Unit Three 1D Lists

sieveOfEratosthenes(n) (5 points)
Write the program sieveOfEratosthenes that finds all primes less than or equal to n and stores them in a list. Go here to read about the sieve. Return is a list with all primes up to and including n.

areaOfPolygon(L) (10 points)
Write the function areaOfPolygon(L) that takes a list of (x,y) points in the format (x1,y1,x2,y2,…,xn,yn] that are guaranteed to be in either clockwise or counter-clockwise order around a polygon, and returns the area of that polygon, as described here. So as an example, almostEqual(areaOfPolygon([4,10, 9,7, 11,2, 2,2]), 45.5)==True

centroidOfPolygon(L) (8 points)
This goes along with the area of a polygon. A related concept is the centroid of a polygon. You can read about it here. You have to do the area first. Once you have that done, the centroid is a similar program that uses essentially the same technique. As an example, centroidOfPolygon([5,5,5,-6,-8,-6,-8,5])==(-1.5, -0.5)

evalPolynomial(coeffs, x) (7 points)
Background: we can represent a polynomial as a list of its coefficients. For example, [2, 3, 0, 4] could represent the polynomial $2x^3 + 3x^2 + 4$. With this in mind, write the function evalPolynomial(coeffs, x) that takes a list of coefficients and a value x and returns the value of that polynomial evaluated at that x value. Do not alter the data in the list coeffs. For example, evalPolynomial([2,3,0,4], 4) ==180
($2*4^3 + 3*4^2 + 4 = 2*64 + 3*16 + 4 = 128 + 48 + 4 = 180$).

multiplyPolynomials(p1, p2) (15 points)
Write the function multiplyPolynomials(p1, p2) which takes two polynomials as defined in the previous problem and returns a third polynomial which is the product of the two. The input data (p1 & p2) cannot be altered.
For example, multiplyPolynomials([2,0,3], [4,5]) ==[8, 10, 12, 15] represents the problem $(2x^2 + 3)(4x + 5) = 8x^3 + 10x^2 + 12x + 15$.

repeatingPattern(L) (15 points)
Write the function repeatingPattern(L) that takes a list L and returns a tuple of (pattern, repeat), where pattern is a list and repeat is an integer >= 2, where the list L is composed of "repeat" consecutive instances of pattern. Return the string 'No Repeating Pattern' if no such pattern exists. For example, repeatingPattern([1,2,3,1,2,3]) returns ([1,2,3], 2).

isNearlySorted(L) (7 points)
Write the function isNearlySorted(L) that takes a possibly-empty list L and returns True if the list is "nearly sorted", and False otherwise, where a "nearly sorted" list is one which is not sorted and requires exactly one swap of two elements to become sorted. L should not be altered in the process.

dotProduct(a,b) (5 points)
Background: the "dot product" of the lists [1,2,3] and [4,5,6] is (1*4)+(2*5)+(3*6), or 4+10+18, or 32. In general, the dot product of two lists is the sum of the products of the corresponding terms. With this in mind, write the function dotProduct(a,b). This function takes two non-empty lists and non-destructively returns the dotProduct of those lists. If the lists are not equal length, ignore the extra elements in the longer list.

duplicates(a) (5 points)
Write the function duplicates(a) that takes a list (which your function must not modify) of unsorted values and returns a sorted list of all the duplicates in that first list. For example, duplicates([1, 3, 5, 7, 9, 5, 3, 5, 3]) would return [3, 5] If there are no duplicates, return an empty list.

smallestDifference(a) (5 points)
Write the function smallestDifference(a) that takes a list of integers and returns the smallest absolute difference between any two integers in the list. If the list is empty, return -1.

lookAndSay(a) (10 points)
First, read about look-and-say numbers here. Then, write the function lookAndSay(a) that takes a list of numbers and returns a list of tuples that results from "reading off" the initial list using the look-and-say method. For example:

lookAndSay([]) == []
lookAndSay([1,1,1]) == [(3,1)]
lookAndSay([-1,2,7]) == [(1,-1),(1,2),(1,7)]
lookAndSay([3,3,8,-10,-10,-10]) == [(2,3),(1,8),(3,-10)]

inverseLookAndSay(a) (5 points)
Write the function inverseLookAndSay(a) that does the inverse of the previous problem, so that, in general:
inverseLookAndSay(lookAndSay(a)) == a
Or, in particular: inverseLookAndSay([(2,3),(1,8),(3,-10)]) == [3,3,8,-10,-10,-10]

makeLookAndSay(L,g) (7 points)
Write the function makeLookAndSay (L,g) that takes in a non-empty list of integers along with a number of generations and returns a list of 'g' generations of lookAndSay. For example: makeLookAndSay ([3,3,7,7,7],4)== [2, 1, 2, 2, 1, 1, 1, 3, 3, 1, 1, 7], the process would look like this:

[3,3,7,7,7]	Generation 0
[2,3,3,7]	Generation 1
[1, 2, 2, 3, 1, 7]	Generation 2
[1, 1, 2, 2, 1, 3, 1, 1, 1, 7]	Generation 3
[2, 1, 2, 2, 1, 1, 1, 3, 3, 1, 1, 7]	Generation 4

areClockwise(a) (15 points)
Write the function areClockwise(a) that takes a list of (x,y) values and returns True if the points are in clockwise order and False otherwise. Return False if the (x,y) pairs do not form a convex polygon.

rotateList(a,n) (7 points)
Write the function rotateList(a, n) which takes a list (a)and an integer (n), and destructively modifies the list so that each element is shifted to the right by n indices (including wraparound). The function should then return the modified list. For example:

rotateList([1,2,3,4], 1) -> [4,1,2,3]
rotateList([4,3,2,6,5], 2) -> [6, 5, 4, 3, 2]
rotateList([1,2,3], 0) -> [1,2,3]
rotateList([1, 2, 3], -1) -> [2, 3, 1]

Do this without creating another list of length len(a).

moveToBack(a,b) (7 points)
Write the function moveToBack(a,b) which takes two lists a and b, and destructively modifies a so that each element of a that appears in b moves to the end of a in the order that they appear in b. The rest of the elements in a should still be present in a, in the same order they were originally. The function should return a. Examples:

moveToBack([2, 3, 3, 4, 1, 5], [3]) == [2, 4, 1, 5, 3, 3]
moveToBack([2, 3, 3, 4, 1, 5], [2, 3]) == [4, 1, 5, 2, 3, 3]
moveToBack([2, 3, 3, 4, 1, 5], [3, 2]) == [4, 1, 5, 3, 3, 2]

Do this without creating another list of length len(a).

linearRegression(L) (15 points)
Write the function linearRegression(L) that takes a list of (x,y) points (in this format: [x1,y1,x2,y2,…xn,yn])and finds the line of best fit through those points. Specifically, your function should return a tuple of floats (a,b,r) such that y = ax+b is the line of best fit through the given points and r is the correlation coefficient. Those of you who had me for Honors Advanced Precalculus might remember these formulas:

$$slope = \frac{\sum x \sum y - n\sum xy}{\left(\sum x\right)^2 - n\sum x^2} \qquad int = \frac{\sum x \sum xy - \sum x^2 \sum y}{\left(\sum x\right)^2 - n\sum x^2} \qquad r^2 = \frac{SS_{dev} - SS_{res}}{SS_{dev}}$$

If you don't remember (how dare you!), or if you didn't take precalc with me, we'll try to catch you up on the whole idea.
For example, linearRegression([3,2,4,7,6,5]) should return a tuple of 3 values approximately equal to (0.7143,1.5714,0.4335), indicating that the line of best fit is y = 0.7143x + 1.5714, with a correlation coefficient of 0.4335 (you should recall this means the data is not very well correlated). Note that the result is approximately equal to these values. Recall that the correlation coefficient can be positive or negative depending on the slope of the regression line. You may ignore the case of a vertical line.

nthLuckyPrime (n) (25 points)
Write the function nthLuckyPrime that takes a non-negative int n and returns the nth number that is both lucky and prime, where a lucky number is as defined here . Here are the first several lucky primes: 3, 7, 13, 31, 37, 43, 67, 73, 79, 127, 151, 163, 193, 211...
Hint: to do this, you will need to use a list to implement a sieve, similar to how we created the Sieve of Eratosthenes. However, this sieve is very tricky; that's why it's worth 25 points.

crossProduct(a,b) (5 points)

Write the function crossProduct that takes in two 3 element lists and performs a vector cross product on them. Look here for a description of how to calculate the cross product. Return is a 3 element tuple representing a 3 dimensional vector.

functionOfBestFit(L) (20 points)

Recall that the grapher (you?) calculates functions of best fit for linear, power, exponential and logarithmic functions by performing linear regression on the (sometimes modified) data set found in its (your) lists. Data modification is accomplished by taking logarithms of either the x or y coordinates of each data point—or both. In all cases, the correlation coefficient (r) is calculated using the linear data created by modification. The function possibilities and return values can be summed up like this:

Function	Data	Model	Return
Linear	(x,y)	f(x)=a*x+b	(a,b,'L')
Power	(ln(x),ln(y))	f(x)=a*x**b	(a,b,'P')
Exponential	(x,ln(y))	f(x)=a*b**x	(a,b,'E')
Logarithmic	(ln(x),y))	f(x)=a+b*ln(x)	(a,b,'N')

Note that everything is natural logarithm. In python, math.log(x)=ln(x). Other bases are possible, but we'll stick to that which is "natural."

With all of this as background, write the function functionOfBestFit(L) which takes in a list of x-y data and returns a tuple of coefficients a&b for the function of best fit and the type of function as a one letter string (Capital). So:

functionOfBestFit([1,3,2,5,3,7,4,9]) returns(2,1,'L')

functionOfBestFit([1,6,2,12,3,24,4,48]) returns (3.000,2.000,'E')

functionOfBestFit([1,3,2,12,3,27,4,48,5,75]) returns (3.000,2.000,'P')

functionOfBestFit ([1,3,10,5,100,7,1000,9]) returns (2.999, 0.868,'N').

Also note that the return values will be approximate because everything is a float. You can assume that all data will be positive.

This program should be set up as a series of helper functions (none of which will work unless you have completed linearRegression).

So the grading will be:

exponentialRegression(L) (5 points) return (a,b,r)

powerRegression(L) (5 points) return (a,b,r)

logarithmicRegression(L) (5 points) return (a,b,r)

functionOfBestFit(L) (5 points) return(a,b,typeOfFunction)

binaryListToDecimal(a) (5 points)

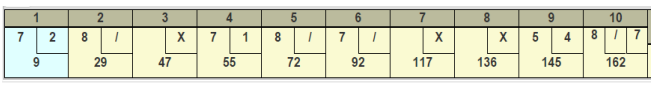
Write the function binaryListToDecimal(a) which takes a list of 1s and 0s, and returns the integer represented by reading the list from left to right as a single binary number. So binaryListToDecimal([0])==0, binaryListToDecimal([1,0])==2, binaryListToDecimal([1,0,1,1])==11 and binaryListToDecimal([1,1,1,1,0,0,0,1,0,0,1,0,0,0,0,0,0])==123456

bowlingScore(a) (20 points)

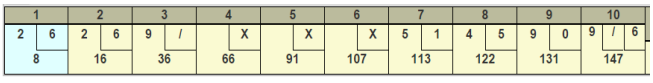
Background: In bowling, a bowler gets 2 throws per frame for 10 frames, where each frame begins with 10 pins freshly positioned, and the score is the sum of all the pins knocked down. However, if the bowler knocks down all 10 pins on the first throw of a frame, it is called a "strike", and they do not get a second throw in that frame. After a strike, the number of pins knocked down in the next two throws, regardless of the frame in which they are thrown, are added to the score of that frame. Also, if the bowler knocks down the rest of the 10 pins on the second throw in a frame, that is called a "spare", and the number of pins knocked down in the first throw of the next frame are added to the score of that frame. Finally, if there is a spare or strike in the final frame, then the bowler gets one extra throw in that frame (if there is a second strike in the final frame, a third throw is awarded). With all this in mind, write the function bowlingScore that takes a list of the number of pins knocked down on each throw and returns the score. Note that throws skipped due to strikes are not listed, so the best possible result is a list of 12 10's (all strikes), which would score 300 points.

Here are some lists with associated scores. You can go here to generate more scores.

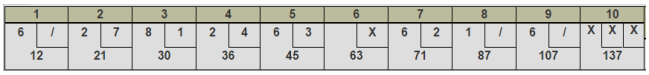
a=[7,2,8,2,10,7,1,8,2,7,3,10,10,5,4,8,2,7]



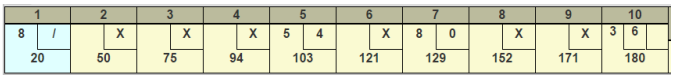
a=[2,6,2,6,9,1,10,10,10,5,1,4,5,9,0,9,1,6]



a=[6,4,2,7,8,1,2,4,6,3,10,6,2,1,9,6,4,10,10,10]



a=[8,2,10,10,10,5,4,10,8,0,10,10,3,6]



carrylessMultiply(x,y) (10 points)

Write the function carrylessMultiply that takes in two positive integers (x,y) and returns a positive integer that is the result of 'carryless multiply' of x and y. You can go back here and read up on how to accomplish the task. Hint: carrylessMultiply is in the 1D List section for a reason—even though both inputs and the output are not lists. You might want to read a little deeper into the paper to uncover the most straightforward method of calculation.

nthRootsOfComplexNumber(z,n) (10 points)

If you enter (-27)**(1/3) into the shell, you would probably expect Python to tell you the result is -3. You would be wrong. Python will tell you that it is equal to (1.5000000000000004+2.598076211353316j). Go ahead and try it. Recall that there are n "nth" roots of any complex number (You should remember DeMoivre's theorem from PreCalc). The result given by Python is one of three cube roots of -27. You have to find the other two. You also may notice that the imaginary component is denoted with j instead of i—that is something related to engineering. You might want to do a little research into the Python capabilities if `import cmath` is employed.

So write the function nthRootsOfComplexNumber(z,n) that takes in a complex number (z) and a positive integer (n) and returns a list of n roots of z in rectangular form. You should do all of your calculations in polar form but the results have to be in rectangular form. A complex version of almostEqual--complexAlmostEqual(z1,z2) has been placed in the starting point for testing purposes.

As an example: nthRootsOfComplexNumber(-27,3) should return [(1.5000000000000004+2.598076211353316j), (-3+3.6739403974420594e-16j), (1.4999999999999978-2.598076211353317j)]

segmentsIntersect(L1,L2) (25 points)

Given two lists that represent line segments L1 & L2 in the format [x1,y1,x2,y2], determine if the segments intersect in a single point. You might think that this is an easy task, and it is in routine situations. However, special cases make it a much more difficult problem. I'm telling you this because the test cases for this problem will be very demanding. You will need to think through the possibilities and design your own test cases that will really stretch the limits of your function. The test cases in the starting point are simple examples. The grading program will be far more thorough. The high point value here is a reward for self-testing. You can assume that all non-zero coordinate values have a magnitude in the interval [10**-8,10**12] and that any floating point calculations you perform need to have accuracy no worse than 10**-8.