Data Structures - Test 2

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

```python
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.

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

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.

```python
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?

\[
\binom{n-1}{2} + \binom{n-2}{1} + \cdots + 2 + 1 = n \left( \frac{n-1}{2} \right) = \frac{n^2}{2} - \frac{n}{2}
\]

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

\[ (n-1) \times 3 \]

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.,

```
Sorted Part | Unsorted Part
```

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

```python
def bubbleSort(myList):
    for firstUnsorted in range(len(myList)-1):
        for test in range(len(myList)-1, firstUnsorted, -1):
            if myList[test-1] < myList[test]:
                temp = myList[test-1]
                myList[test-1] = myList[test]
                myList[test] = temp
```
Question 6. (15 points) Recall the common rehashing strategies we discussed for open-address hashing:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>Check next spot (counting circularly) for the first available slot, i.e., ((\text{home address} + (\text{rehash attempt} #)) % (\text{hash table size}))</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>Check the square of the attempt-number away for an available slot, i.e., [\text{home address} + \left( \left(\text{rehash attempt} #\right)^2 + (\text{rehash attempt} #) \right)/2 ] % (hash table size), where the hash table size is a power of 2. Integer division is used above</td>
</tr>
</tbody>
</table>

a) Insert "Paul Gray" and then "Kevin O'Kane" using Linear (on left) and Quadratic (on right) probing.

<table>
<thead>
<tr>
<th>Hash Table with Linear Probing</th>
<th>Hash function</th>
<th>Hash Table with Quad. Probing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash(John Doe) = 6</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>hash(Philip East) = 3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>hash(Mark Fienup) = 5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>hash(Ben Schafer) = 0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>hash(Paul Gray) = 3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>hash(Kevin O'Kane) = 3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

b) Indicate below if each rehashing strategy suffers from primary clustering and/or secondary clustering? $3 + \frac{2^k + 2}{2} = 3 + \frac{9}{4} = \frac{9}{4} \\ \checkmark \text{linear probing} \quad \checkmark \text{both}\\ \checkmark \text{quadratic probing} \quad \checkmark \text{only suffers from primary clustering}

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

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

\[ \frac{n^2}{2} \cdot \frac{n}{3} \cdot \frac{2}{n} \leq O(n^2) \]
Question 8. (15 points) In class we discussed the 2-way merge sort below.

```python
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.
def mergeSort4(alist):
    if len(alist) > 4:
        quarter = len(alist) // 4
        firstQ = alist[0:quarter]
        secondQ = alist[quarter:quarter*2]
        thirdQ = alist[quarter*2:quarter*3]
        fourthQ = alist[quarter*3:]
        mergeSort4(firstQ)
        mergeSort4(secondQ)
        mergeSort4(thirdQ)
        mergeSort4(fourthQ)
        leftHalf = alist[0:len(firstQ)+len(secondQ)]
        rightHalf = alist[0:len(thirdQ)+len(fourthQ)]
        merge(leftHalf, firstQ, secondQ)
        merge(rightHalf, thirdQ, fourthQ)
        merge(alist, leftHalf, rightHalf)
else:
    alist.sort()}