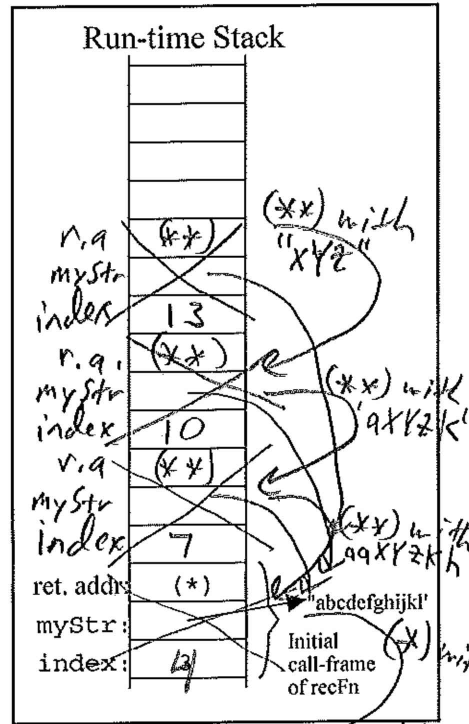
Fall 2018

Name: Mark F.

Data Structures - Test 2



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

```
def recFn(myStr, index):
   print(index)
   if index >= len(myStr):
      return "XYZ"
   else:
      return myStr[0] + recFn(myStr, index + 3) + myStr[index]

print("result =",recFn("abcdefghijkl", 4))
                (*)
```

Output:

4
7
10
13
result = aaaXYZkhe

10

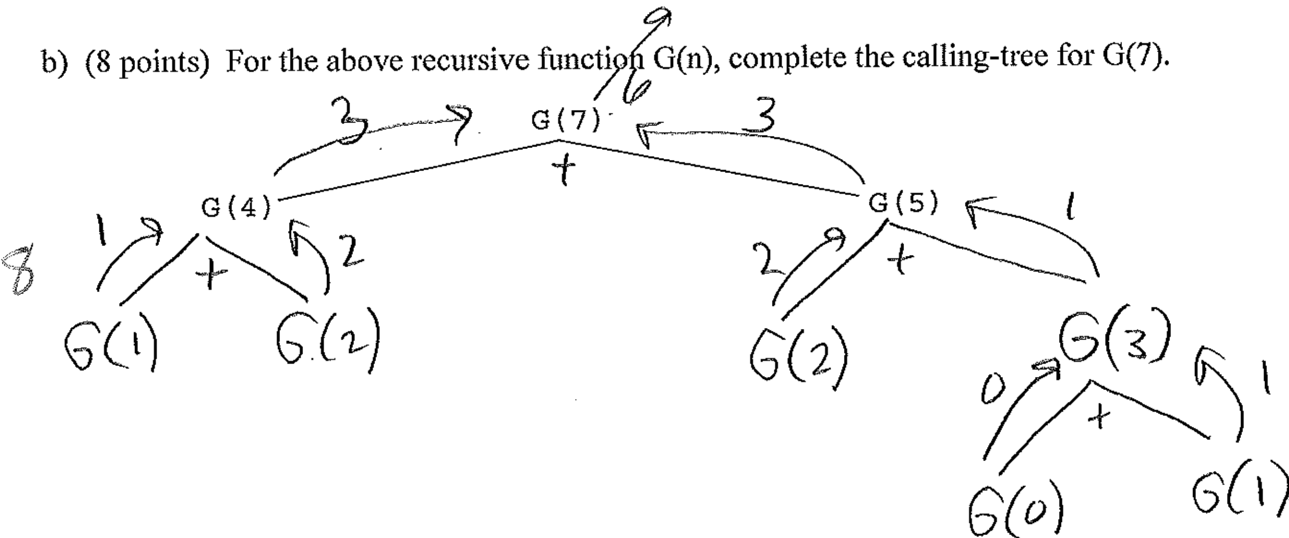
Question 2. a) (12 points) Write a recursive Python function to compute the following mathematical function, G(n):

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

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

12

b) (8 points) For the above recursive function G(n), complete the calling-tree for G(7).



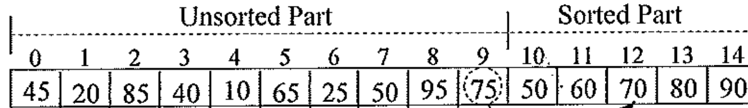
8

3 c) (3 points) What is the value of G(7)? 6

2 d) (2 points) What is the maximum ~~height~~ length of the run-time stack when calculating G(7) recursively? 4 (or 3)

25

Question 3. (15 points) Consider the following insertion sort code which sorts in ascending order, but builds the sorted part on the right-end of the list. For example after the code has ran a while, we need to insert 75 at index 12.




```
def insertionSort(myList):
    for lastUnsortedIndex in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsortedIndex]
        testIndex = lastUnsortedIndex + 1

        while testIndex < len(myList) and myList[testIndex] < itemToInsert:
            myList[testIndex-1] = myList[testIndex]
            testIndex = testIndex + 1

        myList[testIndex - 1] = itemToInsert
```

a) What is the purpose of the `testIndex < len(myList)` while-loop comparison? Keeps testIndex from going past the right-end of the list.

4

b) Consider the modified insertion sort code that eliminates the `testIndex < len(myList)` while-loop comparison.

```
def insertionSortB(myList):
    maxIndex = 0
    for testIndex in range(1,len(myList)):
        if myList[testIndex] > myList[maxIndex]:
            maxIndex = testIndex
    temp = myList[len(myList)-1]
    myList[len(myList)-1] = myList[maxIndex]
    myList[maxIndex] = temp

    for lastUnsortedIndex in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsortedIndex]
        testIndex = lastUnsortedIndex + 1

        while myList[testIndex] < itemToInsert:
            myList[testIndex-1] = myList[testIndex]
            testIndex = testIndex + 1

        myList[testIndex - 1] = itemToInsert
```

Explain how the **bold code** in the modified insertion sort code allows for the elimination of the `testIndex < len(myList)` while-loop comparison. By moving the max. item to the right-end of the list at the start, we are guaranteed that the myList[testIndex] < itemToInsert will prevent testIndex from getting too big.

4

Consider the following timing of the above two insertion sorts on lists of 10000 elements.

Initial arrangement of list before sorting	`insertionSort` - at the top of page	`insertionSortB` - modified version in middle of the page
Sorted in descending order: 10000, 9999, ..., 2, 1	14.0 seconds	12.3 seconds
Already in ascending order: 1, 2, ..., 9999, 10000	0.005 seconds	0.004 seconds
Randomly ordered list of 10000 numbers	7.3 seconds	6.4 seconds

c) Explain why `insertionSortB` (modified version in middle of page) out performs the original `insertionSort`. Added code is $O(n)$, but eliminated testIndex < len(myList) gets done $O(n^2)$ times.

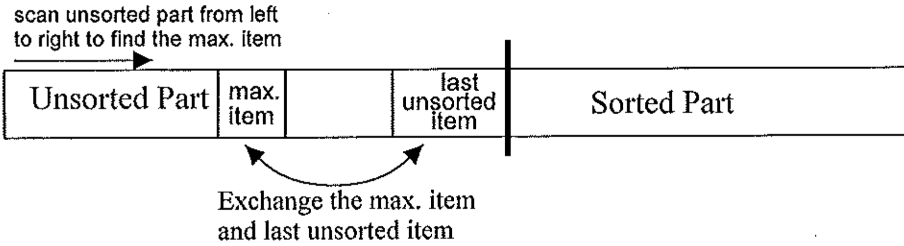
4

d) In either version, why does sorting the initially ascending order list take less time than sorting the initially descending ordered list? Inner-while does not run in ascending order, but runs across whole sorted part in descending order.

3

15

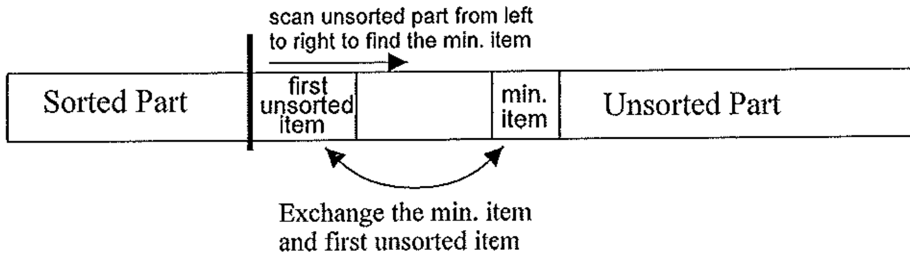
Question 4. In class we developed the following selection sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list, i.e.:




```
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
```

(20 points) For this question write a variation of the above selection sort that:

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




```
def selectionSortVariation(myList):
    for firstUnsortedIndex in range(0, len(myList)-1, 1):        → 5 range
        minIndex = firstUnsortedIndex
        for testIndex in range(firstUnsortedIndex+1, len(myList), 1):   → 5 range
            if myList[testIndex] < myList[minIndex]:             +5
                minIndex = testIndex

        temp = myList[firstUnsortedIndex]                        swap +5
        myList[firstUnsortedIndex] = myList[minIndex]
        myList[minIndex] = temp
```

20

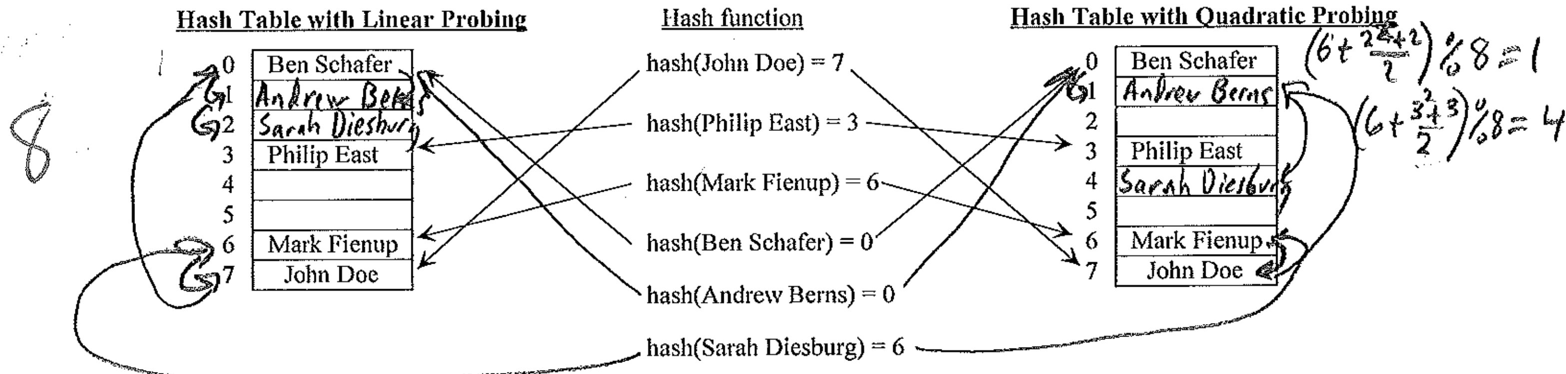
Fall 2018 Name: ______________________

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

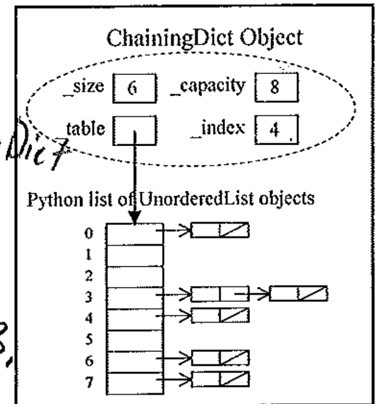
quadratic probing	Check the square of the attempt-number away for an available slot, i.e., [home address + ((rehash attempt #)^2 +(rehash attempt #))//2] % (hash table size), where the hash table size is a power of 2. Integer division is used above

a) (8 points) Insert "Andrew Berns" **and then** "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.

8



b) (7 points) Open-address hashing above uses rehashing (e.g., linear or quadratic probing) when collisions occur. Briefly describe how closed-address hashing (e.g., ChainingDict) handles collisions.



Each home addr. has a data structure that can hold all the items hashed to it. For example the ChainingDict has an UnorderedList at each home addr. In the picture 2 items hashed to home addr. 3 so it has 2 items in the UnorderedList at index 3.

7

Question 6. (15 points) Use the below diagram to ***explain*** the worst-case big-oh notation of merge sort. Assume "n" items to sort.

15

# Compares	Diagram	# Moves
0	Unsorted size n	n
0	Unsorted size n/2, Unsorted size n/2	n
	n/4, n/4, n/4, n/4	.
0	2 2 2 2 2 ... 2 2 2 2 2	n
	1 1 1 1 1 1 1 1 1 1 . . . 1 1 1 1 1 1 1 1 1 1	
$1\times\frac{n}{2}$	2 2 2 2 2 ... 2 2 2 2 2	n
$3\times\frac{n}{4}$	4 4	n
$7\times\frac{n}{8}$	n/4, n/4, n/4, n/4	.
.	Sorted size n/2, Sorted size n/2	n
$(n-1)$	Sorted size n	n

0 compares in top half

$\log_2 n$ levels so top half is $n\log_2 n$

$\log_2 n$ levels with each value between $\frac{n}{2}$ and $(n-1)$ so $O(n\log_2 n)$

$\log_2 n$ levels so bottom half is $n\log_2 n$

$2n\log_2 n$ moves total

Overall $O(n\log_2 n)$

30