Spring 2019

Name: Mark F.

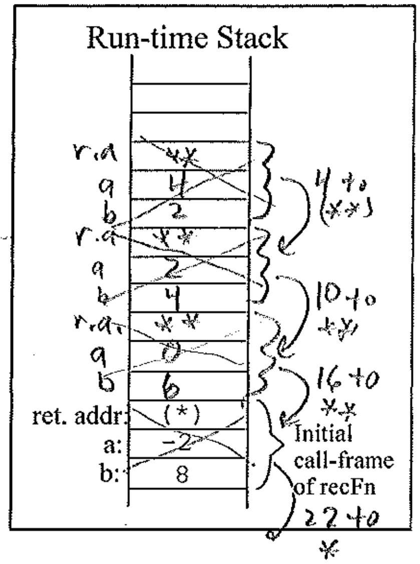
Data Structures - Test 2

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

```
def recFn(a, b):
    print( a, b )
    if  a == b:
        return b
    elif a > b:
        return a
    else:
        return a + recFn(a + 2, b - 2) + b
               2 +   (**)     4          + 4
              -2              18           8
print("Result =", recFn(-2, 8))
                   (*)
```

10

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



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`)

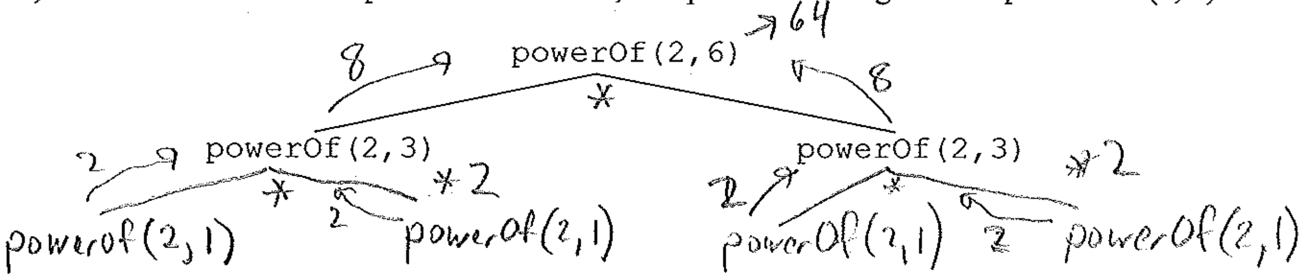
$a^n = a^{(n-1)/2}a^{(n-1)/2}a$, for odd n > 1

a) (8 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
```

8

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



7

5 c) (5 points) Suggest a way to speedup the above `powerOf` function. Instead of doing two identical recursive calls, just do one and square its result, e.g. simple dynamic programming: e.g. else:

return (powerOf(a, (n-1)//2)**2)*a

30

Question 3. (16 points) Consider the following simple sorts discussed in class -- all of which sort in ascending order.

```
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
```

Timings of Above Sorting Algorithms on 10,000 items (seconds)

Type of sorting algorithm	Initial Ordering of Items		
	Descending	Ascending	Random order
bubbleSort.py	23.3	7.7	15.8
insertionSort.py	14.2	0.004	7.3
selectionSort.py	7.1	7.7	6.8

a) Explain why insertionSort on a descending list (14.2 s) takes about **<u>twice as long</u>** 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 half way down sorted part, so it takes about half the time.

5

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

5

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:

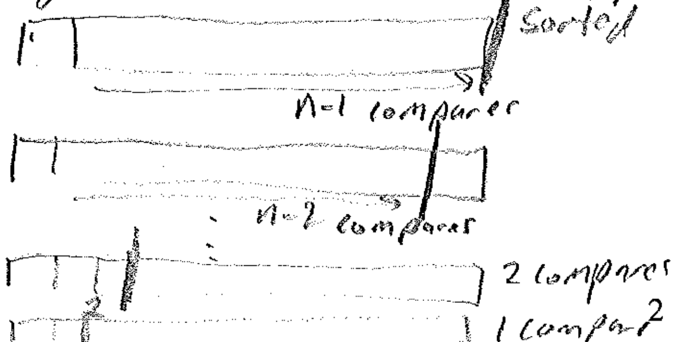
6

$$\overbrace{(n-1)+\underbrace{(n-2)+(n-3)+\cdots+3+2}_{}+1}^{n-2+2=n}$$

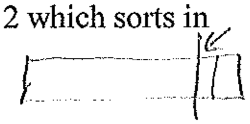
$(n-1+1)=n$

16

$$= \underbrace{n+n+\cdots+n}_{(n-1)/2\ \text{pairs}} = n\frac{(n-1)}{2}\quad O(n^2)$$

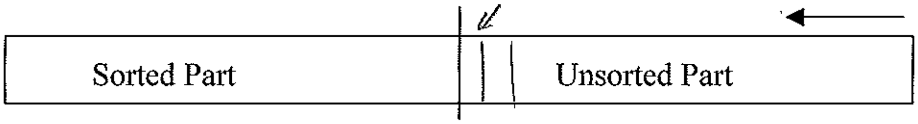


Question 4. (20 points) In class we discussed the bubbleSort code shown in question 3 on page 2 which sorts in ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list.



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.,



Inner loop scans from right to left across the unsorted part swapping adjacent items that are "out of order"

```
def bubbleSortVariation(myList):
    for firstUnsortedIndex in range(0, len(myList)-1, +1):   +4
        didSwap = False
        for testIndex in range(len(myList)-1, firstUnsortedIndex, -1):   +4
            if myList[testIndex-1] > myList[testIndex]:
                temp = myList[testIndex-1]
                myList[testIndex-1] = myList[testIndex]   +5
                myList[testIndex] = temp
                didSwap = True
        if not didSwap:
            return
```

Question 5. Recall the general idea of Quick sort:

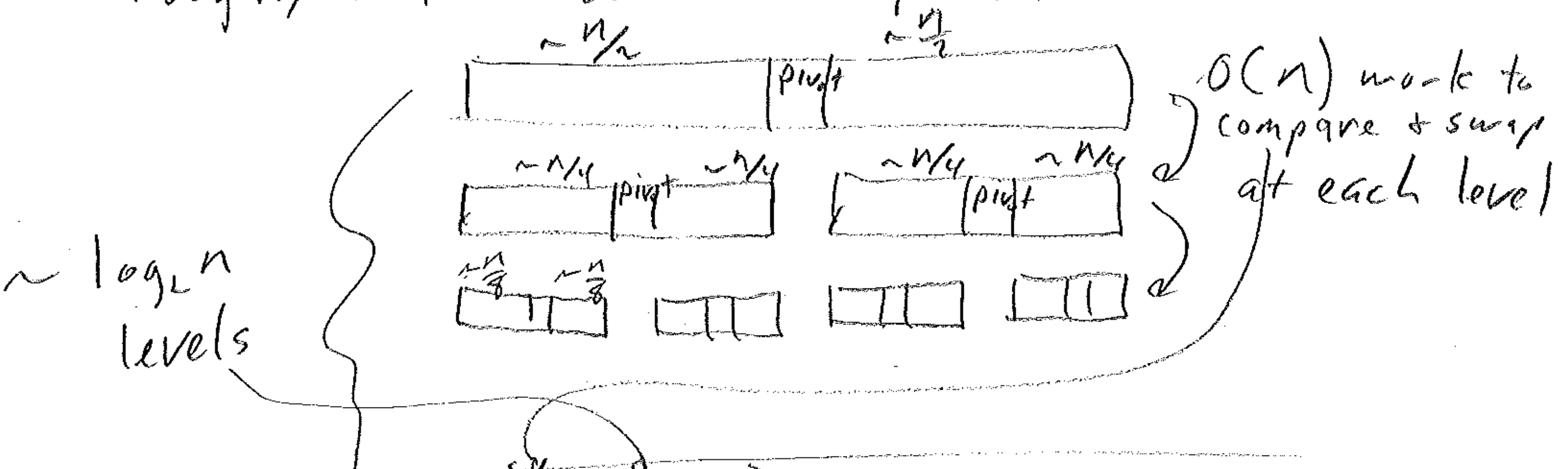
- Partition by selecting a pivot item at "random" and then rearrange (*partitioning*) the unsorted items such that::
- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

Pivot Index

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

(10 points) Explain why quick sort is $O(n \log_2 n)$ 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 partition



Thus, $O(n \log_2 n)$ for

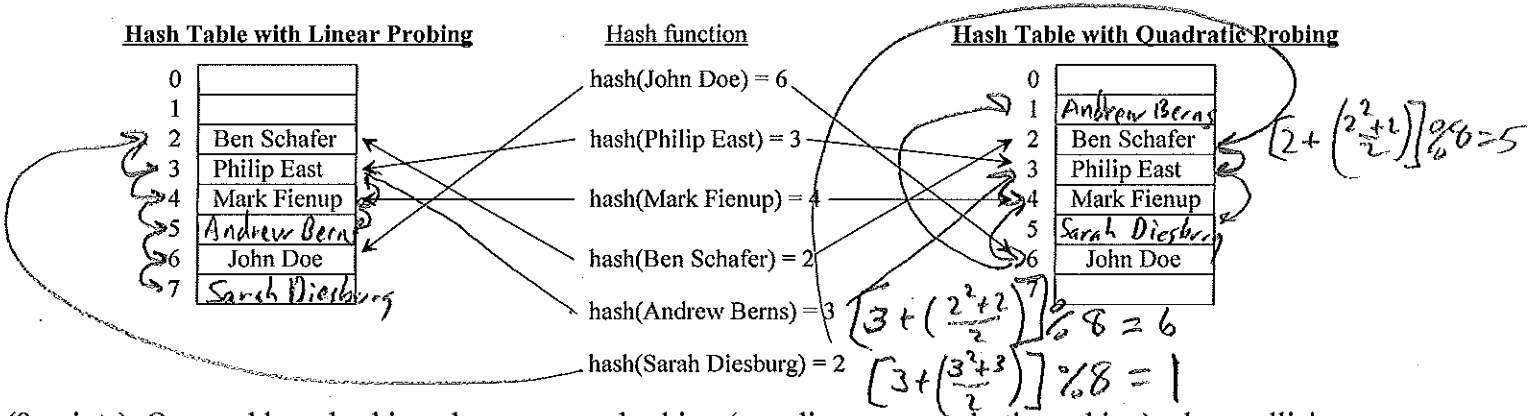
30

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.

8



$\left[2+\left(\frac{2^2+2}{2}\right)\right]\%6=5$

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

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

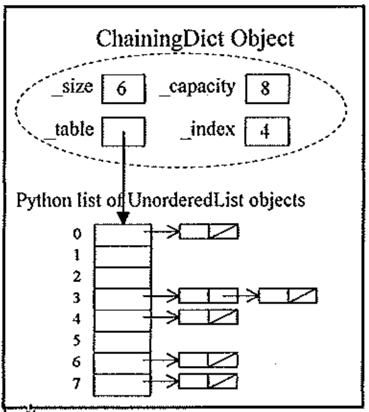
b) (8 points) Open-address hashing above, uses rehashing (e.g., linear or quadratic probing) when collisions occur. Initially, we used None to indicate that a hash table slot is "empty" and True to indicate that a slot had a "deleted" value. Explain why empty and deleted slots are treated differently.

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

8

(More space)

c) (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 (ie., store all valus that hash to that slot/home address).

8

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: ① self._index = hash(item)
② self._table[self._index].remove(item)

24