Problem 4: Secret Sharing and Lagrange Steganography [30 points; individual only]

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/SecretSharingBlack on 3/22/2017

You've been hired by the Bank of Pasadena (BoP) to help them with their secret-sharing security protocols. The plan is to use the Lagrange Polynomials method that we saw in class! Specifically, we want to distribute pieces of a secret to a large number `n` of "trustees" so that *any* `k` of those "trustees" can reproduce the secret but *any* fewer trustees who collude have essentially no useful information. Both `n` and `k` will be parameters that can be selected by the bank administrators.

Recall that when we wish for `k` trustees to recreate the secret:

- The secret is a random polynomial of degree `k-1`, i.e., a polynomial of the form `a * X**(k-1) + a * X**(k-2) + ...` where `a`, `b`, etc. are random coefficients. For simplicity, we'll limit each of those coefficients to be random positive integers in the range 1 to 1000. (*In real life*, we'd probably use a much larger range of positive integers—say between 1 and a trillion—but you're welcome to use numbers as small as 1000 here if you wish.)
- **In fact**, rather than having the secret be the polynomial itself, we can make the secret a *single point* on that polynomial! Better yet, we can make the secret a single number: The y-intercept of the polynomial (the y-coordinate when the polynomial is evaluated at x=0). This number will be the actual secret that the trustees will attempt to reconstruct.
- Once we have that polynomial, we provide each of the `n` trustees with a randomly selected point on the polynomial such that each point has a unique x-coordinate.
- Any `k` trustees who get together with their pieces of the secret (their points) can uniquely recreate the polynomial. Any fewer than `k` trustees have no more information about the polynomial than they could have guessed by themselves! ("Super spiffy!" is BoP's corporate motto…)

Your boss at BoP has asked you to write four functions:

- `create_random_poly(degree, yint)` takes two arguments:

  - `degree`, the degree of the almost-entirely-random polynomial to create
  - `yint`, the desired y-intercept of the (otherwise) random polynomial it creates

  With these arguments, `create_random_poly` should return an almost-entirely-random polynomial of degree `degree` whose only non-random coefficient is the y-intercept, which should be `yint`. You should use random coefficients from `1` to `1000` (it doesn't matter if it's inclusive or not…) for the other pieces of the polynomial.

  In addition, you may ***represent*** your polynomial however you'd like! The reason we don't specify this is that we will only test it with the following *evaluation* function:

- `eval_poly(poly, x)` also takes two arguments:
  - `poly`, which is the result returned by your `create_random_poly`, above
  - `x`, the x value to be fed into the polynomial `poly`

  With these arguments, `eval_poly` should return *poly(x)*, the numeric value of the polynomial `poly` at the value `x`. We will test this mostly at `x = 0`, but you will want to use it to generate other points, specifically in this next function, `distributeSecret`:

- `distributeSecret(n, k, secret)` takes three arguments:
  - `n`, which is the number of points this function will return (as a list of lists). You may assume that `n ≥ 1`.
  - `k`, the number of the above `n` points required to reconstruct the "secret." You may assume that `k ≥ 1` and `k ≤ n`
  - `secret`, the secret value that should be the y-intercept (`yint`) of the secret polynomial you will create!

  With these arguments, `distributeSecret` should return a list of `n` lists. Each of those inner lists is of the form `[x,y]` and is a point on the secret polynomial with a distinct x value. With any `k` of those `n` points, the secret should be recoverable. In fact the fourth and final function will do that recovery:

- `eval_pointspoly(points, x)` takes two arguments:
  - `points` is a list of points (in the form returned by `distributeSecret`)

- `x` is simply an x-value at which the polynomial (defined by `points`) is to be evaluated.

With these arguments, `eval_pointspoly` should create the polynomial that passes through the points in `points` and then evaluate that polynomial at `x`. It should return the resulting y-value. For our application, as long as `points` has `k` of the points returned by `distributeSecret`, the result of calling `eval_pointspoly(points, 0)` should be the original secret. With fewer than `k` points, it shouldn't work (at least, not in general). There may be a bit of floating-point imprecision (we ask you to comment on this—see below). Also, check your implementations against the examples provided.

There is likely to be some numerical imprecision when `eval_poly_from_points` attempts to "build" and evaluate the Lagrange polynomial. Due to floating-point imprecision, the value returned by `eval_poly_from_points(pointList, 0)` may differ from the original y-intercept, which is the original "secret."

In a comment in your code, explain how much "slack" would be needed to allow for this imprecision; that is, try several tests, and write a few sentences about what tests you tried and ***how far away*** the result of `eval_poly_from_points(pointList, 0)` is from the original secret.

If you get stuck or are unsure how to approach the problem, refer to the lecture slides or check out Wolfram's page on Lagrange polynomials.

Examples to try

Try these out to make sure everything is working:

```
In [1]: eval_pointspoly([[1, 1], [2, 2]], 4)
Out[1]: 4.0

In [2]: eval_pointspoly([[1, 1], [2, 4], [3, 9]], 4)
Out[2]: 16.0

In [3]: POLY = create_random_poly(2, 42)

In [4]: print POLY
```

```
Out[4]: [480, 551, 42]   # your representation may differ!!

In [5]: eval_poly(POLY, 0)    # the y-int should be 42
Out[5]: 42

In [6]: eval_poly(POLY, 1)    # your value will probably be
different
Out[6]: 1073

In [7]: eval_poly(create_random_poly(3, 9001), 0)
Out[7]: 9001

In [8]: POINTS = distributeSecret(5, 3, 42)

In [9]: POINTS   # your points will be different!
Out[9]: [[1.0, 1042.0], [2.0, 2216.0], [3.0, 3564.0], [4.0,
5086.0], [5.0, 6782.0]]

In [10]: eval_pointspoly(POINTS[0:3], 0)  # but with 3 of the 5,
evaluated @ 0, it should yield the secret
Out[10]: 42.0

In [11]: eval_pointspoly(POINTS[0:2], 0)  # with too few, it
won't be the secret (in general)
Out[11]: -132.0

In [12]: eval_pointspoly(POINTS[1:3], 0)  # again
Out[12]: -480.0

In [13]: eval_pointspoly(POINTS[1:4], 0)  # any three should
suffice to obtain the secret.
Out[13]: 42.0
```

Please design your program carefully. You'll certainly need some helper functions beyond the two functions that we've asked for here. Please follow the same design guidelines outlined in the Nim and Mastermind problems.

Extra credit: perfectly-precise polynomials!

For up to +10 bonus points, avoid the floating-point imprecision and "slack" value *entirely* by operating exclusively with integers! To do this you'll need to use modular arithmetic where the modulus is a prime number. That is, you'll need to choose some large prime number `p`, and any addition or multiplication that you do will be `mod p` so that numbers `p` or larger wrap back to 0.) For example, if the prime modulus you chose was 5, then 2+4 = 1 (mod 5) and -1 = 4 (mod 5). Of course, your modulus should be much larger than 5. Doing business this way is called operating in a "finite field"; this technique is used in many modern cryptographic protocols.

So that we can check your extra-credit functions, write four variants of the above functions, named with a following `_p`:

- `create_random_poly_p`
- `eval_poly_p`
- `distributeSecret_p`
- `eval_pointspoly_p`

We will check these in the same way as above, but they should not use floating-point numbers at all. You can choose your own large prime for use here (please do make it over 9,000).

Note that you (probably) shouldn't have to change *anything* to make `create_random_poly_p` and `eval_poly_p` — but please do make new functions with these new names. For `distributeSecret_p` you will likely only need to make one small change. (Hint: Remember to keep the points in the finite field!) Depending on your implementation,`eval_pointspoly_p` may require a decent amount of modification. Most importantly, instead of setting up a numerator and dividing it by a denominator, you will need to *multiply*:

```
numerator * mod_inverse(denominator, yourPrime)
```

in addition to your modular addition and multiplication. You will need the following helper functions, `extendedGCD(a,b)` and `mod_inverse(k, prime)`. Try reading them over to understand them, and note that these functions, while not necessarily intuitive, are optimized for speed.

```
def extendedGCD(a,b):
    """calculates the extended greatest common denominator,
       using the extended Euclidean algorithm
       inputs: two integers a and b
       output: a tuple of form (GCD, x, y) where x and y are
               of the form such that a*x + b*y = GCD
    """
    x0, x1, y0, y1 = 1, 0, 0, 1
    while b != 0:
        q, a, b = a // b, b, a % b
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
    return  a, x0, y0
```

```
def mod_inverse(k, prime):
    """calculates the multiplicative inverse of k mod prime
       such that k * mod_inverse(k, prime) % prime = 1
    """
    k = k % prime                       # k is now always positive
    y = extendedGCD(prime, k)[2]    # find y such that 1 =
(prime * x) + (k * y)
    return y % prime                    # note that if we know x and
y to be integers, and setting x
                                        # to be negative, we have (k
* y) - (prime * x) = 1, and if we
                                        # note that -(prime * x) is
equivalent to %prime, then we have found
                                        # y * k % prime = 1, which
means y is the multiplicative inverse of k mod prime
```

Submit

Please submit your code to the usual in a file called `hw8pr4.py`