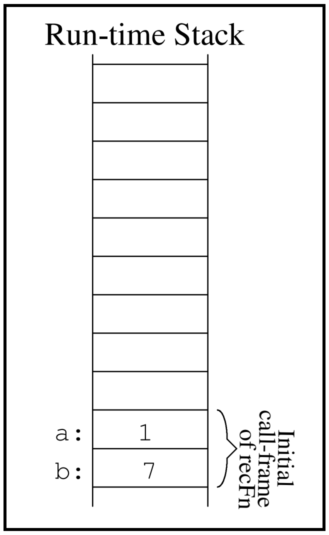
Fall 2014 Name: ______________________

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 a
    elif a == b:
        return a + b
    else:
        return b + recFn(a + 1, b - 1) - a

print("Result = ", recFn(1, 7))
```

Question 2. (8 points) Write a recursive Python function to compute the following mathematical function, G(n):

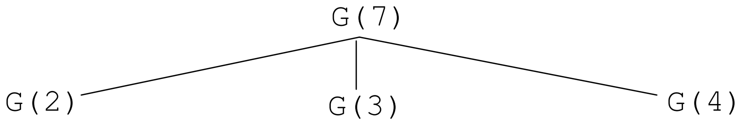
$G(n) = 1$ for all value of $n \leq 0$

$G(n) = 2$ for $n = 1$

$G(n) = G(n-5) + G(n-4) + G(n-3)$ for all values of $n > 1$.

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

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



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

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

Question 4. (10 points.) Consider the following timings of the recursive G(n) function from question 2.

Value of n	40	50	60	70	80
Time to calculate G(n) in seconds	0.05	0.56	8.78	145.98	2,430.46

a) Explain why the recursive `G(n)` function from question 2 is so slow.

b) Explain how we could speed up the calculation of `G(n)`? (no code is necessary just an explanation)

Question 5. (15 points) Consider the timings (in seconds) of various ascending, bubble sorting algorithms on 10,000 items.

Type of bubble sorting algorithm	Initial Ordering of Items		
	Descending	Ascending	Random order
bubble - no check to stop early	23.2	7.7	15.9
bubble - with a check to stop early	24.1	0.002	16.3

a) Why does the bubble sort with no check to stop early take less time on the ascending ordered list than it does on the descending ordered list?

b) Why does the bubble sort with a check to stop early take A LOT less time on the ascending ordered list than the descending ordered list?

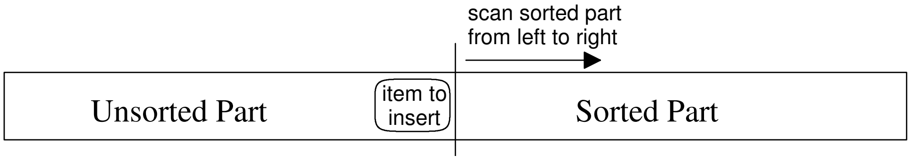
c) Why does the bubble sort with no check to stop early take less time on the descending ordered list than the bubble sort with a check to stop early on the descending ordered list?

Question 6. (20 points) 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.

```
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 insert sort that:

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

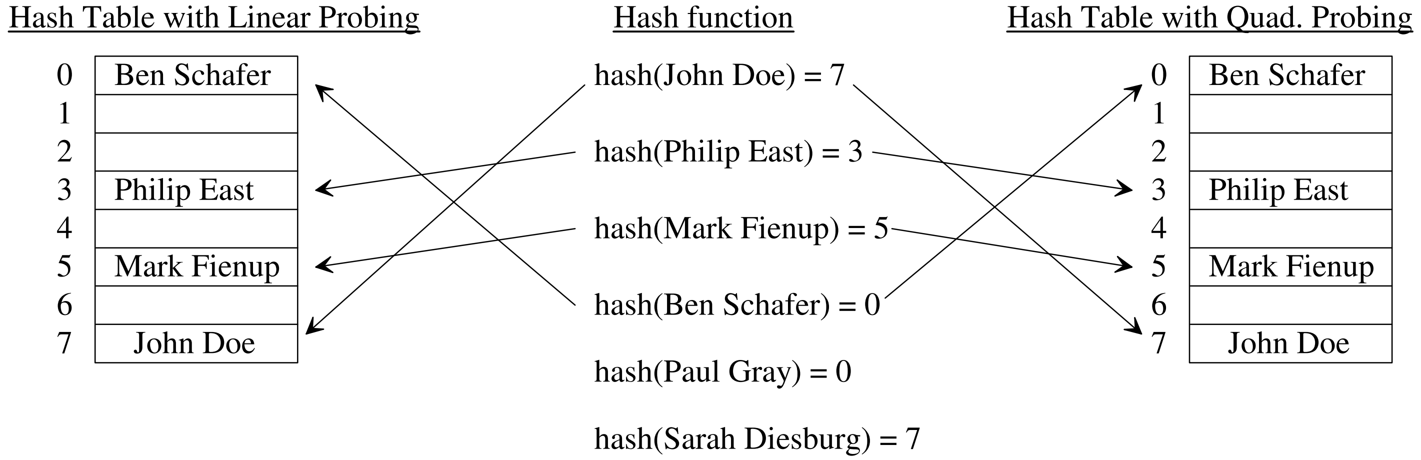


```
def insertionSort(myList):
```

Question 7. (15 points) Recall the common rehashing strategies we discussed for open-address hashing:

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

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



b) What is the purpose of requiring a hash table size that is a power of 2 when using quadratic probing?

Question 8. (15 points) Use the below diagram to explain the worst-case big-oh notation of merge sort. Assume "n" items to sort.