ANSWER SHEET

These problems were generated by TAs and instructors in previous semesters. They may or may not match the actual difficulty of problems on Quiz3.

Lists

1. Write the function `onlyNegative(L)` which takes a list of numbers, `L`, and returns a **new list** that contains only the negative numbers in `L`, in the order they originally appeared. Your function should **not** destructively modify the original list `L`.

For example, `onlyNegative([-2, 1, 2, -1, 0, -3, 3])` would return the list `[-2, -1, -3]`.

ANSWER:

```
def onlyNegative(L):
      result = [ ]
      for item in L:
            if item < 0:
                  result.append(item)
      return result
```

2. Write a function commonElement that takes in two lists and returns True if they share a common element (ie. there is an element that occurs in both lists).

ANSWER:

```
def commonElement(x1, x2):
      result = []
      for elem in x1:
            if elem in x2:
                  return True
      return False
```

3. Given a word search puzzle (format is a 2D list of strings), check to see if a given word is in the board. Words can be left->right or up->down, but not right->left, down->up, or diagonal. For example,

```
puzzle = [ [ 'a', 'q', 'r', 't' ],
           [ 'd', 'o', 'g', 'a' ],
           [ 'w', 'm', 'z', 'c' ] ]

assert(wordSearch(puzzle, "dog") == True)
assert(wordSearch(puzzle, "cat") == False)
```

ANSWER:

```
def wordSearch(puzzle, word):
      allPossible = [] # all lines from our puzzle board
      for row in range(len(puzzle)):
            line = ""
            for col in range(len(puzzle[0])):
                  line += puzzle[row][col]
            allPossible.append(line)

      for col in range(len(puzzle[0])):
            line = ""
            for row in range(len(puzzle)):
                  line += puzzle[row][col]
            allPossible.append(line)

      for elem in allPossible:
            if word in elem:
                  return True
      return False
```

References and Memory

1. Are lists considered mutable or immutable objects? If mutable, what are some built in methods you can use to change a list?

ANSWER:
Mutable. list.append(), list.remove(), list.pop()

2. What does the following code print?

```
def ct1(P, M):
      A = [ [ 1, 5, 7 ], [ 1 ] ]
      P[1][0] = P[1][0] * 3
      M[1] = A[0]
      A = P
      print("A =", A)
      print("P =", P)
      print("M =", M)

ct1([ [ 2, 4, 6 ], [ 2 ] ] , [ [ 3, 6, 9 ], [ 3 ] ])
```

ANSWER:

```
A = [ [ 2, 4, 6 ], [ 6 ] ]
P = [ [ 2, 4, 6 ], [ 6 ] ]
M = [ [ 3, 6, 9 ], [ 1, 5, 7 ] ]
```

3. Given that three lists are originally set up with the following code:

```
L = [ "Y" ]
A = [ 1, L, 2 ]
B = A
C = [ 1, L, 2 ]
```

Fill out the following table so that it shows the values in each variable after the line of code in the left column has run. Any changes made should be cumulative (if A is changed in row 1, the change should carry over to row 2).

Code	List A	List B	List C
Initial List	[1, ["Y"], 2]	[1, ["Y"], 2]	[1, ["Y"], 2]
A.remove(2)	[1, ["Y"]]	[1, ["Y"]]	[1, ["Y"], 2]
B[0] = 9	[9, ["Y"]]	[9, ["Y"]]	[1, ["Y"], 2]
C.append(5)	[9, ["Y"]]	[9, ["Y"]]	[1,["Y"],2,5]
L.append("Z")	[9, ["Y", "Z"]]	[9, ["Y", "Z"]]	[1,["Y","Z"],2,5]
A = [3, 4]	[3, 4]	[9, ["Y", "Z"]]	[1,["Y","Z"],2,5]

Recursion

1. Define what a base case is in terms of a recursive problem.

ANSWER:
The base case gives explicit instruction about what to do when a problem cannot get any smaller, aka the simplest case.

2. In this problem, you will translate the algorithm below into a recursive Python function `b2d(s)` that will compute the decimal value of a binary string of 1's and 0's. For example, `b2d("101") = 5` and `b2d("1011") = 11`.

   A. Base case: If the string is empty, return 0
   B. Recursive case:
      a. Compute the decimal value of the all but the first digit recursively by setting a variable `value` equal to the recursive call on the string from index 1 to end.
      b. Convert the 0 index character in the string to an integer and set a variable `bit` equal to it.
      c. Compute the bit's power of 2 by setting a variable `power` as 2 to the power of the string length minus 1.
      d. Return the value of the string so far as `bit*power + value`

Example: `b2d("1011")` recursively computes $1*2^3 + (0*2^2 + (1*2^1 + (1*2^0 + (0)))) = 11$

ANSWER:

```
def b2d(d):
      if len(s) == 0:
            return 0
      else:
            value = b2d(s[1:]
            bit = int(s[0])
            power = 2**(len(s)-1)
            return bit * power + value
```

3. Write the recursive function addDeck(deck) that takes in a 2D List representing a deck of cards and **recursively** returns the sum of the numbers on the cards. Each list inside the deck is a card, with the first element representing the number on the card and the second element representing the suit. Assume all cards will be number cards. For example:

```
d = [[1, "Hearts"], [4, "Spades"], [3, "Clubs"], [8, "Diamonds"]]
assert(addDeck(d) == 16)
```

ANSWER:

```
def addDeck(deck):
    if len(deck) == 0:
        return 0
    else:
        return deck[0][0] + addDeck(deck[1:])
```

4. Write a recursive function areNearlyEqual(L, M) which takes two lists of integers and returns True if they are “nearly equal” or False otherwise. Two lists are “nearly equal” if they are the same length, and the corresponding values in the lists are all within 1 (inclusive) of each other. Your solution must use recursion.

ANSWER:

```
def areNearlyEqual(L, M):
    if len(L) != len(M):
        return False
    elif L == [] and M == []:
        return True
    else:
        firstNear = abs(L[0] - M[0]) <= 1
        return firstNear and areNearlyEqual(L[1:], M[1:])
```

Search Algorithms

1. Answer the following questions about binary and linear search.
   a. Explain one major difference between binary and linear search
   b. For linear search, at what index would the target element be found in the best case scenario runtime? For binary search? Explain.
   c. For linear search, at what index would the target element be found in the worst case scenario runtime? For binary search? Explain.

ANSWER:

a) Binary search only works on sorted lists; linear search works on any list / Binary search starts checking at the middle; linear search starts checking at the beginning / Binary search is much faster than linear search.
b) Linear: 0th. Binary: middle (len(L)//2). Linear search begins searching with the 0th index and checks by increasing index by 1. Binary search begins checking at the midpoint of the list.
c) Linear: not in list. Binary: not in list. For either search algorithm, the worst-case scenario occurs when the item is not in the list.

2. Binary search algorithm trace: Assuming binary search is called on the following list, write the list argument used in the function call after each split when using binary search to search for 19.

Then: How many function calls are required to find the value? And how many calls would be required if linear search were used?

Original list: [3, 12, 26, 28, 32, 39, 44]

ANSWER:
Value after split 1: [3, 12, 26] ~~[28, 32, 39, 44]~~
Value after split 2: ~~[3, 12]~~ [26]
Value after split 3: [ ] ~~[26]~~
Binary search function calls: 3
Linear Search function calls: 7

Dictionaries

1. The fruits in a bag of groceries are provided as a list (eg. `["apple", "oranges", "banana", "kiwis"]`). For any elements in the list that are plural, like “kiwis”, we say there are 3 of that fruit in the bag. (There are no fruits in the bag where the singular form of the word ends in an “s”.) Write a function `groceryCount` that outputs a dictionary with each fruit name and its frequency as a key-value pair.

For example:

```
groceryCount(["apple", "oranges", "banana", "kiwis", "kiwi"]) ==
{ "apple" : 1, "orange" : 3, "banana" : 1, "kiwi" : 4 }
```

ANSWER:

```
def groceryCount(L):
      d = {}
      for i in range(len(L)):
            fruit = L[i]
            quantity = 1
            if fruit[-1] == "s":
                  fruit = fruit[:-1]
                  quantity = 3
            if (fruit in d):
                  d[fruit] = d[fruit] + quantity
            else:
                  d[fruit] = quantity
      return d
```

2.

a. What does the following code print?

```
def chainedKeys(dict, startkey):
    key = startkey
    while key in dict.keys():
        key = dict[key]
        print(key)
    return key
result = chainedKeys({3:6, 4:9, 5:7, 6:5, 7:4, 8:9, 9:2}, 3)
print(str(result) + " is missing")
```

ANSWER:

```
6
5
7
4
9
2
2 is missing
```

b. What is one key whose **value** could be changed to make the function loop infinitely? What should the value be changed to?

ANSWER:
Many possible answers. One is:
Key: 9, Value: 3

Hashed Search

1. What properties does a hash function need to achieve O(1) lookup in a hashtable?

ANSWER:
Hash function results don't change over time, two unique values are likely hash to different values (collision-resistant)

2. What would happen if two objects happen to have the same hash values? Is it allowed?

ANSWER:
If two objects happen to have the same hash values, collision would happen. However, in general, hash tables are designed to handle collisions. One way to handle that would be create a list of objects with the same hash values and then put them in that bucket.

3. What are some data structures that you can apply hash functions to, and what are the ones you cannot?

ANSWER:
Hash functions can only be applied to immutable data structures such as strings and integers. They cannot be applied to mutable values such as lists and dictionaries.

4. Each student in 15-110 is asked to create a list of their favorite ice cream flavors, which they can update as their tastes change.

Students A and B say ["chocolate", "vanilla", "strawberry"],
Student C says ["mint chocolate chip", "chocolate", "strawberry"], and
Student D says ["vanilla", "mint chocolate chip", "chocolate"].
Later, Student C finds the best strawberry ice cream ever and destructively shuffles their favorite list.

Your professors want to maintain a dictionary of ice cream preferences (keys) to student counts (values) in order to keep track of which combination is the favorite at the time the preferences were first collected. How should they store the keys, and why that approach?

Keys are lists converted into strings using str(). We have to convert the lists because we cannot hash mutable values. But we only want to track the initial preferences of students, not final preferences. Converting a list to a string will capture that initial preference and make it immutable.