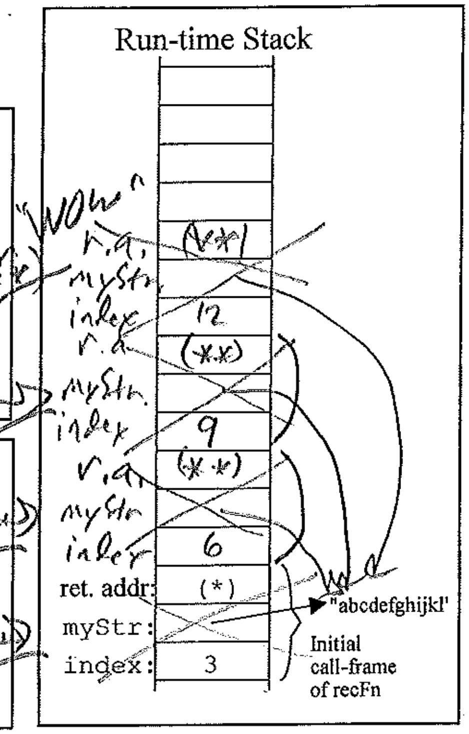
Fall 2015 Name: Mark F.

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 "WOW"
     else:
        return myStr[0] + recFn(myStr, index + 3) + myStr[index]

print("result =",recFn("abcdefghijkl", 3))
                  (*)
```



Output:

3
6
9
12
result = aaaWOWjgd

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 $n \le 2$ (e.g., G(1) value is 1)

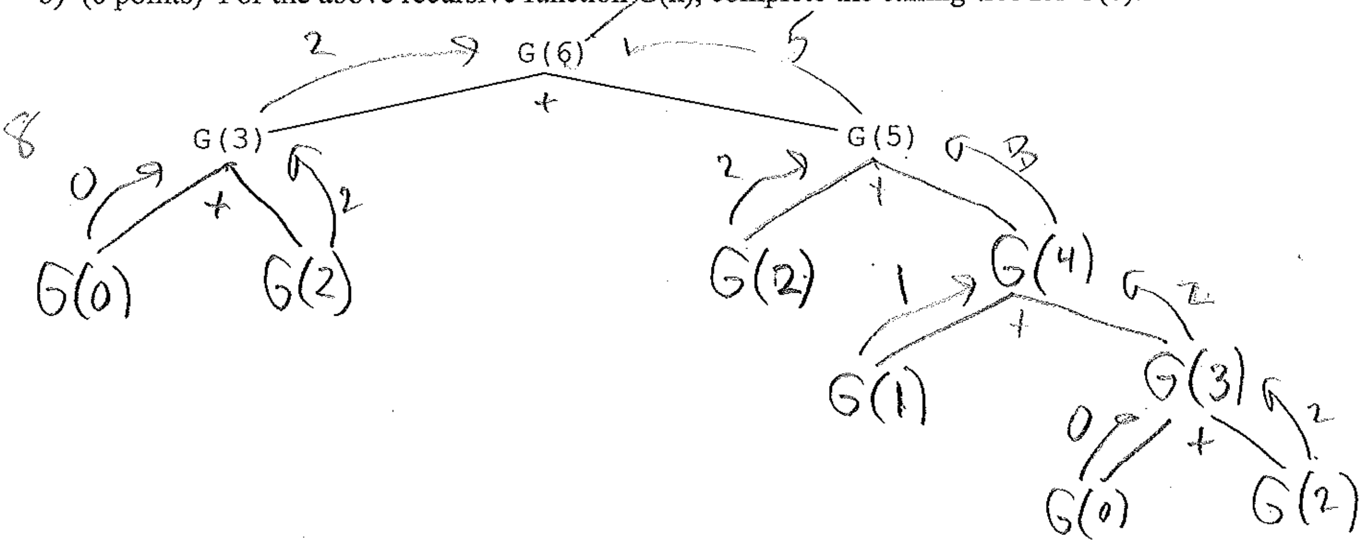
$G(n) = G(n-3) + G(n-1)$ for all values of $n > 2$.

```
def G(n):
    if n <= 2:
        return n
    else:
        return G(n-3) + G(n-1)
```

12

b) (8 points) For the above recursive function G(n), complete the calling-tree for G(6).

8



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

2 d) (2 points) What is the maximum height of the run-time stack when calculating G(6) recursively?. 5

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 bubbleSort on a descending list (24.5 s) takes longer than bubbleSort on a random list (16.5 s).

Descending order causes the if-statement condition to always be True so it always swaps and never stops early. Random order might not always swap and might stop early.

+1

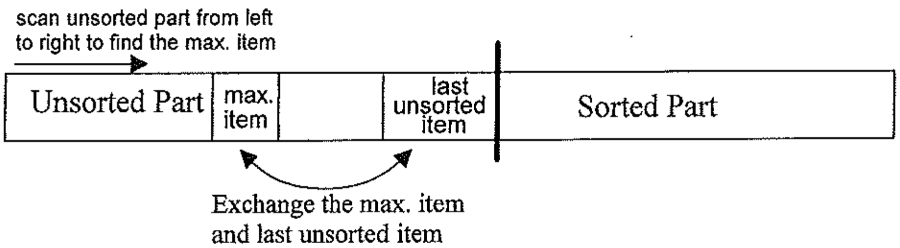
b) Explain why bubbleSort on a descending list (24.5 s) takes longer than insertionSort on a descending list (14.2 s).

Worst case for both: bubble sort compares and swaps down whole unsorted part, and insertion compares and shifts items "up" down whole sorted part. Thus, same # of comparisons, but each bubble sort swap involves 3 moves, while each insertion sort shift takes only one move.

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

Same number of comparisons for both, Selection only does 3 moves (1 swap) to extend the sorted part by one, while insertion sort must shift whole sorted part.

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.:

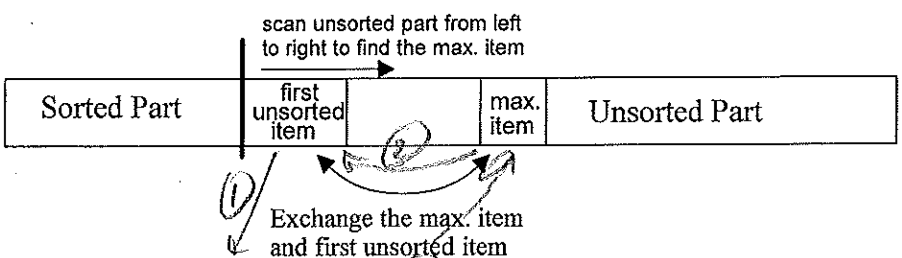



```
def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        maxIndex = 0
        for testIndex in range(1, lastUnsortedIndex+1):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        # exchange the items at maxIndex and lastUnsortedIndex
        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 **descending order** (largest to smallest)
- builds the **sorted part on the left-hand side** of the list, i.e.,

unsorted



temp

sorted

len(myList) -1

range + direction +4

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

+3

+5

Question 5. Recall the quadratic rehashing strategy we discussed for open-address hashing:

Strategy	Description
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.

8

Hash Table with Linear Probing

0	Ben Schafer
1	Paul Gray
2	Sarah Diesburg
3	Philip East
4	
5	
6	Mark Fienup
7	John Doe

Hash function

hash(John Doe) = 7
hash(Philip East) = 3
hash(Mark Fienup) = 6
hash(Ben Schafer) = 0
hash(Paul Gray) = 7
hash(Sarah Diesburg) = 6

Hash Table with Quad. Probing

0	Ben Schafer
1	Sarah Diesburg
2	Paul Gray
3	Philip East
4	
5	
6	Mark Fienup
7	John Doe

$7 + \left(\frac{2^2+2}{2}\right) \% 8 = 2$

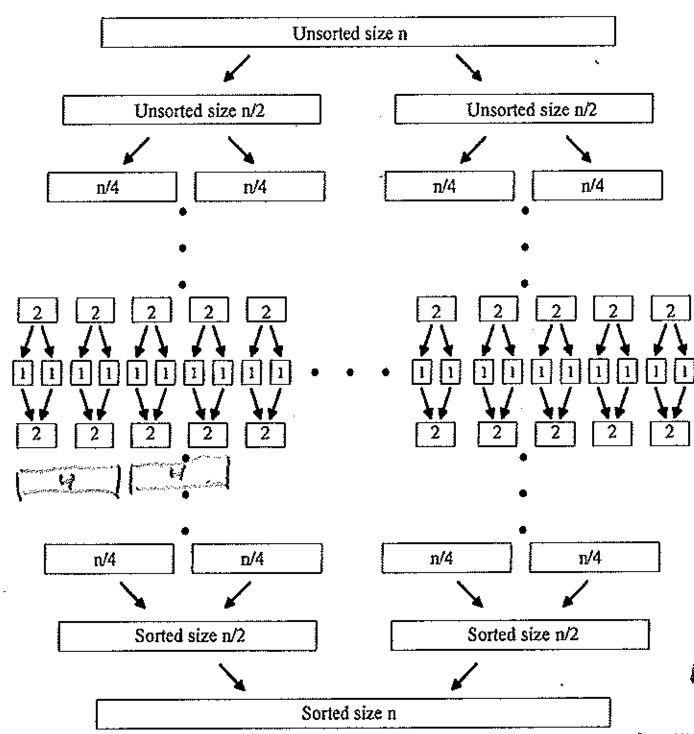
$6 + \left(\frac{2^2+2}{2}\right) \% 8 = 1$

b) (7 points) What is the purpose of requiring a hash table size that is a power of 2 when using quadratic probing?

7 So quadratic probing rehashes to every slot in the hash table before repeating

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	# Moves
0	n
0	n
0	n
$\frac{n}{2}$	n
$\frac{3}{4}n$	n
$(n-1)$	n

$\log_2 n$ values ($\#$ Moves, top half)

$\log_2 n$ values ($\#$ Moves, bottom half)

$\log_2 n$ values all between $\frac{n}{2}$ and $(n-1)$

$O(n \log_2 n)$

$2n\log_2 n$ moves

Overall $O(n\log_2 n)$

30