# EECS 70 Discrete Mathematics and Probability Theory
# Fall 2014 Anant Sahai Homework 6

**This homework is due October 13, 2014, at 12:00 noon.**

1. **Try It Out (optional)**

   Please do the online problems on FLT and RSA. Give us your brief comments on how you found them and how you think that they helped you understand the material better for this homework.

2. **RSA Lab**

   In this week's Virtual Lab, we will implement a toy RSA cryptosystem. We will start by generating the public and private keys for RSA using functions we implemented in the past two weeks (`mod_exp`, `egcd`, etc.). Then, we will encrypt and decrypt messages with the RSA function (for example, you will decrypt a secret and tell us in your written homework what the original message is). Finally, we will transition into Polynomials and implement Lagrange Interpolation for a degree 2 polynomial.

   Please download the IPython starter code from Piazza or the course webpage, and answer the following questions.

   (a) *Warmup*: Implement the function `pairwise_coprime`, which returns True if every pair of numbers in the input list is coprime and False otherwise.

   (b) In this part, we will give you sample implementations of the functions `is_probable_prime` (which uses the Miller-Rabin algorithm) and `gen_prime`, which generates a large pseudorandom prime.

   Implement the function `gen_key`, which generates a pair of public $(N, e)$ and private keys $(d)$ for RSA. Refer to Lecture Note 6 for more details on how RSA key generation works.

   For simplicity, you can generate $e$ as a random integer between 1 and $(p-1)(q-1)$ as long as it's coprime with $(p-1)(q-1)$ and return the keypair as a triple $(N, e, d)$.

   *In practice, e is usually chosen to be the minimum integer coprime with $(p-1)(q-1)$, however.*

   (c) Implement the functions `encrypt_integer` and `decrypt_integer`, which encrypts and decrypts a positive integer $x$, respectively, using the RSA function. Make sure to test your implementation by checking that

   `decrypt_integer(encrypt_integer(x, N, e), N, d) == x`

   for some positive integer $x$ between 2 and $N-1$.

   (d) We now want the ability to encrypt or decrypt actual text messages, instead of just positive integers. In this part, we will give you the code to convert text messages into lists of integers based on ASCII values (between 0 and 127). These numbers are then combined into blocks, each of size $n$, using base 256, which you can then apply the RSA function on.

   Implement the `encrypt` and `decrypt` functions for text messages. Then answer the two subparts below. For both subparts, we will give you the values of $N$, $e$, and $d$ that we want you to work with.

   - What's the secret of your class log-in (*cs70-XYZ*) using a block size of 4? If your login only has two letters, treat the last letter (Z) as a whitespace to keep the message length at 8 characters. Everyone's answer to this question should be quite different from one another!

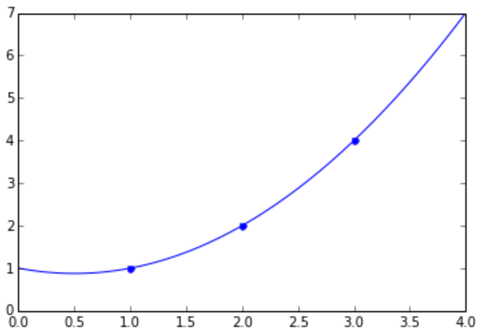
We will use `cs70-ta ` with a whitespace character at the end for the solution to this question (yours should be your own login!).
In this case, the secret turns out to be:
[2447810932550727353987691838700467048278221148712720786064910778322316771088013480762848466975948983782102933077611908382004105020071921300787281039482749456050897021815381244985326603049452469886042947L,
2581152557196637838932821724098901545896292901544401855383915855026780439239644340700306715092811985229978620976222813212541823555444139410374501542555674607875713200849790752745944947567980935590234536L]
Quite a huge number, isn't it?

- What's the original message of the following secret? You will need to figure out the block size yourself.
  *Hint*: the length of the original message is 126 (counting all characters, including whitespace). To figure out the block size, check the numbers from `range(1, len(message))` that divides the message's length.
  The values of $N$, $e$, and $d$, as well as the secret, can be found at `http://pastebin.com/BpesLp6d`. They are also embedded inside the code skeleton.
  Please report the final answers to both parts by copying and pasting from the notebook. Make sure you keep the original message for the second subpart intact. Tell us what block size you used as well (there is only one correct answer).
  The original message is **"One of the powerful things about mathematics is thatit provides you with an alternative to your naive intuition of being right"**, which can be found with a block size of 18.
  (There are other block sizes that "prints" the same message, but the decrypted messages include garble data. If you check the length of the message to be 126, you should see that the only correct block size is 18.)
  The words "that" and "it" are concatenated on purpose to check if students actually do the question or get the answer from someone else.
  This message is taken from one of Prof. Sahai's replies on Piazza, post #237.

(e) Recall from lecture that given three data points $(x_i, y_i)$ where all the $x_i$ are distinct, we can find a unique degree (at most) 2 polynomial $p(x)$ such that $p(x_i) = y_i$.
The three $\Delta_i$ functions can be found as follows.

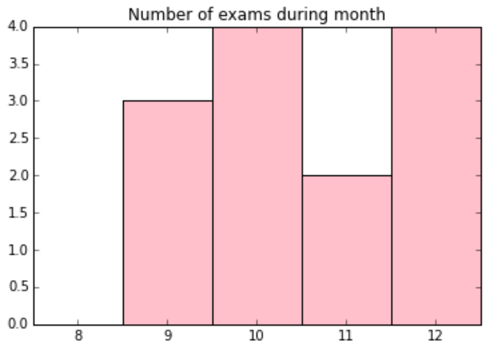
$$\Delta_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}, \tag{1}$$
$$\Delta_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}, \tag{2}$$
$$\Delta_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \tag{3}$$

Implement the function `interpolate_2D` to calculate the polynomial $p(x)$ using Lagrange Interpolation.
Test your implementation with the example in Note 7. If your implementation is correct, when you run the plotting code cell in the skeleton, you should see that the polynomial passes through the three given points $(1,1), (2,2)$ and $(3,4)$.



(f) Last week, we learned how to plot a simple curve using Matplotlib. This week, we will learn to plot a bar chart. By the end of this question, you will plot a bar chart grouping the EECS 70 lecture notes by their last modified months!

Let's first look at an example showing the number of exams a student typically has per month during the Fall semester (this is just a toy dataset!).



The code to make the above plot turns out to be very simple.

Listing 1: Simple Bar Chart Code

```
width = 1
y = [0, 3, 4, 2, 4]
x = np.arange(8, 13)
plt.bar(x, y, width=width, color="pink")
plt.xticks(x + width*0.5, x)  # center the xticks
plt.title('Number of exams during month')
```

We have also introduced four new commands that you will need for this week's lab:

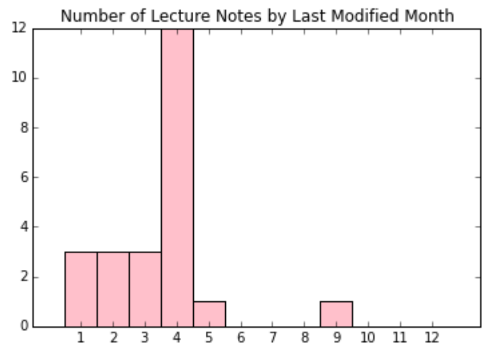
- `np.arange` is very similar to Python's `range` or `xrange`. The main difference is that `np.arange` returns a vectorized array. For example, scalar addition doesn't work on a normal Python's list, but it will add the same scalar to every element in the vectorized array. Those of you who are familar with Matlab will find this very familiar.
- `plt.bar` plots a bar chart, similar to `plt.plot` which plots a line curve from last week.
- `plt.xticks` sets the x-limits of the current tick locations and labels.
- `plt.title` gives the plot a meaningful title.

Believe it or not, you now know more than enough to complete this question! We will give you the code to recursively fetch the lecture note pdfs on the course website, as well as the Python commands to extract the last modified months. **You can tackle this problem in any creative way you want**, but here's our recommended approach.

- Group the lecture notes by month. (*Hint: use a counter dictionary, where the keys are 1 to 12 (the month), and the values are the number of notes that were last modified during that month*).
- Convert the counter dictionary into two lists and make sure to preserve the order. For example, there are three lecture notes that were last modified in January, so 1 should be the first value of the list of months, and 3 should be the first value of the list of counts.
- Plot the list values as a bar chart.

***Important**: the dataset we're working with is very small, and it is possible to create the counter dictionary by inspection. Please do NOT do this. Write the code to convert the dataset into a format usable for* `plt.bar`.

To help you understand what we expect, the final plot should at least look very similar to this.



Be as creative as you want, and feel free to add axes' labels, the count above each bar, etc. to your heart's content.

*Reminder*: When you finish, don't forget to convert the notebook to pdf and merge it with your written homework. Please also zip the `ipynb` file and submit it as `hw6.zip`.

**3. The Enemy**

The land of the Golden Bears has been conquered by an enemy who shall not be named, and even the bears can't bear it. A brave bear decides to gather her fellow bears together in secret to plan an attack against The Enemy. To announce the meeting info, she leaves messages written in secret codes that only bears can decipher. You are a young, strong bear, and you definitely want to join the revolution!

Here are the basic decrypting rules you learned from your *Freshbear Bootcamp*.

- Encryption is done on uppercase letters A to Z. If there are any other characters in the code, remove them, then add them back at their original positions after decoding the rest.
- To get the original text back, you map the letters A to Z to numbers 0 to 25, apply a corresponding decoding function $D$ on this array of numbers, and map the output numbers back to alphabets.

You recall the three encoding functions that you were taught. Let $a$ be the array of numbers, and $a_i$ be the $i^{\text{th}}$ element in $a$, $1 \leq i \leq n$, where $n$ is the number of elements. The functions are,

- *Caesar cipher* with offset $c$: $E_{Caesar}(a_i, c) = (a_i + c) \bmod 26$,

- *Accumulating cipher* with offset $c$: $E_{Acc}(a_i, c) = (\sum_{j=1}^{i} a_j + c) \bmod 26$,
- *RSA*: $E_{RSA}(a_i, p, q, e) = a_i^e \bmod (pq)$.

Unfortunately, you can only remember two decoding functions out of the three,

- *Caesar cipher* with offset $c$: $D_{Caesar}(a_i, c) = (a_i - c) \bmod 26$,
- *RSA*: $D_{RSA}(a_i, p, q, e) = a_i^d \bmod (pq)$, where $d$ is the multiplicative inverse of $e$ modulo $(p-1)(q-1)$,

but that is fine, you can figure out the decoding function for the *Accumulating cipher* by yourself anyway. You then revisit how to actually use these decoding functions. To decrypt "VZQL TO" that is encrypted with Caesar cipher with offset -1:

1. Strip out non-uppercase characters: "VZQL TO" $\rightarrow$ "VZQLTO"
2. Convert the string of characters to an array of numbers: "VZQLTO" $\rightarrow [21, 25, 16, 11, 19, 14]$
3. Apply the decoding function $D_{Caesar}$ to each element:

$$\begin{aligned}
&D_{Caesar}([21,25,16,11,19,14],-1)\\
&= [D_{Caesar}(21,-1), D_{Caesar}(25,-1), D_{Caesar}(16,-1), D_{Caesar}(11,-1), D_{Caesar}(19,-1), D_{Caesar}(14,-1)]\\
&= [(21+1) \bmod 26, (25+1) \bmod 26, (16+1) \bmod 26, (11+1) \bmod 26, (19+1) \bmod 26, (14+1) \bmod 26]\\
&= [22,0,17,12,20,15]
\end{aligned}$$

4. Convert the array of numbers back to a string of characters: $[22,0,17,12,20,15] \rightarrow$ "WARMUP"
5. Insert the stripped out non-uppercase characters back: "WARMUP" $\rightarrow$ "WARM UP"

Now you are ready to go! Find out the name of the enemy, the gathering place, and a secret message from the encoded messages below. Show your work.

It is useful to create a table of the alphabets and their corresponding numbers for fast lookup.

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

(a) The Enemy's name: "OVTLDVYR", encrypted with a Caesar cipher ($c = 7$).
The answer is "HOMEWORK".

1. Strip out non-uppercase characters:
"OVTLDVYR" $\rightarrow$ "OVTLDVYR"
2. Convert the string of characters to an array of numbers:
"OVTLDVYR" $\rightarrow [14, 21, 19, 11, 3, 21, 24, 17]$
3. Apply the decoding function $D_{Caesar}$ to each element:

$$\begin{aligned}
D_{Caesar}&([14,21,19,11,3,21,24,17],7)\\
&= [14-7, 21-7, 19-7, 11-7, 3-7, 21-7, 24-7, 17-7] \bmod 26\\
&= [7,14,12,4,-4,14,17,10] \bmod 26\\
&= [7,14,12,4,22,14,17,10]
\end{aligned}$$

4. Convert the array of numbers back to a string of characters:
$[7,14,12,4,22,14,17,10] \rightarrow$ "HOMEWORK"

5. Insert the stripped out non-uppercase characters back:
"HOMEWORK" $\rightarrow$ "HOMEWORK"

**Alternative way:**
Since Caesar cipher is just shifting alphabets, we can think of two alphabet rings lined up together, where the top line corresponds to the encrypted output and the bottom line corresponds to the original alphabet.

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Output** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| **Input** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

A Caesar cipher with $c = 7$ means sliding the bottom ring to the right 7 places cyclically.

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Output** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| **Input** | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |

From this you can look up the original alphabets (bottom row) directly from the encrypted outputs (top row). The answer is highlighted in yellow. $\square$

(b) The gathering place: "OVZ VJIVDDN YMGTZD", encrypted with an Accumulating cipher $(c = -5)$.
The answer is "THE WOZNIAK LOUNGE".

1. Strip out non-uppercase characters:
"OVZ VJIVDDN YMGTZD" $\rightarrow$ "OVZVJIVDDNYMGTZD"
2. Convert the string of characters to an array of numbers:
"OVZVJIVDDNYMGTZD" $\rightarrow [14,21,25,21,9,8,21,3,3,13,24,12,6,19,25,3]$
3. Let $b_i$ be the $i^{\text{th}}$ element in this array of encrypted numbers, $1 \le i \le n$. We want to recover $a_i$ back from $b_i$. From the encoding function,

$$b_i = (\sum_{j=1}^{i} a_j + c) \bmod 26$$

$$b_i = (a_i + \sum_{j=1}^{i-1} a_j + c) \bmod 26$$

$$a_i = (b_i - (\sum_{j=1}^{i-1} a_j + c)) \bmod 26 \qquad [\text{Because } 0 \le a_i, b_i < 26]$$

If $i = 1$, $\sum_{j=1}^{i-1} a_j + c = 0 + c = c$. Otherwise, $\sum_{j=1}^{i-1} a_j + c = b_{i-1}$. Therefore, the decoding function is,

$$D_{Acc}(b_i) = \begin{cases} (b_i - c) \bmod 26 & i = 1 \\ (b_i - b_{i-1}) \bmod 26 & i > 1 \end{cases}$$

Applying the decoding function $D_{Acc}$ to each element,

$$\begin{aligned} & D_{Acc}([14,21,25,21,9,8,21,3,3,13,24,12,6,19,25,3], -5) \\ &= [14+5,\ 21-14,\ 25-21,\ 21-25,\ 9-21,\ 8-9,\ 21-8,\ 3-21, \\ &\quad 3-3,\ 13-3,\ 24-13,\ 12-24,\ 6-12,\ 19-6,\ 25-19,\ 3-25] \bmod 26 \\ &= [19,7,4,-4,-12,-1,13,-18,0,10,11,-12,-6,13,6,-22] \bmod 26 \\ &= [19,7,4,22,14,25,13,8,0,10,11,14,20,13,6,4] \end{aligned}$$

4. Convert the array of numbers back to a string of characters:
   $[19,7,4,22,14,25,13,8,0,10,11,14,20,13,6,4] \to$ "THEWOZNIAKLOUNGE"
5. Insert the stripped out non-uppercase characters back:
   "THEWOZNIAKLOUNGE" $\to$ "THE WOZNIAK LOUNGE"

□

(c) The secret message: "QKHH JONK!", encrypted with RSA $(p = 2,\ q = 13$, and $e = 5)$.

The answer is "WELL DONE!".

1. Strip out non-uppercase characters:
   "QKHH JONK!" $\to$ "QKHHJONK"
2. Convert the string of characters to an array of numbers:
   "QKHHJONK" $\to [16,10,7,7,9,14,13,10]$
3. The decryption key $d$ is the multiplicative inverse of $e$ mod $(p-1)(q-1) = 5$ mod 12. We find the multiplicative inverse by running the EGCD algorithm,

$$\begin{aligned} 12 &= 5(2) + 2 \\ 5 &= 2(2) + 1 \\ 1 &= 5 - 2(2) \\ 1 &= 5 - (12 - 5(2))(2) \\ 1 &= 5(5) - 12(2) \end{aligned}$$

Therefore, $d = 5$. Applying the decoding function $D_{RSA}$ to each element,

$$\begin{aligned} &D_{RSA}\left([16,10,7,7,9,14,13,10],2,13,5\right) \\ &= [16^5,10^5,7^5,7^5,9^5,14^5,13^5,10^5] \bmod 26 \\ &= [22,4,11,11,3,14,13,4] \end{aligned}$$

4. Convert the array of numbers back to a string of characters:
   $[22,4,11,11,3,14,13,4] \to$ "WELLDONE"
5. Insert the stripped out non-uppercase characters back:
   "WELLDONE" $\to$ "WELL DONE!"

□

**4. Poker mathematics**

A *pseudo-random number generator* is a way of generating a large quantity of random-looking numbers, if all we have is a little bit of randomness (known as the *seed*). One simple scheme is the *linear congruential generator*, where we pick some modulus $m$, some constants $a,b$, and a seed $x_0$, and then generate the sequence of outputs $x_1,x_2,x_3,x_4\ldots$ according to the following equation:

$$x_{t+1} = (ax_t + b) \bmod m$$

(Notice that $0 \le x_t < m$ holds for every $t$.)

You've discovered that a popular web site uses a linear congruential generator to generate poker hands for its players. For instance, it uses $x_0$ to pseudo-randomly pick the first card to go into your hand, $x_1$ to pseudo-randomly pick the second card to go into your hand, and so on. For extra security, the poker site has kept the parameters $a$ and $b$ secret, but you do know that the modulus is $m = 2^{31} - 1$ (which is prime).

Suppose that you can observe the values $x_0$, $x_1$, $x_2$, $x_3$, and $x_4$ from the information available to you, and that the values $x_5, \dots, x_9$ will be used to pseudo-randomly pick the cards for the next person's hand. Describe how to efficiently predict the values $x_5, \dots, x_9$, given the values known to you.

**Answer 1:** We know

$$\begin{aligned} x_1 &\equiv ax_0 + b \pmod{m} \\ x_2 &\equiv ax_1 + b \pmod{m} \end{aligned}$$

Because we know $x_0$, $x_1$, and $x_2$, this is a system of two equations with two unknowns (namely, $a$ and $b$). So we can solve for $a$ and $b$. More explicitly, by subtracting the first equation from the second, we get

$$x_2 - x_1 \equiv a(x_1 - x_0) \pmod{m}$$

If $x_0 \equiv x_1 \pmod{m}$ then by induction on $n$ we see $x_n \equiv x_0 \pmod{m}$ for all $n$ which allows us to immediately calculate $x_5, x_6, x_7, x_8$ and $x_9$. So suppose $x_0 \not\equiv x_1 \pmod{m}$. Then $x_1 - x_0$ is invertible modulo $m$ (because $m$ is prime, therefore $\gcd(x_1 - x_0, m) = 1$), and we see

$$a \equiv (x_2 - x_1)(x_1 - x_0)^{-1} \pmod{m}$$

Once we know $a$, we can plug in the known value of $a$ into the first equation and solve for $b$:

$$b \equiv x_1 - ax_0 \equiv x_1 - x_0(x_2 - x_1)(x_1 - x_0)^{-1} \pmod{m}$$

Since we know $a$ modulo $m$ and $b$ modulo $m$, we can compute $x_5, x_6, x_7, x_8$ and $x_9$.

**Answer 2:** Alternatively, we could start by solving for $b$. Multiplying the first equation by $x_1$, multiplying the second equation by $x_0$, and subtracting gives $x_1^2 - x_2x_0 \equiv b(x_1 - x_0) \pmod{m}$, and then

$$b \equiv (x_1^2 - x_0x_2)(x_1 - x_0)^{-1} \pmod{m}$$

Now we can plug in the known value of $b$ into the first equation and solve for $a$, and continue as before.

**Note:** For both answers, It is important to consider the case when $x_1 \equiv x_0 \pmod{m}$. Solutions that didn't address that were incomplete, as $x_1 - x_0$ would not necessarily have an inverse modulo $m$.

**Comment:** This homework exercise was loosely modelled after a real-life story, where a group of computer scientists discovered serious flaws in the pseudorandom number generator used by several real poker sites. They could have used this to make thousands of dollars off everyone else who played at that site, but rather than cheat, they instead revealed the flaw to the poker site and the public. For more, see their paper "How We Learned to Cheat at Online Poker: A Study in Software Security" (`http://www.cigital.com/papers/download/developer_gambling.php`).

5. **RSA with three primes**
Show how you can modify the RSA encryption method to work with three primes instead of two primes (i.e. $N = pqr$ where $p, q, r$ are all prime), and prove the scheme you come up with works in the sense that $D(E(x)) \equiv x \bmod N$.

$N = pqr$ where $p, q, r$ are all prime. Then, let $e$ be co-prime with $(p-1)(q-1)(r-1)$. Give the public key: $(N, e)$ and calculate $d = e^{-1} \bmod (p-1)(q-1)(r-1)$. People who wish to send me a secret, $x$, send $y = x^e \bmod N$. I decrypt an incoming message, $y$, by calculating $y^d \bmod N$.

Does this work? We prove that $x^{ed} - x \equiv 0 \bmod N$ and thus $x^{ed} \equiv x \bmod N$. To prove that $x^{ed} - x \equiv 0 \bmod N$, we factor out the $x$ to get
$x \cdot (x^{ed-1} - 1) = x \cdot (x^{k(p-1)(q-1)(r-1)+1-1} - 1)$ because $ed \equiv 1 \bmod (p-1)(q-1)(r-1)$. As a reminder, we are considering the number: $x \cdot (x^{k(p-1)(q-1)(r-1)} - 1)$
We now argue that this number must be divisible by $p$, $q$, and $r$. Thus it is divisible by $N$ and $x^{ed} - x \equiv 0 \bmod N$.
To prove that it is divisible by $p$:

- if $x$ is divisible by $p$, then $x \cdot (x^{k(p-1)(q-1)(r-1)} - 1)$ is divisible by $p$.
- if $x$ is not divisible by $p$, then that means we can use FLT on the inside to show that $(x^{p-1})^{k(q-1)(r-1)} - 1 \equiv 1 - 1 \equiv 0 \bmod p$. Thus it is divisible by $p$.

To prove that it is divisible by $q$:

- if $x$ is divisible by $q$, then $x \cdot (x^{k(p-1)(q-1)(r-1)} - 1)$ is divisible by $q$.
- if $x$ is not divisible by $q$, then that means we can use FLT on the inside to show that $(x^{q-1})^{k(p-1)(r-1)} - 1 \equiv 1 - 1 \equiv 0 \bmod q$. Thus it is divisible by $q$.

To prove that it is divisible by $r$:

- if $x$ is divisible by $r$, then $x \cdot (x^{k(p-1)(q-1)(r-1)} - 1)$ is divisible by $r$.
- if $x$ is not divisible by $r$, then that means we can use FLT on the inside to show that $(x^{r-1})^{k(p-1)(q-1)} - 1 \equiv 1 - 1 \equiv 0 \bmod r$. Thus it is divisible by $r$.

**6. $d+2$ points vs. a polynomial of degree $d$**

(a) Given 3 points $(0,1)$, $(1,1)$, and $(2,3)$, use Lagrange interpolation to construct the degree-2 polynomial which goes through these points.
The interpolating polynomial is given by $P(x) = y_0\Delta_0(x) + y_1\Delta_1(x) + y_2\Delta_2(x)$ where

$$\begin{aligned}
\Delta_0(x) &= \frac{(x-1)(x-2)}{(0-1)(0-2)} \\
&= \frac{1}{2}x^2 - \frac{3}{2}x + 1 \\
\Delta_1(x) &= \frac{(x-0)(x-2)}{(1-0)(1-2)} \\
&= -x^2 + 2x \\
\Delta_2(x) &= \frac{(x-0)(x-1)}{(2-0)(2-1)} \\
&= \frac{1}{2}x^2 - \frac{1}{2}x
\end{aligned}$$

Thus, we have

$$\begin{aligned}
P(x) &= y_0\Delta_0(x) + y_1\Delta_1(x) + y_2\Delta_2(x) \\
&= 1\left(\frac{1}{2}x^2 - \frac{3}{2}x + 1\right) + 1\left(-x^2 + 2x\right) + 3\left(\frac{1}{2}x^2 - \frac{1}{2}x\right) \\
&= \boxed{x^2 - x + 1}
\end{aligned}$$

(b) Given 4 points $(0,1)$, $(1,1)$, $(2,3)$, and $(-1,3)$, does there exist a degree-2 polynomial which goes through these points? If yes, find the polynomial; if no, explain why none exists.

Yes. The polynomial in the previous question passes through the first three points. Evaluating $P(-1) = 3$ verifies that it also passes through the fourth point, $(-1,3)$.

(c) Given 4 points $(0,1)$, $(1,1)$, $(2,3)$, and $(-1,0)$, does there exist a degree-2 polynomial which goes through these points? If yes, find the polynomial; if no, explain why none exists.

No. If there existed a polynomial through those four points, then that same polynomial must necessarily pass through the first three points. However, we know that there is only one degree-2 polynomial $P(x)$, which we found in the first part, that passes through the first three points. Since $P(-1) = 3 \neq 0$, it does not pass through the fourth point, $(-1,0)$, and we have a contradiction.

(d) Design a machine (i.e. give the pseudocode for an algorithm) with the following function: given four points $(x_1,y_1),(x_2,y_2),(x_3,y_3),(x_4,y_4)$ with all the $x_i$ distinct, the machine outputs YES if there exists a polynomial $p(x)$ of degree at most 2 such that $p(x_i) = y_i$ for all $i$; otherwise, it outputs NO.

Following the intuition from the previous part: If a degree-2 polynomial passes through the four points, it must pass through the first three. Thus, we may use Lagrange interpolation to construct the unique polynomial, call it $p(x)$ through the first three points. Finally, we simply need to verify if $p(x_4) = y_4$.

```
check_points((x0, y0), (x1, y1), (x2, y2), (x3, y3)):
y = y0*(x3-x1)*(x3-x2)/((x0-x1)*(x0-x2))
+ y1(x3-x0)*(x3-x2)/((x1-x0)*(x1-x2))
+ y2*(x3-x0)*(x3-x1)/((x2-x0)*(x2-x1))
return y == y3
```

## 7. Because the Moth just doesn't cut it

Gandalf the Grey (a good wizard) wanders about on his merry adventures but frequently runs into some troubles with goblins and orcs along the way. Always being the well-prepared wizard that he is, Gandalf has enlisted the service of the Great Eagles to fly him out of sticky situations at a moment's notice. To do this, he broadcasts a short message detailing his dilemma and a nearby eagle will come to his aid.

While this is all well and good, Saruman the White (an evil wizard) wants in on this eagle concierge service. The eagles can no longer trust just any distress call they receive! Gandalf needs you (a cryptography master) to help him devise a simple scheme that will allow the eagles to verify his identity whenever he broadcasts a message out. Not only that, but the eagles need to know when the message they receive from Gandalf has been tampered with.

Once you have devised this scheme, Gandalf will tell it to the eagle lord Gwaihir, who will relay it out to the rest of the world (they are loudmouths so they can't keep a secret).

To summarize:

(a) Gandalf broadcasts a message $m$ to all of Middle-Earth.

(b) He is able to attach to the message an extra piece of information $s$ that verifies his identity (i.e. cannot be forged) to whomever receives it.

(c) If the message has been modified in transit, recipients of the modified message should be able to detect that it is not original.

(d) Everyone in Middle-Earth knows the scheme (i.e. the algorithm itself is not a secret)

Your job in this problem is to devise an algorithm (like RSA) that meets the above criteria. In your answer, you should formally prove that Gandalf's messages can be successfully verified. You do not need to formally prove (though it should still be the case) that it is difficult to forge/tamper with messages, but you should provide some informal justification.

For those that did not understand the title of the problem: http://lotr.wikia.com/wiki/The_Moth

This type of scheme is known as a digital signature and is closely related to public-key cryptography.

Main Idea: We let Gandalf have a private key that only he knows and release to all the Eagles (and potentially all of Middle-Earth) a public key. The scheme must be such that it is only possible to generate a valid signature with the private key (i.e. only Gandalf knows how to sign messages). Secondly, any Eagle with the message and signature must be able to verify the authenticity using the public key.

Nitty Gritty Details: We generate the public/private key pair in the same fashion as in RSA: find two large primes $p,q$ and compute $N = pq$. Next choose $e$ relatively prime to $(p-1)(q-1)$ and compute its multiplicative inverse $d$ modulo $(p-1)(q-1)$. In RSA, only the person with the private key knows how to *decrypt* the message, while everyone else knows how the encrypt. However, with digital signatures, only the person with the private key knows how the *encrypt* the message.

We sign the message with $s = m^d \mod N$. Anyone receiving the message + signature pair can easily verify the signature by computing $s^e \mod N$ and seeing that it equals the original $m$.

An outsider like Saruman cannot forge the signature because he does not know Gandalf's private key to encrypt the message. Similarly, if the message is modified in transit, then we have that $m \neq s^e \mod N$. Coming up with this scheme is all that is required to get full credit on this problem.

For Intrepid Students Who Like Cryptography (not required to get full credit):
The above scheme isn't perfect. An outsider can generate a random signature $s$ and compute a message $m = s^e \mod N$. The message + signature pair $(m,s)$ will be valid. However, finding a signature $s$ that will generate a non-gibberish message $m$ is hard. This is like a prank call, annoying but not extremely dangerous. A second attack works when the attacker knows two or more valid message + signature pairs $(m_1,s_1),(m_2,s_2)$. Then we can construct another message + message pair: $m_3 = m_1m_2$ and $s_3 = s_1s_2$. Both of these can be fixed by requiring Gandalf to prefix all messages, for example with "I am Gandalf".

Finally, Gandalf may want to secure his scheme against "replay attacks", where Saruman copies a valid message $(m,s)$ from Gandalf, and re-broadcasts it later to the eagles. Such attacks may be prevented in various ways, for example by requiring all messages to begin with the current time (validated upon receipt).

## 8. Breaking RSA

(a) Eve is not convinced she needs to factor $N = pq$ in order to break RSA. She argues: "All I need to know is $(p-1)(q-1)$... then I can find $d$ as the inverse of $e$ mod $(p-1)(q-1)$. This should be easier than factoring $N$". Prove Eve wrong, by showing that finding $(p-1)(q-1)$ is at least as hard as factoring $N$ (that is, show that if she knows $(p-1)(q-1)$, she can easily factor $N$). Assume Eve has a friend Wolfram, who can easily return the roots of polynomials over $\mathbb{R}$ (this is, in fact, easy).

Let $a = (p-1)(q-1)$. If Eve knows $a = (p-1)(q-1) = pq-(p+q)+1$, then she knows $p+q = pq-a+1$ (note that $pq = N$ is known too). In fact, $p$ and $q$ are the two roots of polynomial $f(x) = x^2-(p+q)x+pq$ because $x^2-(p+q)x+pq = (x-p)(x-q)$. Since she knows $p+q$ and $pq$, she can give the polynomial $f(x)$ to Wolfram to find the two roots of $f(x)$, which are exactly $p$ and $q$.

(b) When working with RSA, it is not uncommon to use $e = 3$ in the public key. Suppose that Alice has sent Bob, Carol, and Dorothy the same message indicating the time she is having her birthday party. Eve, who is not invited, wants to decrypt the message and show up to the party. Bob, Carol, and

Dorothy have public keys $(N_1, e_1), (N_2, e_2), (N_3, e_3)$ respectively, where $e_1 = e_2 = e_3 = 3$. Furthermore assume that $N_1, N_2, N_3$ are all different. Alice has chosen a number $0 \le x < \min\{N_1, N_2, N_3\}$ which indicates the time her party starts and has encoded it via the three public keys and sent it to her three friends. Eve has been able to obtain the three encoded messages. Prove that Eve can figure out $x$. First solve the problem when two of $N_1, N_2, N_3$ have a common factor. Then solve it when no two of them have a common factor. Again, assume Eve is friends with Wolfram as above.

Eve first tests the GCD of all pairs of $N_1, N_2, N_3$. Let $d_1 = gcd(N_1, N_2)$, $d_2 = gcd(N_2, N_3)$ and $d_3 = gcd(N_1, N_3)$. Then there are two cases:

case 1 If one of the $d_1, d_2$ and $d_3$ is greater than 1, it must be one of the prime factors $p$ of the two $N_i$'s. The other prime factor $q$ can be recovered by $q = \frac{N_i}{p}$. Therefore we can factorize one of the $N_i$'s and once we do that RSA is broken.

case 2 If $d_1 = d_2 = d_3 = 1$, it means all pairs of the $N_i$'s are coprime. Let the three encoded messages be $y_1, y_2, y_3$. Since the messages are encoded by RSA with public keys $(N_1, 3)$, $(N_2, 3)$ and $(N_3, 3)$, we have:

$$x^3 \equiv y_1 \bmod N_1$$
$$x^3 \equiv y_2 \bmod N_2$$
$$x^3 \equiv y_3 \bmod N_3$$

Since all pairs of $N_1, N_2, N_3$ are coprime, By using the Chinese Remainder Theorem, we can solve the above system of congruence equations. Let the solution be

$$x^3 \equiv x_0 \bmod N_1 N_2 N_3$$

with $0 \le x_0 < N_1 N_2 N_3$. Since $x < N_1, N_2, N_3$, $x^3 < N_1 N_2 N_3$ and thus $x^3 = x_0$. We can take the cube root of $x_0$ and recover the original message $x = x_0^{1/3}$. In this problem, the trick is that we were able to convert a problem of finding cube-roots mod a prime (which is hard) into finding cube-roots in the integers (which is easy).

## 9. Coin tosses over text messages

Alice and Bob want to flip a fair coin. However, they are on different continents, so they try to flip a coin by exchanging text messages. Neither one trusts the other, so they want to make sure that neither Alice nor Bob can affect the result of the coin toss (being able to affect the result would be obviously problematic if they are betting on the result of the random bit, for instance). Obviously, one person cannot simply toss the coin and "report" the results.

Bob proposes the following: "Each of us flips our own coin, with some result in $\{0, 1\}$. Then we will send our result to the other, and agree that the final coin should be the XOR of our individual coins"

(a) What is wrong with this method?

In the real world, we cannot guarantee that both messages are received at exactly the same time. But then the first person to receive a message can set the result of the final coin, by faking his/her individual coin toss. For example, if Alice sends Bob a 1 first, Bob could force the final coin to be 0 by claiming he tossed a 1 too, or he could force it to be 1 by claiming he tossed a 0. The core problem is that the first person to send a message reveals his/her choice, allowing the second person to cheat.

(b) Propose a secure coin-flipping protocol using RSA. You do not need to give a formal proof of security, but explicitly list all properties you require from RSA, and why you need them (such that, given these properties, you could carry through a proof of security). *Be careful! Remember your protocol should be secure against cheating at any stage.*

To fix the issues with the above protocol, the main idea is to use RSA as a **commitment scheme**. We would like to implement the digital analog of the following physical scheme: Alice flips a coin, puts the result in a locked safe, and sends the safe to Bob. She has "committed" to her result, but Bob cannot see it. Now Bob flips his coin, and sends his bit in the clear to Alice. Finally, Alice sends Bob the key to the safe, and they agree to use the XOR of their two coins.

Let consider what properties we need from a "digital safe" to implement the above scheme. Roughly, the safe above had the following two properties:

1. **Hiding Property:** The safe hides information about Alice's coin from Bob.
2. **Binding Property:** Once Alice has sent the safe ("committed"), she cannot later change the result of her coin (she is bound to the coin she sent).

Notice the first property ensures Bob can't cheat, while the second ensures Alice can't cheat. We will use RSA as a digital commitment scheme in the following protocol:

1. **Alice:** Sample large primes $p, q$ such that $N = pq > 2^{1000}$, and publish an RSA public-key $(N, e)$. Then generate a 1000-bit string $r$ by flipping 1000 coins, and send the encryption $y = E_e(r)$ to Bob.
2. **Bob:** Flip a coin to get $b \in \{0, 1\}$, and send $b$ to Alice.
3. **Alice:** Reveal $r, p, q$ to Bob.
4. **Bob:** Confirm $y = E_e(r)$, and validate the RSA keys: Test the primality of $p, q$, confirm $N = pq$, and $gcd(e, (p-1)(q-1)) = 1$.

Then they agree to use the XOR of all bits in $r$ and $b$ as their fair coin. (That is, the final coin is $b \oplus \bigoplus_i r_i$).

Intuitively, the encryption $E_e(r)$ is like the locked safe, which Alice uses to commit to her coin. Then Bob chooses his coin, and later Alice reveals the private keys, thus unlocking the safe. There are many subtleties here, so let us first list the properties we require from our RSA-based commitment scheme:

1. **Hiding Property:** For any valid public key $(N, e)$: If the 1000-bit string $r$ is chosen randomly, no computationally-bounded adversary will be able to determine the XOR of bits in $r$ from the encryption $E_e(r)$ and the public key $(N, e)$. (That is, no adversary will be able to guess the XOR with probability much more than 1/2 – same as random guessing).
2. **Binding Property:** For any $r$, the pair $(N, e)$ together with the encryption $E_e(r)$ is binding: There does not exist any primes $p', q'$ and bit-string $r'$ such that $p'q' = N$, $gcd(e, (p'-1)(q'-1)) = 1$, and $E_e(r) = E_e(r')$, but $r \neq r'$. (In particular, this must hold even if the pair $(N, e)$ is NOT a valid RSA keypair).

Notice the Hiding Property is slightly weaker than before, since now we only require security against computationally-bounded adversaries. (In fact, no digital commitment scheme can achieve the full Hiding and Binding properties as our safe – but we will be happy with the realistic assumption of computationally-bounded adversaries). Further, we are requiring a stronger property from RSA: not only should it be hard for an adversary to find $r$ given $E_e(r)$ (the usual security requirement), but it should be hard to find even the XOR of bits in $r$. This is a non-trivial strengthening... it is possible to construct an encryption scheme that reveals the XOR of all bits in the message, but remains hard to fully invert.

The Binding Property is as strong as before. Our RSA-based commitment scheme is in fact perfectly binding, due to the uniqueness of prime factorization, and the fact that encryption under a valid public key is a bijection.

Now we can argue the security of our coin-flipping protocol, using the above properties. First, we will show that Bob cannot cheat against an honest Alice: If Alice chooses a valid public-key, then the Hiding Property guarantees that Bob cannot find the XOR of bits in $r$ before he sends his bit $b$. Therefore, he learns nothing that could give him an advantage in influencing the final coin. To show that Alice cannot cheat: By the Binding Property, Alice cannot reveal a different $r$ to Bob in Step 3 than she chose in Step 1. Therefore her contribution to the final coin is determined before receiving any information from Bob, so she can't cheat. We have shown that no dishonest player can cheat against an honest player. Finally, by construction, if at least one honest player abides by the protocol, the final coin will be fair (unbiased).

Note that if Bob did not confirm Alice's key pair $(N,e)$ was valid in Step 4, Alice could generate $(N,e)$ such that the encryption function $E_e(x)$ is no longer a bijection. Then the Binding Property would not hold, and she may be able to send Bob some $r' \neq r$ such that $E(r') = E(r)$. If $r$ and $r'$ have a different number of ones, then Alice can cheat (by seeing Bob's $b$, and deciding which $r, r'$ to send).

**Common Mistakes:**

- If Alice does not send Bob $p, q$, then Bob cannot confirm Alice's key pair is valid, and Alice could cheat as described above. (For example, if Alice sends only a private key $d$ instead of $p, q$, she may be able to cheat by choosing an appropriate fake private key, after receiving Bob's bit $b$).
- If either party sends a deterministic encryption of their bit, then the Hiding Property will not hold. For example, if Alice only flips one coin to get a bit $k$, and sends the encryption $E_e(k)$ to Bob, then Bob can easily discover her bit by trying to encrypt $k = 0$ vs $k = 1$. The correct protocol prevents this, by allowing the bit 0 to be represented by an exponential number of messages $r$ (all those with an even number of 1s).

(c) Find at least one flaw in the following alternate protocol:

1. Alice: Publish a public-key $(N_A, e_A)$.
2. Bob: Publish a public-key $(N_B, e_B)$.
3. Alice: Generate a 1000-bit string $r_A$ by flipping 1000 coins. Send the encryption $y_A = E_{e_A}(r_A)$ to Bob.
4. Bob: Generate a 1000-bit string $r_B$ by flipping 1000 coins. Send the encryption $y_B = E_{e_B}(r_B)$ to Alice.
5. Alice: Reveal $r_A$ to Bob.
6. Bob: Reveal $r_B$ to Alice.
7. Alice: Confirm $y_B = E_{e_B}(r_B)$.
8. Bob: Confirm $y_A = E_{e_A}(r_A)$.

Then they agree to use the XOR of all 2000 bits in $r_A, r_B$ as their fair coin.

The most significant flaw is: If Bob just copies Alice exactly at every step (publishing her public-key as his public-key, and her encrypted message as his own), then the result of the XOR will always be zero! (since $r_A = r_B$).

Notice that Bob does not need access to any private information: He simply relays every message Alice sends to him back to her ("What a coincidence! I chose the same thing").

Another flaw: Since Alice doesn't check Bob's public key is valid, Bob could also cheat as described in the previous part.

10. **Write your own problem**

Write your own problem related to this week's material and solve it. You may still work in groups to brainstorm problems, but each student should submit a unique problem. What is the problem? How to formulate it? How to solve it? What is the solution?