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

```python
def recFn(myString, index):
    if index >= len(myString):
        return ""
    else:
        return recFn(myString, index + 2) + myString[index]

print(recFn("panther", 0))
```

Question 2. (12 points) Write a recursive Python function to compute the following mathematical function, G(n):

- G(0) value is 0
- G(1) value is 1
- G(2) value is 2
- G(n) = G(n-3) + G(n-2) + G(n-1) for all values of n > 2.

```python
def G(n):
    if n <= 2:
        return n
    else:
        return G(n-3) + G(n-2) + G(n-1)
```

Question 3. (13 points) a) For the above recursive function G(n), complete the calling-tree for G(5).

b) What is the value of G(5)? 11

c) What is the maximum height of the run-time stack when calculating G(5) recursively? 4
Question 4. (10 points) Consider the following selection sort code which sorts in ascending order.

```python
def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        # look for maximum item in unsorted part of list
        maxIndex = 0
        # Assume maximum is the first item in the unsorted part
        maxIndex = 0
        # scan the unsorted part of the list to correct the assumption
        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
```

3 moves

a) Let "n" be the number of items in the list. How many total comparisons does the if-statement perform in the selection sort?

\[
\frac{n+1}{2} + \frac{n+3}{2} + \cdots + 3 + 2 + 1 = \frac{n + (n+1) + \cdots + 3}{2} = \frac{n(n+1)}{2}
\]

b) Let "n" be the number of items in the list. How many total item moves are performed in the selection sort?

\[3(n-1) = 3n - 3\]

Question 5. (25 points) Write a variation of bubble sort that:
- sorts in descending order (largest to smallest)
- 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>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

Inner loop scans from right to left, across the unsorted part swapping adjacent items that are "out of order"

(Your code does NOT need to stop early if a scan of the unsorted part has no swaps)

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

```
```

```
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., ((\text{home address} + (\text{rehash attempt} #)) \mod \text{ (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., ([\text{home address} + ((\text{rehash attempt} #)^2 + (\text{rehash attempt} #))/2] \mod \text{ (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 “Kevin O’Kane” using Linear (on left) and Quadratic (on right) probing.

b) Indicate below if each rehashing strategy suffers from primary clustering and/or secondary clustering:

- linear probing  **suffers from both primary and secondary clustering**
- quadratic probing  **suffers only from primary clustering**

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

Explain why the worst-case performance is **\(O(n^2)\)**.

Partition loops across unsorted part

If pivot always at end, then worst-case

\[
\frac{n(n-1)}{2} = O(n^2)
\]