Please read the following guidelines carefully!

- Please write your name on the front and back of the exam.
- This examination has 4 questions. There are a total of 5 pages, DOUBLE-SIDED.
- You may always write helper functions/methods unless explicitly asked not to.
- Any question you leave blank or clearly cross out your work and write “I don’t know” is worth 10% of the marks.

Take a deep breath.

This is your chance to show us

How much you’ve learned.

We WANT to give you the credit

That you’ve earned.

A number does not define you.

Good luck!
1. Consider a function `count_item` which takes a nested list and an item, and returns how many times that item appears in the nested list. Note: this should return 0 if the item doesn’t appear at all in the nested list.

(a) [1] State the output of `count_item([1, [3, 4, [4, -10]], [], [[]]], 4)`.

Solution: 3.

(b) [1] State the output of each of the following: `count_item(1, 4)`, `count_item([3, 4, [4, -10]], 4)`, `count_item([], 4)`, and `count_item([], 4)`. Be sure to label which one is which.

Solution:

```
>>> count_item(1, 4)
0
>>> count_item([3, 4, [4, -10]], 4)
2
>>> count_item([], 4)
0
>>> count_item([], 4)
1
```

(c) [4] Implement `count_item` in the space below.

```python
1  def count_item(obj, item):
  2      """Return the number of times <item> appears in <obj>.
  3      """
  4      @type obj: list | int
  5      @type item: int
  6      @rtype: int
  7      
  8      # SOLUTION
  9      if isinstance(obj, int):
     10          if obj == item:
     11              return 1
     12          else:
     13              return 0
     14      else:
     15          s = 0
     16          for lst in obj:
     17              s = s + count_item(lst, item)
     18          return s
```
2. (a) [2] Draw two different binary search trees which contain the items -4, 2, 3, 10, 15, 200 (and no other items).
   One should have the maximum possible height and the other the minimum possible height for a BST containing these items.

   **Solution:** Many possible answers. Minimum possible height is 3; maximum possible height is 6 (in this case, the BST looks like a list).

   (b) [2] Here is a tree. Is it a binary search tree? Why or why not?

```
    6
   /|
  3 30
 /|
-2 4 10
|
20
```

   **Solution:** No, this is not a BST; the 20 is on the right side of the 30, which violates the BST property for 30 (all items to its right must be ≥ it).

(c) [2] Assume that we have implemented the `insert` method for `BinarySearchTree` correctly. Consider a `BinarySearchTree` method which takes a non-empty BST with an empty left subtree, and attempts to “promote” the right subtree up to be the full tree, and insert the old root element into the new tree. Explain what is wrong with the following implementation.

```python
def promote_right(self):
    """ASSUME self is non-empty and self._left is empty!""
    temp = self._root
    self = self._right
    self.insert(temp)
```

   **Solution:** assigning `self = self._right` doesn’t actually mutate `self`, it changes what data `self` refers to. So this method won’t mutate the BST that calls it.

   Note: some students said you couldn’t assign `self` to something. That’s not true - Python does allow you to do it, it’s just that we don’t normally see it because it rarely helps.

(d) [2] Implement a correct version of `promote_right`. No docstring is necessary. Again, you may assume that `insert` is already implemented, and `self` is non-empty and `self._left` is empty.

```python
def promote_right(self):
    """ASSUME self is non-empty and self._left is empty!""
    temp = self._root
    #self = self._right
    self._root = self._right._root
    self._left = self._right._left
    self._right = self._right._right
    self.insert(temp)
```
3. Consider the **BinarySearchTree** method `big_internal`, which takes an integer and returns the number of internal (i.e., non-leaf) values in the BST which are greater than or equal to that integer. You may assume there are only integers in the BST.

(a) [1] Suppose we have a Python variable `bst` which corresponds to the above tree. What is the output of `t.big_internal(11)`?

   **Solution**: 1 (the value 13). Note that there was a typo in the above tree (the item 3 violates the BST property for 5), but this didn’t change the answer. There was also typo in the question, so we also accepted “error because t is not defined.”

(b) [2] To compute `bst.big_internal(11)` (bst refers to the above tree), would we need to make both recursive calls `bst.left.big_internal(11)` and `bst.right.big_internal(11)`, or just one, and if so, which one? Why?

   **Solution**: only the right recursive call; because the argument (11) is bigger than the root of `bst` (10), we know that all internal values in the left subtree are ≤ 10, and won’t be counted.

(c) [5] Implement `big_internal` below. Use the BST property to make as few recursive calls as possible – but think carefully about all the different cases! You may **not** use any BST methods other than `is_empty`.

```python
def big_internal(self, num):
    
    """Return the number of internal values in <self> which are >= num.
    """

    # SOLUTION
    if self.is_empty() or (self._left.is_empty() and self._right.is_empty()):
        # tree has no internal values
        return 0
    elif self._root < num:
        return self._right.big_internal(num)
    else: # self._root >= num
        # A few students forgot the "+ 1".
        return 1 + self._left.big_internal(num) + self._right.big_internal(num)
```

**Solution**: O(1) – independent of the “input size”, regardless of whether we’re talking about the height or size of the tree. The best case is when we input a num which is larger than every item in the tree, so we keep making recursive calls to the right subtree, i.e., go down the right side of the tree. It is tempting in this case to say that the total number of recursive calls is $O(h)$, where $h$ is the height of the tree. However, keep in mind that the right subtree could be very short compared to the left subtree; in the extreme case, it could be empty.

A true “best case” is when the input num is larger than every item in the tree, and the tree has an empty right subtree. In this case, after the root is checked the base case is reached immediately; neither the size nor the height of the tree matter in this case.

(b) [3] What is the worst-case Big-Oh running time of the following BST method? Your answer should involve the height of the BST. Justify your answer. (Hint: think carefully about the number of recursive calls.)

```python
def my_method(self, n):
    """n is a positive integer.""
    if self.is_empty():
        return n
    elif n == 0:
        return 10
    else:
        return 1 + self._left.my_method(n - 1)
```

**Solution**: $O(\min(n, h))$, where $h$ is the height of the BST. Without the “n”, the worst-case would be the same as for BST search, insert, or delete: $O(h)$ recursive calls are made, all down the left side of the tree. However, because there are two base cases (input tree is empty, or $n$ is 0), the sequence of recursive calls could stop because of either case, whichever is reached first. Since the “n” argument passed to the recursive call decreases by 1 on each recursive call, if the original $n$ is smaller than the height of the tree, then there will be $n$ recursive calls made – in other words, the recursion could stop before reaching the bottom of the tree.

Note that “worst case,” like “best case,” doesn’t allow you to specify an input size; in particular, there are really two inputs for this method, and you can’t say something like “the worst case happens when $n$ is less than the height of the tree, and...” because this restricts the size of one of the inputs, and both input sizes matter!

(c) [2] David says, “there exists an algorithm for inserting an item into a general tree that always runs in constant time.” Is this statement true or false? If you think it’s true, define an insert method for the Tree class that runs in constant time. If you think it’s false, explain why not using words (and possibly a diagram).

**Remember**: we’re talking about general trees, not binary search trees.

**Solution**: Such an algorithm exists; the easiest one is to add the item as the “last” child of the root of the tree. (Recall that we’ve already discussed how adding an item to the end of a Python list takes constant time.)

```python
def insert(self, item):
    self._subtrees.append(Tree(item))
```