Data Structures - Test 2

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

```
def recFn(a, b):
    print( a, b )
    if  a < 0:
        return 100
    elif b < 0:
        return 1000
    elif a > b:
        (**)
        return a + recFn(a - 3, b - 5)
    else:
        return recFn(a - 1, b - 3) - b
(***)
print("Result = ", recFn(8, 10))

(*)
```

**Output:**

```
10 8
13 3
14 2
17 1
20
Result = 20
```

Question 2.  

a) (12 points) Write a recursive Python function to compute the binomial coefficient using the following recursive definition of $C(n, k)$:

\[
C(n, k) = \begin{cases} 
C(n-1, k-1) + C(n-1, k) & \text{for } 1 \leq k \leq (n-1), \\
C(n, k) = 1 & \text{for } k = 0 \text{ or } k = n
\end{cases}
\]

```
def C(n,k):
    ...
```

b) (8 points) For the above recursive function C(n,k), complete the calling-tree for C(4,2).

```
C(4,2)
  /
C(3,1)  C(3,2)
```

c) (3 points) What is the value of C(4,2)?

d) (2 points) What is the maximum number of call-frames of C on the run-time stack when calculating C(4,2) recursively?
Question 3. (15 points) Consider the following simple sorts discussed in class -- all of which sort in ascending order.

```
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1, 0, -1):
        for testIndex in range(lastUnsortedIndex):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
```

```
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>23.3</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 (23.3 s) takes longer than bubbleSort on an ascending list (7.7 s).

b) Explain why bubbleSort on a descending list (23.3 s) takes longer than insertionSort on a descending list (14.2 s).

c) Explain why selectionSort is $O(n^2)$ in the worst-case.
Question 4. 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 + ( (rehash attempt #)^2 + (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 “Andrew Berns” and then “Sarah Diesburg” using Linear (on left) and Quadratic (on right) probing.

Hash Table with Linear Probing

| 0 | Ben Schafer |
| 1 | Philip East |
| 2 | Mark Fienup |
| 3 | John Doe |

Hash Table with Quadratic Probing

| 0 | Ben Schafer |
| 1 | Philip East |
| 2 | Mark Fienup |
| 3 | John Doe |

Hash function

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

b) In open-address hashing (like the pictures above), the average number probes/compares for various load factors is:

<table>
<thead>
<tr>
<th>Probing Type</th>
<th>Search outcome</th>
<th>Load Factor, λ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>unsuccessful</td>
<td>0.25</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>unsuccessful</td>
<td>0.25</td>
</tr>
<tr>
<td>Linear Probing</td>
<td>successful</td>
<td>0.5</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>successful</td>
<td>0.5</td>
</tr>
<tr>
<td>Linear Probing</td>
<td>successful</td>
<td>0.67</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>successful</td>
<td>0.67</td>
</tr>
<tr>
<td>Linear Probing</td>
<td>successful</td>
<td>0.8</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>successful</td>
<td>0.8</td>
</tr>
<tr>
<td>Linear Probing</td>
<td>successful</td>
<td>0.99</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>successful</td>
<td>0.99</td>
</tr>
</tbody>
</table>

The "general rule of thumb" tries to keep the load factor (i.e., # items / hash-table size) between 0.5 and 0.67.

- (4 points) Why don’t you want the load factor to exceed 0.67?

- (3 points) Why don’t you want the load factor to be less than 0.5?

c) (5 points) In closed-address hashing (e.g., ChainingDict picture to the right) if the load factor (# items / hash table size) is 10, what would you expect for the average number of probes/compares of a successful search? (Justify your answer)
Question 5. (20 points) In class we discussed the bubbleSort code shown in question 3 on page 2 which sorts in
ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list.

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

```
Sorted Part     Unsorted Part
```

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

```python
def bubbleSortVariation(myList):
```

Question 6. Recall the general idea of Quick sort:
- Partition by selecting a pivot item at "random" and then 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>
</tr>
</thead>
<tbody>
<tr>
<td>All items &lt; to Pivot</td>
</tr>
</tbody>
</table>

a) (5 points) Explain why quick sort is $O(n \log n)$ when sorting initially randomly ordered items. ($n$ is the len(myList))

b) (5 points) Explain why quick sort is $O(n^2)$ is the worst-case. ($n$ is the len(myList))