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

```python
def recFn(x, y):
    print( x, y )
    if x <= y:
        return x
    else:
        return x + recFn(x // 10, y + 1) + y
print("Result = ", recFn(1000, 2))
```

Question 2. (8 points) Write a recursive Python function to compute the following mathematical function, G(n):

\[ G(n) = n \text{ for all values of } n \leq 0 \]

\[ G(n) = G(n-6) + G(n-4) + G(n-2) \text{ for all values of } n > 0. \]

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

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?
Consider the following insertion sort code which sorts in ascending order.

```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
```

a) What is the purpose of the `testIndex >= 0` while-loop comparison?

b) Consider the modified insertion sort code that eliminates the `testIndex >= 0` while-loop comparison.

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

b) Explain how the bolded code in the modified insertion sort code above allows for the elimination of the `testIndex >= 0` while-loop comparison.

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`.

d) In either version, why does sorting the randomly order list take about half the time of sorting the initially descending ordered list?
Question 5. (20 points) Write a variation of selection sort that:
- sorts in ascending order (smallest to largest)
- builds the sorted part on the left-hand side of the list, i.e.,

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
</table>

def selectionSort(myList):
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.,</td>
</tr>
<tr>
<td></td>
<td>(home address + (rehash attempt #)) % (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.,</td>
</tr>
<tr>
<td></td>
<td>[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</td>
</tr>
</tbody>
</table>

a) Insert “Paul Gray” 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 Quad. Probing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Ben Schafer</td>
<td>hash(John Doe) = 6 0 Ben Schafer</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2 Philip East</td>
<td>hash(Philip East) = 3 2 Philip East</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4 Mark Fienup</td>
<td>hash(Mark Fienup) = 5 4 Mark Fienup</td>
<td></td>
</tr>
<tr>
<td>5 John Doe</td>
<td>hash(Ben Schafer) = 0 5 John Doe</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>hash(Paul Gray) = 5 7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hash(Sarah Diesburg) = 6</td>
<td></td>
</tr>
</tbody>
</table>

b) Explain how deletions in an open-address hash table are handled.

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

<table>
<thead>
<tr>
<th>Pivot Index</th>
<th>All items &lt; to Pivot</th>
<th>Pivot Item</th>
<th>All items &gt;= to Pivot</th>
</tr>
</thead>
</table>

Explain why the best-case performance is \( O(n \log_2 n) \).
Question 8. (15 points) Recall the general idea of Heap sort which uses a min-heap (class BinHeap) to sort a list. (BinHeap methods: BinHeap(), insert(item), delMin(), isEmpty(), size())

Generl idea of Heap sort:  

myList  

| unsorted list with n items |

1. Create an empty heap  
2. Insert all n list items into heap  

heap with n items  

3. delMin heap items back to list in sorted order  

myList  

| sorted list with n items |

a) If we insert all of the list elements into a min-heap, what item would we easily be able to determine?

b) Complete the code for heapSort so that it sorts in descending order

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

c) Determine the overall $O(\ )$ for heap sort and justify your answer.