Data Structures - Test 2

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

```python
def recFn(a, b):
    if a == b:
        return b
    elif a > b:
        return a
    else:
        return a + recFn(a + 2, b - 2) + b

print("Result = ", recFn(-2, 8))
```

Output:

```
-2 8
0 6
4 2
Result = 22
```

Run-time Stack

---

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

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

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)`.

```
64
```

---

c) (5 points) Suggest a way to speed up the above `powerOf` function. Instead of doing two identical recursive calls, just do one and square its result.

```python
return (powerOf(a, (n-1)/2) * a)**2
```
Question 3. (16 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>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>7.1</td>
</tr>
</tbody>
</table>

a) Explain why insertionSort on a descending list (14.2 s) takes about **twice as long** as insertionSort on a random list (7.3 s). On descending list, itemToInsert is compared across whole sorted part with whole sorted part being shifted. On random list we expect to insert itemToInsert about halfway down sorted part, so it takes about half the time.

b) Explain why bubbleSort on a descending list (23.3 s) takes longer than Sort on a descending list (14.2 s). Bubble sort on descending order needs to swap across whole unsorted part and insertion sort must shift whole of sorted part. Since swapping takes 3 moves per swap while shifting only requires one move, insertion sort is faster. Both do about the same # of compares.

c) Explain why selectionSort is $O(n^2)$ in the worst-case. Selection sort does $O(n^2)$ compares:

To find maxIndex in unsorted part you need:

\[
\begin{align*}
    (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 &= (n-1) + (n-2) + \cdots + 2 + 1 + n \\
    &= \frac{n(n+1)}{2} \\
    &\in O(n^2)
\end{align*}
\]
Question 4. (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 **ascending order** still (smallest to largest), but **+4**
- adds a **check to stop early** if no swap occurs when scanning the unsorted part of the array, AND **+4**
- 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):
        didSwap = False
        for testIndex in range(1, len(myList)-firstUnsortedIndex):
            if myList[testIndex-1] > myList[testIndex]:
                temp = myList[testIndex-1]
                myList[testIndex-1] = myList[testIndex]
                myList[testIndex] = temp
                didSwap = True
        if not didSwap:
            return
```

Question 5. 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>

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

- Selecting a pivot at random from a random list should fall roughly in the middle after partitioning.
- $O(n)$ work to compare & swap at each level.
- $\log n$ levels.

Thus, $O(n \log n)$ for
Question 5. 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](image)

**Hash function**

- hash(John Doe) = 6
- hash(Philip East) = 3
- hash(Mark Fienup) = 4
- hash(Ben Schafer) = 2
- hash(Andrew Berns) = 0
- hash(Sarah Diesburg) = 3

**Hash Table with Quadratic Probing**

- hash(John Doe) = 6
- hash(Philip East) = 3
- hash(Mark Fienup) = 4
- hash(Ben Schafer) = 2
- hash(Andrew Berns) = 0
- hash(Sarah Diesburg) = 3

b) (8 points) Open-address hashing above, uses rehashing (e.g., linear or quadratic probing) when collisions occur. Initially, we used None to indicate that a hash table slot is "empty" and True to indicate that a slot had a "deleted" value. Explain why empty and deleted slots are treated differently.

(Consider the linear probing example in part (a). If we delete "Andrew Berns" and then search for "Sarah Diesburg", we need some way at slot 5 to know we cannot stop searching which is the reason for a "deleted" value.)

(Use more spec)

b) (8 points) Briefly describe how closed-address hashing (e.g., ChainingDict) handles deletions.

In closed-address hashing we have a data structure at each slot in the hash table to handle collisions (i.e., store all values that hash to that slot/home address).

For deletion of an item, we

1. hash the item to get the home addr.
2. delete the item from the data structure at that slot/home addr.

In the ChainingDict example, the UnorderedList object has a remove method we rely on:

- self. index = hash(item)
- self. table [self. index]. remove (item)