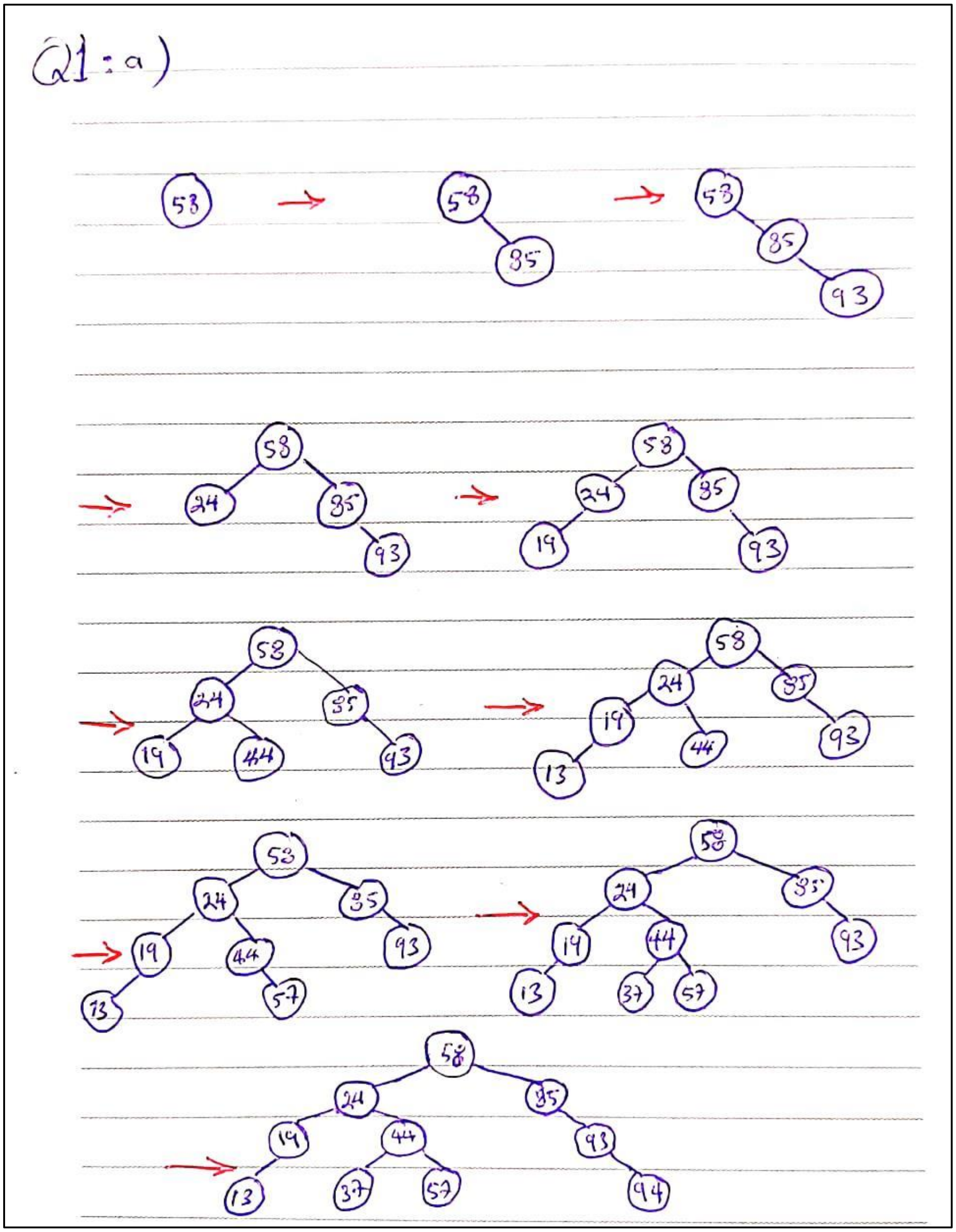
Mohammed S. Yaseen
21801331
Section 3
Assignment 2

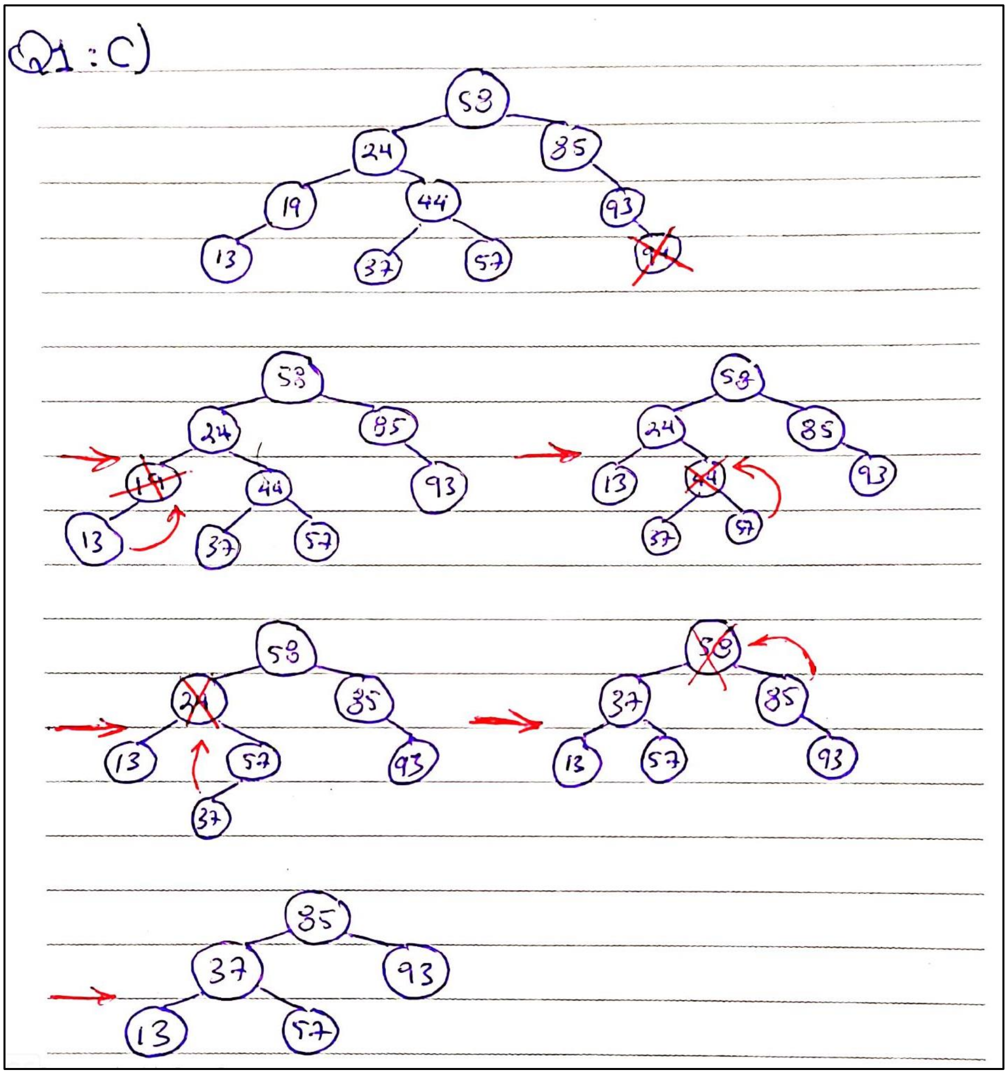
Q1: a)



Q1: b)

Preorder: 58, 24, 19, 13, 44, 37, 57, 85, 93, 94
Inorder: 13, 19, 24, 37, 44, 57, 58, 85, 93, 94
Postorder: 13, 19, 37, 57, 44, 24, 99, 93, 85, 58

Q1: c)



Q3)

Here, I am going to mention each function I wrote and firstly go over how I implemented it and the logic behind it and then discuss its time complexity.

- **CalculateEntropy function**

  Implementation: I first went through all the class counts stored in classCounts array and calculated the total number of observations. Then, one more time I go over the class counts, and on every iteration I divide the number of observation of the i'th class by the total number of observations to obtain the ratio. At the same time, I find the product of that ratio and its $\log_2$ and add this product to the entropy. Finally, I return the final value of the entropy.

  Time complexity: assuming we have N classes and we have two for loops each of them iterates through the N classes, the total number of instructions will be 2N; therefore, the time complexity is O(logN).

- **CalculateInformationGain function**

  Implementation: In this function the goal is to find the information gain of a feature and the information needed to do so are: the entropy of parent, the weighted entropy of both left and right children.

  Firstly, because the function needs to create an array of the classes' counts, we need to figure out the available classes, and that is handled by the private function findClasses, where it finds the classes and the number of classes. After, figuring out the number of classes we basically will execute the following pseudo code algorithm:

```
Create 3 2D-arrays of classCounts of the size of number of classes
found by findClasses function: parentClassCounts, leftClassCounts,
and rightClassCounts. The first dimension would hold the classes
themselves, and the second dimension would hold each class' counts.

// find the class counts
   For each usable sample
       For each of the classes (found by findClasses function)
           If the i'th sample's label = the k'th class
                Increment the k'th parentClassCounts
                If the featureId value of the sample == 0
                    Increment the k'th leftClassCounts
                    Increment samplesForLeft
                If the featureId value of the sample == 1
                    Increment the k'th rightClassCounts
                    Increment samplesForRight


// calculate the entropy for the parent, left, and right child
```

```
// using calculateEntropy function

// calculate the weighted entropy of both children as follows
H(s) = smplesForLeft/numSamples*LEntropy
       + samplesForRight/numSample*REntropy

// calculate the information gain
IG = H(p) - H(s)
Return the IG
```

Time complexity: In this function the major process is the double for loop, which will dominate the time complexity of the function. If we have M samples and N classes, we would have the outer loop iterates through M samples and the inner loop through N classes. Also everything inside both for loops are O(1) operations; therefore, the time complexity of this algorithm is O(MN).

- **FindClasses function**

Implementation: Given that the classes are integers, we firstly would find the first number that has not been used as a class label then initialize an array of a large size with this integer. Finally, go through all labels, and if the label doesn't already exist (check it using a private function called exists), add it.

```
// variables
Classes            // the classes found
numForInit         // the number that doesn't exist in labels
number of classes

// Find a number that doesn't exist in labels
    For each sample
       If the sample's label == numForInit
           numForInit++
           Repeat the loop

// Initialize classes with numForInit

// Go through all labels if a label doesn't exist already add it
// and increment number of classes
    For each sample
        If the sample's label doesn't exist in classes
            Add it to classes
            Increment number of classes
```

Time complexity: The function in general has two for loops. Assuming we have M samples and N different classes, the first for loop would iterate through M samples and would repeat

N times, in the worst case scenario that is samples numbering starts with 0 until N, but in this homework assignment the numbering starts with 1, so this first for loop would repeat only once. In terms of the second for loop, it iterates through M samples and would execute only once; however, it calls exists function at each iteration, and exists function has a time complexity of array size. Given that we give it N as array size its complexity would be O(N). As a result, the final time complexity of the second for loop is O(MN) an is the time complexity of the function given that it is the heaviest in the function.

- **exists function**

  Implementation: This function goes through all the items of an array and checks if the wanted item does exist or not in the array.

  Time complexity: assuming there are N items, and since we are looping through all of them the time complexity is O(N).

- **train function**

  Implementation: Firstly, using a function called findFeatureId, we find the featureId with the highest information gain for the root node of the decision tree. Then pass the root to a function called buildDecisionTree to continue building the tree.

  Time complexity: in this function the heaviest time complexity is the part of calling buildDecisionTree which has the time complexity of O($2^N MN^2$), and since it is being called only once, the final time complexity of this function is O($2^N MN^2$).

- **findFeatureId function**

  Implementation: This function goes through all features that weren't used before and finds the information gain of each of them then finds the one with the highest information gain and returns its index value, its pseudo code as follows:

```
// Local variables
Create two 2D-arrays temporaryHighestIG and highestIG, the first
dimension of both arrays would hold the index of the feature and
the second dimension would hold the information gain of that
feature.

// Base cases
    Return -1 if all samples are used
    Return -2 if all features are used
```

```
// Find the feature with highest entropy
    For each feature
        If this feature was not used before
            temporaryHighstIG[0] = this feature
            temporaryHighstIG[1] = its information gain (using the
                                 function calculateInformationGain)

            // If a feature with a higher IG is found update
               highestIG
            If information gain in temporaryHighstIG > highestIG
                Update the information gain of highestIG
                Update the feature id of highestIG

// Mark the feature found as used not to be used anymore

// return the index of feature with highest IG
    return feature id of highestIG
```

Time complexity: assuming we have M samples and N features, firstly we go through all samples and all features to check whether they are all used or not and this is MN complexity, after that there is a for loop that goes through all features and calls calculateInfomationGain function which has a complexity of O(MN), therefore the final complexity is the complexity of the for loop since it is the heaviest that is $O(MN^2)$.

- **buildDecisionTree function**

Implementation: This is a recursive function that traverses the tree in a preorder and each time it finds the best feature for left and right then continues with left till it hits a base case that is the node is a leaf node. The Base case is being a leaf node that is all samples that reached this node are of the same class. The recursive step is composed of two steps the first is to go left and the second is to go right. The pseudo code is as follows:

```
// Base case - Leaf Node -
    If all samples are of same class or the node is marked leaf
        // find the class
        For each usable sample
            // find number of repetition for each class
            For each class
                Store this class's repetition in k'th position in
                TheClassSoFar array

        // find the class with the highest repetition
        For each class
```

```
        // Update the class if a class with more repetition found
            If repetition of theClass < theClassSoFar's
            The theClass = theClassSoFar

        // fill out the leaf information
        root->theClass = theClass
        root->leaf = true;
        root->left = NULL;
        root->right = NULL;

        return

    // - NOT A LEAF -
    // update used samples to pass it to left (0 for left)
    For each item in usedSamples
        if sample is usable and it has 1 for parent feature
            mark it unusable for the left node
        else
            increment the number of samples going left

    If there is at least one sample going left
        // find the next feature
        featureId = findFeatureIDToSplit(...)
        root->left = new DecisionTreeNode();
        root->left->featureId = featureId;

        // featureId = -1 all samples used
        // featureId = -2 all features used
        mark it as leaf if featureId < 0

        // pass it to left
        buildDecisionTree(..., root->left);
     else (no samples going left)
        root->left = NULL // there is no left

    // Do the same as left for right sub-tree
```

Time complexity: assuming M samples, N features, and k different classes, firstly we calculate the time for one function call. Each function call will either perform the functionality for an inner node or a leaf node. For leaf node the complexity equation is as follows: $T(leaf) = 2k + M + MK + K$, and for an inner node as follows : $T(inner) = 2K + 2N + 2M + 2MN^2$. We also know that for the worst case scenario all tree nodes will be used that is the number of features will need to be used up before reaching a leaf node, which means half of the nodes will be leaf nodes and the other half will be inner nodes according to the relation: leaf nodes = $2^N - 1 - (2^{N-1} - 1) = \frac{2^N}{2}$, which leaves $\frac{2^N}{2} - 1$ as inner nodes. Therefore, our leaf equation would be executed $\frac{2^N}{2}$ times and hence $T(leaf) = \frac{2^N}{2}\ (2k + M + MK + K)$, and the inner nodes equation $T(inner) = (\frac{2^N}{2} - 1)\ (2K + 2N + 2M + 2MN^2)$. Given the -1 in the second equation is not making any big effect, it can be dropped, and since both equations are multiplied by the same factor, they can be combined in one equation $T = \frac{2^N}{2}\ (5k + 3M + MK + 2N + 2MN^2)$. From the final equation we can drop the $5k + 3M + MK + 2N$ since their complexity is much less than $2MN^2$; therefore we are left with $T = 2^N MN^2$, and we can conclude that the time complexity id $O(2^N MN^2)$

- ## predict function

  Implementation: in this function we have a pointer pointing to the root node of the tree, and then it start traversing through the tree until it hits a leaf node. To traverse it, each time, finds the value of the sample at the feature that is equal to the featureId in the current node, so it the value was 0 it goes to the left node, or the right node otherwise. Once it hits the leaf node, it returns its class value.

  Time complexity: since we are traversing the tree by level that is each time we are choosing one node out of a level, and since there are log(N) levels, given there are N features, the time complexity of this function is O(logN).

- ## test function

  Implementation: this function receives a list of observation and their labels, so it would go through all observations, generates a prediction for each observation, and checks it against the corresponding label. We also keep track of all correct predictions, the value predicted equal to the corresponding label. Finally, the function returns the quotient of the correct predictions over all observations.

  Time complexity: assuming there are M observations, and since the function loops through all of them and calls predict function, the time complexity would be O(MlogN)

- ## printTree function

  Implementation: this function would traverse the tree in preorder traversal, and at each time it prints the value of either the class or the feature id. To keep track of the number of tabs to print, I added a new parameter that is level, so at each recursive call for the children the level is incremented then passed. The base case is if the root is NULL, otherwise it would firstly print itself then precedes to left node then to right node recursively.

  Time complexity: assuming we have N features, the maximum number of nodes that we can have is $2^N - 1$ and since we are traversing all the nodes, the time complexity of this function is: $O(2^N)$.