Problem 1

Suppose that you start with an empty BST (binary search tree), and then insert the following values (in this order): 7, -2, 10, 0, 13, 14, 1, 3. Draw the tree that results (to save time, you can draw just the tree, not all of the steps to get there).

Then, give the in-order, pre-order, and post-order traversals of this tree. Compare these to the original values you inserted - is there any similarity?

Finally, give rearrange the numbers in such a way that, if you started over and inserted them again, you would end up with exactly the same tree (even though you inserted in another order).
In-order traversal: -2, 0, 1, 3, 7, 10, 13, 14
Pre-order traversal: 7, -2, 0, 1, 3, 10, 13, 14
Post-order traversal: 3, 1, 0, -2, 14, 13, 10, 7

The pre-order traversal is similar to the original insertion order - for instance, 7 is inserted first (and becomes the root), and so it’s the first element in the pre-order. Likewise, throughout the tree, a parent is always inserted before the child (and it is also before the child in the pre-order). Of course, when two nodes are not directly related (like -2 and 10), the relative order of those two nodes doesn’t matter.

Here are two new sequences which build the same tree:
(the pre-order traversal): 7, -2, 0, 1, 3, 10, 13, 14
(the same, but swapping the left and right trees): 7, 10, 13, 14, -2, 0, 1, 3
Problem 2

Suppose that you have a simple BSTNode class, which has fields `left`, `right`, and `val`. (For simplicity, assume that they are public fields. In a real program, they would probably be private.)

Write a function, `print_pre_order(node)`, which will print out a pre-order traversal of all of the values in the tree (one value per line). **Use recursion!**

```python
def print_pre_order(root):

    # SOLUTION:
    if root is None:
        return

    print(root.val)
    print_pre_order(root.left)
    print_pre_order(root.right)
```
Problem 3

Repeat Problem 2, but this time, **DO NOT** use recursion. Instead, use a list to keep a stack of "TODO" items - that is, trees that you need to print out.

It's easy to get this wrong. After you've written it on paper, open up your computer and try out your code. Does it work? Debug it if not.

(My hope is that, after trying this, you'll decide that recursion was the easier way to get this done!)

**Instructor's Note:**
This problem was a lot more difficult than I anticipated. If you didn't get it, don't worry - neither did lots of other students.

The key in this algorithm is to use a list to emulate a call stack. Take a look at the solution from Problem 2 - when do we “leave” the current function, into a recursive call? It’s when we recurse into the left side. What do we need to later, when we return from the function? We need to recurse into the right side.

Look at how this works with our example tree from Problem 1. We start by calling the function on 7; we recurse into -2, but need to come back (later) and print the entire subtree rooted at 10. Likewise, we recurse to the left side again, from -2 into an empty subtree; we need to come back later, and print the entire subtree rooted at 0.

So, in the recursive version, this is what the stack looks like:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>print_pre_order(None)</td>
<td>Will return immediately</td>
</tr>
<tr>
<td>print_pre_order(-2)</td>
<td><strong>Remember:</strong> print out -2.right - which is 0 - when we come back</td>
</tr>
<tr>
<td>print_pre_order(7)</td>
<td><strong>Remember:</strong> print out 7.right - which is 10 - when we come back</td>
</tr>
<tr>
<td>main()</td>
<td></td>
</tr>
</tbody>
</table>

We use our list to keep track of the “Remember” notes on the right.
So, version 1 of the solution does **lots of leftward recursing**, with the stack representing the “things to get back to”:

```python
def print_pre_order(root):
    SOLUTION (v1):
    stack = [root]
    while len(stack) > 0:
        r = stack.pop()

        # iterate through the left children, saving rights to
        # the stack.
        while r is not None:
            print(r.val)
            stack.append(r.right)
            r = r.left
```

**Version 2 recognizes that the list represents a “list of subtrees to recurse into”** - meaning that we don’t actually need the nested loop inside!

```python
def print_pre_order(root):
    SOLUTION (v2):
    stack = [root]
    while len(stack) > 0:
        r = stack.pop()
        if r is None:
            continue
        print(r.val)
        stack.append(r.right)
        stack.append(r.left)
```
Problem 4

In this problem, you will recreate a BST, using only a pre-order traversal. The input to this function will be a list (or tuple) of values, representing the pre-order traversal of some old tree. Recreate the tree, and return it. (You may use the nested-tuples format, or you can actually create BSTNodes. Your choice.)

Use recursion. The key to this is to realize that, in a pre-order traversal, the first node in the list is the root of the subtree - and what follows is the left subtree, followed by the right subtree. And you can tell the place where the left subtree ends, and the right begins, because you know that this is a BST. (What do you know about the values in each subtree?)

Your function must handle the empty list (return an empty tree).

Bonus. Rewrite your function (if you didn’t do it this way the first time) to use list comprehensions.

```python
def build_bst_from_pre_order(elems):
    SOLUTION:
    if len(elems) == 0:
        return None

    val = elems[0]
    left = [x for x in elems if x < val]
    right = [x for x in elems if x > val]

    left = build_bst_from_pre_order(left)
    right = build_bst_from_pre_order(right)
    return BSTNode(left, val, right)
```