Name:
Student Number:

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 \texttt{count\_pos} which takes a nested list and returns the sum of all positive numbers in the nested list, or returns 0 if there are no positive numbers in the nested list. 0 is not positive.

(a) [1] State the output of \texttt{count\_pos([1, [3, -4, [4, -10]], [], [[5]]])}.

\textbf{Solution:} 13. (Note: if you answered all three parts of this question by computing the number of positive numbers rather than their sum, you could have received full marks.)

(b) [1] State the output of each of the following: \texttt{count\_pos(1), count\_pos([3, -4, [4, -10]]), count\_pos([]), and count\_pos([[5]])}. Be sure to label which one is which.

>>> count\_pos(1)
1
>>> count\_pos([3, -4, [4, -10]])
7
>>> count\_pos([])
0
>>> count\_pos([[5]])
5

(c) [4] Implement \texttt{count\_pos} in the space below.

```
def count_pos(obj):
    """Return the sum of all positive numbers in <obj>.
   "
    @type obj : list | int
    @rtype : int
   "
    # SOLUTION
    if isinstance(obj, int):
        if obj > 0:
            return obj
        else:
            return 0
    else:
        s = 0
        for lst in obj:
            s = s + count_pos(lst)
        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] Define the Binary Search Tree property.

Solution: (item version) – every value in an item’s left subtree is ≤ the item, and every value in an item’s right subtree is ≥ the item. (tree version) for each item in the tree, every value in its left subtree is ≤ it, and every value in its right subtree is ≥ it.

(c) [2] Suppose we want a BinarySearchTree method root_dup, which takes a non-empty BST and returns whether there is a copy of the root item in one of the subtrees of the BST (remember that BSTs are allowed to contain duplicates), and leaves the BST unchanged. Assume that __contains__ and delete_item are implemented for this class already. Explain the problem with the following implementation.

```python
def root_dup(self):
    """ASSUME that self is not empty!""
    copy = self
    copy.delete_item(self._root)
    return copy.__contains__(self._root)
```

Solution: the line copy = self creates an alias of the calling object, so if copy changes (by calling the mutating method delete_item), then self will also change. However, this method is supposed to be non-mutating, i.e., “leave the BST unchanged.”

(d) [2] Implement a correct version of root_dup. No docstring is necessary. Again, assume __contains__ and delete_item are already implemented, and that self is non-empty.

```python
def root_dup(self):
    return self._left.__contains__(self._root) or self._right.__contains__(self._root)
    # or,
    # return self._root in self._left or self._root in self._right
```
3. Consider the **BinarySearchTree** method `small_leaves`, which takes an integer and returns the number of leaf values in the BST which are less than or equal to that integer. You may assume there are only integers in the BST.

![BinarySearchTree Diagram]

(a) **1** Suppose we have a Python variable `bst` which corresponds to the above tree. What is the output of `t.small_leaves(5)`?

**Solution**: 2. Although there was a typo in the question, so we also accepted “error because `t` is not defined.”

(b) **2** To compute `bst.small_leaves(5)` (`bst` refers to the above tree), would we need to make both recursive calls `bst._left.small_leaves(5)` and `bst._right.small_leaves(5)`, or just one, and if so, which one? Why?

**Solution**: only the left recursive call; because the argument (5) is smaller than the root of `bst` (6), we know that all leaves in the right subtree are $\geq 6$, and won’t be counted.

(c) **5** Implement `small_leaves` 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
1 def small_leaves(self, num):
2     """Return the number of leaves in <self> which are <= num.
3     @type self: BinarySearchTree
4     @type num: int
5     @rtype: int
6     """
7     # SOLUTION
8     if self.is_empty():
9         return 0
10    elif self._root > num:
11        return self._left.small_leaves(num)
12    else: # self._root <= num
13        if self._left.is_empty() and self._right.is_empty():
14            # root is a leaf
15            return 1
16        else:
17            return self._left.small_leaves(num) + self._right.small_leaves(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 smaller than every item in the tree, so we keep making recursive calls to the left subtree, i.e., go down the left 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 left subtree could be very short compared to the right subtree; in the extreme case, it could be empty.

A true “best case” is when the input num is smaller than every item in the tree, and the tree has an empty left 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: look carefully at the base case.)

```python
def my_method(self, n):
    """n is a positive integer.""
    if self.is_empty():
        for i in range(n):
            print(i)
        return n
    else:
        return self._left.my_method(n)
```

Solution: Let \(h\) represent the height of self. As with BST search, insert, and delete, the number of recursive calls in the worst-case is \(h + 1\). Each recursive call takes constant time, except for the base case, which takes \(n\) steps. So the running time in the worst-case is \(O(h+n)\) – adding up the cost of each recursive call (proportional to \(h\)) and the cost of the single base case call (proportional to \(n\)).

(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))
```