Fall 2013 Name: ______________________

Data Structures - Test 2

Question 1. (5 points) What is printed by the following program?

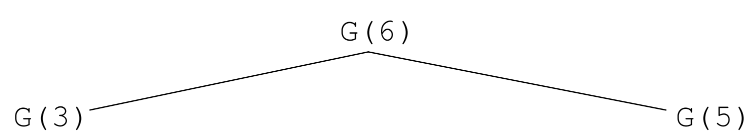
```
def recFn(myString, index):
    if index >= len(myString):
        return ""
    else:
        return recFn(myString, index + 3) + myString[index]

print(recFn("Go panthers!", 0))
```

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

G(0) value is 0
G(1) value is 1
G(2) value is 2
G(n) = G(n-3) + G(n-1) for all values of n > 2.

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

Question 3. (7 points) a) For the above recursive function G(n), complete the calling-tree for G(6).



b) What is the value of G(6)?

c) What is the maximum height of the run-time stack when calculating G(6) recursively?

Question 4. (10 points.) Consider the following selection sort code which sorts in ascending order.

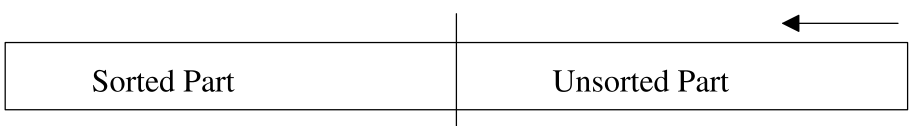
```
def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        # look for maximum item in unsorted part of list
        # Assume maximum is the first item in the unsorted part
        maxIndex = 0
        # scan the unsorted part of the list to correct the assumption
        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
```

a) Let "n" be the number of items in the list. How many total comparisons does the `if`-statement perform in the selection sort?

b) Let "n" be the number of items in the list. How many total item moves are performed in the selection sort?

Question 5. (25 points) Write a variation of bubble sort that:

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



Inner loop scans from right to left across the unsorted part swapping adjacent items that are "out of order"

(Your code does NOT need to stop early, i.e., if a scan of the unsorted part has no swaps)

```
def bubbleSort(myList):
```

Question 6. (15 points) Recall the common rehashing strategies we discussed for open-address hashing:

Strategy	**Description**
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)
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) Insert "Paul Gray" and then "Kevin O'Kane" using Linear (on left) and Quadratic (on right) probing.

Hash Table with Linear Probing

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

Hash function

hash(John Doe) = 6

hash(Philip East) = 3

hash(Mark Fienup) = 5

hash(Ben Schafer) = 0

hash(Paul Gray) = 3

hash(Kevin O'Kane) = 3

Hash Table with Quad. Probing

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

b) Indicate below if each rehashing strategy suffers from primary clustering and/or secondary clustering?

- linear probing

- quadratic probing

Question 7. (15 points) The general idea of *Quick sort* is as follows:

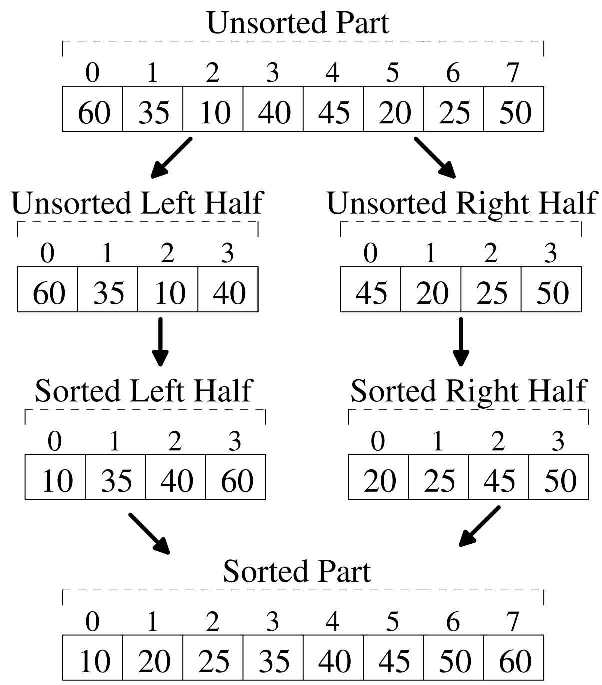
- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (*partitioning*) the unsorted items such that:
- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

	Pivot Index	
All items < to Pivot	Pivot Item	All items >= to Pivot

Explain why the **worst-case** performance is $O(n^2)$.

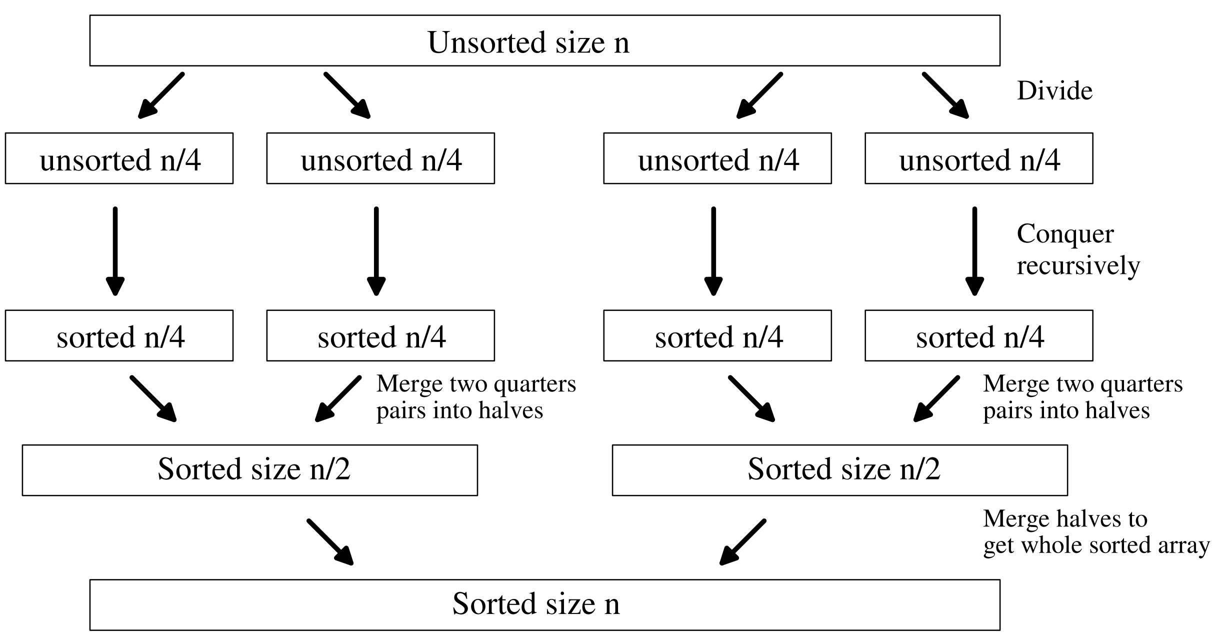
Question 8. (15 points) In class we discussed the **2-way merge sort** below.

```
def merge(alist, lefthalf, righthalf):
    i=j=k=0
    while i<len(lefthalf) and j<len(righthalf):
        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):
        alist[k]=lefthalf[i]
        i=i+1
        k=k+1

    while j<len(righthalf):
        alist[k]=righthalf[j]
        j=j+1
        k=k+1

def mergeSort(alist):
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)
        merge(alist, lefthalf, righthalf)
```



The general idea of **4-way merge sort** is as follows. Assume "n" items to sort.

- Divide the unsorted part into quarters to get four smaller sorting problems of about equal size = n/4
- Conquer/Solve the smaller problems recursively using 4-way merge sort
- "Merge" the solution to the smaller problems together using two levels of merging



Write the Python code for the 4-way merge sort. NOTE: Use the same `merge` code as used as in the 2-way merger sort code given above. Just call the 2-way `merge` three times as shown in the above diagram to merge the four quarters. You do not need to rewrite the `merge` code.