Spring 2015 Name: Mark F.

Data Structures - Test 2

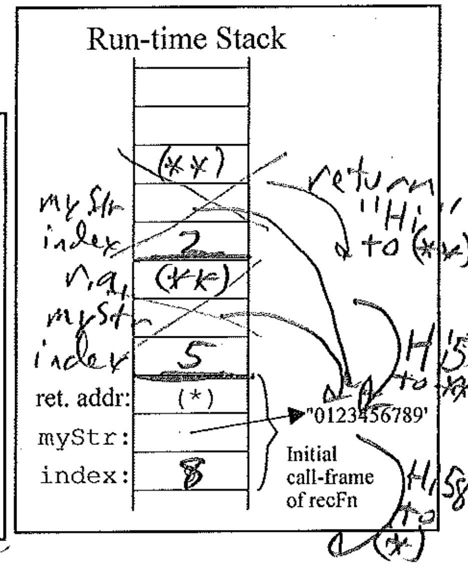
Question 1. (10 points) What is printed by the following program? Output:

```
def recFn(myStr, index):
   print(myStr[index], index)
   if index < 4:
      return "Hi"
    else:
      return recFn(myStr, index - 3) + myStr[index]
                (**)


print("result =",recFn("0123456789", 8))
                   (*)
```

8 8
5 5
2 2
result = Hi58

10



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

$a^0 = 1$, for n = 0
$a^1 = a$, for n = 1
$a^n = a^{n/2}a^{n/2}$, for even n > 1 (recall we can check for this in Python by `n % 2 == 0`)
$a^n = a^{(n-1)/2}a^{(n-1)/2}a$, for odd n > 1

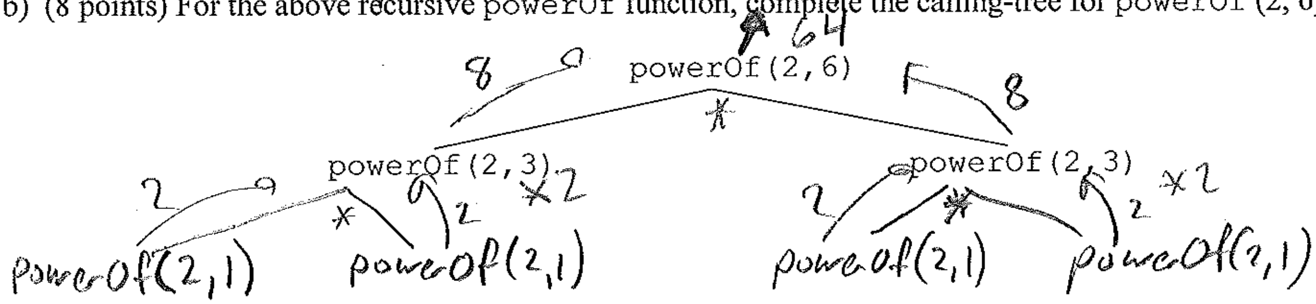
a) (12 points) Complete the below `powerOf` **recursive** function

```
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
```

if stmts +5
recursive calls +5
returns +2

12

b) (8 points) For the above recursive `powerOf` function, complete the calling-tree for `powerOf(2, 6)`.



8

c) (5 points) Suggest a way to speedup the above `powerOf` function. - Don't do two recursive calls that are exactly the same. Instead, do one and square the result. e.g.: else:
temp = powerOf(a,(n-1)//2)
return temp*temp*a

5

35

Question 3. Consider the following insertion sort which sorts in ascending order, but builds the sorted part on the right.

```
def insertionSort(myList):
    myListLength = len(myList)
    for lastUnsortedIndex in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsortedIndex]
        testIndex = lastUnsortedIndex + 1
        while testIndex < myListLength and myList[testIndex] < itemToInsert:
            myList[testIndex-1] = myList[testIndex]
            testIndex = testIndex + 1
        myList[testIndex - 1] = itemToInsert
```

a) (5 points) What is the purpose of the `testIndex < myListLength` while-loop comparison?

When inserting a new largest value into the sorted part, we don't want to access an index off the right end of the list.

Consider the modified insertion sort code that eliminates the `testIndex < myListLength` while-loop comparison, but adds the **bold** code.

```
def insertionSortB(myList):
    myList.append(max(myList))
    for lastUnsortedIndex in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsortedIndex]
        testIndex = lastUnsortedIndex + 1
        while myList[testIndex] < itemToInsert:
            myList[testIndex-1] = myList[testIndex]
            testIndex = testIndex + 1
        myList[testIndex - 1] = itemToInsert
    myList.pop()
```

b) (5 points) Explain how the **bolded** code in the modified insertion sort code above allows for the elimination of the `testIndex < myListLength` while-loop comparison. We temporarily add the largest item to the right end of the list, so the myList[testIndex] < itemToInsert condition will fail when we compare itemToInsert to it.

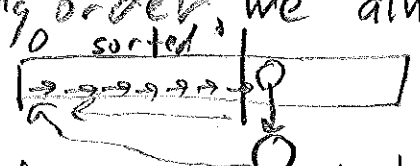
Consider the following timing of the above two insertion sorts on lists of 10000 elements.

Initial arrangement of list before sorting	`insertionSort` - at the top of page	`insertionSortB` - modified version in middle of the page
Sorted in descending order: 10000, 9999, ..., 2, 1	14.1 seconds	12.6 seconds
Already in ascending order: 1, 2, ..., 9999, 10000	0.004 seconds	0.004 seconds
Randomly ordered list of 10000 numbers	7.3 seconds	6.5 seconds

c) (5 points) Explain why `insertionSortB` (modified version in middle of page) out performs the original `insertionSort`. The while-loop has one less condition to check every time it loops, so it is faster.

d) (5 points) In either version, why does sorting the randomly order list take about halve the time of sorting the initially descending ordered list?

In descending order, we always insert at index 0:

(e.g. 

) so whole sorted-part is compared and shifted right one spot.

On random order, we expect the itemToInsert to insert in the middle of the sorted part on average, so only half the compares and moves.

Question 4. (20 points) In class we discussed the following bubble sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list.

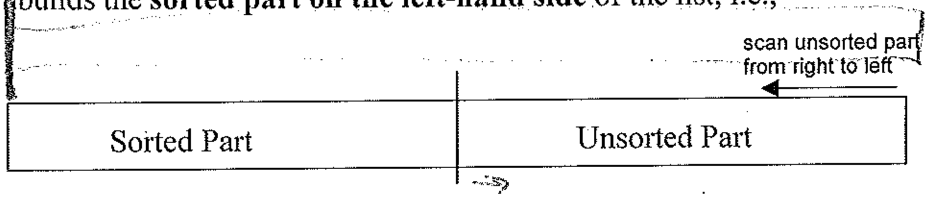
```
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1,0,-1):
        # scan the unsorted part at the beginning of myList
        for testIndex in range(lastUnsortedIndex):

            # if we find two adjacent items out of ascending order, then switch them
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
```

unsorted

For this question write a variation of the above bubble sort that:

- sorts in **descending order** (largest to smallest)
- builds the **sorted part on the left-hand side** of the list, i.e.,




```
def bubbleSortVariation(myList):
    for firstUnsorted in range(0, len(myList)-1):
        for testIndex in range(len(myList)-1, firstUnsorted, -1):
            if myList[testIndex-1] < myList[testIndex]:
                temp = myList[testIndex-1]
                myList[testIndex-1] = myList[testIndex]
                myList[testIndex] = temp
```

Spring 2015 Name: ______________________

Question 5. Recall the common rehashing strategies we discussed for open-address hashing:

Strategy	Description
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 "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 function

hash(John Doe) = 7

hash(Philip East) = 3

hash(Mark Fienup) = 6

hash(Ben Schafer) = 0

hash(Paul Gray) = 3

hash(Sarah Diesburg) = 3

Hash Table with Quad. Probing

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

$\left(3+\frac{(1^2+1)}{2}\right) \% 8 = 4$

$\left(3+\frac{(2^2+2)}{2}\right) \% 8 = 6$

$\left(3+\frac{(3^2+3)}{2}\right) \% 8 = 1$

b) (7 points) Explain why both linear and quadratic probing both suffer from primary clustering?

Both linear + quad. probing rehash formulas are based solely on home addr and rehash attempted #, so all values hashing to a home addr. follow the same rehash pattern which is our definition of primary clustering.

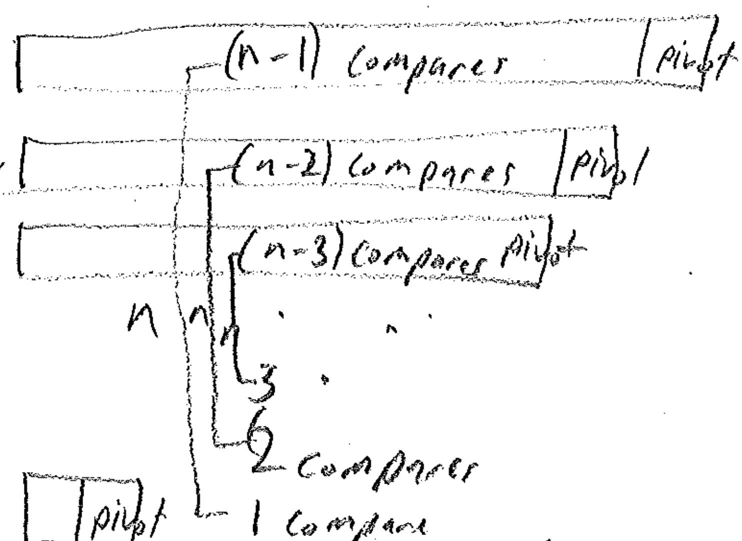
Question 6. (10 points) The general idea of *Quick sort* is as follows:

- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (*partition*) the unsorted items as shown in diagram on right:
- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

Pivot Index

All items < to Pivot	Pivot Item	All items >= to Pivot

Explain why the **worst-case** performance is **$O(n^2)$**. In the worst-case the pivot repeatedly falls on an end



$$\# \text{ compares} = (n-1)+(n-2)+(n-3)+\dots+3+2+1 = \overbrace{n+n+\dots+n}^{\frac{(n-1)}{2} \text{ times}} = n \times \frac{(n-1)}{2} \in O(n^2)$$