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

```python
def recFn(a, b):
    print(a, b)
    if a == b:
        return 100
    elif a > b:
        return 10
    else:
        return recFn(a + 1, b - 2) + a

print("result =", recFn(1, 9))
```

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

- \( G(n) = n \) for all values of \( n \leq 1 \)
- \( G(2) = 5 \) if \( n = 2 \)
- \( G(n) = G(n-5) + G(n-3) + G(n-2) \) for all \( n \) values > 2.

```python
def G(n):
    # Your code goes here
```

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

b) (2 points) What is the value of \( G(7) \)?

c) (1 point) What is the maximum height of the run-time stack when calculating \( G(7) \) recursively?
Question 3. The insertion sort code discussed in class is:

```python
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
```

Consider the following `insertMergeSort` code which calls the above `insertionSort` code twice with copies of each half of the array, and then merges the two sorted halves back together using the merge code from merge sort.

```python
def insertMergeSort(aList):
    halfSize = len(aList) // 2
    lefthalf = aList[:halfSize]
    righthalf = aList[halfSize:]
    insertionSort(lefthalf)
    insertionSort(righthalf)

    ### BELOW IS THE MERGE CODE FROM MERGE SORT ###
    i=0  # index into lefthalf
    j=0  # index into righthalf
    k=0  # index into aList
    while i<len(lefthalf) and j<len(righthalf):  # compare and copy until one half runs out
        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):       # copy the remaining items from lefthalf if any
        aList[k]=lefthalf[i]
        i=i+1
        k=k+1
    while j<len(righthalf):       # copy the remaining items from righthalf if any
        aList[k]=righthalf[j]
        j=j+1
        k=k+1
```

Consider the following timing of `insertionSort` vs. `insertMergeSort` 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>insertMergeSort - modified version in middle of the page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted in descending order: 10000, 9999, ..., 2, 1</td>
<td>14.3 seconds</td>
<td>7.1 seconds</td>
</tr>
<tr>
<td>Already in ascending order: 1, 2, ..., 9999, 10000</td>
<td>0.005 seconds</td>
<td>0.009 seconds</td>
</tr>
<tr>
<td>Randomly ordered list of 10000 numbers</td>
<td>7.4 seconds</td>
<td>3.6 seconds</td>
</tr>
</tbody>
</table>

a) (10 points) Explain why `insertMergeSort` (modified version in middle of page) out performs the original `insertionSort`.

b) (10 points) In either version, why does sorting the randomly order list take about halve the time of sorting the initially descending ordered list?
Question 4. (20 points) In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the below example) and "inserts" it into the sorted part of the list "at the correct spot."

```
10 20 35 40 45 60 25 50 90
```

In class we discussed the following insertion sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list, i.e.:

```
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
```

For this question write a variation of the above insertion sort that:

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

```
    Unsorted Part                     Sorted Part
  0 1 2 3 4 5 6 7 8                   25 40 90 60 50 45 35 20 10
```

```
def insertionSortVariation(myList):
```
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 } \#)/2\] % (hash table size), where the hash table size is a power of 2. Integer division is used above |

a) (8 points) Insert “Paul Gray” and then “Sarah Diesburg” using Linear (on left) and Quadratic (on right) probing.

Hash Table with Linear Probing

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

Hash function

- hash(John Doe) = 7
- hash(Philip East) = 3
- hash(Mark Fienup) = 6
- hash(Ben Schafer) = 0
- hash(Paul Gray) = 6
- hash(Sarah Diesburg) = 7

Hash Table with Quad. Probing

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

b) (7 points) Explain why both linear and quadratic probing both suffer from primary clustering?

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

```
from bin_heap import BinHeap

def heapSort(myList):
    myHeap = BinHeap()  # Create an empty heap
```

a) (10 points) Complete the code for `heapSort` so that it sorts in descending order

b) (5 points) Determine the overall `O( )` for your heap sort and briefly justify your answer.