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 < b:
        return a
    elif a == b:
        return a + b
    else:
        return a + recFn(a - 2, b + 1) - b

print("Result = ", recFn(8, 0))
```

```
8 0
6 1
4 2
2 3
Result = 17
```

Question 2. Write a recursive Python function to calculate \(a^n\) (where \(n\) is an integer) based on the formulas:

\[
\begin{align*}
a^0 &= 1, & \text{for } n &= 0 \\
a^1 &= a, & \text{for } n &= 1 \\
a^n &= a^{n/2}a^{n/2}, & \text{for even } n \geq 1 \quad (\text{recall we can check for this in Python by } n \% 2 == 0) \\
a^n &= a^{(n-1)/2}a^{(n-1)/2}a, & \text{for odd } n > 1
\end{align*}
\]

a) (8 points) Complete the below `powerOf` recursive function

```python
def powerOf(a, n):
    if n == 0:
        return 1
    elif n == 1:
        return a
    elif n % 2 == 0:
        return powerOf(a, n//2) * powerOf(a, n//2)
    else:
        return powerOf(a, (n-1)//2) * powerOf(a, (n-1)//2) * a
```

b) (7 points) For the above recursive `powerOf` function, complete the calling-tree for `powerOf(2, 6).

```
8 9
  / \
2    powerOf(2, 6)
     / \
   2    powerOf(2, 3)
      /    a=2
  powerOf(2, 1)  powerOf(2, 1)

(6, 2) = 2*2
(3, 2) = 2*2
```

c) (5 points) Suggest a way to speedup the above `powerOf` function.

You could do dynamic programming or avoid the duplicate recursive calls: (8, 0) == 0.

```python
def powerOf(a, n):
    if n % 2 == 0:
        return powerOf(a, n//2) * powerOf(a, n//2)
    else:
        return (powerOf(a, (n-1)//2) * 2) * a
```

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) (10 points) Insert "Paul Gray" and then "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.

Hash Table with Linear Probing

| 0 | Ben Schafer |
| 1 |
| 2 |
| 3 | Philip East |
| 4 | Paul Gray |
| 5 | Sarah Diesburg |
| 6 | Mark Fienup |
| 7 | John Doe |

Hash Table with Quad. Probing

| 0 | Ben Schafer |
| 1 | Sarah Diesburg |
| 2 |
| 3 | Philip East |
| 4 | Paul Gray |
| 5 | Mark Fienup |
| 6 | John Doe |

In lab 7, we inserted the 10000 even values 0, 2, 4, 6, 8, ..., 19996, 19998 is ascending order into various hash tables and then timing searching for the 20000 values 0, 1, 2, 3, 4, 5, ..., 19996, 19997, 19998, 19999. On my office computer the timings for these searches are:

<table>
<thead>
<tr>
<th>Type of Hashing</th>
<th>Timings of 20,000 Searches on Various Hash Table Sizes (Load Factors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open-address with Linear probing</td>
<td>16,384 (0.61) 12.19 seconds 0.064 seconds 0.062 seconds</td>
</tr>
<tr>
<td>Open-address with Quadratic probing</td>
<td>32,768 (0.31) 0.575 seconds 0.064 seconds 0.065 seconds</td>
</tr>
<tr>
<td>Closed-address with unsorted linked list at each home address (i.e., ChainingDict)</td>
<td>65,536 (0.15) 0.128 seconds 0.114 seconds 0.115 seconds</td>
</tr>
</tbody>
</table>

b) (8 points) For load factor 0.61, why did the open-address with Linear probing perform much worse than open-address with Quadratic probing? Because linear probing suffers from secondary clustering, rehashing patterns for home addresses 0, 2, 4, ... have merged into one big cluster. Small odd search (1, 3, 5, ...) rehashes down table to first empty slot at 3617.

Quadratic doesn't suffer secondary cluster so empty found sooner.

c) (7 points) For load factors 0.31 and 0.15, why is the closed-address (e.g., ChainingDict) version slower than the open-address versions? Open-address for these load factors has no rehashing, so keys compared to hash table at home addresses. Close-address must search linked list of size 1, but adds an extra layer of method calls which slows it slightly.
Question 5. (25 points) 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.:

```python
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
```

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.,

```python
def selectionSortVariation(myList):
    for firstUnsortedIndex in range(0, len(aList)-1, 1):
        maxIndex = firstUnsortedIndex
        for testIndex in range(firstUnsortedIndex+1, len(aList), 1):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        temp = aList[firstUnsortedIndex]
        aList[firstUnsortedIndex] = aList[maxIndex]
        aList[maxIndex] = temp
```
Question 6. (10 points) Use the below diagram to explain the worst-case big-oh notation of merge sort. Assume "n" items to sort.

Question 7. (10 points) Quick sort general idea is as follows.
- Select a "random" item in the unsorted part as the pivot
- Rearrange (partitioning) the unsorted items such that →→:
  - All items < to Pivot
  - Pivot Item
  - All items >= to Pivot

Explain how quick sort performs $O(n^2)$ in the worst-case.