Data Structures - Test 2

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

```python
def recFn(a, b):
    print(a, b)
    if a < 0:
        return 100
    elif b < 0:
        return 1000
    elif a > b:
        return recFn(a - 3, b - 5)
    else:
        return recFn(a - 1, b - 3) - b

print("Result = ", recFn(8, 10))
```

Output:

```
8 10
7 7
6 4
3 1
Result = 989
```

Question 2. a) (12 points) Write a recursive Python function to compute the binomial coefficient using the following recursive definition of $C(n, k)$:

$C(n, k) = C(n-1, k-1) + C(n-1, k)$

for $1 \leq k \leq (n-1)$, and

$C(n, k) = 1$

for $k = 0$ or $k = n$

```python
def C(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return C(n-1, k-1) + C(n-1, k)
```

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

![Calling-tree for C(4,2)]

C(4,2) = C(3,1) + C(3,2)

C(3,1) = C(2,0) + C(2,1)

C(3,2) = C(2,1) + C(2,2)

C(2,0) = C(1,0)

C(2,1) = C(1,0) + C(1,1)

C(2,2) = C(1,0) + C(1,1)

C(1,0) = C(0,0)

C(1,1) = C(0,1)

C(0,0) = 1

C(0,1) = 1

c) (3 points) What is the value of $C(4,2)$? 6

d) (2 points) What is the maximum number of call-frames of $C$ on the run-time stack when calculating $C(4,2)$ recursively? 4
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):
        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>Descending</th>
<th>Ascending</th>
<th>Random order</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubbleSort.py</td>
<td>23.3</td>
<td>7.7</td>
<td>15.8</td>
</tr>
<tr>
<td>insertionSort.py</td>
<td>14.2</td>
<td>0.004</td>
<td>7.3</td>
</tr>
<tr>
<td>selectionSort.py</td>
<td>7.3</td>
<td>7.7</td>
<td>6.8</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).

Same # of comparisons, but ascending order will never find any items to swap. Descending order will always swap which is why it takes longer.

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

Insertion comparer and shifts across whole sorted part (1 move/shift), but bubble compares and swaps (3 moves/swap)

b) Summary: same # comparer, but insertion does 1/3 fewer moves so its faster.

c) Explain why selectionSort is $O(n^2)$ in the worst-case.

Selection: n-1 comparers to find max + 3 moves to swap

i) Selection: n-1 comparers to find max

n! comparers: (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n + n + ... + n

n^2/2 pairs

$O(n^2)$
Question 4. Two common rehashing strategies for open-address hashing are linear probing and quadratic probing:

- **Linear Probing**
  - 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.

a) (8 points) Insert “Andrew Berns” and then “Sarah Diesburg” using Linear (on left) and Quadratic (on right) probing.

b) (4 points) Why don’t you want the load factor to exceed 0.67? Over a load factor of 0.67 there starts to be more probes.

- (3 points) Why don’t you want the load factor to be less than 0.5? Under a load factor of 0.5, the hash table wastes space since it is less than half used.

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)

If the hash function is doing a good job randomizing keys to home addresses, then each list has about 10 items. We’d need to go about half way down a list on average, so 5 compares would be expected on a successful search.
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):
    for firstUnsortedIndex in range(0, len(myList) - 1):
        for testIndex in range(len(myList) - 1 - firstUnsortedIndex):
            if myList[testIndex - 1] < myList[testIndex]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex - 1]
                myList[testIndex - 1] = temp
```

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

a) (5 points) Explain why quick sort is $O(n \log n)$ when sorting initially randomly ordered items. ($n$ is the $\text{len}(\text{myList})$) For random data we expect the pivot to roughly split the list in about half

```
\log_2 n \approx \frac{n}{\ln(2)} \\
\leq n / \ln(2) \\
\leq 10 \ln(n)
```

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

If the pivot is always picked as the largest value

```
\text{partition}:
\begin{array}{c}
\underbrace{\ldots}_{\text{pivot} = \text{max}(	ext{myList})}
\end{array}
```

```
\begin{array}{c}
\ldots
\end{array}
```

```
\begin{array}{c}
\underbrace{\ldots}_{\text{pivot} = \text{min}(	ext{myList})}
\end{array}
```

```
\begin{array}{c}
\ldots
\end{array}
```