) = 2\,T(\frac{n}{2}) + O(n)$$

And the problem states that comparisons are guaranteed to happen in constant time. Furthermore, we now that T(n) = O(n logn), a typical complexity for recursion. So, the divide and conquer approach yields Logarithmic runtime.

b) Pair up the elements of A arbitrarily to get n/2 pairs. Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them.
If we had some 2k number of elements in an array (the number could be odd, regardless), then there should be at least k+1 number of the same element in order for that element to be the majority element. This means that no matter how we pair up the elements, there should be at least one pair which is a matching pair. Call this chosen element e. This takes n/2 comparisons.

Now, we check if e is the actual majority element of A. Take e and compare it with every element in A. Calculate how many times e occurs in A. If e appears more than array.size times, or in other words n/2 times, it means that e is the majority element. If not, then A does not have a majority element.