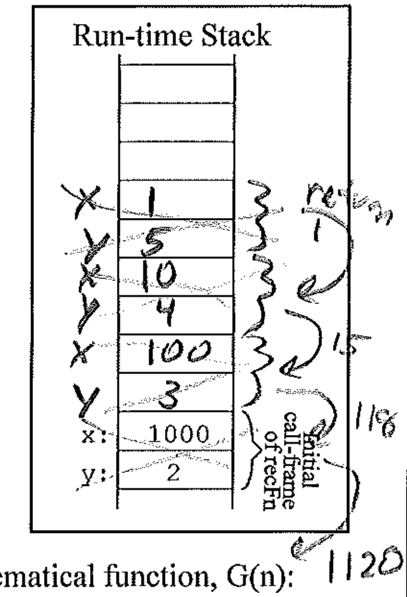
Name: Mark F.

Data Structures - Test 2



Question 1. (5 points) What is printed by the following program?

```
def recFn(x, y):
    print( x, y )
    if x <= y:
        return x
    else:
        return x + recFn(x // 10, y + 1) + y

print("Result = ", recFn(1000, 2))
```

Output:

1000 2
100 3
10 4
1 5
Result = 1120

5

Question 2. (8 points) Write a recursive Python function to compute the following mathematical function, G(n):

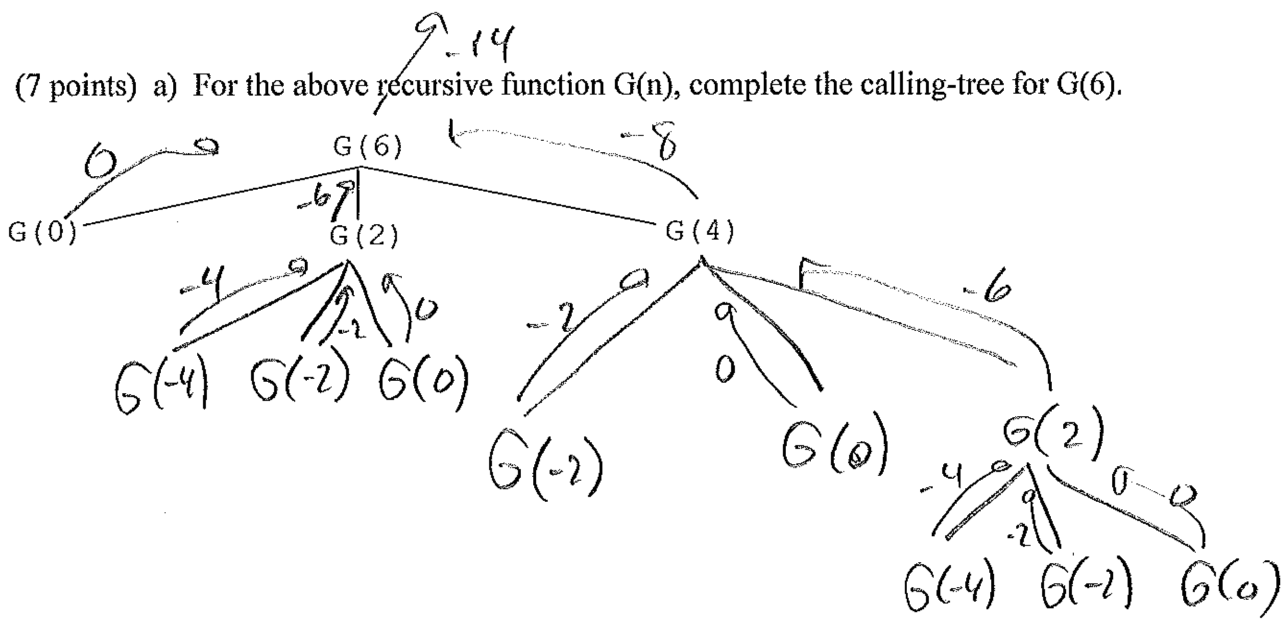
$G(n) = n$ for all value of $n \leq 0$

$G(n) = G(n-6) + G(n-4) + G(n-2)$ for all values of $n > 0$.

```
def G(n):
    if n <= 0:
        return n
    else:
        return G(n-6) + G(n-4) + G(n-2)
```

8

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



7

b) What is the value of G(6)? -14

c) What is the maximum height of the run-time stack when calculating G(6) recursively? 4

20

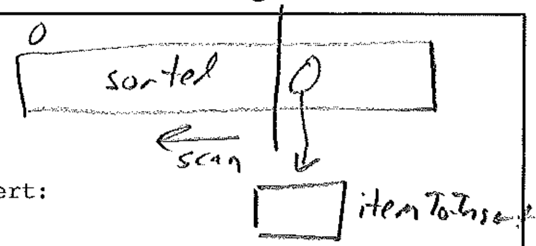
Question 4. (15 points.) Consider the following insertion sort code which sorts in ascending order.

```
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?

3 To avoid running off the left end of the list when scanning the sorted part from right-to-left.

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
```

b) Explain how the bolded code in the modified insertion sort code above allows for the elimination of the `testIndex >= 0` while-loop comparison.

3 The bolded code places the minimum list item at index 0, so the remaining while-condition must be false when testIndex is 0. Thus, the while cannot run of the left end of the 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`.

4 The bold code is $O(n)$, but it replaces the "testIndex>=0" check which is $O(n^2)$

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

4 We expect to insert a random item about half way down the sorted part on average, but in descending order we must insert at spot 0.

Question 5. (20 points) Write a variation of selection sort that:

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

Sorted Part	Unsorted Part

```
def selectionSort(myList):
    for firstUnsorted in range(len(myList)-1):
        minIndex = firstUnsorted

        for testIndex in range(firstUnsorted+1, len(myList)):
            if myList[testIndex] < myList[minIndex]:
                minIndex = testIndex

        temp = myList[firstUnsorted]
        myList[firstUnsorted] = myList[minIndex]
        myList[minIndex] = temp
```

Question 6. (15 points) Recall the common rehashing strategies we discussed for open-address hashing:

Strategy	Description
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)
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) 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	Sarah Diesburg
2	
3	Philip East
4	
5	Mark Fienup
6	John Doe
7	Paul Gray

4

Hash function

hash(John Doe) = 6

hash(Philip East) = 3

hash(Mark Fienup) = 5

hash(Ben Schafer) = 0

hash(Paul Gray) = 5

hash(Sarah Diesburg) = 6

Hash Table with Quad. Probing

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

$6 + \frac{2^2+2}{2}$

$9 \% 8 = 1$

5

$\left(5 + \frac{2^2+2}{2}\right) \% 8 = 8 \% 8 = 0$

$\left(5 + \frac{3^2+3}{2}\right) \% 8 = 11 \% 8 = 3$

$\left(5 + \frac{4^2+4}{2}\right) \% 8 = 15 \% 8 = 7$

b) Explain how deletions in an open-address hash table are handled.

Deleted keys are replaces by a "DELETED" marker (e.g. True) to indicate that it once held a key. The "DELETED" mark means continue to search, but can be replaced on an insertion.

Question 7. (15 points) The general idea of *Quick sort* 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

Explain why the **best-case** performance is $O(n \log_2 n)$.

Ideally (best-case) the pivot item divides the unsorted part in half, so we have $\log_2 n$ levels. At each level, all the partitions combine take $O(n)$ amount of work, so overall $O(n \log_2 n)$

work per level

~n/2 | ~n/2 — $O(n)$

~n/4 | ~n/4 | ~n/4 | ~n/4 — $O(n)$

~n/8 ... — $O(n)$

$O(\log_2 n)$ levels

+

$O(n \cdot \log_2 n)$

30

Question 8. (15 points) Recall the general idea of Heap sort which uses a min-heap (class `BinHeap`) to sort a list. (`BinHeap` methods: `BinHeap(), insert(item), delMin(), isEmpty(), size()`)

<u>Generl idea of Heap sort:</u>

myList: unsorted list with n items

1. Create an empty heap
2. Insert all n list items into heap

heap with n items

3. delMin heap items back to list in sorted order

myList: sorted list with n items

3 a) If we insert all of the list elements into a min-heap, what item would we easily be able to determine?

the minimum item.

b) Complete the code for `heapSort` so that it **sorts in descending order**

```
from bin_heap import BinHeap
def heapSort(myList):
    # Create an empty heap
    myHeap = BinHeap()

    for item in myList:
        myHeap.insert(item)

    for index in range(len(myList)-1, -1, -1):
        myHeap[index] = myHeap.delMin()
```

7

c) Determine the overall $O()$ for heap sort and justify your answer.

5

Overall $O(n \log_2 n)$

Step #1: $O(1)$

Step #2: loops n times with each insert taking $O(\log_2 n)$ since thats the max. height of the heap. Thus, $O(n \log_2 n)$ for step #2

Step #3: loops n times with each delMin taking $O(\log_2 n)$, so $O(n \log_2 n)$ for step #3