15-110 Hw4 Check-in - Written Portion

Name:

AndrewID:

#1 - Running a Hash Function - 10pts

We've designed our own hash function for strings, as shown below:

```
def hash(s):
    s = s.lower()
    if len(s) == 0:
        return 0
    elif len(s) == 1:
        return ord(s[0]) - ord("a")
    else:
        # Note that abs(x) is the absolute value of x
        return abs(ord(s[0]) - ord(s[len(s)-1]))
```

The table below represents a hashtable with six buckets. Use our hash function to add the following strings to that hashtable, according to the formula we discussed in class:

`"f", "apple", "hello", "TEST", "Zebra"`

#2 - Mystery Code Comprehension - 10pts

The following function takes a list of strings as input, takes two major algorithmic steps, then returns something useful:

```
def mystery(lst):
    ### BLOCK A ###
    d = { }
    for item in lst:
        if len(item) != 0:
            tmp = item[0]
            if tmp not in d:
                d[tmp] = 0
            d[tmp] = d[tmp] + 1

    ### BLOCK B ###
    x = 0
    y = 0
    for k in d:
        if d[k] > x:
            x = d[k]
            y = k
    return y
```

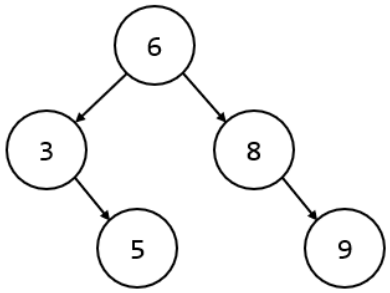
What does the BLOCK A part of the function do?

What does the BLOCK B part of the function do?

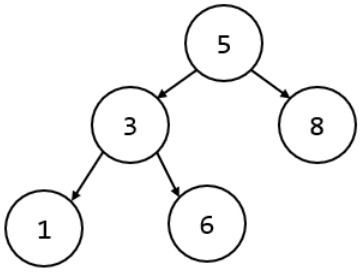
What does the function return?

#3 - Binary Search Tree Identification - 15pts

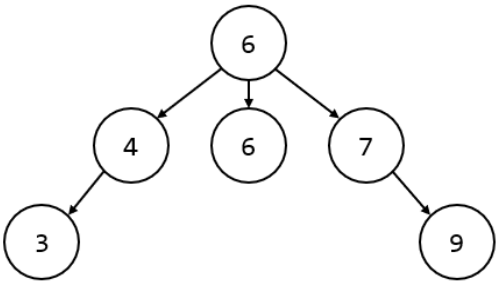
For each of the following trees, determine whether or not it is a Binary Search Tree.



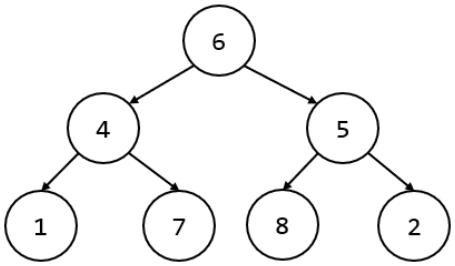
☐ BST
☐ Not BST



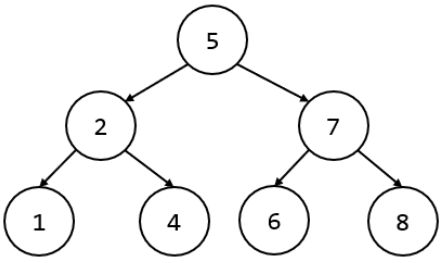
☐ BST
☐ Not BST



☐ BST
☐ Not BST



☐ BST
☐ Not BST



☐ BST
☐ Not BST

#4 - Calculating Big-O Families - 10pts

Note: this problem is on content from Week7.
For each of the following functions, check the **Big-O function family** that function belongs to. You should determine the function family by considering how the number of steps the algorithm takes grows as the size of the input grows.

```
# n = len(L)
def countEven(L):
  result = 0
  for i in range(len(L)):
    if L[i] % 2 == 0:
      result = result + 1
  return result
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ $O(n^2)$

```
# n = len(L)
def sumFirstTwo(L):
  if len(L) < 2:
    return 0
  return L[0] + L[1]
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ $O(n^2)$

```
# n = len(L1) = len(L2)
def linearSearchAll(L1, L2):
  count = 0
  for item in L1:
    # Hint: what's the complexity of
    # linear search?
    if linearSearch(L2, item) == True:
      count = count + 1
  return count
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ $O(n^2)$

```
# n = len(L1) = len(L2)
def binarySearchAll(L1, L2):
  count = 0
  for item in L1:
    # Hint: what's the complexity of
    # binary search?
    if binarySearch(L2, item) == True:
      count = count + 1
  return count
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ $O(n^2)$

```
# n = len(L)
# Hint: consider the # of recursive call
def recursiveSum(L):
  if len(L) == 0:
    return 0
  else:
    return L[0] + recursiveSum(L[1:])
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ $O(n^2)$

#5 - Tracing Sorting Algorithms - 15pts

Note: this problem is on content from Week7.

For the two tables below, each row represents a 'pass' - a single iteration of the outer loop in the function. Fill in the number of comparisons and swaps that happen in each pass, and the state of the list at the *end* of that pass, for the specified sort function as implemented in class.

Selection Sort			
Pass #	**Comparisons**	**Swaps**	**List State**
Start	-	-	**[3, 5, 1, 2, 4]**
1			
2			
3			
4			

Insertion Sort			
Pass #	**Comparisons**	**Swaps**	**List State**
Start	-	-	**[3, 5, 1, 2, 4]**
1			
2			
3			
4			

Merge Sort:
How many times is the function mergeSort called when we run `mergeSort([ 3, 5, 1, 2, 4 ])`?

Programming Problems

Each of these problems should be solved in the starter file available on the course website. They should be submitted to the Gradescope assignment Hw4 Check-in (Programming) to be autograded.

Most programming problems may also be checked by running the starter file, which calls the function testAll() to run test cases.

#1 - createPhonebook(nameList, numberList) - 20pts

Write the function createPhonebook that takes two lists, a list of names and a list of phone numbers (both strings), and returns a dictionary mapping names to phone numbers. You may assume that each person is in the same index as their phone number.

If a person occurs in the nameList multiple times (in other words, if they have multiple phone numbers), you should map their name to the **first** phone number they were paired with. For example, given the list of names ["Kelly", "Stephanie", "Kelly"] and the list of numbers ["0000", "1234", "9876"], the function would return the dictionary { "Kelly" : "0000", "Stephanie" : "1234" }.

#2 - sumTree(t) - 20pts

Write the function sumTree(t) that takes a binary tree and returns the sum of all the values in the tree. Remember that you'll have to use recursion to iterate over all the elements of the tree!

The tree will be in the dictionary structure discussed in class: each node is a dictionary with three keys; "value", "left", and "right". If the left or right sub-tree doesn't exist, that key maps to None.