Data Structures - Test 2

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

```python
def recFn(myString, index):
    print(index)
    if index == 0:
        return "here"
    elif index == 1:
        return "there"
    elif index > 4:
        return recFn(myString, index-1) + myString[index-2]  
        (***) 'chere'
    else:
        return myString[index] + recFn(myString, index-2)  
        (****) 'here'
print("result =", recFn("abcdefg", 5))
```

```
result = 'chere'
```

Question 2. (10 points) Write a recursive Python function to compute the following mathematical function, G(n):

\[ G(n) = \begin{cases} 
  n & \text{if } n \leq 3 \text{ (e.g., } G(2) \text{ is 2)} \\
  6 & \text{if } n = 4 \\
  G(n-5) + G(n-4) + G(n-2) & \text{for all } n \text{ values } > 4.
\end{cases} \]

```python
def G(n):
    if n <= 3:
        return n
    elif n == 4:
        return 6
    else:
        return G(n-5) + G(n-4) + G(n-2)
```

Question 3. a) (7 points) For the above recursive function G(n), complete the calling-tree for G(9).

b) (2 points) What is the value of G(9)? 19

c) (1 point) What is the maximum height of the run-time stack when calculating G(9) recursively? 4
Question 4. Consider the following simple sorts discussed in class -- all of which sort in ascending order.

```python
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1, 0, -1):
        for testIndex in range(lastUnsortedIndex):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp

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
```

<table>
<thead>
<tr>
<th>Type of sorting algorithm</th>
<th>Initial Ordering of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descending</td>
</tr>
<tr>
<td>bubbleSort.py</td>
<td>23.3</td>
</tr>
<tr>
<td>insertionSort.py</td>
<td>14.2</td>
</tr>
<tr>
<td>selectionSort.py</td>
<td>7.1</td>
</tr>
</tbody>
</table>

a) (5 points) Explain why selectionSort's timings are roughly the same regardless of the initial ordering of items.

Selection sort always needs to compare across the whole unsorted part to find the maxIndex, then always does 3 moves to extend sorted part. Thus, it is relatively unaffected by

d) (5 points) Explain why insertionSort on a descending list (14.2 s) takes longer than insertionSort on a ascending list (0.004 s). Descending order causes insertionSort to scan (right-to-left) and shift right whole sorted part before inserting at index 0. With ascending order only a single comparison to right-most sorted item is needed.

c) (5 points) Explain why bubbleSort is O(n^2) in the worst-case, where n is the size of the list being sorted.

\[
\text{Pairs sum to } n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} = \frac{n(n-1)}{2} + \frac{n^2}{2} = O(n^2)
\]

\[
\text{Total comparisons} = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}
\]
Question 5. (20 points) In class we discussed the insertionSort code shown in question 2 which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list.

For this question write a variation of insertion sort that:
- sorts in **descending order** (largest to smallest), and
- builds the **sorted part on the right-hand side** of the list, i.e.,

<table>
<thead>
<tr>
<th>Unsorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>7 8</td>
<td></td>
</tr>
</tbody>
</table>

```
myList: [0, 1, 2, 25, 40, 90, 60, 50, 45, 35, 20, 10]
```

```python
def insertionSortVariation(myList):
    for lastUnsorted in range(len(myList) - 2, 1, -1):
        itemToInsert = myList[lastUnsorted]
        test = lastUnsorted + 1
        while test < len(myList) and myList[test] < itemToInsert:
            myList[test - 1] = myList[test]
            test += 1
        myList[test - 1] = itemToInsert
```

Question 6. Recall the general idea of Heap sort which uses a min-heap (class BinHeap with methods: BinHeap(), insert(item), delMin(), isEmpty(), size()) to sort a list.

**General idea of Heap sort:**
1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order

```
myList  unsorted list with n items
        |__________________________|
        |                         |
        |__________________________|
        |                         |
        |__________________________|
        |                         |
        |__________________________|
        |                         |
        |__________________________|
```

a) (5 points) Complete the code for heapSort so that it sorts in **descending order**

```python
from bin_heap import BinHeap
def heapSort(myList):
    myHeap = BinHeap()  # Create an empty heap

    for item in myList:
        myHeap.insert(item)

    for index in range(len(myList) - 2, 1, -1):
        myList[index] = myHeap.delMin()
```

b) (5 points) Determine the overall \( O() \) for your heap sort and briefly justify your answer. Let \( n = len(myList) \).

\[ O(n \log_2 n) \]  Each for-loop loops \( n \) times and call a heap method that is \( O(\log_2 n) \), since the height of the heap is at most \( \log_2 n \).
Question 7. 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) (10 points) Insert "Andrew Berns" and then "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.

Hash Table with Linear Probing

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben Schafer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>John Doe</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andrew Berns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Philip East</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Hash Table with Quadratic Probing

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben Schafer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>John Doe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Andrew Berns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mark Fienup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Philip East</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

b) In open-address hashing (like the pictures above), the average number probes/compares for various load factors is:

<table>
<thead>
<tr>
<th>Probing Type</th>
<th>Search outcome</th>
<th>Load Factor (i.e., # items / hash-table size)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td>Linear</td>
<td>unsuccessful</td>
<td>1.39</td>
</tr>
<tr>
<td>Probing</td>
<td>successful</td>
<td>1.17</td>
</tr>
<tr>
<td>Quadratic</td>
<td>unsuccessful</td>
<td>1.37</td>
</tr>
<tr>
<td>Probing</td>
<td>successful</td>
<td>1.16</td>
</tr>
</tbody>
</table>

The "general rule of thumb" tries to keep the load factor (i.e., # items / hash-table size) between 0.5 and 0.67.

- (4 points) Why don’t you want the load factor to exceed 0.67? Over 0.67 the # of probes grows and we want 1 probe ideally.

- (4 points) Why don’t you want the load factor to be less than 0.5? Mostly empty hash table wastes space due to

---

c) (7 points) In closed-address hashing (e.g., ChainingDict picture below), if the load factor (# items / hash table size) is close to 1, say 0.99, would you expect average-case searches of O(1) or O(n)? (Justify your answer).

Yes, with good hash function each list should have about 1 item in it with occasional collision. Thus, 1 or 2 problems is expected.