Week 8 Practice Problems

Let's start with some questions on lists.

#1. Ask the user for a size (`size`) and a maximum value (`max_value`). Write a program that randomly assigns values between 0 and `max_value` to a `size`-by-`size` matrix. Print the matrix to the screen, and then calculate the sum of values around the "outside" (border) of the matrix: that is, the sum of the entries in the first and last rows and first and last columns but without double counting the corner entries. For example, if the random matrix is:

```
8 7 2 0 4
8 4 0 2 5
1 9 4 6 5
0 2 6 9 3
1 7 3 9 5
```

then the sum around the outside is 68. You will need to import the random module and use an appropriate function for generating the numbers.

#2. Write a program that asks the user for a matrix size (call it, `n`) and then randomly assigns the values 1 to n^2 to a `n` x `n` matrix, not repeating any values!

Then print the matrix to the screen. For example:

```
 3 25 12  4 19
 5 17 21 10  6
 1  8 16  2  7
13 22 24 11 20
23 14 15  9 18
```

Notice that you'll need some way to make sure there are no repeated numbers.

If you do this in a simple, but inefficient way, you'll notice it takes longer and longer to place the next value or find the next location. That's OK for a 5 x 5 matrix but won't be appropriate for a much larger (say, 1000 x 1000) matrix. So then what? Try writing a version that's efficient, that takes the same amount of time to place the first number as the last. Maybe you can think of more than two different ways?

Bonus: Investigate the Python `time` module and time your different implementations.

#3. Assume a list of N elements contains a sorted list of integers. Write a function that corresponds to the following definition:

```
def binary_search (array, value)
```

The function does a binary search, and returns the index in the list where `value` is stored. If `value` isn't in the list, the function returns -1.

A binary search is a way to search a sorted list of numbers. Begin at the middle of the list and decide if the value you're looking for is in the first half of the list or the second half. If it is the first half, then look at the middle element of the first half (i.e., a quarter of the way through the initial list) and decide if the element is in the first quarter or second quarter. If it is in the second half, look at the middle element of the second half (i.e. three-quarters of the way through the original list) to decide if you should like in the third quarter or fourth quarter. Then look halfway again, and halfway again, and …

Test your function on an array of size 100 filled in with sorted integers.

Hint: Draw a picture of what the binary search is supposed to do and while you are writing the code, use a short list (e.g., less than 10 elements) to debug it.

Now let's move to some work on sets and dictionaries.

#4. Write a function called `find_dups` that takes a list of integers as its input argument and returns a set of those integers occurring two or more times in the list. [Question source: Gries et al., *Practical Programming*, 3rd Edition].

#5. Generalize the function from Q4 to `find_multiples`. The function should take an additional parameter, *k*, and return a set of those elements that appear *k* or more times. Note: depending on how you answered Q4, this function may require a quite different approach.

You should implement the function it in two ways: using one of the lists methods and using a dictionary. Your code should work with a list of any type.

#6.

a) Write a function that takes in two equal size sets with arbitrary element types and returns a set of element pairs containing each element from the input sets in exactly

one pair. For example, input {'a','b'} and {1,2} could produce output {('a',1), ('b', 2)}. [A variation on a question from Gries et al., *Practical Programming*, 3rd Edition].

b) Write a variation on the function in part a that returns a much larger set with each element in the first set matched with each element in the second set. The two input sets no longer need to be the same size.

#7. [Question source: Gries et al,. *Practical Programming*, 3rd Edition]

A sparse vector is a one whose entries are almost all zero, e.g., [1, 0, 0, 0, 0, 0, 3, 0, 0, 0]. To save memory, such sparse vectors can be stored as a dictionary that just keeps the non-zero entries. For example, the vector above could be represented as {0:1, 6:3}: the value 1 at index 0 and the value 3 at index 6.

a) The sum of two vectors is the element-wise sum of their elements. For example, the sum of [9, 9, 9] and [1, 2, 3] is [10, 11, 12]. Write a function called `sparse_add` that takes two sparse vectors stored as dictionaries and returns a new dictionary representing their sum.

b) The dot product of two vectors is the sum of the products of corresponding elements. For example, the dot product of [1, 2, 3] and [4, 5, 6] is 4+10+18, or 32. Write a function called `sparse_dot` that calculates the dot product of two sparse vectors.

c) It would be useful to automatically "sparsify" a vector. Write a function called `sparsify` that takes in a list of integers and returns a dictionary representing a sparse vector.

d) What is the output sparse vector of the following command (that uses a and c)?

sparse_add(sparsify([0,0,1,-2,0,0]), sparsify([0,1,0,2,0,0])

Depending on how you implemented the method above, the result may not be a well-formed sparse vector (i.e., does it have an entry for a zero element?). Write a function that takes in a sparse vector that may not be well-formed in this way and return a well-formed sparse vector.

e) As an alternative to d), re-write your `sparse_add` function to always return a valid sparse vector.

#8. Write a program that asks a user to enter a string. Output letter that appears the most times and the number of times it appears. You need to deal with both capital and lower-case letters.

For example:

Enter a word: Bubble

The letter b/B appears most often: 3 times.

Hint: Do it once using a list and then again using a dictionary. Don't look at Bonus #2 yet as it gives a big hint.

Bonus #1: Did you use the string `count()` function? This function reads through the entire string each time it is called. Since you need to count the number of occurrences of each letter in the word (let's say there are *n* letters), that means count will make *n* "steps" *n* times (i.e., you will look for each letter (n steps) once for each letter (n letters)). So your program will execute n^2 steps. See if you can write the solution without using `count()` such that you only look at each letter in the word once (or twice) not *n* times.

Bonus #2: The simplest solutions will use two loops (once through the word to count the letters and once through the dictionary (or list) to find the one that appears most). So you will look at each letter twice, once per loop. Can you re-do the solution using a dictionary with only one loop?