

#2 - Recursion Tracing - 5pts

Trace the following code, then fill out the table below to indicate all the **recursive function calls** that are made, and which **value** is returned by each function call. You may not need all of the rows.

```
def gcd(x, y):  
    if y == 0:  
        return x  
    return gcd(y, x % y)  
  
print(gcd(20, 12))
```

Note: in the second column, make sure to indicate the actual returned value, not a function call!

Function Call	Returned Value
gcd(20, 12)	

#3 - Tracing Towers of Hanoi - 8pts

Recall the algorithm we discussed in class to solve the Towers of Hanoi problem. Use that algorithm to fill out all the steps needed to move three discs from Peg A to Peg C in the table below. You might not need to use all the rows.

The three discs are called 1, 2, 3 (where 1 is the smallest and the disc on top). So the algorithm starts with the discs 1, 2, 3 on Peg A, and should end with 1, 2, 3 on Peg C. We've done the first step for you.

	Peg A	Peg B	Peg C
Start	1, 2, 3		
Step 1	2, 3		1
Step 2			
Step 3			
Step 4			
Step 5			
Step 6			
Step 7			
Step 8			
Step 9			
Step 10			
Step 11			

How many steps would it take to move 4 discs instead of 3?

--

#4 - Binary Search - 12pts

In the following table, write out the recursive calls that our implementation of `binarySearch` would make while searching the given list for the given item. Make sure to write out the **function call**, not the result. You might not need to use all the rows.

Q1: Search for 5

Original Call	<code>binarySearch([3, 5, 6, 7, 9, 11, 11, 15, 15, 19], 5)</code>
Recursive Call 1	
Recursive Call 2	
Recursive Call 3	
Recursive Call 4	
Recursive Call 5	

Q2: Search for 14

Original Call	<code>binarySearch([3, 5, 6, 7, 9, 11, 11, 15, 15, 19], 14)</code>
Recursive Call 1	
Recursive Call 2	
Recursive Call 3	
Recursive Call 4	
Recursive Call 5	

#5 - Best Case and Worst Case - 12pts

For each of the following functions, give an example of an input that would result in **best-case efficiency**, then an example of an input that would result in **worst-case efficiency**. This example can either be a specific input, or a general description of an input.

```
def getEmail(words):  
    # words is a list  
    for token in words:  
        if "@" in token:  
            return token  
    return "No email found"
```

```
def isPrime(num):  
    for factor in range(2, num):  
        if num % factor == 0:  
            return False  
    return True
```

What is a **best case input** for getEmail?

What is a **worst case input** for getEmail?

What is a **best case input** for isPrime?

What is a **worst case input** for isPrime?

#6 - Calculating Big-O Families - 15pts

For each of the following functions, check the **Big-O function family** that function belongs to. You should determine the function family by considering how the number of steps the algorithm takes grows as the size of the input grows.

```
def countEven(L): # n = len(L)
    result = 0
    for i in range(len(L)):
        if L[i] % 2 == 0:
            result = result + 1
    return result
```

- 0(1)
 - 0(logn)
 - 0(n)
 - 0(nlogn)
 - 0(n²)
-

```
# n = len(L)
def sumFirstTwo(L):
    if len(L) < 2:
        return 0
    return L[0] + L[1]
```

- 0(1)
 - 0(logn)
 - 0(n)
 - 0(nlogn)
 - 0(n²)
-

```
# n = len(L1) = len(L2)
def linearSearchAll(L1, L2):
    count = 0
    for item in L1:
        # Hint: what's the complexity of
        # linear search?
        if linearSearch(L2, item) == True:
            count = count + 1
    return count
```

- 0(1)
 - 0(logn)
 - 0(n)
 - 0(nlogn)
 - 0(n²)
-

```
# n = len(L1) = len(L2)
def binarySearchAll(L1, L2):
    count = 0
    for item in L1:
        # Hint: what's the complexity of
        # binary search?
        if binarySearch(L2, item) == True:
            count = count + 1
    return count
```

- 0(1)
 - 0(logn)
 - 0(n)
 - 0(nlogn)
 - 0(n²)
-

```
#Hint:consider the # of recursive calls
def recursiveSum(L): # n = len(L)
    if len(L) == 0:
        return 0
    else:
        return L[0] + recursiveSum(L[1:])
```

- 0(1)
- 0(logn)
- 0(n)
- 0(nlogn)
- 0(n²)

#7 - Tracing Sorting Algorithms - 15pts

For the two tables below, each row represents a 'pass' - a single iteration of the outer loop in the function. Fill in the number of comparisons and swaps that happen in each pass, and the state of the list at the **end** of that pass, for the specified sort function as implemented in class.

Selection Sort			
Pass #	Comparisons	Swaps	List State
Start	-	-	[3, 5, 1, 2, 4]
1			
2			
3			
4			

Insertion Sort			
Pass #	Comparisons	Swaps	List State
Start	-	-	[3, 5, 1, 2, 4]
1			
2			
3			
4			

Merge Sort:

How many times is the function
mergeSort called when we run
mergeSort([3, 5, 1, 2, 4])?

15-110 Hw3 - Programming Portion

Each of these problems should be solved in the starter file available on the course website. Submit your code to the Gradescope assignment Hw3 - Programming for autograding.

All programming problems may also be checked by running the starter file, which calls the function `testAll()` to run test cases on all programs.

#1 - `hiddenMessage(s)` - 5pts

Write a function `hiddenMessage(s)` which takes a string, splits it by spaces, and returns a new string that is composed of the n -th letters of the n -th words. You may assume that the string only contains letters and spaces.

For example, `hiddenMessage("I'm here")` returns "Ie", and `hiddenMessage("Come to office hours")` returns "Cofr".

#2 - `letterFrequency(s)` - 5pts

Write a function `letterFrequency(s)` which takes a string and returns a list of 26 elements, where each element is the number of times that the corresponding letter of the alphabet occurs in the string. The 0th index corresponds to "a", the 1st corresponds to "b", etc., until the 25th element corresponds to "z".

For example, `letterFrequency("Hello World")` should return:
[0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0].

Note: you can ignore any non-letter characters that occur in the string, but you should make sure both upper- and lower-case letters are counted as the same.

Hint: the easiest way to get an index based on a letter is to use the `ord(c)` method, which takes a one-character string as input and returns the ASCII value of that character. Offset this number by `ord("a")` or `ord("A")` to get the index you need.

#3 - onlyPositive(lst) - 5pts

Write a function `onlyPositive(lst)` that takes as input a 2D list and returns a new 1D list that contains only the positive elements of the original list, in the order they originally occurred. You may assume the list only has numbers in it.

Example: `onlyPositive([[1, 2, 3], [4, 5, 6]])` returns `[1, 2, 3, 4, 5, 6]`,
`onlyPositive([[0, 1, 2], [-2, -1, 0], [10, 9, -9]])` returns `[1, 2, 10, 9]`,
and `onlyPositive([[-4, -3], [-2, -1]])` returns `[]`.

#4 - recursiveMax(lst) - 5pts

Write a function `recursiveMax(lst)` that takes a list as input and returns the maximum value in the list. You may assume the list contains at least one element. This function must use **recursion** in a meaningful way; a solution that uses a loop or built-in max functions will receive no points.

For example, `recursiveMax([1, 2, 3])` returns 3, and
`recursiveMax([2, 4, 6, 9, 10, 2, 6])` returns 10.

#5 - recursiveCount(lst, item) - 5pts

Write a function `recursiveCount(lst, item)` that takes a list and a value as input and returns a count of the number of times that item occurs in the list. This function must use **recursion** in a meaningful way; a solution that uses a loop or built-in count functions will receive no points.

For example, `recursiveCount([2, 4, 6, 8, 10], 6)` returns 1,
`recursiveCount([4, 4, 8, 4], 4)` returns 3, and
`recursiveCount([1, 2, 3, 4], 5)` returns 0.