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

```python
def recFn(myStr, index):
    if index >= len(myStr):
        return "WOW"
    else:
        myStr[0] + recFn(myStr, index + 3) + myStr[index]
    print("result =", recFn("abcdefghijklmnopqrstuvwxyz", 3))
```

Output:
```
result = qaqWOWjgd
```

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

\[ G(n) = n \text{ for all values of } n \leq 2 \text{ (e.g., } G(1) \text{ value is 1) } \]
\[ G(n) = G(n-3) + G(n-1) \text{ for all values of } n > 2. \]

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

b) (8 points) For the above recursive function G(n), complete the calling-tree for G(6).

c) (3 points) What is the value of G(6)?

d) (2 points) What is the maximum height of the run-time stack when calculating G(6) recursively?
Question 3. (15 points) 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):
        alreadySorted = True
        for testIndex in range(lastUnsortedIndex):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
                alreadySorted = False
        if alreadySorted:
            return

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>24.5</td>
</tr>
<tr>
<td>insertionSort.py</td>
<td>14.2</td>
</tr>
<tr>
<td>selectionSort.py</td>
<td>7.3</td>
</tr>
</tbody>
</table>

a) Explain why bubbleSort on a descending list (24.5 s) takes longer than bubbleSort on a random list (16.5 s).

**Descending order causes the if-statement condition to always be True, so it always swaps and never stops early. Random order might not always swap and might stop early.**

b) Explain why bubbleSort on a descending list (24.5 s) takes longer than insertionSort on a descending list (14.2 s).

**Worst case for both: bubble sort compares and swaps down whole unsorted part, and insertion compares and shifts items up down whole sorted part. Thus, same # of comparisons, but each bubble sort swap involves 3 moves, while each insertion sort shift takes only one move.**

c) Explain why insertionSort on a descending list (14.2 s) takes longer than selectionSort on a descending list (7.3 s).

**Same number of comparisons for both. Selection only does 3 moves (1 swap) to extend the sorted part by one, while insertion sort must shift whole sorted part.**
Question 4. In class we developed the following selection sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list, i.e.:

```
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
```

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

```
def selectionSortVariation(myList):
    for firstUnsortedIndex in range(0, len(myList)-1):
        maxIndex = firstUnsortedIndex
        for testIndex in range(firstUnsortedIndex+1, len(myList)):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        temp = aList[firstUnsortedIndex]
        aList[firstUnsortedIndex] = aList[maxIndex]
        aList[maxIndex] = temp
```

```
Question 5. Recall the quadratic rehashing strategy we discussed for open-address hashing:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>[\text{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</td>
</tr>
</tbody>
</table>

(a) (8 points) 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) = 7</td>
<td>0 Ben Schafer</td>
</tr>
<tr>
<td>1 Paul Gray</td>
<td>hash(Philip East) = 3</td>
<td>1 Paul Gray</td>
</tr>
<tr>
<td>2 Sarah Diesburg</td>
<td>hash(Mark Fienup) = 6</td>
<td>2 Sarah Diesburg</td>
</tr>
<tr>
<td>3 Philip East</td>
<td>hash(Ben Schafer) = 0</td>
<td>3 Philip East</td>
</tr>
<tr>
<td>4</td>
<td>hash(Paul Gray) = 7</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>hash(Sarah Diesburg) = 6</td>
<td>5</td>
</tr>
<tr>
<td>6 Mark Fienup</td>
<td></td>
<td>6 Mark Fienup</td>
</tr>
<tr>
<td>7 John Doe</td>
<td></td>
<td>7 John Doe</td>
</tr>
</tbody>
</table>

\[7 + \left(\frac{2+1}{2}\right) \% 8 = 2\]

\[6 + \left(\frac{2+1}{2}\right) \% 8 = 1\]

(b) (7 points) What is the purpose of requiring a hash table size that is a power of 2 when using quadratic probing?

So quadratic probing rehashes to every slot in the hash table before repeating.

Question 6. (15 points) Use the below diagram to explain the worst-case big-oh notation of merge sort. Assume “n” items to sort.

\[O(n \log_2 n)\]

(\(n = \frac{\text{values}}{2} = (n-1)\) \text{ all between } \frac{n}{2} \text{ and } (n-1)\)