Data Structures - Test 2

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

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

q = 10
*6
q + (q*q + 1) + myStr[index]
q + a0 q q q k h e

print("result =", recFn("abcdefghijklmnopqrstuvwxyz", 4))
```

Output:
```
4 7 10 13
result = qqqXYYZkhe
```

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

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

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

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

d) (2 points) What is the maximum height of the run-time stack when calculating G(7) recursively?
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 run 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
```

<table>
<thead>
<tr>
<th>Unsorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>20</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Initial arrangement of list before sorting</th>
<th>insertionSort - at the top of page</th>
<th>insertionSortB - modified version in middle of the page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted in descending order: 10000, 9999, ..., 2, 1</td>
<td>14.0 seconds</td>
<td>12.3 seconds</td>
</tr>
<tr>
<td>Already in ascending order: 1, 2, ..., 9999, 10000</td>
<td>0.005 seconds</td>
<td>0.004 seconds</td>
</tr>
<tr>
<td>Randomly ordered list of 10000 numbers</td>
<td>7.3 seconds</td>
<td>6.4 seconds</td>
</tr>
</tbody>
</table>

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.**

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.**
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.:

```python
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.,

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

20
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 + \((\text{rehash attempt #})^2 + (\text{rehash attempt #}) \div 2\)] \% (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.

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.

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