Fall 2017 Name: Mark F.

Data Structures - Test 2

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

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

print("Result = ", recFn(1, 10))
                      (*)
```

1 + (-5) - 10 = -14

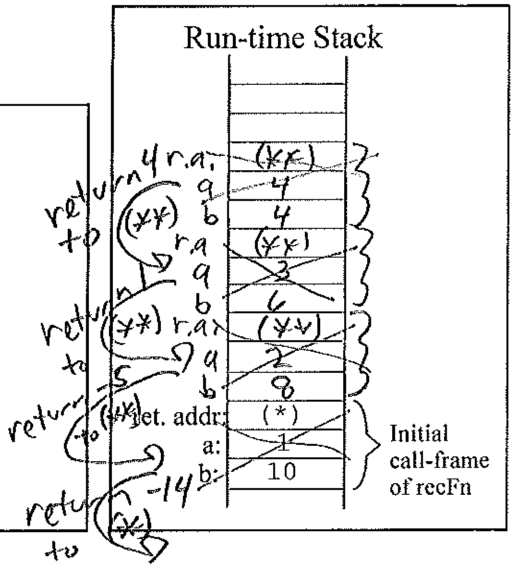
2 + 1 - 8 = -5

3 + 4 - 6 = 1

-14

Output:

```
1  10
2  8
3  6
4  4
Result = -14
```



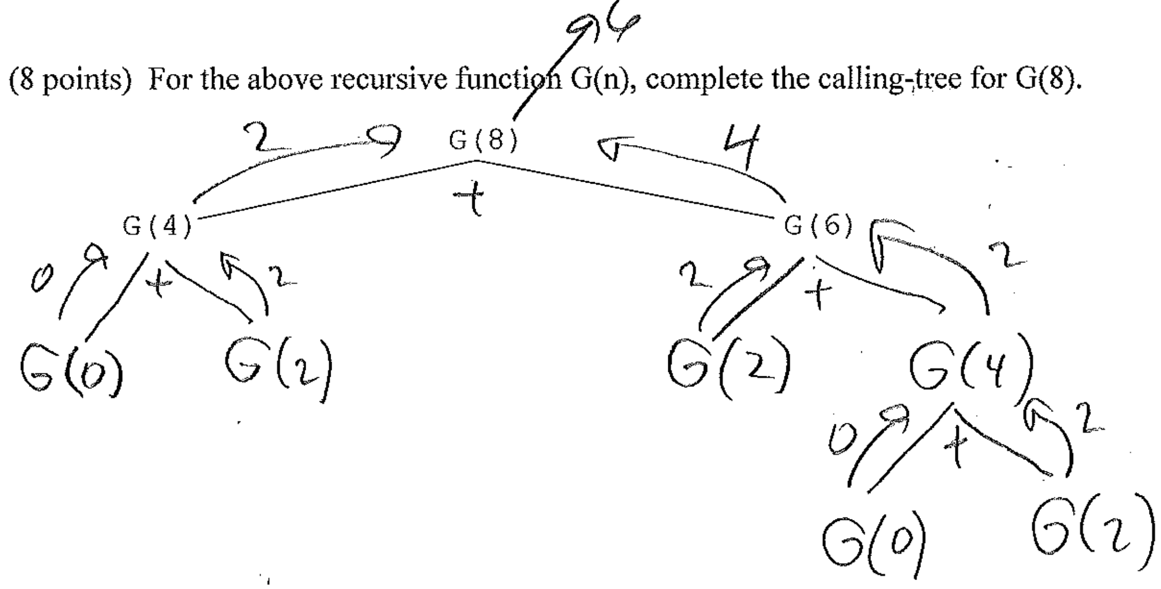
Question 2. a) (12 points) Write a recursive Python function to compute the following mathematical function, G(n):

$G(n) = n$ for all values of $n \leq 3$ (e.g., G(2) value is 2).

$G(n) = G(n-4) + G(n-2)$ for all values of $n > 3$.

```
def G(n):
    if n <= 3:
        return n
    else:
        return G(n-4) + G(n-2)
```

b) (8 points) For the above recursive function G(n), complete the calling-tree for G(8).



c) (3 points) What is the value of G(8)? 6

d) (2 points) What is the maximum number of call-frames of G on the run-time stack when calculating G(8) recursively? 4

35

Question 3. (15 points) Consider the following simple sorts discussed in class -- all of which sort in ascending order.

```
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1,0,-1):
        alreadySorted = True
        for testIndex in range(lastUnsortedIndex):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
                alreadySorted = False
        if alreadySorted:
            return
```

```
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
```

```
def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        maxIndex = 0
        for testIndex in range(1, lastUnsortedIndex+1):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        # exchange the items at maxIndex and lastUnsortedIndex
        temp = aList[lastUnsortedIndex]
        aList[lastUnsortedIndex] = aList[maxIndex]
        aList[maxIndex] = temp
```

Timings of Above Sorting Algorithms on 10,000 items (seconds)			
Type of sorting algorithm	Initial Ordering of Items		
	Descending	Ascending	Random order
bubbleSort.py	24.5	0.002	16.5
insertionSort.py	14.2	0.004	7.3
selectionSort.py	7.3	7.7	6.8

a) Explain why insertionSort on a descending list (14.2 s) takes longer than insertionSort on a random list (7.3 s).

For descending order, the whole sorted part is compared and shifted before inserting at index 0. On random order, we expect only needing to compare and shift half way down sorted part.

b) Explain why insertionSort on a descending list (14.2 s) takes longer than selectionSort on a descending list (7.3 s).

Insertion sort must compare and shift whole sorted part. Selection sort compares whole unsorted part to find max. item's index, but only needs 3 moves to swap max. item. Insertion does more moves.

c) Explain why bubble sort is $O(n^2)$ in the worst-case.

Descending order is the worst-case for bubble sort since item at index 0 must compare and swap down whole unsorted part. Since the unsorted part shrinks by one on each iteration of the outer-loop the # of compares and swaps is $(n-1)+(n-2)+(n-3)+\cdots+3+2+1 = \underbrace{n+n+\cdots+n}_{(n-1)/2}$

15 $= n \times \frac{(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$ which is $O(n^2)$

n

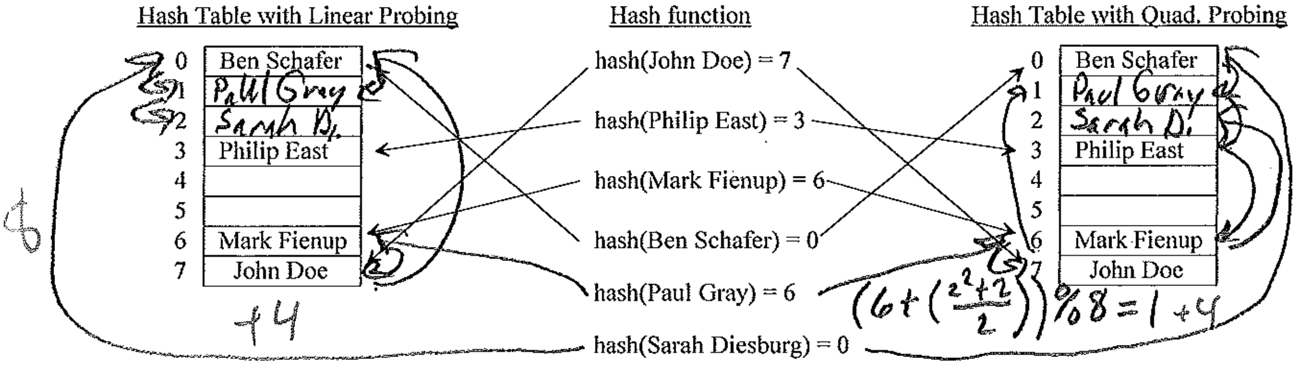
$(n-2)+2 = n$

$(n-1)+1 = n$

Question 4. 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.



$(0+(\frac{2^2+2}{2}))\%8=3$

$(0+(\frac{3^2+3}{2}))\%8=6$

$(0+(\frac{4^2+4}{2}))\%8=2$

b) (4 points) In open-address hashing (like the pictures above), how do we handle deleting items in the hash table?

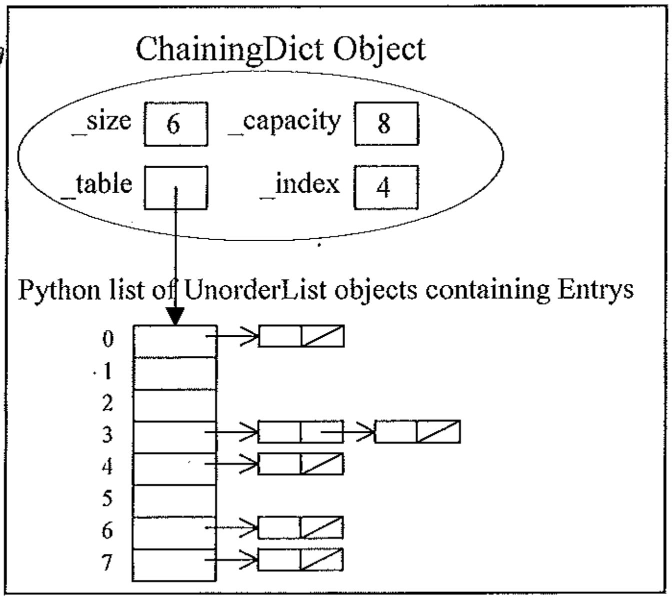
We overwrite deleted item with special "deleted" value
4 (e.g. True in OpenAddrHashTable class) which is different than None for empty slot.

c) (3 points) In open-address hashing (like the pictures above), how do deleted items effect performance of searching?

Slots with "deleted" values are treated as non-matchs
3 for target key, i.e., need to keep search until target key or None/empty slot.

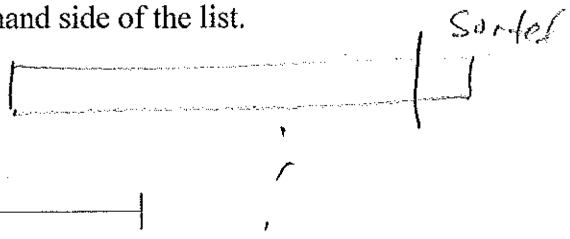
d) (5 points) In closed-address hashing (e.g., ChainingDict like picture below), if the load factor (# items / hash table size) is close to 1, say 0.99, would you expect average-case searches of O(1)? (Justify your answer)

Yes, if hash function is doing a "good" job, then each
5 slot has a list of about 1 in size. Search a list of 1 (or a few items) is constant.



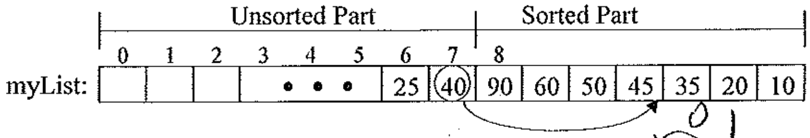
28

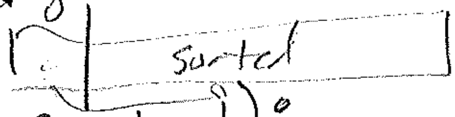
Question 5. (20 points) In class we discussed the insertionSort code shown in question 3 on page 2 which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list.



For this question write a variation of insertion sort that:

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



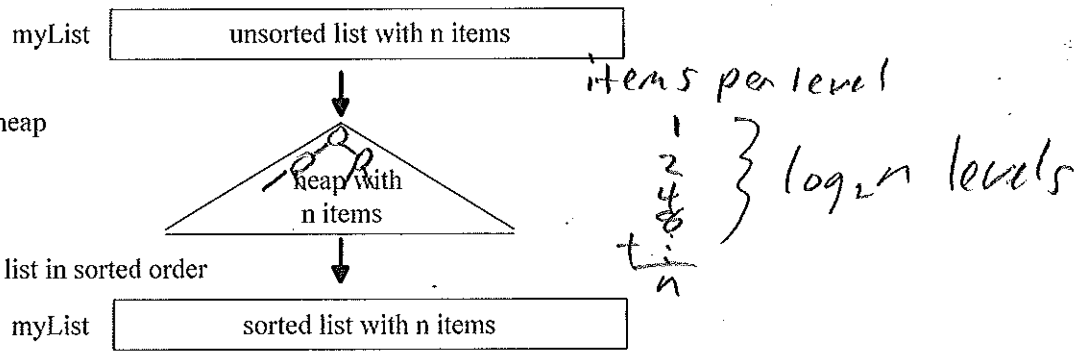


```
def insertionSortVariation(myList):
    for lastUnsorted in range(len(myList)-2,-1,-1):
        itemToInsert = myList[lastUnsorted]
        testIndex = lastUnsorted+1
        while testIndex <= len(myList)-1 and myList[testIndex] > itemToInsert:
            myList[testIndex-1] = myList[testIndex]
            testIndex += 1
        myList[testIndex-1] = itemToInsert
```

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.

Generl idea of Heap sort:

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



a) (5 points) Complete the code for `heapSort` so that it **sorts in descending order**

+2

```
from bin_heap import BinHeap
def heapSort(myList):
   myHeap = BinHeap()   # Create an empty heap
   for item in myList:
       myHeap.insert(item)
   for index in range(len(myList)-1,-1,-1):
       myList[index] = myHeap.delMin()
```

b) (5 points) Determine the overall $O(\,)$ for your heap sort and briefly justify your answer. Let n = len(myList).

+2

$O(n\log_2 n)$: The two for-loops are not nested so we add their contributions. Each for-loop loops n times and calls insert or delMin which are $O(\log_2 n)$ because the heap has height $\log_2 n$ (see above diagram)

30