Data Structures - Test 2

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

```
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))
```

**Output:**

```
1000 2
100 3
10 4
1 5
Result = 1120
```

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

\[
G(n) = \begin{cases} 
    n & \text{for all value of } n \leq 0 \\
    G(n-6) + G(n-4) + G(n-2) & \text{for all values of } n > 0.
\end{cases}
\]

```
def G(n):
    if n <= 0:
        return n
    else:
        return G(n-6) + G(n-4) + G(n-2)
```

**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)?**  14

**c) What is the maximum height of the run-time stack when calculating G(6) recursively?**  4

\[ \]
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?

To avoid running off the left end of the list when scanning the sorted part from right-to-left.

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
        firstUnsortedIndex = 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.

The bolded code places the minimum list item at index 0, so the remaining while-condition must be false when `testIndex` is 0. Thus, the while cannot run at the left end of the list.

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`.

The bold code is O(n), but it replaces the "testIndex >= 0" check which is O(n).

d) In either version, why does sorting the randomly order list take about half the time of sorting the initially descending ordered list?

We expect to insert a random item about halfway down the sorted part on average, but in descending order we must insert at spot 0.
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>

```python
def selectionSort(myList):
    for firstUnsorted in range(len(myList)-1):
        minIndex = firstUnsorted
        for testIndex in range(firstUnsorted+1, len(myList)):
            if myList[testIndex] < myList[minIndex]:
                minIndex = testIndex
        temp = myList[firstUnsorted]
        myList[firstUnsorted] = myList[minIndex]
        myList[minIndex] = 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.,&lt;br&gt; (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.,&lt;br&gt; [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.

b) Explain how deletions in an open-address hash table are handled. Deletions keys are replaced by a "DELETED" marker (e.g. "True") to indicate that it once held a key. The "DELETED" marker means continue to search, but can be replaced on an insertion.

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 best-case performance is $O(n \log_2 n)$.

Ideally (best-case) the pivot item divides the unsorted part in half, so we have $\log_2 n$ levels. At each level, all the partitioning combine take $O(n)$ amount of work, so overall $O(n \log_2 n)$ work per level.
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())

**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

<table>
<thead>
<tr>
<th>myList</th>
<th>unsorted list with n items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--------------------------------</td>
</tr>
<tr>
<td></td>
<td>heap with n items</td>
</tr>
<tr>
<td></td>
<td>--------------------------------</td>
</tr>
<tr>
<td>myList</td>
<td>sorted list with n items</td>
</tr>
</tbody>
</table>

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

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()

    for item in myList:
        myHeap.insert(item)

    for index in range(len(myList)-1, -1, -1):
        myHeap[index] = myHeap.delMin()
```

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

5

- **Step #1:** $O(1)$
- **Step #2:** loops n times with each insert taking $O(n \log n)$ since that's the max height of the heap. Thus, $O(n \log n)$ for step #2
- **Step #3:** loops n times with each delMin taking $O(\log n)$, so $O(n \log n)$ for step #3