Lab 1

Introduction to Python, Part I

In order to start writing and running code as quickly and painlessly as possible, we will use an open-source web application called Jupyter, run through Google's cloud service Colaboratory[1]. Start by visiting colab.research.google.com and login with your Google account credentials. On the dialog that appears, create a new Python 3 Notebook (blue text at the bottom of the dialog) and call it Sandbox (or something similar). You can also create new notebook for each lab throughout the semester. For today, create a second notebook called Math 495R Lab 1. You can have both of these notebooks open in separate tabs in your browser.

For the first three labs, we will be following the Introduction to Python book written for the BYU ACME program. You can find it here (click on "Python Essentials"):

https://foundations-of-applied-mathematics.github.io/

Start reading "Python Basics" on page 6. All you need to know about Jupyter is that to run a chunk of code, type SHIFT + ENTER.

The best way to learn a new coding language is by actually writing code. Follow along with the examples in the yellow code boxes in the Introduction to Python book by executing them in your Sandbox file. Avoid copy and paste for now; your fingers need to learn the language as well.

1. When you get to Problem 2 in the blue box on page 8, follow the instructions below instead. Write your program in your Lab 1 file.

The volume of a sphere with radius r is $V = \frac{4}{3}\pi r^3$. Define a function called sphere_volume() that accepts a single parameter r. Return the volume of the sphere of radius r, using 3.14159 as an approximation for π (for now). Also write an appropriate docstring for your function.

```
>>> sphere_volume(1.59)
16.83757779948
```

[1] If you would like to download Python to run locally on your machine, follow the instructions on page 3 of the ACME Introduction to Python book. Make sure you are using Python 3.

Now define a one-line function `V`, using the `lambda` keyword, that also computes the volume of a sphere.

```
>>> V(24)
57905.78687999999
```

2. When you get to Problem 3 in the blue box on page 10, stop reading the book (for today) and do that problem. It's copied here for your convenience.

   Write a function called `isolate()` that accepts five arguments. Print the first three separated by 5 spaces, then print the rest with a single space between each output. For example:

   ```
   >>> isolate(1, 2, 3, 4, 5)
   '1     2     3 4 5'
   ```

3. Experiment and figure out the order of operations that Python uses for the operations `+`, `-`, `*`, `/` (division), `**` (exponentiation), and `%` (remainder after division) when more than one operation occurs in the same expression. Explain your process and conclusions.

Some of you already have experience with programming coming in to Math 495R; those of you who do probably finished this lab very quickly. If there is still lab time remaining, try to solve the following challenge problems. Don't turn these in for credit; they are just for your personal edification.

Challenge 1. 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder. What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20? (Project Euler #5)

Challenge 2. If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000. (Project Euler #1)

Lab 2

Introduction to Python, Part II

In the ACME Introduction to Python book, start reading where we left off last time, beginning at "Data Types and Structures." Remember to follow along with the examples in the yellow code boxes in the book by executing them in your `Sandbox` file.

1. After reading about float division vs. integer division, do the following in your `Math 495R Lab 2` notebook.

   Write a function `last_two_deleted()` that accepts a positive integer and returns the integer with the last two digits deleted. The function should return zero if the input is in the range $[1, 99]$. Use integer division.

   ```
   >>> last_two_deleted(246810)
   2468
   ```

   Write a function `last_two()` that accepts a positive integer and returns the last two digits (as an integer). The function should return a one-digit integer if the second-to-last digit is zero. Use the modulo operator `%`.

   ```
   >>> last_two(246810)
   10
   ```

2. Do Problem 4 in the Introduction to Python book.

   Write a function called `first_half()`, which should accept a (string) parameter and return the first half of it, excluding the middle character if the string has an odd number of characters (use integer division for this). Use the built-in function `len()` to get the length of the input string.

   ```
   >>> first_half('diophantine')
   'dioph'
   ```

Write a function called `backward()`, which should accept a (string) parameter and reverse the order of its characters using slicing, then return the reversed string. The `step` parameter in `[start:stop:step]` can be negative.

```
>>> backward('desserts')
'stressed'
```

3. Feel free to do problems 5 and 6 on your own, but do not turn them in for credit. When you get to problem 6, instead do the following.

Write a function `int_parity()` using `if` statements that asks the user to type in a number and then prints whichever of the following statements is true.

- `That integer is even.`
- `That integer is odd.`
- `That is not an integer.`

Use `input()` to ask for user input. You can check if a string of characters represents an integer by using the `isdigit()` function, e.g. `mystr.isdigit()`. Once you have determined that the input is an integer, use the modulo operator `%` to determine its parity (evenness or oddness).

```
>>> int_parity()
Enter a number: 8675309
'That integer is odd.'
```

If there is still lab time remaining, try to solve the following challenge problem. Don't turn this in for credit; it is just for your personal edification.

Challenge 3. The sum of the squares of the first ten natural numbers is

$$1^2 + 2^2 + \ldots + 10^2 = 385.$$

The square of the sum of the first ten natural numbers is

$$(1 + 2 + \ldots + 10)^2 = 3025.$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$. Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum. (Project Euler #6)

Lab 3

Introduction to Python, Part III

In the ACME Introduction to Python book, start reading where we left off last time, beginning at "The While Loop." Remember to follow along with the examples in the yellow code boxes in the book by executing them in your `Sandbox` file.

1. After reading about `while` loops, write the following function.

   Write a function `int_to_str26` that, given an integer m, returns the corresponding string in the 26 character alphabet

   `A = 00, B = 01, C = 02, D = 03,..., Z = 25,`

   or returns an error message if the integer does not correspond to a string. Use a `while` loop to look at the last two digits of m and convert those digits to a character, then repeat with a smaller integer $m' =$ (m with the last two digits deleted).

```
>>> int_to_str26(20019)
'CAT'
>>> int_to_str26(1904181922141703)
'TESTWORD'
```

2. After you have read about `for` loops, do the following problem.

   Write a function `str_to_int26` that, given a string of capital letters, returns an integer using the 26 character alphabet. Use a `for` loop. If you prefer, you can keep track of the integer using a string (e.g. '123456'), then convert it to an integer at the end (e.g. `int('123456')`).

```
>>> str_to_int26('CAT')
20019
>>> str_to_int26('DOESTHISFUNCTIONWORK')
314041819070818052013021908141322141710
```

3. After you have read about list comprehension, do Problem 8 in the ACME Introduction to Python book.

The alternating harmonic series is defined as

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots = \ln(2).$$

Write a function called `alt_harmonic()` that accepts an integer n. Use a list comprehension to quickly compute the first n terms of this series (be careful not to compute only $n-1$ terms). The sum of the first 500 000 terms of this series approximates $\ln(2)$ to five decimal places. Use the python built-in function `sum()`.

```
>>> alt_harmonic(10)
0.6456349206349207
>>> import math
>>> math.log(2)
0.6931471805599453
```

If there is still time remaining, try to solve the following challenge problem (no credit).

Challenge 4. Consider all integer combinations of a^b for $2 \le a \le 5$ and $2 \le b \le 5$:

$$\begin{array}{llll} 2^2 = 4, & 2^3 = 8, & 2^4 = 16, & 2^5 = 32 \\ 3^2 = 9, & 3^3 = 27, & 3^4 = 81, & 3^5 = 243 \\ 4^2 = 16, & 4^3 = 64, & 4^4 = 256, & 4^5 = 1024 \\ 5^2 = 25, & 5^3 = 125, & 5^4 = 625, & 5^5 = 3125 \end{array}$$

If they are then placed in numerical order, with any repeats removed, we get the following sequence of 15 distinct terms:

$$4,\ 8,\ 9,\ 16,\ 25,\ 27,\ 32,\ 64,\ 81,\ 125,\ 243,\ 256,\ 625,\ 1024,\ 3125$$

How many distinct terms are in the sequence generated by a^b for $2 \le a \le 100$ and $2 \le b \le 100$? (Project Euler #6)

Lab 4

Introduction to LaTeX

In this assignment, we will practice typesetting some basic math in LaTeX. For today, visit `overleaf.com` and sign up for a personal free account.[1]

Change the title to `EMC2 Lab 4`, change the author to your name, change the date to `\date{\today}`, and delete the `\section` line. Then click "Recompile" to generate an updated pdf. Now read the following short crash-course in using LaTeX. Another excellent resource can be found at `www.overleaf.com/learn`. See also Prof. Doud's lecture notes.

To display

$$a \in A,$$

one would write the text

```
\[a \in A\].
```

The command `\[` tells LaTeX that you are beginning an equation and you want it to be centered on its own line. The command `\]` tells LaTeX that your equation is over. The `a` and `A` are hopefully self explanatory. The `\in` is very important. Most keyboards don't have a key with an $\in$, or most other non alphanumeric symbols for that matter, on it. Every symbol command in LaTeX begins with an \ followed by a word. Most of the words make sense. For example, the command `\in` makes sense, since the sentence "$a \in A$" can be read as "The element a is *in* the set A."

In order to type an equation or mathematical symbol in the line of text that you are typing, you need to go into math mode. Surround the mathematics with single `$`. So if you wanted to type an equation like $ax^2 + bx + c = 0$, you would type `$ax^2+bx+c=0$`.

At this point you know very few symbols. Some of the important ones are illustrated below.

[1]If you would like to download LaTeX for local use on your personal machine, visit `www.latex-project.org/get/` and follow the instructions for your platform.

TeX Code	Output	
`$a\in A$`	$a\in A$	
`$A\times B$`	$A\times B$	
`$A\cdot B$`	$A\cdot B$	
`$\emptyset=\{\}$`	$\emptyset=\{\}$	
`$P\implies (Q\lor R)$`	$P\implies (Q\lor R)$	Note that `\lor` stands for "logical or"
`$P\implies (Q\land R)$`	$P\implies (Q\land R)$	Note that `\land` stands for "logical and"
`$2\neq 3$`	$2\neq 3$	
`$x\notin\{a,b,c\}$`	$x\notin\{a,b,c\}$	
`$A\subseteq B$`	$A\subseteq B$	
`$X\subsetneq Y$`	$X\subsetneq Y$	

A vertical line can be obtained by a shifted backslash (above the enter key on your keyboard). It can also be obtained by `\vert`, so you can type $|a|$ by either `$|a|$` or `$\vert a\vert$`. Note also that set brackets need to have a backslash in front of them–they have a different meaning to TeX if the backslash is omitted.

Your assignment

Turn in a replica of the following two paragraphs (don't worry about font size, or getting the line breaks to match up–just produce readable text that says the same thing, with the same appearance):

The Cartesian product (or simply the product) $A\times B$ of two sets A and B is the set consisting of all ordered pairs whose first coordinate belongs to A and whose second coordinate belongs to B. In other words,

$$A\times B=\{(a,b):a\in A \text{ and } b\in B\}.$$

For example, if $A=\{x,y\}$ and $B=\{1,2,3\}$, then

$$A\times B=\{(x,1),(x,2),(x,3),(y,1),(y,2),(y,3)\};$$

while

$$B\times A=\{(1,x),(1,y),(2,x),(2,y),(3,x),(3,y)\}.$$

Since, for example, $(x,1)\in A\times B$ and $(x,1)\notin B\times A$, these two sets do not contain the same elements; so $A\times B\neq B\times A$. If $A=\emptyset$ or $B=\emptyset$, then $A\times B=\emptyset$.

For the sets A and B just mentioned, $|A|=2$ and $|B|=3$; while $|A\times B|=|B\times A|=6$. Indeed, for all finite sets A and B,

$$|A\times B|=|A|\cdot|B|.$$

Hints

1. Pay close attention to your logs and output files in Overleaf (click the button just to the right of "Recompile"). Among other things, this will help you check that you have matching open/close delimiters, e.g. `$$, \{\}, \[\]`.

2. Note that 'A' is different than 'A'. The first is written as `$A$` and the second as `A`. The difference in font is the difference between a math symbol and an indefinite article.

3. To get the word 'and' to show up in the equation as 'and' and not as '*and*', write '`\text{ and }`'.

Challenge

If there is time left, try to replicate every displayed math line (meaning all of the `\[...\]` lines) on page 5 of these Analytic Number Theory lecture notes:

`http://math.byu.edu/~nick/ant-notes.pdf`

Do not turn this in for credit. You may find the Detexify symbol lookup helpful.

Lab 5

Vector Arithmetic Using Python Lists

In this lab you will program certain basic vector operations using standard Python Lists to represent vectors. In each problem your program must allow for vectors of any size. However, if your functions are given vectors of incompatible sizes, your program should raise an exception to notify the user. You can do this by including the line

```
raise Exception('Error:  Vectors are different sizes')
```

Unless appropriately caught, an exception will immediately terminate not only the current function, but also every function above it in the stack. So for instance if function A calls function B which calls function C, and C raises an exception, then all three functions will terminate without returning a value, and the exception message will be printed.

1. Write the following function to perform scalar multiplication on a vector.

   Write a function `scalar_mult` that takes as input a scalar s and a vector $\mathbf{v}$ and returns the vector $s \cdot \mathbf{v}$. The input and output vectors should be represented as Python List data types.

   ```
   >>> scalar_mult( 4, [ 1, 2 ] )
   [ 4, 8 ]
   >>> scalar_mult( 3, [ 1., 0., 0.5 ] )
   [ 3., 0., 1.5 ]
   ```

2. Write the following function that performs vector addition.

Write a function `vector_add` that takes as input two vectors $\mathbf{v}$ and $\mathbf{w}$ and returns the vector $\mathbf{v}+\mathbf{w}$. The input and output vectors should be represented as python list data types. Your function should check to ensure the vectors are the same size. If not, your function should raise an exception with an appropriate message.

```
>>> vector_add( [ 1, -1, 0 ], [ 1, 2, 3 ] )
[ 2, 1, 3 ]
>>> vector_add( [ 1.5, -.5 ], [ -1, 1 ] )
[ -0.5, .5 ]
>>> vector_add( [ 0, 2 ], [ 1, 5, -4 ] )
Error: Vectors are different sizes
```

3. After reviewing the definition of the dot product of two vectors given in your linear algebra text, write the following function.

Write a function `dot_product` that takes as input two vectors $\mathbf{v}$ and $\mathbf{w}$ and returns the standard dot product $\mathbf{v}\cdot\mathbf{w}$. The input and output vectors should be represented as python list data types. Your function should check to ensure the vectors are the same size. If not, your function should raise an exception with an appropriate message.

```
>>> dot_product( [ 1, -1, 0 ], [ 1, 2, 3 ] )
-1
>>> dot_product( [ 1, 3 ], [ 4, 0 ] )
4
>>> dot_product( [ 0, 2 ], [ 1, 5, -4 ] )
Error: Vectors are different sizes
```

Challenge

Remember the Law of cosines tells us that

$$\cos\theta = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|},$$

where θ is the angle between vectors $\mathbf{u}$ and $\mathbf{v}$. Write a function that takes as input two vectors $\mathbf{u}$ and $\mathbf{v}$, and computes the cosine of the angle between them. Do not turn this in for credit. You may find the math.sqrt function useful to compute the norms of vectors.

Lab 6

The Caesar Cipher

The Caesar (or shift) cipher with integer key n encodes a string of letters by doing the following to each letter:

(a) translate the letter to an integer using the 26 character alphabet

`A = 00, B = 01, ..., Z = 26,`

(b) add n,

(c) reduce the result modulo 26 (take the remainder after dividing by 26), and

(d) translate the result back to a letter.

The effect is to shift each letter forward in the alphabet by n spaces, wrapping around at the end of the alphabet.

1. Write a function `shift_encrypt` that takes an integer n and a string of capital letters and encrypts the string using the Caesar cipher with key n.

   ```
   >>> shift_encrypt(3, ‘CAT’)
   ‘FDW’
   ```

   You may use your functions `int_to_string26` and `string_to_int26` from Lab 3.

2. Write a function `shift_decrypt` that takes an integer and a string of capital letters and decrypts the string.

   ```
   >>> shift_decrypt(3, ‘FDW’)
   ‘CAT’
   ```

3. Decrypt the coded message `KYV TRBV ZJ R CZV` by running your `shift_decrypt` function for every n in $\{1, 2, 3, \ldots, 25\}$ and reading off the only one that makes sense.

Lab 7

Matrix Row Reduction

In this lab you will write a functions to performs basic matrix operations related to row-reducing a matrix. We will represent a matrix as an array of arrays where each inner array represents a **row** of the matrix. So, for example, the matrix $\left(\begin{smallmatrix}1&2\\3&4\end{smallmatrix}\right)$ would be represented in the program as `[[1,2],[3,4]]`. As in the previous lab, we will represent a column vector as a simple list of numbers. The functions you write for this lab should work for matrices of any size.

1. Write the following function to swap rows in a given matrix.

   Write a function `row_swap` which takes as input a matrix A, and two indexes i and j. Your function should return the matrix obtained from A with rows i and j swapped.

   ```
   >>> row_swap( [ [ 1, -1, 1 ], [ 0, 1, 3 ], [ 2, -2, 0] ], 0, 2)
   [ [ 2, -2, 0], [ 0, 1, 3 ], [ 1, -1, 1 ] ]
   >>> row_swap( [ [ 2, -1, 3 ], [ 1, 2, 3 ] ], 0, 1)
   [ [ 1, 2, 3 ], [ 2, -1, 3 ] ]
   ```

2. Write the following function to multiply a row of a given matrix by a constant.

   Write a function `row_mult` which takes as input a matrix A, one index i, and a scalar c. Your function should return the matrix obtained from A with row i multiplied by c.

   ```
   >>> row_mult( [ [ 1, 1 ], [ 2, 3 ] ], 1, 3)
   [ [ 1, 1 ], [ 6, 9 ] ]
   >>> row_mult( [ [ 1, 1 ], [ 6, 9 ] ], 0, 0)
   [ [ 0, 0 ], [ 6, 9 ] ]
   ```

3. Write the following function that adds a multiple of one row to another.

Write a function `row_add` which takes as input a matrix A, two indexes i and j, and a scalar c. Your function should return the matrix obtained from A with row i replaced with itself plus c times row j.

```
>>> row_add( [ [ 0, 1, 1 ], [ 1, -1, 3 ], [ 1, 3, 2] ], 2, 0, -3)
[ [ 0, 1, 1 ], [ 1, -1, 3 ], [ 1, 0, -1] ]
>>> row_add( [ [ 2, 1 ], [ 1, -2 ] ], 0, 1, 0)
[ [ 2, 1 ], [ 1, -2 ] ]
```

Challenge

a) Write a function that determines whether or not a matrix is in echelon form.

b) Write a function that row-reduces a matrix to echelon form. The hard part of this problem is determining when to swap rows.

Lab 8

The Vigenère Cipher

The Vigenère cipher is a variant of the Caesar cipher. Its key is a list of integers between 0 and 25; often, the entries correspond to the letters of some key word, e.g.

`KEY = 10 4 24.`

To encrypt a message using the key `KEY`, we shift the first letter of the message by 10, the second letter of the message by 4, the third letter by 24, the fourth letter by 10, the fifth by 4, and so on, starting again at the beginning of the key after reaching the end. These shifts can be represented by addition modulo 26 after converting the message to a string of numbers between 0 and 25.

1. Write a function `vigenere_encrypt` that takes a keyword (written as a string of capital letters) of arbitrary length and a plaintext message (written as a string of capital letters) and encrypts the message using that keyword.

   ```
   >>> vigenere_encrypt('KEY', 'MESSAGE')
   'WIQCEEO'
   ```

2. Write a function `vigenere_decrypt` that takes a keyword (written as a string of capital letters) of arbitrary length and a coded message (written as a string of capital letters) and decrypts the coded message using that keyword.

   ```
   >>> vigenere_decrypt('KEY', 'CIABIR')
   'SECRET'
   ```

3. The coded message

   `GLTYMVWCVBUMCNEUMROQFWTJZRXHHDRMIZPNPAIMEIKLIYIS`

   was encrypted using a Vigenère cipher with one of the keys below:

   `ALMA, ETHER, HELAMAN, MORONI, NEPHI, SARIAH, ZARAHEMLA`

Use a `for` loop to decrypt the ciphertext using each of the keys and determine the coded message.

4. In a future lab we will write a program that decrypts a ciphertext encoded with the Vigenère cipher without knowing any of the possible keys. This is particularly difficult because: not only do we not know the key, we don't even know the length of the key!

   In this problem, we will determine the most likely key length of an encrypted cipher text, using the following measurement. Whenever the characters in a string at positions n and $n + K$ are the same, we call this a *coincidence at distance* K. The most likely key length K is the smallest distance with the most coincidences in the cipher text.[1]

   a) Write a function `num_coincidences` that takes a string of capital letters and a positive integer K, and returns the number of coincidences at distance K.

   ```
   >>> str = 'GLTYMVWCVBUMCNEUMROQFWTJZRXHHDRMIZPNPAIMEIKLIYIS'
   >>> num_coincidences(str, 8)
   3
   ```

   b) Write a function `key_length` that takes a string of capital letters and returns the smallest distance $1 \leq K \leq 20$ with the most coincidences (if there are several distances that give the maximum number of coincidences, your program should return the smallest distance). Try your function with the ciphertext at `pastebin.com/q3k3CKQU`. The key length for this ciphertext is 7.

[1] Later we will see why this is the case.

Lab 9

Matrix Multiplication

In this lab you will program certain basic matrix operations related to matrix multiplication. As before, we will represent a matrix as an array of arrays where each inner array represents a row of the matrix. The functions you write for this lab should work for matrices and vectors of any size. You will find it useful to use the use the `dot_product` function from Lab 5 in problems 1 and 3. If the input matrices or vectors for any function have incompatible sizes your function should raise an exception. If your function calls another function which raises an exception (such as the `dot_product` function from the previous lab), you may not need to add any additional code.

1. Write the following function to perform multiplication of a matrix and a vector.

   Write a function `matXvec` that takes as input a matrix M and a vector $\mathbf{v}$ and returns the vector $M\mathbf{v}$.

   ```
   >>> matXvec( [ [1, -1], [1,2 ] ], [ 2, -1 ] )
   [ 3, 0 ]
   >>> matXvec( [ [0, 2], [1, 4] ], [ 3, -1, 7 ] )
   Error: Vectors are different sizes
   ```

2. Write the following function to extract a column of a matrix.

   Write a function `mat_col` that takes as input a matrix and an index and returns the column of the matrix at that index.

   ```
   >>> mat_col( [ [1, 2], [ 3, 4] ], 0 )
   [ 1, 3 ]
   >>> mat_col( [ [1, 2, 3 ], [3, 4, 5 ], [5, 6, 7 ] ], 2 )
   [ 3, 5, 7 ]
   ```

3. Write the following function to perform matrix multiplication.

Write a function `mat_mult` that takes as input two matrices M and N and returns their product MN.

```
>>> mat_mult( [[ -1, 2], [-1, 1]], [[1, 0],[1, 1]] )
[ [ 1, 2 ], [ 0, 1] ]
>>> mat_mult([[0,1,0],[0,0,1],[1,0,0]],[[1,2,3],[3,4,5],[5,6,7]])
[ [3,4,5], [5,6,7], [1,2,3] ]
```

Challenge

Write a function `mat_pow` that takes as input a square matrix M and a positive integer e and computes the matrix M^e obtained by multiplying M by itself e times.

Lab 10

Counterexamples

In this lab we will use a computer search to find counterexamples to some statements that seem plausible, given a handful of initial data, but are actually false. For each "proposition" listed below, determine the smallest counterexample. In Propositions 3 and 4, the false statement is underlined.

Proposition 1. Every number of the form $2^{2^k} + 1$ is prime, for $k = 0, 1, 2, 3, \ldots$.

Hint: The SymPy package has some useful functions for primality testing and factoring:

```
>>> from sympy import isprime, factorint
```

Proposition 2. Whenever p is a prime number, $2^p - 1$ is also a prime number.

Hint: You can get the n-th prime using `prime(n)` after you import `prime` from `sympy`.

Proposition 3. Every prime larger than 2 is congruent to either 1 or 3 modulo 4. Let $\pi_1(x)$ denote the number of primes $\leq x$ which are 1 modulo 4, and define $\pi_3(x)$ likewise. <u>Then $\pi_3(x)$ is always in first place (or tied for first) in the "prime number race," that is, $\pi_3(x) \geq \pi_1(x)$ for all $x \geq 3$</u>.

Interesting fact: it has been proven that $\lim\limits_{x\to\infty} \frac{\pi_1(x)}{\pi_3(x)} = 1$, meaning that "at infinity" the prime number race ends in a tie.

Proposition 4. For each n, factor the polynomial $x^n - 1$ as much as possible, assuming that only integer coefficients are allowed. For example,

$$x^2 - 1 = (x-1)(x+1), \quad x^3 - 1 = (x-1)(x^2+x+1), \quad x^4 - 1 = (x-1)(x+1)(x^2+1), \ldots.$$

The polynomials $x - 1$, $x + 1$, $x^2 + 1$, $x^2 + x + 1$, $\ldots$, are called *cyclotomic polynomials.* <u>Every coefficient of every cyclotomic polynomial is in the set $\{1, 0, -1\}$</u>.

Hint: For this problem, you should start by running the following code:

```
>>> from sympy import factor_list, symbols, Poly
>>> x = symbols('x')
```

Then you can get a list of factors of a polynomial by running code like `factor_list(x**10-1)[1]`. This will output a list of ordered pairs, with the first element of each pair being a polynomial, and the second element being the exponent of that factor. Then for each factor you can do something like `Poly(fac).coeffs()` to get a list of the coefficients.

Lab 11

Solving Systems of Linear Equations

In this lab you use the Numpy matrix objects and construct a program that will reduce an *integer* matrix to echelon form. You can import Numpy using the following command:

```
>>> import numpy as np
```

Matrix objects in Numpy are stored like arrays of arrays where each inner array represents a row of the matrix. Remember these arrays are zero-indexed!

```
>>> M=np.matrix([[1,2,3],[4,5,6]])
>>> M[1]
matrix([[ 4, 5, 6]])
>>> M[1,0]
4
>>> M[1]=3*M[1]
>>> M
matrix([[ 1,  2,  3],
        [ 12, 15, 18]])
```

The dimensions of the matrix can be accessed using the `shape` property. Note, this is not a method so do not use parentheses!

```
>>> M.shape
(2,3)
```

You can learn more about the numpy matrix object by reading
https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matrix.html

1. Write a function to swap rows if necessary in the row-reduction algorithm.

Write a function `row_swap` that takes as input a matrix object A and two indices i and j of a row and a column respectively. If the entry $A[i, j] \neq 0$, your function should return the matrix A unaltered. Otherwise your function should step through the entries $A[k, j]$ of the j-th column with $k \geq i$. If it finds an entry $A[k, j] \neq 0$, your function should swap the i-th and k-th rows. You may need to use the `.copy()` method to create copy a of the row while you make the swap. If there is no such entry $A[k, j] \neq 0$, return the original matrix A unaltered.

```
>>> A=np.matrix([[1,2],[3,4]])
>>> row_swap(A,0,0)
matrix([[ 1,  2],
        [ 3,  4]])
>>> B=np.matrix([[0,2],[3,4],[5,6]])
>>> row_swap(B,0,0)
matrix([[ 3,  4],
        [ 0,  2],
        [ 5,  6]])
>>> C=np.matrix([[1,2,3],[0,0,6],[0,8,9]])
>>> row_swap(C,1,1)
matrix([[1, 2, 3],
        [0, 8, 9],
        [0, 0, 6]])
```

2. Write a function to add rows in the row-reduction algorithm.

Write a function `row_add` that takes as input a matrix object A and two indices i and j of a row and a column respectively. If the entry $A[i,j] \neq 0$, your function should iterate through the rows $A[k]$ of A with index $k > i$. At each step, replace the row $A[k]$ with the row $A[i,j] * A[k] - A[k,j] * A[i]$. Otherwise, return the original matrix A unaltered. Do not divide! If the input is an integer matrix your function should always return an integer matrix.

```
>>> A=np.matrix([[1,2],[3,4]])
>>> row_add(A,0,0)
matrix([[ 1,  2],
        [ 0,  -2]])
>>> B=np.matrix([[3,4],[0,2],[5,6]])
>>> row_add(B,0,0)
matrix([[ 3,  4],
        [ 0,  2],
        [ 0, -2]])
>>> C=np.matrix([[1,2,3],[0,5,6],[0,8,9]])
>>> row_add(C,1,1)
matrix([[1, 2, 3],
        [0, 5, 6],
        [0, 0, -3]])
```

3. Write a function that row-reduces a given integer matrix.

Write a function `ref` that takes as input a matrix object A and reduces the matrix to echelon form. If the input matrix is an integer matrix, your function should return an integer matrix. Your function should only have one `for`-loop (outside of any function calls) which should iterate through the index j of the columns of A. Create a variable i for the row index which should be initialized to 0 and should be incremented as you identify the leading term in a given row. For instance, your function will begin with $i = j = 0$. Your function should call `row_swap(A,i,j)` which will attempt to place a non-zero entry at $A[0,0]$. If `row_swap` fails to find such an entry, then j should increment, but i should not since you have not yet found a leading term in the i-th row. Otherwise, you have found a leading term. Your function should now call `row_add(A,i,j)` to reduce the rows below the i-th row. Then *both* i and j should increment. The process should terminate once either $i > m$ or $j > n$, where m and n are the dimensions of A.

```
>>> A=np.matrix([[1,2],[3,4]])
>>> ref(A)
matrix([[ 1,  2],
        [ 0,  -2]])
>>> B=np.matrix([[0,2],[3,4],[5,6]])
>>> ref(B)
matrix([[ 3,  4],
        [ 0,  2],
        [ 0,  0]])
>>> C=np.matrix([[1,2,3],[0,0,6],[0,8,9]])
>>> ref(C)
matrix([[1, 2, 3],
        [0, 8, 9],
        [0, 0, 6]])
```

Challenge

Write a function `rref` that takes as input a matrix A where the entries are decimal numbers, and returns A in reduced echelon form. Since the entries are now decimal numbers instead of integers, your `row_swap` function may have some trouble recognizing 0. You may modify to recognize any number z with $|z| < 10^{-5}$ as zero.

Lab 12

Recursion

Recursion, a close cousin to mathematical induction, is an important problem solving technique used in computer programming. A recursive function is one that calls itself (similar to how a proof by induction uses $P(n)$ to prove $P(n+1)$). Consider the following program that computes the sum of all the integers from 1 to n:

```
def recursive_sum(n):
    if n==1:
        return 1
    else:
        return n + recursive_sum(n-1)
```

As with proof by induction, we start with a base case (the `if` statement that checks if $n = 1$). Among other things, this helps ensure that our program will terminate eventually. The "inductive step" tells the program how to compute the sum up to n once it knows the sum up to $n-1$. Here is what is happening in the background when we call `recursive_sum(5)`:

```
recursive_sum(5) = 5 + recursive_sum(4)
                 = 5 + (4 + recursive_sum(3))
                 = 5 + (4 + (3 + recursive_sum(2)))
                 = 5 + (4 + (3 + (2 + recursive_sum(1))))
                 = 5 + (4 + (3 + (2 + (1))))
```

Once the base case is reached, the recursion terminates and the whole stack is evaluated. Then `recursive_sum(5)` returns `15`, which is the sum of all the integers between 1 and 5.

1. Recall that the factorial function is defined on positive integers as

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

Write a function `fac(n)` that computes $n!$ recursively. Your program should raise an exception if the input is not a positive integer.

```
>>> fac(7)
5040
>>> fac(30)
265252859812191058636308480000000
```

2. In section 15.C of the Math 290 textbook math.byu.edu/~doud/Transition an algorithm is given (in the form of the proof of Proposition 15.3) to compute the binary expansion of a given integer. The idea is:

```
the binary expansion of 1 is '1'
if n is odd:
  the binary expansion of n is [binary expansion of (n-1)/2] followed by '1'
if n is even:
  the binary expansion of n is [binary expansion of n/2] followed by '0'
```

If you are unfamiliar with binary arithmetic, see

https://en.wikipedia.org/wiki/Binary_number#Counting_in_binary

Write a function `binary(n)` that returns the binary expansion of n as a string. Your program should raise an exception if the input is not a positive integer.

```
>>> binary(45)
'101101'
>>> binary(2**30)
'1000000000000000000000000000000'
```

3. The Fibonacci numbers are a collection of natural numbers labeled $F_1, F_2, F_3, \ldots$ and defined by the rule

$$F_1 = F_2 = 1,$$

and for $n \geq 3$,

$$F_n = F_{n-1} + F_{n-2}.$$

Write a function `fib(n)` that recursively computes the n-th Fibonacci number F_n. Your program should raise an exception if the input is not a positive integer.

```
>>> fib(15)
610
>>> fib(30)
832040
```

4. Follow the proof of Proposition 14.6 in the Math 290 textbook to write the following function computing powersets.

Write a function `power_set(n)` that recursively computes the powerset of a given set of integers, input as a list. The order of the resulting list does not matter.

```
>>> power_set([1,2,3])
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
>>> power_set([])
[[]]
```

Hint 1: the `append()` function is probably not helpful here, since it does not return a new list. You might try using the code `list+[a]` to add the element `a` to your list and return the result.

Hint 2: another way to state the idea in Proposition 14.6 is to say that: given $a \in S$,

$$\mathcal{P}(S) = \mathcal{P}(S - \{a\}) \bigcup \{X \cup \{a\} : X \in \mathcal{P}(S - \{a\})\}.$$

Lab 13

Inverting matrices

In this lab you will use the Sympy Matrix objects and construct a program that will invert a matrix. Note that Sympy Matrix objects behave a little differently than Numpy matrix objects. You can import Sympy using the following command:

```
>>> import sympy as sp
```

Rows and entries of Sympy Matrix objects can be accessed in the following ways.

```
>>> M=sp.Matrix([[1,2,3],[4,5,6]])
>>> M.row(1)
Matrix([[4, 5, 6]])
>>> M[1,0]
4
>>> M[4]
5
```

Notice that if you only use a single bracket index you get the entry as though the matrix were a single list, not the row! The dimensions of the matrix can be accessed using the `shape` property, same as for Numpy matrix object. Remember, this is not a method so do not use parentheses.

```
>>> M.shape
(2,3)
```

You can learn more about the Sympy Matrix object by reading https://docs.sympy.org/0.7.2/modules/matrices/matrices.html or the Acme introduction to Sympy, starting from page 8:
http://www.acme.byu.edu/wp-content/uploads/2017/08/Sympy.pdf.

1. Write a function to augment a given matrix by the identity matrix.

   Write a function `aug_id` that takes as input an $m \times n$ matrix object A and augments the matrix by the identity matrix. There is no built-in sympy function to do this, so you will need to create a new matrix object of the appropriate size and edit its entries.

   ```
   >>> A=sp.Matrix([[1,2],[3,4]])
   >>> aug_id(A)
   Matrix([
   [ 1,  2, 1, 0],
   [ 3,  4, 0, 1]])
   >>> B=sp.Matrix([[0,2],[3,4],[5,6]])
   >>> aug_id(B)
   Matrix([
   [ 0,  2, 1, 0, 0],
   [ 3,  4, 0, 1, 0],
   [ 5,  6, 0, 0, 1]])
   ```

   Sympy Matrix objects have a built-in method `rref` that returns the matrix in reduced echelon form.

   ```
   >>> A=sp.Matrix([[1,2],[3,4]])
   >>> aug_id(A).rref()
   (Matrix([
    [1, 0, -1],
    [0, 1,  2]]), (0, 1))
   ```

   The first returned value is the reduced matrix; the second is a tuple contianing the indexes of the columns containing leading terms.

2. Write a function to invert a matrix.

Write a function `mat_inv` that takes as input an $m \times n$ matrix object A and returns a potential inverse. Use the function from part one to augment the input matrix by the identity, then use the `rref` method to row reduce the augmented matrix. Your function should then extract the matrix consisting of the final $n \times n$ columns of the row-reduced matrix. If A is invertible, this will be the inverse. Your function should still return a matrix even if the input matrix was not invertible.

```
>>> A=sp.Matrix([[1,2],[3,4]])
>>> mat_inv(A)
Matrix([
[ -2,    1],
[3/2, -1/2]])
>>> B=sp.Matrix([[1,2,3],[4,5,6]])
>>> mat_inv(B)
Matrix([
[-5/3,  2/3],
[4/3,  -1/3]])
```

Challenge

Write a function `inv_true` which calculates the potential inverse as in part 2, and then verifies that the potential inverse is a true inverse for the input matrix.

Lab 14

Basis-finding

In this lab you will write a function that will compute a basis for a subspace of $\mathbb{R}^n$ spanned by a given set of vectors and any linear dependencies among the vectors. You will want to use Sympy Matrix objects and the `rref` method for this lab.

1. Write a function to compute a basis for a given subspace of $\mathbb{R}^n$.

   Write a function `vec_basis` that takes as input a list of vectors in $\mathbb{R}^n$ and returns a basis for the subspace spanned by the vectors. The input vectors should be given as lists. If the vectors are of different lengths, your function should throw an appropriate error. The output should be a list of lists, where each inner list represents a vector in the original list. Your function should compute the basis following the procedure to compute a basis of the column space of a matrix. Remember that the Sympy Matrix `rref` method returns a tuple, where the second value contains the indexes of the columns which have leading terms.

```
>>> vec_basis([[1,2],[2,1],[0,1]])
[[1, 2], [2, 1]]

>>> vec_basis([[1,2,3],[3,4,5],[6,7,8]])
[[1, 2, 3], [3, 4, 5]]
```

2. Write a function to find linear dependencies among a list of vectors in $\mathbb{R}^n$.

Write a function `lin_deps` that takes as input a list of list of vectors in $\mathbb{R}^n$ and returns a list of linear dependencies among the vectors. The input vectors should be given as lists. If the input vectors are of different lengths, your function should throw an appropriate error. The output should be a list of lists, where each inner list represents a linear dependence among the input vectors and should represent a basis for the null space of the matrix A whose column vectors are the input vectors.

You should have one output vector per free variable in the matrix equation $A\mathbf{x} = \mathbf{0}$. If B is the reduced echelon form of A and x_i is a free variable, we create the linear dependence corresponding to x_i as follows.

a) If m is the number of input vectors, create a list of length m where every entry is zero.

b) Change the ith entry to a 1.

c) For each basic variable x_j let k be the index of the leading term in the jth column of the reduced so that $B_{k,j} \neq 0$. Then set jth entry of the list to $-B_{k,i}$.

```
>>> M=sp.Matrix([[1,2,3,4],[5,10,6,11]])
>>> M.rref()
(Matrix([
 [1, 2, 0, 1],
 [0, 0, 1, 1]]), (0, 2))

>>> lin_deps([[1,5],[2,10],[3,6],[4,11]])
[[-2,1,0,0], [-1,0,-1,1]]


>>> M=sp.Matrix([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
>>> M.rref()
(Matrix([
 [1, 0, -1, -2, -3],
 [0, 1,  2,  3,  4],
 [0, 0,  0,  0,  0]]), (0, 1))

>>> lin_deps ([[1,6,11],[2,7,12],[3,8,13],[4,9,14],[5,10,15]])
[[1,-2,1,0,0], [2,-3,0,1,0], [3,-4,0,0,1]]
```

Lab 15

The Euclidean Algorithm

1. The Euclidean Algorithm, iteratively.

   Write a function `gcd_it(a,b)` that computes the GCD of a and b using a `while` loop. See Algorithm 17.20 in the Math 290 textbook. Your function should raise an exception if a and b are both zero.

   ```
   >>> gcd_it(12345,67890)
   15
   >>> gcd_it(-56,98)
   14
   ```

2. The Euclidean Algorithm, recursively.

   Write a function `gcd(a,b)` that computes the GCD of a and b recursively. See Algorithm 17.21 in the Math 290 textbook. Your function should raise an exception if a and b are both zero.

   ```
   >>> gcd(112233,445566)
   33
   ```

3. The Extended Euclidean Algorithm, recursively.

   In addition to computing the GCD d of two integers a and b, the extended Euclidean algorithm also computes integers x and y such that d can be written as the linear combination $d = ax + by$.

   How does this work recursively? Suppose that we start with a and b where $|a| \geq |b|$ (otherwise, swap them). Then we know that $d = \gcd(a, b) = \gcd(b, r)$, where $a = qb+r$ with $0 \leq r < |b|$. If we have some way of computing integers x' and y' such that

$d = bx' + ry'$ then we can work backwards as follows:

$$\begin{aligned} bx' + ry' &= d \qquad\qquad r = a - qb, \\ bx' + (a - qb)y' &= d \\ b(x' - qy') + ay' &= d. \end{aligned}$$

Then we can compute the original x and y we wanted as $x = y'$ and $y = x' - qy'$.

It remains to compute $d = ax + by$ in some simple base case. If $b = 0$ then we know that $d = |a|$ and therefore we can take $x = \pm 1$ (depending on the sign of a) and $y = 0$.

Write a recursive function `xgcd(a,b)` that performs the extended Euclidean Algorithm on the pair a, b. Your function should return a list of three integers `[d,x,y]` such that $d = ax + by$. See Section 18.B in the Math 290 textbook. Your function should raise an exception if a and b are both zero.

```
>>> xgcd(57,89)
[1, 25, -16]
>>> xgcd(112233,445566)
[33, -4633, 1167]
>>> xgcd(12,-53)
[1, -22, -5]
```

4. Integer 2×2 matrices with determinant 1.

Write a function `sl2mat(a,b)` that takes as input two relatively prime integers a and b (i.e. $\gcd(a, b) = 1$) and returns a matrix `[[a,b],[c,d]]` such that c and d are integers and the matrix has determinant 1 (recall that the determinant is $ad - bc$). Your function should raise an exception if a and b are not relatively prime.

```
>>> sl2mat(1,7)
[[1,7],[0,1]]
>>> sl2mat(17,-89)
[[17, -89], [-4, 21]]
```

Lab 16

Stochastic Matrices

In this lab you will write a function to verify that a matrix is a stochastic matrix, and then compute limiting distributions of the dynamical system described by the matrix.

You will want to use Sympy Matrix objects for this lab. You can calculate the eigenspaces of a Sympy Matrix object using the `eigenvects()` command, see

docs.sympy.org/latest/tutorial/matrices.html.

This method will return a list where each entry is a tuple (λ, d, B) representing each eigenspace. Here λ is the eigenvalue, d is the dimension of the eigenspace, and B is a basis for the eigenspace.

Recall that an $n \times n$ stochastic matrix M represents a dynamical system with n states, where the entry $M_{i,j}$ represents the probability of at each time step of transitioning from state j to state i. Therefore a stochastic matrix must satisfy the following properties:

1. The matrix is square,
2. Every entry is non-negative,
3. The entries of each column add up to 1.

If every entry of a stochastic matrix is positive (rather than non-negative), the system has a unique limiting distribution independent of the starting state. This state is represented by an eigenvector for the matrix with eigenvalue 1.

If any entry of the matrix is 0, the limiting distribution may depend on the initial conditions, or the distribution may oscillate among several states rather than settling on a single limiting distribution. This happens because the eigenspace for $\lambda = 1$ may have dimension larger than one, and the matrix may have multiple eigenvalues λ with $|\lambda| = 1$.

1. Write a function to verify that a given matrix is stochastic.

Write a function `is_stochastic(M)` to verify that a given matrix M is stochastic.

```
>>> A = sp.Matrix([[.3,.9],[.7,.1]])
>>> is_stochasitc(A)
True

>>> B = sp.Matrix([[1,.5,.4],[0,.5,.3],[0,0,.3]])
>>> is_stochasitc(B)
True

>>> C = sp.Matrix([[1,2,3],[4,5,6]])
>>> is_stochasitc(C)
False

>>> D = sp.Matrix([[.6,.3,.2],[.5,.5,.3],[-.1,.2,.5]])
>>> is_stochasitc(B)
False
```

2. If every entry of a stochastic matrix is positive (rather than non-negative), the system has a unique limiting distribution independent of the starting state. This state is represented by an eigenvector for the matrix with eigenvalue 1. Write a function to find this limiting distribution.

Write a function `stochastic_limit(M)` that takes as input a stochastic matrix M and attempts to calculate the limiting distribution of the implied system. If the input matrix is not stochastic, your function should throw an appropriate error. Examples are on the next page.

a) If every entry of the matrix is positive, your function should return an eigenvector with positive entries which add up to one. This eigenvector will be unique.

b) If any entry of the matrix is 0, the limiting distribution may depend on the initial conditions, and the distribution may oscillate among several states without settling on a limiting distribution. This happens because the eigenspace for $\lambda = 1$ may have dimension larger than one, and the matrix may have multiple eigenvalues λ with $|\lambda| = 1$. In this case you should return a basis for the subspace of $\mathbb{R}^n$ spanned by the eigenvectors with eigenvalue $|\lambda| = 1$.

```
>>> A = sp.Matrix([[.3,.9],[.7,.1]])
>>> stochastic_limit(A)
Matrix([
[0.5625],
[0.4375]])

>>> B = sp.Matrix([[1,.4,0],[0,.4,0],[0,.2,1]])
>>> stochastic_limit(B)
[Matrix([
 [1.0],
 [  0],
 [  0]]), Matrix([
 [  0],
 [  0],
 [1.0]])]

>>> C = sp.Matrix([[0,0,1],[1,0,0],[0,1,0]])
>>> stochastic_limit(C)
[Matrix([
 [1],
 [1],
 [1]]), Matrix([
 [   -1/(1/2 + sqrt(3)*I/2)],
 [(1/2 + sqrt(3)*I/2)**(-2)],
 [                        1]]), Matrix([
 [   -1/(1/2 - sqrt(3)*I/2)],
 [(1/2 - sqrt(3)*I/2)**(-2)],
 [                        1]])]
```

Lab 17

Recursion Reloaded

Part 1: In Section 15 of the Math 290 textbook math.byu.edu/∼doud/Transition it is shown that any finite set of cities with a system of one-way roads connecting them has a valid path (re-read that seciton if you don't remember what these terms mean). We will write a program that finds a valid path through any such system of roads.

First, we need to generate systems of roads. A system of one-way roads through n cities can be represented as a list of n city names together with an $n \times n$ matrix (a_{ij}), where

$$a_{ij} = \begin{cases} 0 & \text{if } i = j, \\ 1 & \text{if there is a road from } i \text{ to } j, \\ -1 & \text{if there is a road from } j \text{ to } i. \end{cases}$$

For example, the system of roads through five cities pictured on p. 108 of the Math 290 textbook would be represented by list `L=['A','B','C','D','E']` together with the matrix

$$M = \begin{bmatrix} 0 & -1 & 1 & -1 & -1 \\ 1 & 0 & -1 & 1 & 1 \\ -1 & 1 & 0 & -1 & 1 \\ 1 & -1 & 1 & 0 & -1 \\ 1 & -1 & -1 & 1 & 0 \end{bmatrix}.$$

Notice that M is antisymmetric, meaning that $M^T = -M$.

Write a function `random_system(n)` that takes as input a positive integer n and returns a random system of one-way roads through n cities as an $n \times n$ matrix as described above. If you need help generating random integers, read about the function `randint` online.

To find a valid path through n cities: let X be the first city in the list L. For each other city Y say that Y is a red city if there is a road from X to Y (i.e. $a_{XY} = 1$) and say that Y is a blue city if there is a road from Y to X (i.e. $a_{XY} = -1$). Then form two smaller systems of one way roads, one comprising the red cities, and the other comprising the blue cities (where the order of the labels is preserved). For example, for the matrix M above, we have $X = A$ and

$$M_{\text{red}} = \begin{bmatrix}0\end{bmatrix}, \quad L_{\text{red}} = [C] \quad \text{and} \quad M_{\text{blue}} = \begin{bmatrix}0 & 1 & 1\\ -1 & 0 & -1\\ -1 & 1 & 0\end{bmatrix}, \quad L_{\text{blue}} = [B, D, E].$$

Now we (recursively) find a valid path through the red cities and a valid path through the blue cities and combine them to make the path

$$[\text{path through blue cities}] \to X \to [\text{path through red cities}].$$

Write a function `valid_path(M,L)` that takes as input an $n \times n$ matrix M and a list L of length n, and returns a valid path through the cities as a list of length n. For example, the function should return $[B, E, D, A, C]$ as the path for the example above.

As an extra (ungraded) challenge, write a function that computes all valid paths for the given input.

Part 2: Pull up your Fibonacci function `fib(n)` from Lab 12 and see how long it takes to compute the 40-th Fibonacci number. To do this, first run the following code:

```
>>> from timeit import default_timer as timer
```

Then you can measure how long it takes (in seconds) to compute `fib(40)` by running:

```
>>> start = timer()
>>> fib(40)
>>> end = timer()
>>> print(end-start)
```

The thing you should learn from this is: recursion makes for beautiful code but often expensive results. The reason it takes so long to compute `fib(40)` is that all of the earlier Fibonacci numbers are computed *several times* in order to compute the 40-th. To test this out, add the following print command in the base case of your `fib` function:

```
>>> if n==1 or n==2:
>>>     print('I'm calling the base case')
>>>     return 1
```

Unsurprisingly, this prints 'I'm calling the base case' every time the base case is called. Now run `fib(10)` and watch the flood of output. In theory, computing the 10-th Fibonacci number "by hand" should only call the base case once or twice.

In order to speed up our Fibonacci function, we'll use a programming technique called "memoization." (The name is awkward, but the technique is great.) The idea is that to compute `fib(41)` we shouldn't have to do much work, because we already computed `fib(40)` and, while computing `fib(40)`, we computed `fib(39)`. So it would be nice if Python would just remember that for us. Luckily, there's a built-in Python tool just for that:

```
>>> import functools
>>> @functools.lru_cache(maxsize=256)
>>> def fib(n):
>>>     blah blah blah
```

The line starting with `@` needs to come immediately before your function definition. Replace `blah blah blah` by the code you already wrote for the `fib` function. Now see how fast your function computes `fib(40)`. And just for fun, see how fast it computes `fib(200)`.

Part 3: There is a legend about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it. At the beginning of time, the leftmost post was surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks in accordance with the immutable rules of Brahma since that time, in an effort to move the disks to the rightmost post. The rules are:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty post.
3. No larger disk may be placed on top of a smaller disk.

When the last move of the puzzle is completed, the world will end. Your task is to determine how soon the world will end, given that the priests can move one disk per second.

Start by playing the Towers of Hanoi game here. Play with 3 disks first until you have a solution that takes the minimal number of moves (in this case 7 moves). Then move on to 4 disks until you have a solution that takes 15 moves. Keep playing with more disks until you see the pattern. Don't read on until you're done with this step.

Let's call the leftmost and rightmost posts the "source" and "target," respectively. We'll call the middle post "aux." How do we solve the puzzle when there are two disks? First, move the top disk from source to aux, then move the second disk from source to target, then move the top disk from aux to target. When there are three disks? First, move the top two disks from source to aux (using the solution for two disks), then move the third disk from source to target, then move the top two disks from aux to target (again using the solution for two disks).

Write a function `hanoi(n, source, aux, target)` that takes in an integer n and the names of the source, aux, and target posts as strings. Do not implement memoization for this function. Your function should move the top n disks from source to target, using aux for help. At each move, print something like "move from post A to post C."

```
>>> hanoi(3,'A','B','C')
'move from post A to post C'
'move from post A to post B'
'move from post C to post B'
'move from post A to post C'
```

```
‘move from post B to post A’
‘move from post B to post C’
‘move from post A to post C’
```

For how many years will the priests be moving disks? (Remember that priests can move one disk per second.)

Lab 18

Modular Arithmetic

1. Roughly speaking, an *elliptic curve* is an equation of the form

$$E(a,b) : y^2 = x^3 + ax + b$$

for fixed integers a, b. For example, when we set $a = b = 1$ we get the elliptic curve $E(1,1) : y^2 = x^3 + x + 1$. For any prime p, we can count the number of points on $E(a,b)$ over $\mathbb{Z}/p\mathbb{Z}$ by finding all of the pairs $(x,y) \in (\mathbb{Z}/p\mathbb{Z}) \times (\mathbb{Z}/p\mathbb{Z})$ that satisfy

$$y^2 \equiv x^3 + ax + b \pmod{p}.$$

Write a function `num_points(a,b,p)` that takes as input two integers a and b and a prime p, and returns the number of points on $E(a,b)$ over $\mathbb{Z}/p\mathbb{Z}$.

```
>>> num_points(1,1,5)
8
>>> num_points(0,1,97)
83
```

2. The textbook for Math 687R (a graduate number theory course at BYU) in Fall 2018 was *Auxiliary Polynomials in Number Theory* by David Masser. One of the exercises in that book states: do there exist $a, b, c, d, e, f \in \mathbb{Z}/11\mathbb{Z}$ such that $a \neq 0$ and

$$y^2 = ax^5 + bx^4 + cx^3 + dx^2 + ex + f \tag{18.1}$$

has no solutions $(x,y) \in (\mathbb{Z}/11\mathbb{Z}) \times (\mathbb{Z}/11\mathbb{Z})$? The author states that he does not know the answer to this question.

Find $a, b, c, d, e, f \in \mathbb{Z}/11\mathbb{Z}$, all nonzero, such the equation (18.1) has no solutions $(x,y) \in (\mathbb{Z}/11\mathbb{Z}) \times (\mathbb{Z}/11\mathbb{Z})$.

3. Powers of 2 modulo n.

Write a program that finds every $n \in \{3, 4, \ldots, 500\}$ that satisfies the congruence $2^{n-1} \equiv 1 \pmod{n}$. What do you notice about these n? Is your observation true about *every* n in your list?

4. Powers of n modulo p.

For every prime p satisfying $20 < p < 100$, do the following. For every $n \in \{1, \ldots, p-1\}$ find the smallest positive k such that $n^k \equiv 1 \pmod{p}$, if such a k exists. What do you notice about these values of k? Formulate a conjecture. (We'll prove your conjecture in Math 371.)

If you're annoyed at the way Python outputs a long list returned by a function, try printing the list instead. For example, suppose the function `allk(p)` returns a list of length $p - 2$ consisting of all the k values asked for in this problem for the prime p. Then the following code should give you a relatively compact way to view all of your results at once:

```
for p in range(20,100):
     if isprime(p):
        print(p,‘:’, allk(p))
```

Lab 19

Lucas Sequences

If P and Q are integers, the Lucas sequence $(U_n(P,Q))$ is defined as follows: $U_0(P,Q) = 0$, $U_1(P,Q) = 1$, and for $n > 1$,

$$U_n(P,Q) = P \cdot U_{n-1}(P,Q) - Q \cdot U_{n-2}(P,Q).$$

Notice that when $P = 1$ and $Q = -1$, we have that $U_n(1,-1) = F_n$ is the nth Fibonacci number. The Lucas numbers can be computed efficiently using matrix multiplication. The recursive definition of $(U_n(P,Q))$ means that

$$\begin{bmatrix} U_{n+1}(P,Q) \\ U_n(P,Q) \end{bmatrix} = \begin{bmatrix} P & -Q \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U_n(P,Q) \\ U_{n-1}(P,Q) \end{bmatrix}.$$

Starting with $U_0(P,Q) = 0$, $U_1(P,Q) = 1$, and iterating this formula, we find

$$\begin{bmatrix} U_{n+1}(P,Q) \\ U_n(P,Q) \end{bmatrix} = \begin{bmatrix} P & -Q \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

We can also give an exact formula for $U_n(P,Q)$ using the eigenvalues of the matrix, as long as $Q \neq P^2/4$. In this case, the matrix will have two distinct eigenvalues λ_1 and λ_2. Suppose $\mathbf{v}_1$ and $\mathbf{v}_2$ are eigenvectors for these two eigenvalues. Then $B = \{\mathbf{v}_1, \mathbf{v}_2\}$ is a basis for $\mathbb{R}^2$. If $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] = [\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]]_B$ are the coordinates of $\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$ with respect to the basis B, then $\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] = a\mathbf{v_1} + b\mathbf{v_2}$, and so

$$\begin{bmatrix} U_{n+1}(P,Q) \\ U_n(P,Q) \end{bmatrix} = \begin{bmatrix} P & -Q \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = a\lambda_1^n \mathbf{v_1} + b\lambda_2^n \mathbf{v_2}.$$

1. Write a function to calculate a value of the Lucas sequence using matrix powers.

Write a function `LucasU_pow(P,Q,n)` that calculate the value $U_n(P,Q)$ using the matrix powers method described above.

```
>>> LucasU_pow(1,-1,8)
21
>>> LucasU_pow(1,-1,167)
35600075545958458963222876581316753
>>> LucasU_pow(3,4,8)
-93
>>> LucasU_pow(2,-5,50)
1576294185744813172441960 82
```

2. Write a function to calculate a value of the Lucas sequence using eigenvalues.

Write a function `LucasU_eig(P,Q,n)` that calculates the value $U_n(P,Q)$ using the eigenvalue method described above. Explain how you calculated the coordinates $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] = [\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]]_B$. If you use Sympy you may need to convert to floating point numbers to get a decimal expansion rather than a symbolic expression.

```
>>> LucasU_eig(1,-1,8)
21.0
>>> LucasU_eig(1,-1,167)
3.5600075545958458e+34
>>> LucasU_eig(3,4,8)
-93.0
>>> LucasU_eig(2,-5,50)
1.5762941857448133e+26
```

Challenge

You can test if a number is probably prime using a Lucas sequence. For simplicity, we'll stick to the case $P = 1$ and $Q = -1$. If m is a positive integer not divisible by 5, let $d = \begin{cases} 1 & \text{if } m \equiv \pm 1 \pmod 5 \\ -1 & \text{if } m \equiv 2,3 \pmod 5 \end{cases}$. If m is prime then $U_{m-d}(P,Q) \equiv 0 \pmod m$. If m is not prime, this will usually fail, so if a number passes this test we say it is probably prime. Write a function `Lucas_test` to implement this test to see if a number is probably prime. In order to make this test effective for large numbers, you will need to be able to compute large powers of the matrix, $\left[\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, reducing $\pmod m$ at each step. How can you do this effectively?

Lab 20

Cracking the Vigenère Cipher

To refresh your memory of the Vigenère Cipher, go back to Lab 8 and read about it. You will need your `key_length` function from that lab today. There is an encrypted ciphertext at `pastebin.com/q3k3CKQU`; your ultimate goal today is to determine the plaintext message.

The main fact we will use to break the cipher is that in most English texts the frequencies of letters are not equal (see Figure 20.1). For example, the letters `RSTLNE`, which you may recognize from watching Wheel of Fortune, are among the most frequent.

a	b	c	d	e	f	g	h	i	j
.082	.015	.028	.043	.127	.022	.020	.061	.070	.002
k	l	m	n	o	p	q	r	s	t
.008	.040	.024	.067	.075	.019	.001	.060	.063	.091
u	v	w	x	y	z				
.028	.010	.023	.001	.020	.001				

Figure 20.1: Frequencies of letters in English

The following is an outline for how to decrypt a ciphertext encoded with the Vigenère Cipher with an unknown key. (Read until the end of the lab before you start working on any individual part.)

(a) Compute the key length. (We already did this.)

(b) For each j from 0 to $K-1$: build a string $S[j]$ comprising the letters of the ciphertext at indices congruent to j modulo K. For example, if `ciphertext='ABCDEFGHIJK'` and the key length is $K = 3$, then

```
S[0] = 'ADGJ'
S[1] = 'BEHK'
S[2] = 'CFI'
```

For each j, every letter of $S[j]$ has been encrypted with the same letter of the key.

(c) For each j from 0 to $K-1$, and for each m from 0 to 25, let
`dS[j, m] = vigenere_decrypt('l',S[j])`,
where `'l'` is the m-th letter of the alphabet.

(d) Measure the frequency of each letter in each of the strings `dS[j,m]`, and record these frequencies in a vector $f[j,m]$ of length 26.

(e) For each j from 0 to $K-1$: find the value of m for which the dot product $f[j,m] \cdot F$ is maximized, where F is the vector of letter frequencies
`F=[.082, .015, .028, ..., .020, .001]` from the table above. Then the j-th letter of the key is the m-th letter of the alphabet.

(f) Decrypt the ciphertext using the key you found in part (e).

Your assignment has four parts:

1. Split the ciphertext.

Write a function `str_split(s,j,k)` that takes in a string s and two integers j, k, where $k \geq 1$, and returns a string consisting of each letter of s whose index is congruent to j modulo k.

```
>>> str_split('ABCDEFGHIJK',1,3)
'BEHK'
```

2. Measure letter frequencies.

Write a function `letter_freq(s)` that takes as input a string s and outputs a vector of length 26 whose i-th element is the frequency of the i-th letter of the alphabet in the string s. The built-in function `count` may be helpful here.

```
>>> s = 'AAAAAAABZZ'
>>> s.count('A')
7
>>> letter_freq('AAAAAAABZZ')
[0.7,0.1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.2]
```

3. Maximize the dot product.

Write a function `maximize_dot(str_list)` that takes as input a list of 26 strings `str_list` and outputs the integer m for which the dot product `letter_freq(str_list[m])`·F is maximized.

```
>>> cipher = 'KYVHLZTBSIFNEWFOALDGJFMVIKYVCRQPUFX'
>>> A = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> S = [vigenere_decrypt(letter,cipher) for letter in A]
>>> maximize_dot(S)
17
```

4. Decrypt the ciphertext.

Write a function `vigenere_crack(str)` that takes in a string and outputs a list of two strings: the most likely key and the most likely plaintext.

As a test input, use the ciphertext found at pastebin.com/q3k3CKQU. It will be very clear if you have computed the correct key and plaintext.

You can get other test input strings at math.byu.edu/∼doud/Vigenere by clicking on "Get Random Ciphertext."

Lab 21

Determinants

In this lab you will write two functions to compute the determinant of a matrix. The first method is recursive using the co-factor expansion of the matrix. The second is much better and uses row reduction. For our purposes we will use the scipy.linalg LU factorization method for the determinant. You can read more about this method here:

docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.linalg.lu.html

The operations allowed in an LU factorization do not change the determinant of a matrix, so if $LU = A$ is a factorization of A, then $\det A = \det U$. Since U is upper triangular, its determinant is the product of entries down the main diagonal.

You can use the LU factorization method as follows.

```
>>> import scipy.linalg as la
>>> la.lu([[1,2],[1,1]])
(array([[1., 0.],
        [0., 1.]]), array([[1., 0.],
        [1., 1.]]), array([[ 1.,  2.],
        [ 0., -1.]]))
```

The output is a triple (P, L, U) where P is a permutation matrix, L is a lower-triangular matrix, and U is an upper triangular matrix.

1. Write a function to compute the determinant of a matrix recursively using the co-factor expansion.

Write a function `det_rec(M)` which takes as input a matrix M and compute the determinant of a matrix recursively using the co-factor expansion. Your base case should be if M is 1, in which case the determinant is just the entry.

```
>>> det_rec([[1,2],[1,1]])
-1.0
>>> det_rec([[1,2,3],[1,1,1],[1,0,1]])
-2.0
>>> det_rec([[1,2,3,4,5],[1,1,1,0,0],[1,0,1,0,1],[0,-1,0,-1,-2],
[1,1,1,1,1]])
-4.0
```

2. Write a function to compute the determinant of a matrix using LU factorization.

Write a function `det_LU(M)` which takes as input a matrix M and compute the determinant of a matrix by first computing the LU factorization of M as described above, and then returning the product of values on the main diagonal of U. For this problem, you may ignore the permutation matrix. This may mean your determinant is off by a sign since swapping rows changes the sign of the determinant.

```
>>> det_LU([[1,2],[1,1]])
-1.0
>>> det_LU([[1,2,3],[1,1,1],[1,0,1]])
2.0
>>> det_LU([[1,2,3,4,5],[1,1,1,0,0],[1,0,1,0,1],[0,-1,0,-1,-2],
[1,1,1,1,1]])
4.0
```

21.1 Challenge

Fix your `det_LU(M)` function so that it accounts for the change of sign indicated by the permutation matrix.

```
>>> det_LU([[1,2],[1,1]])
-1.0
>>> det_LU([[1,2,3],[1,1,1],[1,0,1]])
-2.0
>>> det_LU([[1,2,3,4,5],[1,1,1,0,0],[1,0,1,0,1],[0,-1,0,-1,-2],
[1,1,1,1,1]])
-4.0
```

Lab 22

The Gram-Schmidt Process

In this lab you will use the python `numpy` module to perform the Gram-Schmidt process on a collection of vectors to find an orthonormal basis for their span. Recall that to import the numpy module you use the command

```
>>> import numpy as np
```

Use the numpy array constructor for vectors:

```
>>> A=np.array([1,2,3])
```

This will allow you to use the built-in method to compute dot products. To compute the dot product of two vectors `A` and `B`, use the syntax `A.dot(B)` or `B.dot(A)`.

1. Write a function to project one vector onto the subspace spanned by another.

Write a function `projection(A,B)` that takes as input two vectors `A` and `B` of the same length n and computes the projection of `B` onto the subspace of $\mathbb{R}^n$ spanned by `A`. Recall the formula for this projection is

$$\mathrm{Proj}_{\boldsymbol{A}}(\boldsymbol{B}) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{\boldsymbol{A} \cdot \boldsymbol{A}} \boldsymbol{A},$$

where $\cdot$ is the usual dot product.

```
>>> projection([1,0,-1],[1,2,3])
array([-1,0,1])

>>> projection([1,2,0,-1],[2,3,4,5])
array([ 0.5,  1. ,  0. , -0.5])
```

2. Write a function to perform the Gram Schmidt process on list of vectors to obtain an orthogonal basis for a given span.

Write a function `GramSchmidt(X)` that takes as input a list `X` of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \dots \boldsymbol{v}_p\}$ and returns a list of vectors $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \dots \boldsymbol{u}_q\}$ which forms an orthogonal basis for the space spanned by the original vectors. You may assume the original vectors are the same length, however you should not assume the original vectors are linearly independent. As you go through the process you should throw out any 0 vectors that come up.

```
>>> GramSchmidt([[1,0,-1],[1,2,3]])
[array([1,0,-1]), array([2,2,2])]

>>> GramSchmidt([[1,0,-1,0],[1,2,3,4],[0,1,2,2]])
[array([1,0,-1,0]), array([2,2,2,4])]
```

3. Write a function to perform the extended Gram Schmidt process on list of vectors to obtain an orthognormal basis for a given span.

Write a function `Orthonormal(X)` that takes as input a list `X` of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \dots \boldsymbol{v}_p\}$ and returns a list of vectors $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \dots \boldsymbol{u}_q\}$ which forms an orthonormal basis for the space spanned by the original vectors. Use the `GramSchmidt` function from the previous part to construct an orthogonal basis, and then normalize each element to have norm 1.

```
>>> GramSchmidt([[1,0,-1],[1,2,3]])
[array([ 0.70710678,  0.        , -0.70710678]),
array([0.57735027, 0.57735027, 0.57735027])]

>>> GramSchmidt([[1,0,-1,0],[1,2,3,4],[0,1,2,2]])
[array([ 0.70710678,  0.        , -0.70710678,0]),
array([0.37796447, 0.37796447, 0.37796447, 0.75592895])]
```

Lab 23

Image Compression with SVD

In this lab you will perform image compression using singular value decomposition. You will need to import the following:

```
>>> import numpy as NP
>>> import numpy.linalg as LA
>>> import imageio as IO
>>> import matplotlib.pyplot as PLT
>>> import scipy.misc
```

For this lab we will use a grayscale image of a racoon from the scipy.misc module, which can be accessed by

```
>>> F = scipy.misc.face(gray=True)
```

To display the image, use the command

```
>>> PLT.imshow(F, cmap=‘gray’)
```

The command `F.shape` shows that the image is stored as a numpy array of dimensions $m \times n$. These dimensions represent the coordinates of a pixel in the image with $(0,0)$ in the top left corner. The first dimension is the vertical dimension and the second is the horizontal dimension. We can take a part of the image by using a command like

```
>>> F1 = F[200:400, 500:700]
>>> PLT.imshow(F1, cmap=‘gray’)
```

Each entry is an integer representing how dark a pixel is (0 = black, 255 = white). Since `F` is a two-dimensional array we can treat it as a matrix and perform a singular value decomposition:

```
>>> U,S,VT = LA.svd(F)
```

Recall that the singular value decomposition writes F in the form

$$F = U\Sigma V^T. \tag{23.1}$$

The matrix `U`$= U$ is an $m \times m$ matrix with orthonormal columns, and `VT`$= V^T$ is an $n \times n$ matrix with orthonormal columns. Σ is a diagonal matrix whose nonzero entries are the singular values of F. Python represents this as `S`, a list of the singular values of `F`. The `NP.diag` function will turn a list into a diagonal matrix. If r is the length of S, the command

```
>>> Ftest = U[:,:r].dot(NP.diag(S[:r])).dot(VT[:r])
```

will create a matrix `Ftest` that is the same as `F` (up to Python precision issues). If you display this as an image it will look indistinguishable from the original picture. Python may complain about a potential loss of precision, but you may ignore this. Now try using the command

```
>>> Ftest = U[:,:s].dot(NP.diag(S[:s])).dot(VT[:s])
```

for some $s < r$. This is the *rank s approximation of F*.

1. Display the rank s approximation of the image for several values of s. What happens when $s = 500$? What about $s = 100$? What about $s = 1$?

2. How small can you make s and still have the image recognizable? Don't worry about a little graininess.

3. We can store a grayscale image as a list of integers of length $u + r + v$, where u (or v) is the number of entries in U (or V). Stored that way, the amount of memory that the image takes up is proportional to that number $u + r + v$. When we reduce r to the smaller number s, this reduces u and v as well to numbers $u(s)$ and $v(s)$. How does your choice of s in part (b) affect the amount of memory required to store the image? Compute the number $u(s) + s + v(s)$ for the SVD of `Ftest` for each choice of s as a percentage of $u + r + v$.

Lab 24

The Lean Theorem Prover

[From the Lean Theorem Prover community website] "A proof assistant is a piece of software that provides a language for defining objects, specifying properties of these objects, and proving that these specifications hold. The system checks that these proofs are correct down to their logical foundation.

These tools are often used to verify the correctness of programs. But they can also be used for abstract mathematics.... In a formalization, all definitions are precisely specified and all proofs are virtually guaranteed to be correct."

In this lab, we will get a sense of how a theorem prover works by proving basic properties of the natural numbers in the interactive theorem prover Lean. Visit

https://wwwf.imperial.ac.uk/~buzzard/xena/natural_number_game/

[From that page] "In this game, you get own version of the natural numbers, called `mynat`, in an interactive theorem prover called Lean. Your version of the natural numbers satisfies something called the principle of mathematical induction, and a couple of other things too (Peano's axioms). Unfortunately, nobody has proved any theorems about these natural numbers yet! For example, addition will be defined for you, but nobody has proved that $x + y = y + x$ yet. This is your job. You're going to prove mathematical theorems using the Lean theorem prover. In other words, you're going to solve levels in a computer game."

Start by clicking on the blue dot labeled "Tutorial World." Carefully read the instructions for each level and complete the code at the bottom of the webpage. When the top right frame of the page reads "Proof complete!" you can advance to the next level. Continue playing until you have finished "Tutorial World," then go back to the main menu and start "Addition World."

When you have completed the final boss of "Addition World," send Elizabeth an email saying that you have completed this lab. There is nothing to turn in.