Fall 2016 Name: Mark F.

Data Structures - Test 2

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

```
def recFn(myString, index):
    print(index)
    if index == 0:
        return "here"
    elif index == 1:
        return "there"
    else:
        return myString[index] + recFn(myString,index-2)
                      (**)

print("result =",recFn("abcdefgh", 5))
                  (*)
```

Output:

5
3
1
result = fdthere

Run-time Stack

r.a.	(**)
myString	
index	1
r.a.	(**)
myString	
index	3
ret. addr:	(*)
myString:	"abcdefgh"
index:	5

Initial call-frame of recFn

"there" returned to (**); "dthere" to (**); "fdthere" to (*)

10

Question 2. (5 points) Write a recursive Python function to compute the Fibonacci mathematical function, fib(n):

fib(0) = 0 (if n is 0)
fib(1) = 1 (if n is 1)
fib(n) = fib(n-1) + fib(n-2) for all n values > 1.

```
def fib(n):
    if n==0:
        return 0
    elif n==1
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

5

Question 3. a) (7 points) For the above recursive function `fib(n)`, complete the calling-tree for `fib(6)`.

fib(6) → 8
- fib(5) → 5
  - fib(4) → 3
    - fib(3) → 2
      - fib(2) → 1
        - fib(1) → 1
        - fib(0) → 0
      - fib(1) → 1
    - fib(2) → 1
      - fib(1) → 1
      - fib(0) → 0
  - fib(3) → 2
    - fib(2) → 1
      - fib(1) → 1
      - fib(0) → 0
    - fib(1) → 1
- fib(4) → 3
  - fib(3) → 2
    - fib(2) → 1
      - fib(1) → 1
      - fib(0) → 0
    - fib(1) → 1
  - fib(2) → 1
    - fib(1) → 1
    - fib(0) → 0

7

b) (2 point) What is the value of `fib(6)`? 8

c) (6 point) What would be a better approach to calculating a fibonacci value than the above recursive function?

Use dynamic programming, to avoid recalculation of smaller problems:

```
fibonacci = [0,1]
for next in range(2,n+1):
    fibonacci.append(fibonacci[-1] + fibonacci[-2])
```

6

30

Question 4. (20 points.) Consider the following insertion sort code which sorts in ascending order.

```
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
```

a) What is the purpose of the `testIndex >= 0` while-loop comparison? To avoid scanning past left-end of list.

5

b) Consider the modified insertion sort code that eliminates the `testIndex >= 0` while-loop comparison.

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

    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1

        while myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        myList[testIndex + 1] = itemToInsert
```

Explain how the **bold code** in the modified insertion sort code allows for the elimination of the `testIndex >= 0` while-loop comparison.

5 It puts the smallest item in myList at index 0, so the remaining while-loop condition "myList[testIndex] > itemToInsert" is guaranteed to be False before scanning past left-end of list.

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	14.0 seconds	12.3 seconds
Already in ascending order: 1, 2, ..., 9999, 10000	0.005 seconds	0.004 seconds
Randomly ordered list of 10000 numbers	7.3 seconds	6.4 seconds

c) Explain why `insertionSortB` (modified version in middle of page) out performs the original `insertionSort`.

5 One less while-loop condition to check which occurs $O(n^2)$ times in original code at the expense of $O(n)$ to put minimum item at index 0.

d) In either version, why does sorting the initially ascending order list take less time than sorting the initially descending ordered list? In ascending order the inner while-loop does not execute, but in descending order the inner while-loop executes across the whole sorted part.

5

20

Question 5. (25 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):
        alreadySorted = True
        # scan the unsorted part at the beginning of myList
        for testIndex in range(lastUnsortedIndex):
            # if we find two adjacent items out of order, switch them
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
                alreadySorted = False
        if alreadySorted:
            return
```

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

- sorts in **descending order** (largest to smallest),
- stops early if the inner-loop makes no swaps, and
- builds the **sorted part on the left-hand side** of the list. Sample picture after running a while:

Sorted Part								Unsorted Part						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
95	90	80	75	73	68	65	64	25	60	50	15	35	45	10

Inner-loop should "bubble" the largest item down the unsorted part from right to left

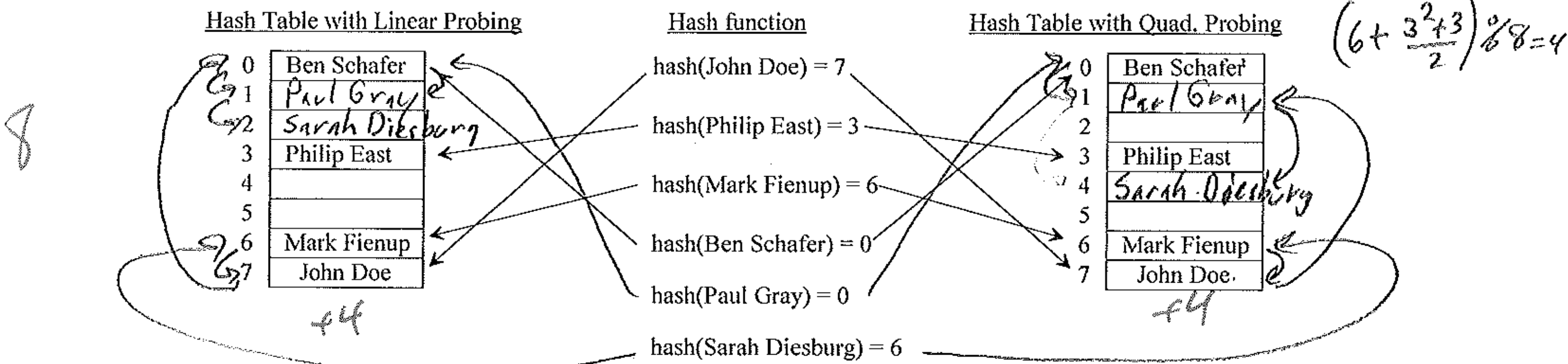
```
def BubbleSortVariation(myList):
    for firstUnsorted in range(0, len(myList)-1):
        alreadySorted = True

        for testIndex in range(len(myList)-1, firstUnsorted, -1):
            if myList[testIndex-1] < myList[testIndex]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex-1]
                myList[testIndex-1] = temp
                alreadySorted = False
        if alreadySorted:
            return
```

Question 6. 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 "Paul Gray" and then "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.



b) (7 points) Explain why linear probing suffers from secondary clustering? (Recall that in *secondary clustering* rehash patterns from initially different home addresses merge together)

Consider two home addresses 5 and 6 for example: Once things with home addr. of 6 rehash to 7, then things with home addr. of 7 or 6 will rehash to 8, 9, 10, ... linearly together.

7

Question 7. *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:
- 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

a) (10 points) Explain how quick sort performs $O(n \log_2 n)$ on random data.

We expect a random pivot item to split the list about in half, so the number of levels should be $O(\log_2 n)$. The partitioning at each level is $O(n)$ work. Overall, $O(n \log_2 n)$

10

25