

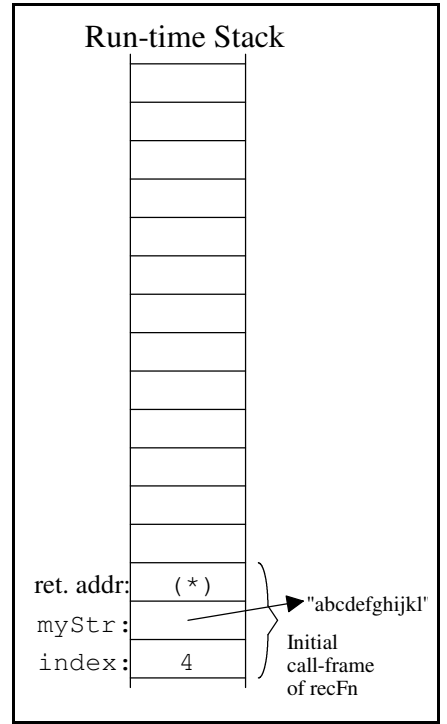
Data Structures - Test 2

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

```
def recFn(myStr, index):
    print(index)
    if index >= len(myStr):
        return "XYZ"
    else:
        return myStr[0] + recFn(myStr, index + 3) + myStr[index]
        (**)

print("result =", recFn("abcdefghijkl", 4))
        (*)
```

Output:



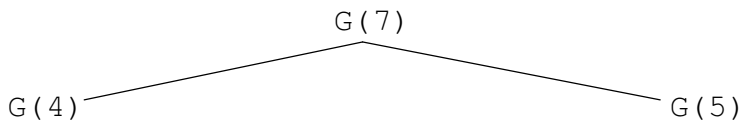
Question 2. a) (12 points) Write a recursive Python function to compute the following mathematical function, $G(n)$:

$$G(n) = n \text{ for all values of } n \leq 2 \text{ (e.g., } G(2) \text{ value is 2)}$$

$$G(n) = G(n-3) + G(n-2) \text{ for all values of } n > 2.$$

def G(n):

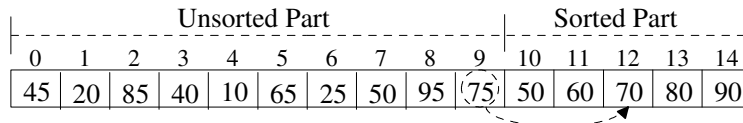
b) (8 points) For the above recursive function $G(n)$, complete the calling-tree for $G(7)$.



c) (3 points) What is the value of $G(7)$?

d) (2 points) What is the maximum length/size of the run-time stack when calculating $G(7)$ recursively?

Question 3. (15 points) Consider the following insertion sort code which sorts in ascending order, but builds the sorted part on the right-end of the list. For example after the code has ran a while, we need to insert 75 at index 12.



```
def insertionSort(myList):
    for lastUnsortedIndex in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsortedIndex]
        testIndex = lastUnsortedIndex + 1
        while testIndex < len(myList) and myList[testIndex] < itemToInsert:
            myList[testIndex-1] = myList[testIndex]
            testIndex = testIndex + 1
        myList[testIndex - 1] = itemToInsert
```

a) What is the purpose of the `testIndex < len(myList)` while-loop comparison?

b) Consider the modified insertion sort code that eliminates the `testIndex < len(myList)` while-loop comparison.

```
def insertionSortB(myList):
    maxIndex = 0
    for testIndex in range(1, len(myList)):
        if myList[testIndex] > myList[maxIndex]:
            maxIndex = testIndex
    temp = myList[len(myList)-1]
    myList[len(myList)-1] = myList[maxIndex]
    myList[maxIndex] = temp
    for lastUnsortedIndex in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsortedIndex]
        testIndex = lastUnsortedIndex + 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 < len(myList)` while-loop comparison.

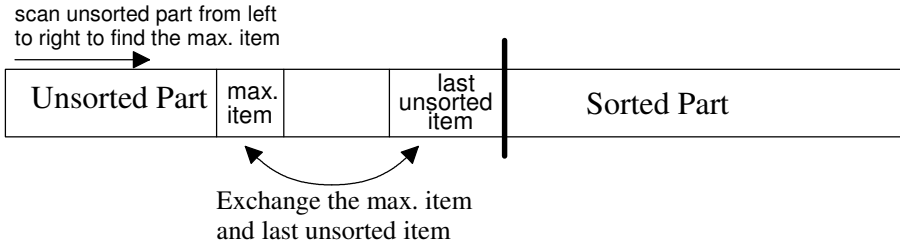
Consider the following timing of the above two insertion sorts on lists of 10000 elements.

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`.

d) In either version, why does sorting the initially ascending order list take less time than sorting the initially descending ordered list?

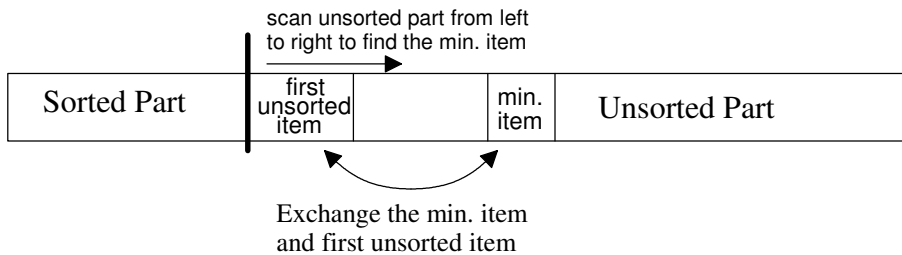
Question 4. In class we developed the following selection sort code which sorts in ascending order (smallest to largest) and builds the sorted part on the right-hand side of the list, i.e.:



```
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
```

(20 points) For this question write a variation of the above selection sort that:

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



```
def selectionSortVariation(myList):
```

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.

Hash Table with Linear Probing

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

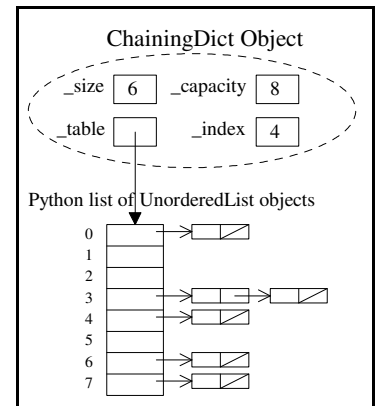
Hash function

hash(John Doe) = 7
 hash(Philip East) = 3
 hash(Mark Fienup) = 6
 hash(Ben Schafer) = 0
 hash(Andrew Berns) = 0
 hash(Sarah Diesburg) = 6

Hash Table with Quadratic Probing

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

b) (7 points) Open-address hashing above uses rehashing (e.g., linear or quadratic probing) when collisions occur. Briefly describe how closed-address hashing (e.g., ChainingDict) handles collisions.



Question 6. (15 points) Use the below diagram to **explain** the worst-case big-oh notation of merge sort. Assume “n” items to sort.

