Assignment-03-B403-cmtidmar

Clare Tidmarsh, cmtidmar@iu.edu

March 2021

1 Introduction

Assignment 3 for CSCI-B403 Spring 2021. First assignment two day late assignment policy was used for Assignment 3. I am aware that any assignment turned in is point onward is deducted for lateness for the future assignments.

2 Problem 1

A male player tosses two fair coins into the air. He wins 1 dollar for the number of heads he will get. However, he will lose 5 dollars if neither coin is a head. Calculate the expected value of this game and determine whether it is favorable for the player.

Using the examples from February 17 (217.pdf) as an example for this problem, let

$$x_i = \begin{cases} 2 & \text{if two head appears (HH)} \\ 1 & \text{if one head appears (HT)} \\ 1 & \text{if one head appears (TH)} \\ -5 & \text{if no head appears (TT)} \end{cases}$$

If the coins are truly fair, then the probability for each would be $1/4 = 0.25$ for each possibility. Assuming that this game is only played once,

$$=> x = x_1 + x_2$$

$$=> E(x) = E(x_1 + x_2) = E(x_1) + E(x_2)$$

Calculate $E(x_i)$,

$$E(x_i) = (2 * 1/4) + (1 * 1/4) + (1 * 1/4) + (-5 * 1/4)$$

$$= (1/2) + (1/4) + (1/4) + (-5/4)$$

$$= (1/2) + (2/4) + (-5/4)$$

$$= (1/2) + (1/2) + (-5/4)$$

$$= 1 - 5/4$$

$$= -1/4$$

Since $E(x_i) = -1/4$,

$$E(x) = 2 * E(x_i) = 2 * (-1/4) = -2/4 = -1/2$$

If you play this one time, it is favorable for the player since he will most likely win at least 1 dollar. There is only a 1/4 chance that he will have to pay 5 dollars.

3 Problem 2

A company makes electronic gadgets. One out of every 20 gadgets is faulty, but the company doesn't know which ones are fault until a buyer complains. Suppose the company makes a 3 dollar profit on the sale of any working gadget, but suffers a loss of 150 dollars for every faulty gadget because they have to repair the unit. Check whether the company can expect a profit in the long term.

Similarly to Problem (1), use the expected value formula for this problem, let

$$x_i = \begin{cases} 3 & \text{profit per unit} \\ -150 & \text{loss per unit} \end{cases}$$

$$=> x = x_1 + x_2 + ... + x_10$$

Using linearity of expectation,

$$=> E(x) = E(x_1 + x_2 + ... + x_20) = E(x_1) + E(x_2) + ... + E(x_20)$$

Since the company receives 3 dollars per every working gadget (19/20) and losses 150 dollars for every non-working gadget (1/20),

$$E(x_i) = 3 * (19/20) + (-150) * (1/20)$$

$$= (57/20) - (150/20)$$

$$= -93/20$$

Since $E(x_i) = -93/20 = -4.65$,

$$E(x) = 20 * E(x_i) = 20 * -4.65 = -93$$

The company will not expect a profit in the long term. From a batch of 20, where 19/20 are working, the company will only make 57 dollars, however, since one is gong to faulty, then they lose 150 dollars in that batch and the company will lose -4.65 dollars from each unit and lose 93 dollars for each batch of gadgets. Therefore, if the company is losing 93 dollars every batch, then the company will never see profit since the cost of repair is more significant than the profit for the working gadgets.

4 Problem 3

Describe and write an algorithm, and also explain its complexity, that decodes and returns the decoded form of a string encoded according to the following rule, k[encodedstring] where k is a positive integer and the encodedstring inside the square brackets is being repeated exactly k times. You may assume that the input encoded string is always validly encoded. No extra white spaces, and square brackets are well formed, etc.

An algorithm that can decode a string like the ones described here uses the data structure of a stack. Since we are assuming that the string is always validly encoded, we do not need to write cases for the possibility that it there are unbalanced brackets or no brackets at all. To start, initiate a counter to count the number of times you need to repeat the process, two new empty stacks one to hold the integers and the other to hold the characters, and an empty string to hold the final output. You will then need to look at the first character in the string and identify the integer value of that character position. Push() the integer into the stack that can hold integer elements. Next, use the peek() method to peek at the next character ensuring it is an open bracket. There is the possibility to skip this step since you already know that it will be an open bracket. Then, Iterate to the next position in the string and identify the character. You will then push() this character and place it into the empty stack that you created. Use peek() again to peek at the next character, if it is another character, then push() that character into the character stack. Repeat the peek() and push() methods until you peek at a closing bracket. If it is another integer, then push it to the integer stack and repeat the steps that you did before after pushing the integer.Once you get to a closing bracket, take the element in the integer stack and set that number to a variable. Using this variable, create a for-loop to iterate through the number of times that the variable value states. In the for-loop, during each iteration, pop() the elements in the character stack and place it in the empty output string you created. Use the counter to keep track of how many times you iterated through the for loop, ensuring that by the end, the counter equates to the value of the variable that you used to set the integer to. Coming out of the for-loop, peek() at the element in the next position of the for-loop. If it is an integer, then push that integer into the integer stack and repeat all of the steps that you did previously after pushing the first integer. If it is nothing, then the string is done and you should return the final output.

```
// java syntax
import java.util.Stack;

stringDecoder(String s){

// initialize
Int counter = 0;
Stack<Integer> intStack = new Stack<Integer >();
Stack<Character> charStack = new Stack<Character >();
String final = ""; // final output

for (int i = 0; i < s.length; i++){
    if (Character.isDigit(s.charAt(i)){
        intStack.push(s.charAt(i)); }
    else if (s.charAt(i) == '['){
    i++;} // moves to the next position

    else if (Character.isLetter(s.charAt(i)){
        charStack.push(s.charAt(i));
        charStack.peek();}

    else if (s.charAt(i) == ']'){
        count = intStack.pop(); // pops off the integer element
        for (int j = 0 ;j < count;j++){
            String character = charStack.pop();
            character.push(final.charAt(j))
            final = "";
            }
        }
    }
return final;

}
```

The time complexity should be $O(n^2)$ since both for-loops used in the program each has a complexity of O(n). The outer-loop iterates through the input. The inner-loop only has n number of iterations depending on the integer value and will repeatedly output the number of letters according to how many iterations it uses.

5 Problem 4

Given the roots of a Binary Search Tree and a target number k, describe and write an algorithm to return true if there exists two elements in the BST such that their sum is equal to the given target and also explain the complexity.

For this problem, let's assume that the BST is already set up and the Node class has already been defined. All we need to do is find two pairs from a tree that equals a given target number. Using the tree on the assignment pdf, create an empty array and start at the root/the top of the tree. Add the root to the array. Then Move to the first level of the tree. Sum the left node and the right node together. If the sum of these two nodes is the same as the target value, then return true. Else, add the two node values to the array. Then, look at the children of the left node. If there are at least two children, move to the next level and compare the left and right child of that parent node. If you sum them, and they equate to the target value, return true. If their sum is not equivalent to the target value, add them to the array and then traverse back up and over to the right side of the tree. Check to see if the right side has two children. If there are not, insert the single child into the array. If there are at least two children, sum the children. If their sum equates to the target value, return true. If not, insert both children into the list. Now you should have a filled array full of nodes. Compare each node which each other by summing them at each comparison. If two nodes equals the target value, then return true. If none of them sum to equivalent to the target value, return false.

```
// java syntax
sumOfNodes(Node node){


// initialize
Array<Integer> listOfNodes = new Array<Integer >();
sum = 0;
k = 9;
listOfNodes.append(root);

if (root.getLeft() != null && root.getRight != null){
    sum = root.getLeft() + root.getRight();
    if (sum == k){
        return true;
        } else {
            listOfNodes.append(root.getLeft());
            listOfNodes.append(root.getRight());}
    while (root.getLeft() != null){
        node1 = node.left;
        if (node1.getLeft() != null && node1.getRight() != null){
                sum = node1.getLeft() + node1.getRight()
                if (sum == k){
                    return true;}
        } else {
            listOfNodes.append(node1);
            if (node1.getLeft() != null || node1.getRight != null){
                node2 = node1.getLeft() || node1.getRight();
                listOfNodes.append(node2)}
                }
        }

        while (root.getRight() != null){
            node3 = node.right;
            if (node3.getLeft() != null && node3.getRight() != null){
                sum = node3.getLeft() + node3.getRight();
                if (sum == k){
                    return true;}
            } else {
                listOfNodes.append(node3);
                if (node3.getLeft() != null || node3.getRight() != null){
                    node4 = node3.getLeft() || node3.getRight();
                    listOfNodes.append(node4)}
                    }
        }
    for (int i = listOfNodes[0]; i < listOfNodes.length; i++){
        for (int j = listOfNodes[-1]; j < listOfNodes.length; j--){
            sum = listOfNodes[i] + listOfNodes[j];
            if (sum == k){
                return true;
            }else{
            return false;}
            }
        }
        } else {
            break;}
            }
        }
    }
}
```

The time complexity should be O(n) as it has to look at every node at each level to make the comparisons. Then to look at the list, O(n) + O(n) since it also looks at each element. The complete time complexity should be $O(n^3)$.