15-112, Lab 7, Efficiency and Recursion Practice

See how many of these problems you and a partner can solve during today's lab.

Part 1: Big-O Analysis

For each of these, don't run the code — work out the Big-O on paper. Fill in every blank: the cost of each line, how many times each loop runs, and the overall Big-O of the function. Be ready to explain *why*, not just state the answer.

Use a pencil when solving these problems.

Code #1

```
def bigO1(L):                       # L is a list of length N

    result = 0                      # O(__________)

    for elem in L:                  # loops __________ times

        result += L.count(elem)     # O(__________)

    return result                   # O(__________)

                                    # Overall - O(__________)
```

Code #2

```
def bigO2(a):                       # a is a list of length N

    for i in range(len(a)):         # loops __________ times

        j = 1                       # O(__________)

        while j < len(a):           # loops __________ times

            j *= 2                  # O(__________)

    return a                        # O(__________)

                                    # Overall - O(__________)
```

Code #3

```
def bigO3(L):                                          # L contains N items
    allowedWords = ["surround", "friend", "stop"]      # O(__________)
    newList = []                                       # O(__________)
    for w in L:                                        # loops __________ times
        if w in allowedWords:                          # O(__________)
            newList.append(w)                          # O(__________)
    for w in allowedWords:                             # loops __________ times
        if w in L:                                     # O(__________)
            L.remove(w)                                # O(__________)
    L.sort()                                           # O(__________)
    L.extend(newList)                                  # O(__________)
    return L                                           # O(__________)
                                                       # Overall - O(__________)
```

Code #4

```
def bigO4(L, z):              # L is a list of length N
    n = len(L)                # O(__________)
    x = 0                     # O(__________)
    y = 0                     # O(__________)
    while (x < n):            # loops __________ times
        while (y < n):        # loops __________ times
            y += 1            # O(__________)
            z += y            # O(__________)
        x += 1                # O(__________)
        z += x                # O(__________)
    return z                  # O(__________)
                              # Overall - O(__________)
```

Careful: look closely at where **y** *gets reset.*

Code #5

```
def bigO5(L, z):                    # L is a list of length N

    for i in range(len(L)):         # loops __________ times

        if L[i] == sorted(L)[z]:    # O(__________)

            return i                # O(__________)

    return False                    # O(__________)

                                    # Overall - O(__________)
```

Code #5 is less efficient than it needs to be. How could you change one line to improve its Big-O, and what would the new Big-O be?

Part 2: Recursion

For each problem, you write the requested function *and* testcases for it. See `week01/example.py` in the demo code repository for an example of organizing this.

Rules for every problem below: each function must be written recursively. A solution that uses loops, comprehensions, generators, or iterative built-in functions (such as `range`, `count`, `sum`, `max`, or `min`) will not be accepted.

---

1. multiply(a, b)

Write the recursive function `multiply(a, b)`, which returns the value `a * b`. You may only use the `+` and `-` operators in this function — no `*`, `/`, `//`, or `**`.

---

2. bunnyEars(n)

We have bunnies standing in a line, numbered `1, 2, ...`. The odd bunnies (`1, 3, ...`) have the normal 2 ears. The even bunnies (`2, 4, ...`) we'll say have 3 ears, because they each have a raised foot. Write the recursive function `bunnyEars(n)`, which returns the total number of "ears" in a line of `n` bunnies.

```
assert(bunnyEars(0) == 0)
assert(bunnyEars(1) == 2)
assert(bunnyEars(2) == 5)
```

---

3. removeVowels(s)

Write the non-destructive recursive function `removeVowels(s)`, which takes a string `s` and returns a copy of it with all of the vowels removed. For example, `removeVowels("Hello there")` returns `"Hll thr"`.

---

4. getHiLo(lst)

Write the recursive function `getHiLo(lst)`, which takes a list of integers `lst` and returns a tuple containing the highest and lowest values in the list, in that order. If the list is empty, return `None`.

```
assert(getHiLo([1, 7, 3, 8, 2, 9, 6, 4, 5]) == (9, 1))
assert(getHiLo([5]) == (5, 5))
assert(getHiLo([]) == None)
```

---

5. wordCounter(wordList)

Write the non-destructive recursive function `wordCounter(wordList)`, which takes a list of words and returns a dictionary mapping each word in the list to the number of times it appears. Words are case-sensitive, so `"Of"` and `"of"` are different words. If the list is empty, return an empty dictionary. Note that the built-in `count` method and `collections.Counter` are iterative, so you can't use them. For example:

```
wordList = ["father", "bless", "of", "golf", "of", "golf", "father", "of"]
assert(wordCounter(wordList) == {"father": 2, "bless": 1, "of": 3, "golf": 2})
assert(wordCounter(["hi"]) == {"hi": 1})
assert(wordCounter([]) == {})
```