Problem 1

Give the time cost of each function, using big-Oh notation. Some functions have a parameter n; for the rest, I’ve added a comment, indicating what n represents.

def count_lengths(some_strings):
    # n = len(some_strings)
    counts = [0]*8
    for s in some_strings:
        counts[len(s)%8] += 1
    return counts

O(n)
def triangle(n):
    retval = ""
    for i in range(n):
        retval += "A"*(i+1) + "\n"
    return retval

O(n³) - although O(n²) is an easy mistake (Russ made it!)
- On each pass we add O(n) characters to the string
- Thus, the length of the string grows as O(n²)
- However, since we do O(n²) work per pass (because we are concatenating strings), we do O(n³) work overall!
HINT: The next function has a big-Oh cost which is different than examples we’ve seen before in this class.

```python
def is_prime(n):
    if n%2==0:
        return False
    d = 3
    while d*d <= n:
        if n%d==0:
            return False
        d += 2
    return True
```

\[ O(\sqrt{n}) \] - we iterate until \( d \times d \) passes \( n \)

NOTE: The ‘in’ operator on a dictionary is \( O(1) \). So is indexing into one. The \texttt{keys()} method is \( O(n) \).

```python
def count_vals(integers):
    # n = len(integers)
    counts = {}
    for i in integers:
        if i in counts:
            counts[i] += 1
        else:
            counts[i] = 1
    for k in counts.keys():
        print("{} showed up {} times." .format(k, counts[k]))
```

\( O(n) \) - the first loop takes \( O(1) \) per iteration, and runs \( O(n) \) times. The second loop can run as many as \( O(n) \) times; each pass is \( O(1) \)
For the next function, I've provided TWO variables: n and x, which count the data in two different ways. Give your big-Oh expression using the variable which better expresses the time cost.

```python
def total_len(list_of_strings):
    # n = len(list_of_strings)
    # x = total number of characters
    c = 0
    for s in list_of_strings:
        c += len(s)
    return c
```

**O(n)** - the time cost is proportional to the number of strings in the list

The next two functions do things which are a little bit similar to each other - but they do *NOT* have the same time cost. Give the time cost of each, and explain the difference.

```python
def triple(string):
    # n = len(string)
    return string+string+string
```

**O(n)** - we create a new string which has 3n characters

```python
def triad(string):
    # n = len(string)
    return [string,string,string]
```

**O(1)** - we create a list which has exactly three elements. But each element takes O(1) time to set, since they are references
**HINT:** The next two functions have time costs which are different than most of the examples we’ve seen before in this class.

```python
def pow_of_2_string(steps):
    # n = steps
    retval = "\n"
    for i in range(steps):
        retval = retval + retval
    return retval
```

**O(2^n)** - each pass makes the string twice as long. The length of the string, on the last pass, is \(2^n\) characters

```python
def what_pow_of_2(n):
    v = 1
    p = 0
    while v < n:
        v *= 2
        p += 1
    return p
```

**O(log n)** - if \(n\) gets twice as large, then the loop will have to run one more time.
Problem 2

An “iterator” is a class which allows you to count through a set of values. Simple iterators count over numeric ranges; more advanced iterators allow you to iterate through the elements of a data structure. (Python’s for loop uses iterators to iterate through a list, string, or other data structure.)

Write a class called Counter, which counts through range of integers; each time that the user calls next(), the method returns the next value. next() raises a StopIteration exception when/if there are no more values to deliver. (That is, Counter works a little like range().)

Implement the following methods. Do so without using range() anywhere in the class.

- __init__()
  
  Has one required parameter, which is the start number. Has one optional parameter (provide a default value for this), which is the end number (exclusive). If no end number is provided, then the counter will never terminate; it will keep counting all the way to infinity.

- next()
  
  Returns the next integer in the range (the first call to next() should return the start value, unless the end is <= the start). When the entire range has been returned, the next call to next() should do the following:
  
  raise StopIteration()

I’ve provided a blank page, on the next sheet, for your code.

EXAMPLE

The following code should print out the numbers 10,11,12:

```python
    count = Counter(10)
    print(count.next())
    print(count.next())
    print(count.next())
    print(count.next())
```
Solution will be posted online.
Problem 3

Concatenation is an expensive operation, since it requires us to copy all of the values of the list, tuple, or string into a new data structure. An alternative is to build an object which represents the concatenated values, instead of actually creating a new list.

Create a class named Joiner. It must take two parameters in its __init__() method; you may assume that these parameters are both lists, tuples, strings, or some other Python type which supports both the len() function and indexing.

Your Joiner class will emulate concatenation - that is, it will act as if we had concatenated the two lists together. Implement a method of this class named __getitem__(self,index), which works just like indexing into this list. (For simplicity, you don’t have to support negative indices; you also don’t have to check to see if the indices are valid. Just assume that they are positive, valid values.) Thus, if index=0, then return the [0] element of your first sub-list (unless it’s empty!). If index=len(first_sublist), then return the [0] element of the second sub-list. Etc.

If you have time at the end, type your class into Python, and then try the following code:

```python
j1 = Joiner([1,2,3],[4,5,6])
for i in range(6):
    print(j1[i])
```

It should print out:

```
1
2
3
4
5
6
```

Solution will be posted online.