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

```python
def recFn(myString, index):
    if index == 0:
        return "here"
    elif index == 1:
        return "there"
    elif index > 4:
        return recFn(myString, index-1) + myString[index]
    else:
        return myString[index] + recFn(myString, index-2)

print("result =", recFn("abcdefgh", 5))
```

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

- G(n) = n for all value of n ≤ 3  (e.g., G(2) is 2)
- G(4) = 6 if n = 4
- G(n) = G(n-5) + G(n-4) + G(n-2)  for all n values > 4.

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

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

```
G(9)  
/   
G(4)  G(5)  G(7)
```

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

c) (1 point) What is the maximum height of the run-time stack when calculating G(9) recursively?
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
```

Timings of Above Sorting Algorithms on 10,000 items (seconds)

<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.

b) (5 points) Explain why insertionSort on a descending list (14.2 s) takes longer than insertionSort on a ascending list (0.004 s).

c) (5 points) Explain why bubbleSort is $O(n^2)$ in the worst-case, where n is the size of the list being sorted.
Question 5. (20 points) In class we discussed the insertionSort code shown in question 3 on page 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:
1. sorts in **descending order** (largest to smallest), and
2. 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>myList:</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>25 40 90 60 50 45 35 20 10</td>
</tr>
</tbody>
</table>

```python
def insertionSortVariation(myList):
```

---

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

```python
from bin_heap import BinHeap

def heapSort(myList):
    myHeap = BinHeap()  # Create an empty heap
    
    for item in myList:
        myHeap.insert(item)
    
    sorted_list = []
    while not myHeap.isEmpty():
        sorted_list.append(myHeap.delMin())

    return sorted_list
```

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

---
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.

<table>
<thead>
<tr>
<th>Hash Table with Linear Probing</th>
<th>Hash function</th>
<th>Hash Table with Quadratic Probing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ben Schafer</td>
<td>hash(John Doe) = 2</td>
</tr>
<tr>
<td>1</td>
<td>John Doe</td>
<td>hash(Philip East) = 5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>hash(Mark Fienup) = 4</td>
</tr>
<tr>
<td>3</td>
<td>Philip East</td>
<td>hash(Ben Schafer) = 1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>hash(Andrew Berns) = 1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>hash(Sarah Diesburg) = 4</td>
</tr>
</tbody>
</table>

(b) In open-address hashing (like the pictures above), the average number probes/comparisons 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>Linear Probing</td>
<td>unsuccessful</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td></td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>unsuccessful</td>
<td>1.37</td>
</tr>
<tr>
<td></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?
- (4 points) Why don’t you want the load factor to be less than 0.5?

(c) (7 points) In closed-address hashing (e.g., ChainingDict picture below), if the load factor is close to 1, say 0.99, would you expect the average-case number of probes/comparisons to be between 1 and 2? (Justify your answer).

The ChainingDict object holds 6 items in a hash table of capacity 8, with a size of 4. It contains UnorderList objects containing Entry objects.