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 Test3.

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
```

Aliasing and Mutability

1. 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]

2. 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()

3. 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 ] ]
```

Recursion

1. 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
```

2. 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:])
```

3. 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.

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. 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

2. 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.

Runtime and Big-O Notation

1. For each of the following programs, write the Big-O runtime of that program in terms of N to the right of the code. Assume that all named functions have the runtimes discussed in class. Make sure your Big-O is simplified (no extra terms) and as small as possible.

Program	Big-O Runtime

`# N = length of string S`
`def hasChar(S, char):`
`  for c in S:`
`  if c == char:`
`  return True`
`  return False`
	O(n)

`# N = length of string S`
`def countVowels(S):`
`  vowels = ['A','E','I','O','U']`
`  count = 0`
`  for c in S:`
`  for i in range(5):`
`  if c == vowels[i]:`
`  count = count + 1`
`  return count`
	O(n)

`# N = length of list L`
`def specialSearch(L, item):`
`  L = mergeSort(L)`
`  return binarySearch(L, item)`
	O(nlogn)

`# N = length of list L`
`def getValue(L, i):`
`  return L[i]`
	O(1)

`# N = length of lists L1 and L2`
`def searchAllItems(L1, L2):`
`  count = 0`
`  for item in L1:`
`  if linearSearch(L2, item):`
`  count = count + 1`
`  return count`
	O(n^2)

`# N = length of list L`
`def printStuff(L):`
`    for i in range(10):`
`        print("I love 110!")`
`    for item in L:`
`        print(item)`
	O(n)

2. Why is mergeSort's worst case and best case runtime O(nlogn)?

ANSWER:
No matter the order of the elements, mergesort will still divide up the list into the smallest parts and reassemble and sort them. This does not change even if the list is already sorted.

Sort Algorithms

1. What is the Big-O of insertion sort? Explain your reasoning.

ANSWER:
O(n**2). In the worst case, we would both compare and swap all the elements we have seen at each iteration (given that we would have n iterations for a list with n elements). Algorithmically, this would be represented as 2 * (1 + 2 + … + n-1) which reduces to n**2.

2. Write out the list for each pass of a call to selection sort on the list [3, 7, 2, 9, -3].

ANSWER:
[-3, 7, 2, 9, 3]
[-3, 2, 7, 9, 3]
[-3, 2, 3, 9, 7]
[-3, 2, 3, 7, 9]

3. Explain why someone might choose to sort a list with insertion sort over selection sort. Hint: do both algorithms always run in their worst-case O(n^2)?

Insertion sort will run in O(n) if the given list is already in order, so we might choose insertion sort if we expect a lot of sorted lists as input.