Spring 2017 Name: Mark F.

Data Structures - Test 2

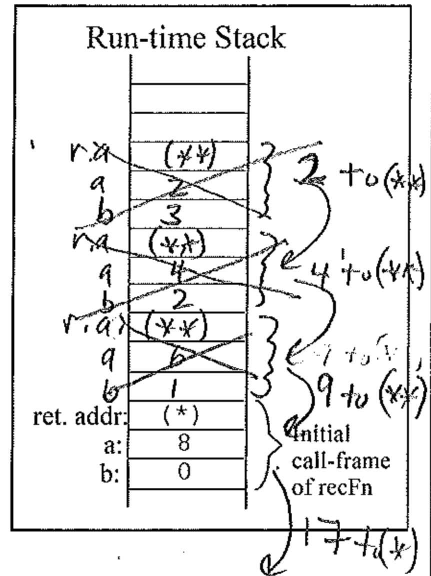
Question 1. (10 points) What is printed by the following program? Output:

```
def recFn(a, b):
    print( a, b )
    if  a < b:
        return a
    elif a == b:
        return a + b
    else:
        return a + recFn(a - 2, b + 1) - b
                      (**)

print("Result = ", recFn(8, 0))
                       (*)
```

4 6 8 2 4 9 - 2 x 1

8 0
6 1
4 2
2 3
Result = 17



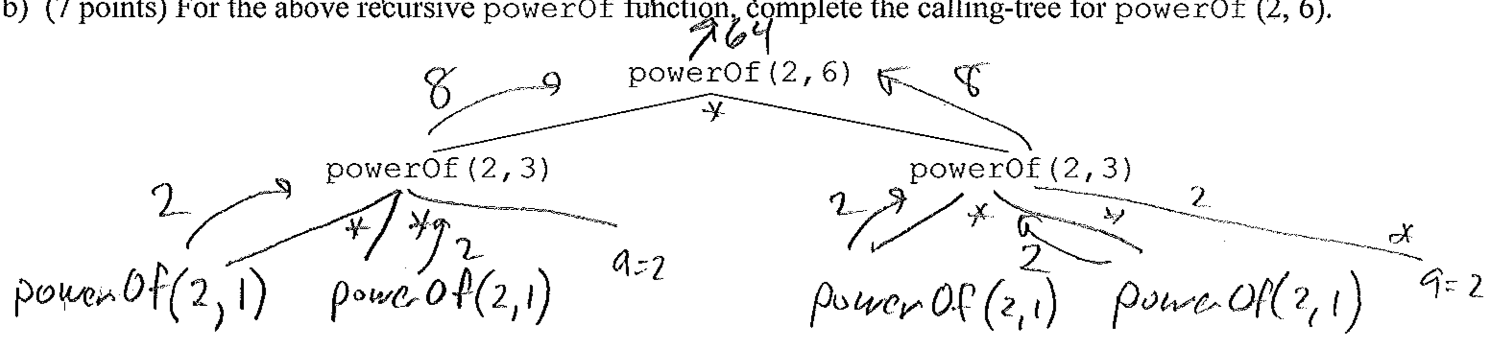
Question 2. Write a recursive Python function to calculate a^n (where n is an integer) based on the formulas:

$a^0 = 1$, 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
```

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



c) (5 points) Suggest a way to speedup the above `powerOf` function.

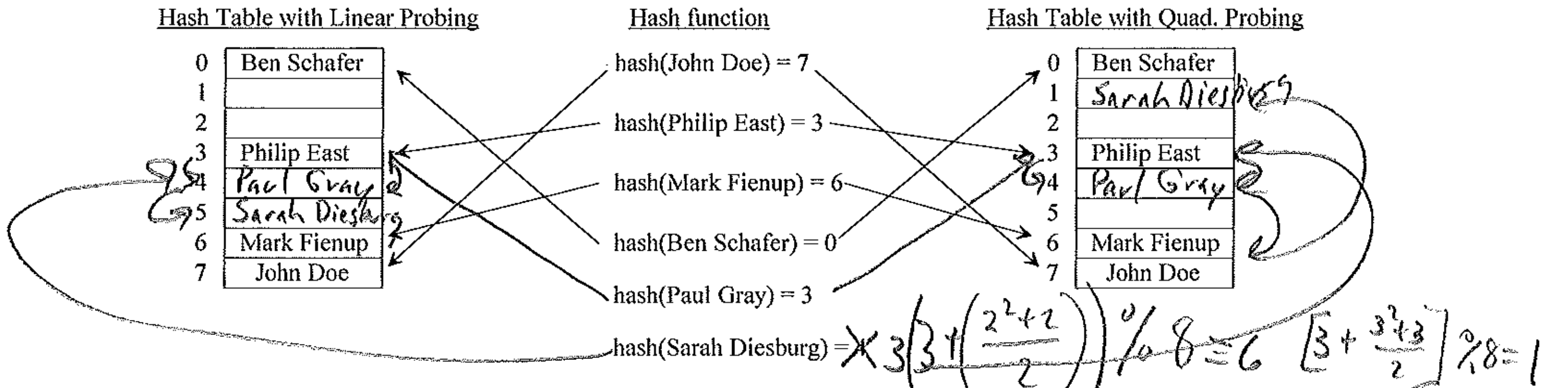
You could do dynamic-programming or avoid the duplicate recursive calls:

```
elif n % 2 == 0:
    return powerOf(a, n//2) ** 2
else:
    return (powerOf(a, (n-1)//2) ** 2) * a
```

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) (10 points) Insert "Paul Gray" and then "Sarah Diesburg" using Linear (on left) and Quadratic (on right) probing.

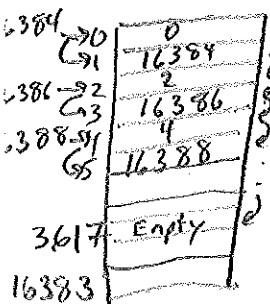


In lab 7, we inserted the 10000 even values 0, 2, 4, 6, 8, ..., 19996, 19998 is ascending order into various hash tables and then timing searching for the 20000 values 0, 1, 2, 3, 4, 5, ..., 19996, 19997, 19998, 19999. On my office computer the timings for these searches are:

Type of Hashing	Timings of 20,000 Searches on Various Hash Table Sizes (Load Factors)		
	16,384 (0.61)	32,768 (0.31)	65,536 (0.15)
Open-address with Linear probing	(b) 12.19 seconds	0.064 seconds	0.062 seconds
Open-address with Quadratic probing	0.575 seconds	0.064 seconds	0.065 seconds
Closed-address with unsorted linked list at each home address (i.e., ChainingDict)	0.128 seconds	0.114 seconds	0.115 seconds

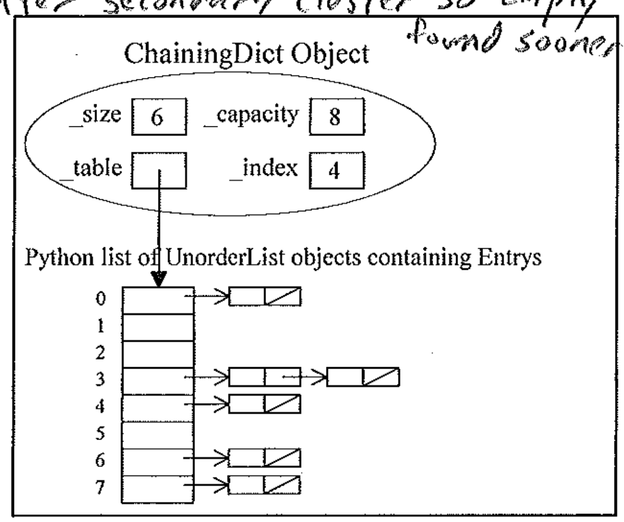
b) (8 points) For load factor 0.61, why did the open-address with Linear probing perform much worse than open-address with Quadratic probing? Because Linear probing suffers from secondary clustering, rehashing patterns for home addresses 0, 2, 4, .. have merged into one big cluster. Small odd search (1, 3, 5, ..) rehash down table to first "Empty" slot at 3617

Quadratic doesn't suffer secondary cluster so Empty found sooner.

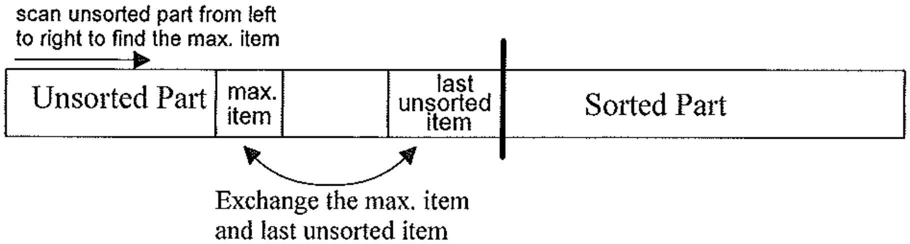


c) (7 points) For load factors 0.31 and 0.15, why is the closed-address (e.g., ChainingDict) version slower than the open-address versions?

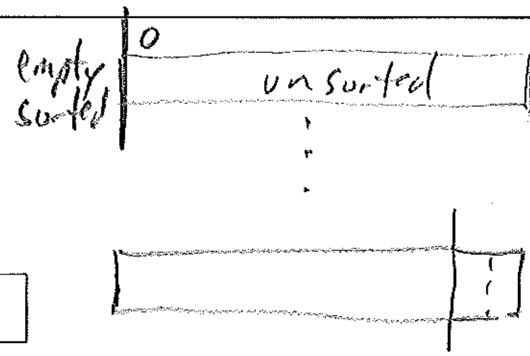
Open-address for these load factors has no rehashing, so keys compared to hash table at home addresses. Close-addr. must search linked list of size 1, but adds an extra layer of method calls which slows it slightly.



Question 5. (25 points) In class we developed the following selection sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list, i.e.:

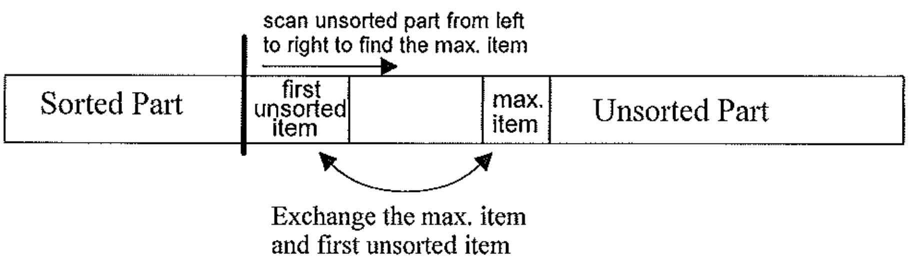



```
def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        maxIndex = 0
        for testIndex in range(1, lastUnsortedIndex+1):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        # exchange the items at maxIndex and lastUnsortedIndex
        temp = aList[lastUnsortedIndex]
        aList[lastUnsortedIndex] = aList[maxIndex]
        aList[maxIndex] = temp
```



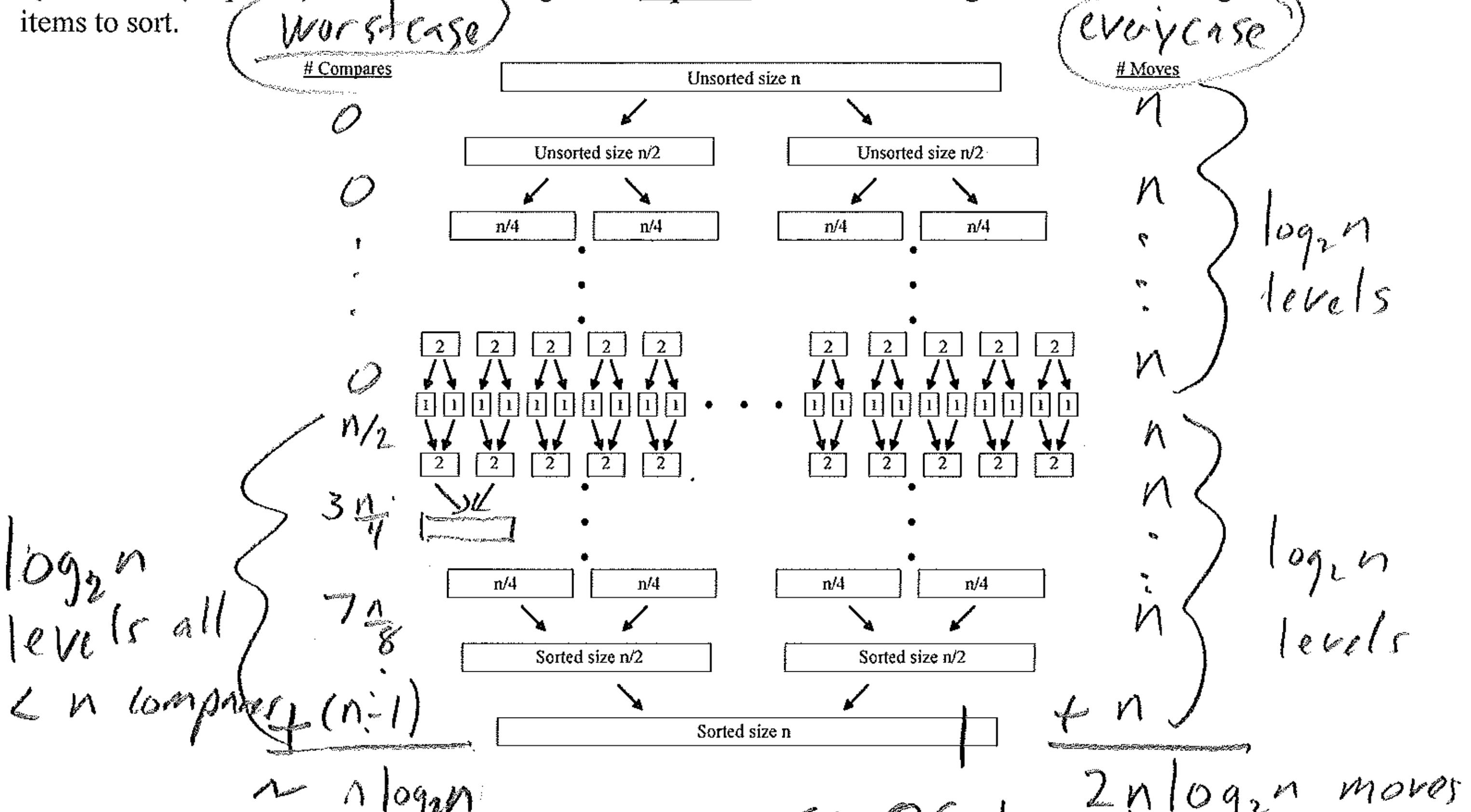
For this question write a variation of the above selection sort that:

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




```
def selectionSortVariation(myList):
    for firstUnsortedIndex in range(0, len(alist)-1, 1):
        maxIndex = firstUnsortedIndex
        for testIndex in range(firstUnsortedIndex+1, len(alist), 1):
            if alist[testIndex] > alist[maxIndex]:
                maxIndex = testIndex
        temp = aList[firstUnsortedIndex]
        aList[firstUnsortedIndex] = alist[maxIndex]
        alist[maxIndex] = temp
```

Question 6. (10 points) Use the below diagram to ***explain*** the worst-case big-oh notation of merge sort. Assume "n" items to sort.



Question 7. (10 points) *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

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

(Pivot Index labels the Pivot Item column.)

Explain how quick sort performs $O(n^2)$ in the worst-case. – pivot alway selected that's largest item