Spring 2018 Name: Mark F.

Data Structures - Test 2

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

```
def recFn(a, b):
    print( a, b )
    if  a < 0:
        return 100
    elif b < 0:
        return 1000
    elif a > b:
        return a + recFn(a - 3, b - 5)
    else:
        return recFn(a - 1, b - 3) - b
                 (***)

print("Result = ", recFn(8, 10))
                            (*)
```

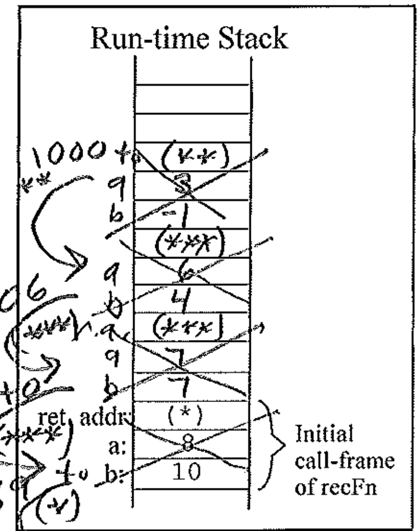
6 (**) + 1000 = 1006

1006 − 7 = 999

999 − 10 = 989

Output:

8 10
7 7
6 4
3 -1
Result = 989

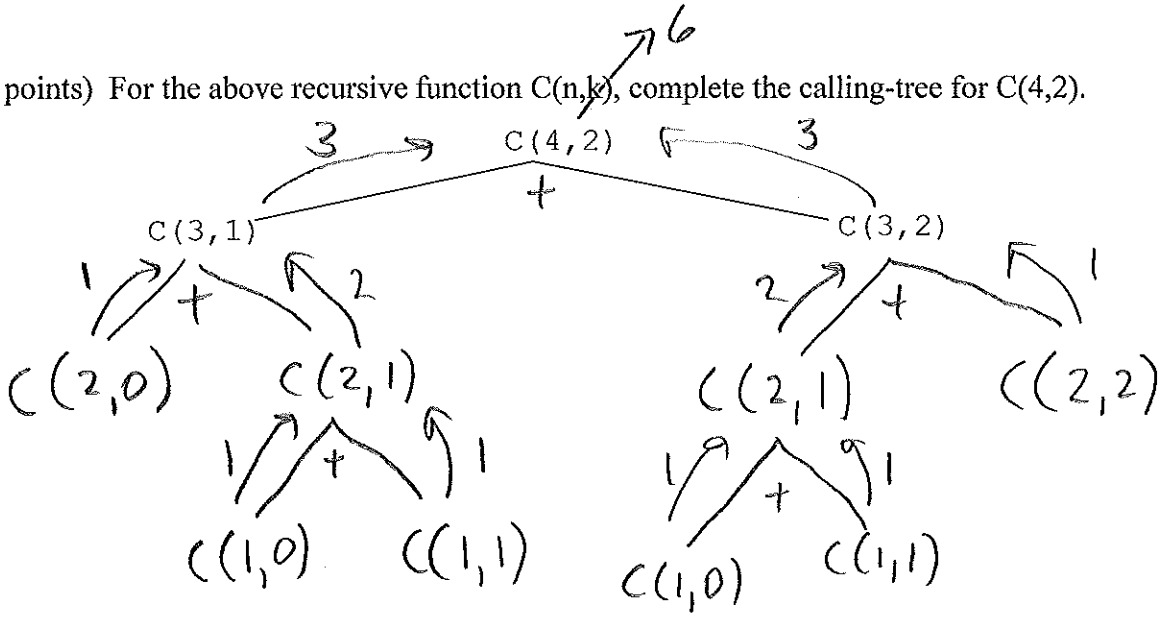


Question 2. a) (12 points) Write a recursive Python function to compute the binomial coefficent using the following recursive definition of $C(n, k)$:

$C(n, k) = C(n-1, k-1) + C(n-1, k)$ for $1 \le k \le (n-1)$, and
$C(n, k) = 1$ for $k = 0$ or $k = n$

```
def C(n,k):
    if k==0 or k==n:
        return 1
    else:
        return C(n-1,k-1) + C(n-1,k)
```

b) (8 points) For the above recursive function C(n,k), complete the calling-tree for C(4,2).



c) (3 points) What is the value of C(4,2)? 6

d) (2 points) What is the maximum number of call-frames of C on the run-time stack when calculating C(4,2) recursively? 4

35

10

Question 3. (15 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	7.3	7.7	6.8

a) Explain why bubbleSort on a descending list (23.3 s) takes longer than bubbleSort on an ascending list (7.7 s).

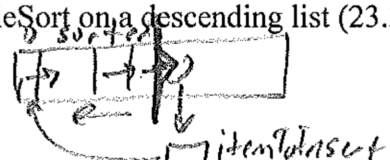
same # of comparison, but ascending order will never find any items to swap. Descending order will always swap which is why it takes longer.

5

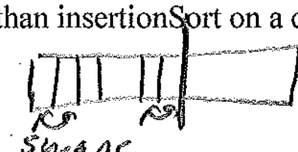
b) Explain why bubbleSort on a descending list (23.3 s) takes longer than insertionSort on a descending list (14.2 s).

5

Insertion



Bubble:



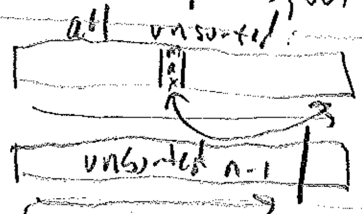
Insertion compares and shifts across whole sorted part (1 move/shift), but bubble compares and swaps (3 moves/swap) across whole unsorted part.

c) Explain why selectionSort is $O(n^2)$ in the worst-case.

b) summary: same # compares, but insertion does ~2/3 fewer moves so its faster.

c) Selection:

5



n-1 compares to find max + 3 moves to swap

n-2 compares to find max

⋮

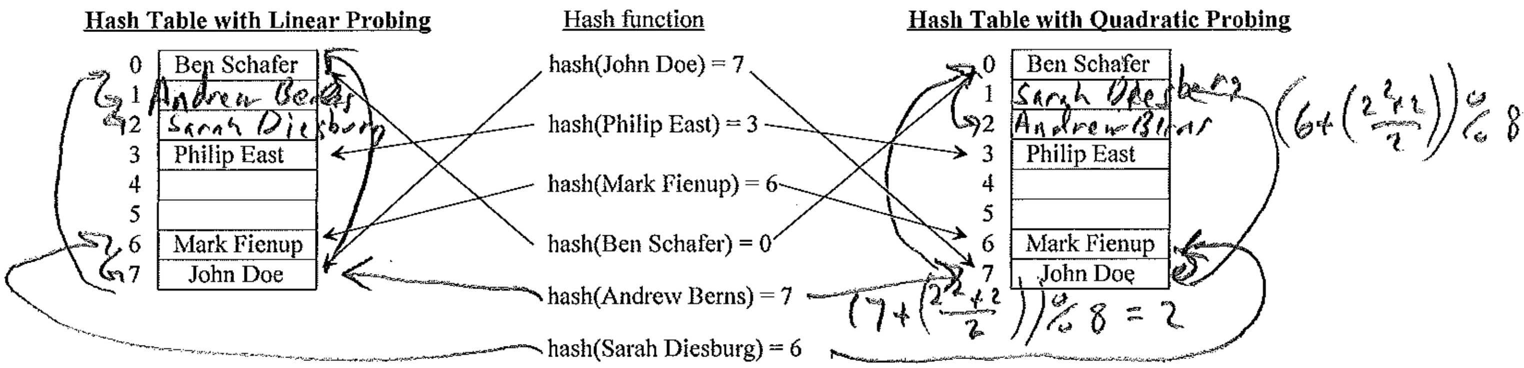
15

compares: $(n-1)+(n-2)+(n-3)+ \cdots +3+2+1 = \underbrace{n+n+\cdots+n}_{\frac{n-1}{2} \text{ pairs}} = n\left(\frac{n-1}{2}\right) = \frac{n^2}{2} - \frac{n}{2}$ $O(n^2)$

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) (8 points) Insert "Andrew Berns" and then "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.



b) In open-address hashing (like the pictures above), the average number probes/compares for various load factors is:

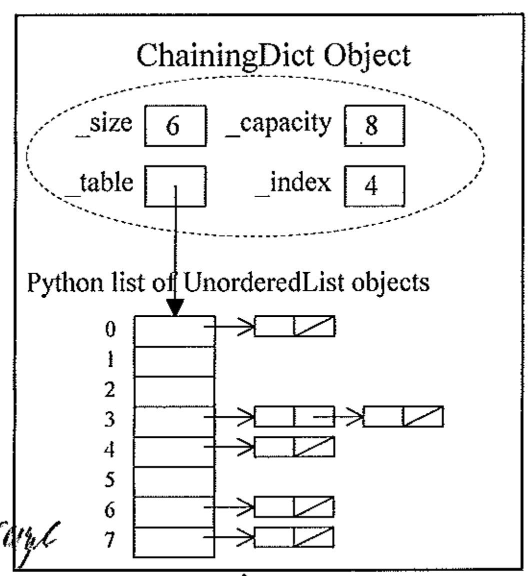
Probing Type	Search outcome	Load Factor, λ 0.25	0.5	0.67	0.8	0.99
Linear Probing	unsuccessful	1.39	2.50	5.09	13.00	5000.50
	successful	1.17	1.50	2.02	3.00	50.50
Quadratic Probing	unsuccessful	1.37	2.19	3.47	5.81	103.62
	successful	1.16	1.44	1.77	2.21	5.11

The "general rule of thumb" tries to keep the load factor between 0.5 and 0.67.

- (4 points) Why don't you want the load factor to exceed 0.67? Over a load factor of 0.67, there starts to be more probes.

- (3 points) Why don't you want the load factor to be less than 0.5? Under a load factor than 0.5, the hash table wastes space, since it's less than half used.

c) (5 points) In closed-address hashing (e.g., ChainingDict picture to the right) if the load factor (# items / hash table size) is 10, what would you expect for the average number of probes/compares of a successful search? (Justify your answer)



If the hash function is doing a good job randomizing keys to home addresses, then each list has about 10 items. We'd need to go about half way down a list on average so 5 compares would be expected on a successful search.

20

Question 5. (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 **descending order** (largest to smallest), and
- 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"

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

20

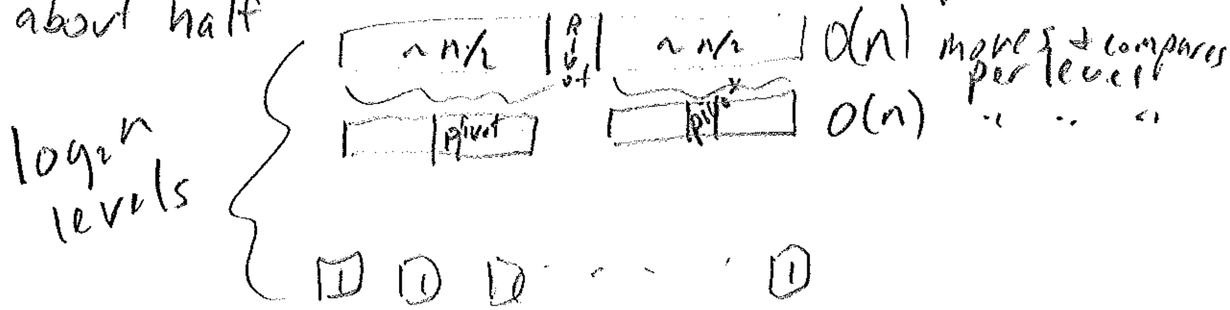
Question 6. Recall the general idea of Quick sort:

- 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

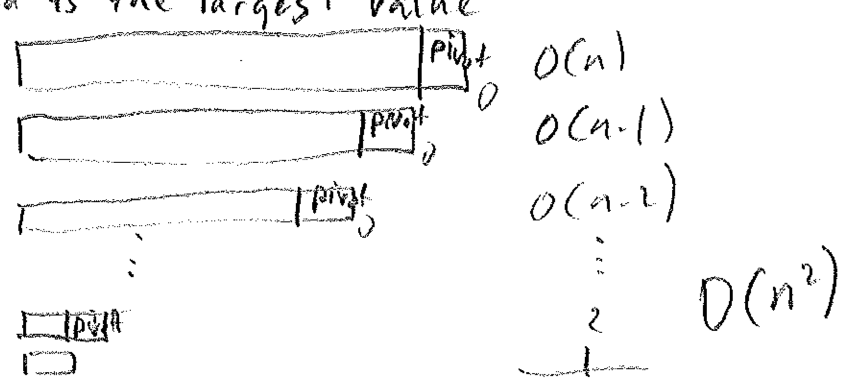
a) (5 points) Explain why quick sort is $O(n \log_2 n)$ when sorting initially randomly ordered items. (n is the len(myList)) For random data we expect the pivot to roughly split the list in about half

5



b) (5 points) Explain why quick sort is $O(n^2)$ is the worst-case. (n is the len(myList)) If the pivot is always picked as the largest value

5



30