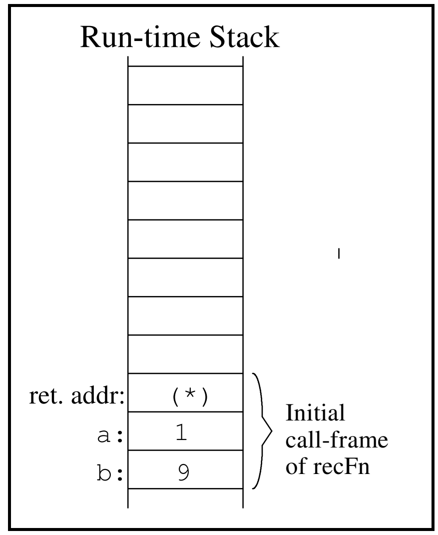
Spring 2016 Name: ________________________

Data Structures - Test 2

Question 1. (10 points) What is printed by the following program? Output:

```
def recFn(a, b):
   print(a, b)
   if a == b:
      return 100
   elif a > b:
      return 10
    else:
      return recFn(a + 1, b - 2) + a
                (**)


print("result =",recFn(1, 9))
                   (*)
```



Question 2. (10 points) Write a recursive Python function to compute the following mathematical function, G(n):

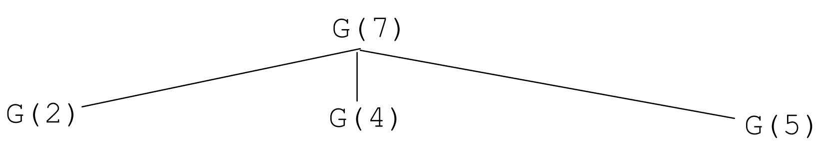
$G(n) = n$ for all value of $n \leq 1$

$G(2) = 5$ if $n = 2$

$G(n) = G(n-5) + G(n-3) + G(n-2)$ for all n values > 2.

```
def G(n):
```

Question 3. a) (7 points) For the above recursive function G(n), complete the calling-tree for G(7).



b) (2 point) What is the value of G(7)?

c) (1 point) What is the maximum height of the run-time stack when calculating G(7) recursively?

Question 3. The insertion sort code discussed in class is:

```
def insertionSort(myList):
    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        myList[testIndex + 1] = itemToInsert
```

Consider the following `insertMergeSort` code which calls the above `insertionSort` code twice with copies of each half of the array, and then merges the two sorted halves back together using the merge code from merge sort.

```
def insertMergeSort(aList):
    halfSize = len(aList) // 2
    lefthalf = aList[ : halfSize]
    righthalf = aList[halfSize : ]
    insertionSort(lefthalf)
    insertionSort(righthalf)

    #### BELOW IS THE MERGE CODE FROM MERGE SORT ####
    i=0  # index into lefthalf
    j=0  # index into righthalf
    k=0  # index into aList
    while i<len(lefthalf) and j<len(righthalf):  # compare and copy until one half runs out
        if lefthalf[i]<righthalf[j]:
            aList[k]=lefthalf[i]
            i=i+1
        else:
            aList[k]=righthalf[j]
            j=j+1
        k=k+1

    while i<len(lefthalf):        # copy the remaining items from lefthalf if any
        aList[k]=lefthalf[i]
        i=i+1
        k=k+1

    while j<len(righthalf):        # copy the remaining items from righthalf if any
        aList[k]=righthalf[j]
        j=j+1
        k=k+1
```

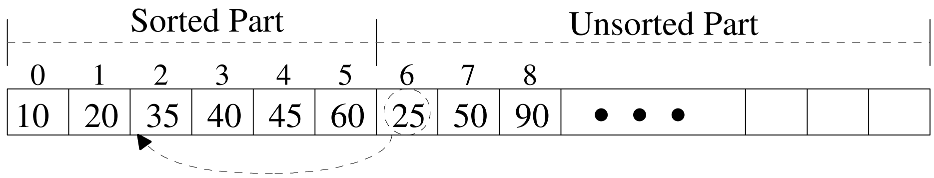
Consider the following timing of `insertionSort` vs. `insertMergeSort` on lists of 10000 elements.

Initial arrangement of list before sorting	`insertionSort` - at the top of page	`insertMergeSort` - modified version in middle of the page
Sorted in descending order: 10000, 9999, ..., 2, 1	14.3 seconds	7.1 seconds
Already in ascending order: 1, 2, ..., 9999, 10000	0.005 seconds	0.009 seconds
Randomly ordered list of 10000 numbers	7.4 seconds	3.6 seconds

a) (10 points) Explain why `insertMergeSort`(modified version in middle of page) out performs the original `insertionSort`.

b) (10 points) In either version, why does sorting the randomly order list take about halve the time of sorting the initially descending ordered list?

Question 4. (20 points) In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the below example) and "inserts" it into the sorted part of the list "at the correct spot."



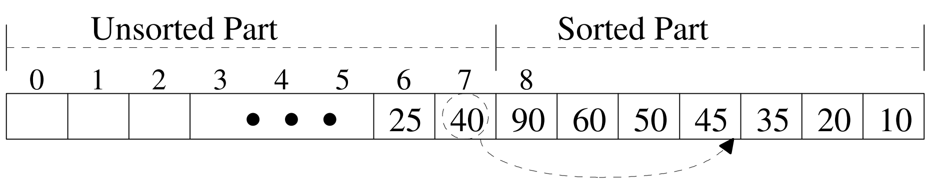
In class we discussed the following insertion sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list, i.e.:

```
def insertionSort(myList):
    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        myList[testIndex + 1] = itemToInsert
```

For this question write a variation of the above insertion sort that:

- sorts in **descending order** (largest to smallest)
- builds the **sorted part on the right-hand side** of the list, i.e.,

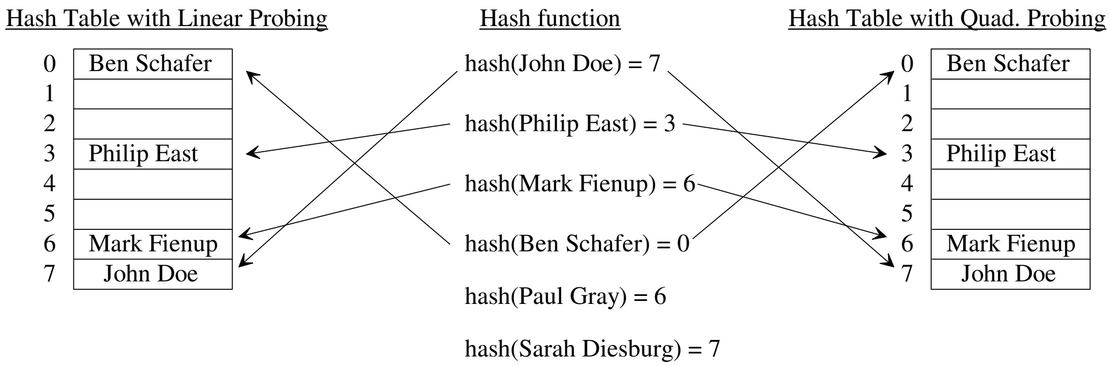



```
def insertionSortVariation(myList):
```

Question 5. Two common rehashing strategies for open-address hashing are linear probing and quadratic probing:

quadratic probing	Check the square of the attempt-number away for an available slot, i.e., [home address + ((rehash attempt #)^2 +(rehash attempt #))/2] % (hash table size), where the hash table size is a power of 2. Integer division is used above

a) (8 points) Insert "Paul Gray" and then "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.

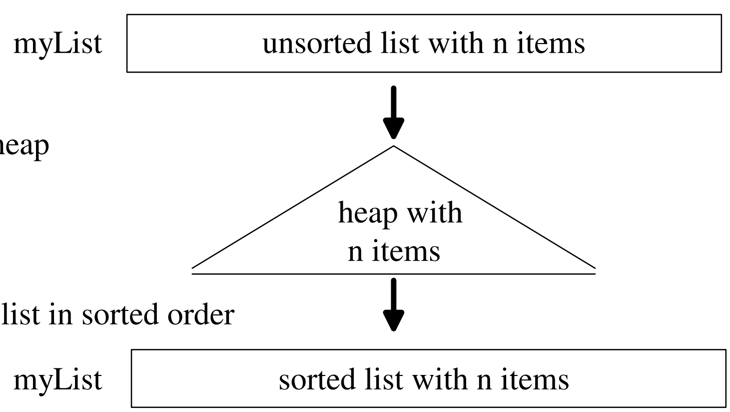


b) (7 points) Explain why both linear and quadratic probing both suffer from primary clustering?

Question 6. Recall the general idea of Heap sort which uses a min-heap (class `BinHeap` with methods: `BinHeap()`, `insert(item)`, `delMin()`, `isEmpty()`, `size()`)) to sort a list.

<u>Generl idea of Heap sort:</u>

1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order



a) (10 points) Complete the code for `heapSort` so that it **<u>sorts in descending order</u>**

```
from bin_heap import BinHeap
def heapSort(myList):
   myHeap = BinHeap()   # Create an empty heap
```

b) (5 points) Determine the overall $O(\ )$ for your heap sort and briefly justify your answer.