LARSON—MATH 353–CLASSROOM WORKSHEET 14
Density of Primes—Analyzing Recursion.

1. Log in to CoCalc.

   (a) Start the Chrome browser.

   (b) Go to `https://cocalc.com`

   (c) Login (**your VCU email address** is probably your username).

   (d) You should see an existing Project for our class. Click on that.

   (e) Click "New", then "Worksheets", then call it **c14**.

   Primes & Gauss

   We wrote a function density_of_primes(n) that **returns the number of primes** in the integers up to n, divided by n. What we see is that the density appears to go down as n increases. Gauss conjectured that $f(n) = \frac{1}{log(n)}$ "approximates" the density of primes.

   Let $\pi(n)$ be the number of primes up to n. Then the density of the primes up to n is $\frac{\pi(n)}{n}$.

   The **Prime Number Theorem** says that $\frac{\pi(n)}{n} \sim \frac{1}{log(n)}$, that is,

$$lim_{n\to\infty} \frac{(\frac{\pi(n)}{n})}{(\frac{1}{log(n)})} = 1$$

2. Use `scatter_plot` to *visualize* the limit expression as values of n get larger and larger.

   Recursion

   A **recursive** function is a function that calls itself. It must always have a *base case* so that the recursion eventually stops.

3. The Fibonacci sequence F_n is defined as follows $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. Here is a recursive function `fibonacci(n)` that computes the n^{th} Fibonacci number.

```
def fibonacci(n):
    if n==0:
        return 0
    if n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

   Test your function to make sure that it works.

   A problem to investigate: Complexity Analysis We saw that this recursive Fibonacci function will recompute sub-case like `fibonacci(0)` repeatedly. It was conjectured that this might be exponential. Let's count, graph, conjecture, and modify!

4. Let's add a global counter to keep track of how many times gets computed for an input n.

```
count = 0
def fibonacci(n):
    global count
    if n==0:
        count = count + 1
        return 0
    if n==1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
print(count)
```

   Try this for a few values of n.

5. Now let wrap all this in a function `fib_count(n)` so that we can investigate systematically and graph. Put all the above code inside your new function and `return` the *count* counter.

6. Now use `scatter_plot` to *visualize* `fib_count(n)` for n in say [1..30].

7. Use a double scatter plot to compare this to 2^n on this interval.

8. The result is not exact. What else might we try? (What other functions can we try? Or can we modify our exponential function somehow?)

Weighted Coin Flip

A *weighted coin* is one that does not come up heads half the time. We'd like to investigate weighted coins. This will be a prelude to an investigation of the behavior of random graphs with specified edge probabilities.

9. Define a function `weighted_flip` which returns "heads" (or 1) 75% of the time, and "tails" (or 0) the rest of the time. How can you test that your function if doing what you want it to do?

10. Flip your weighted coin 100 times. Find the length of a longest streak of heads or tails.

11. How long do you think a longest streak will be (on average)?

12. Write some code to investigate.

13. **Getting your classwork recorded**

    When you are done, before you leave class...

    (a) Click the "Make pdf" (Adobe symbol) icon and make a pdf of this worksheet. (If Cocalc hangs, click the printer icon, then "Open", then print or make a pdf using your browser).

    (b) Send me an email with an informative header like "Math 353—c14 worksheet attached" (so that it will be properly recorded).