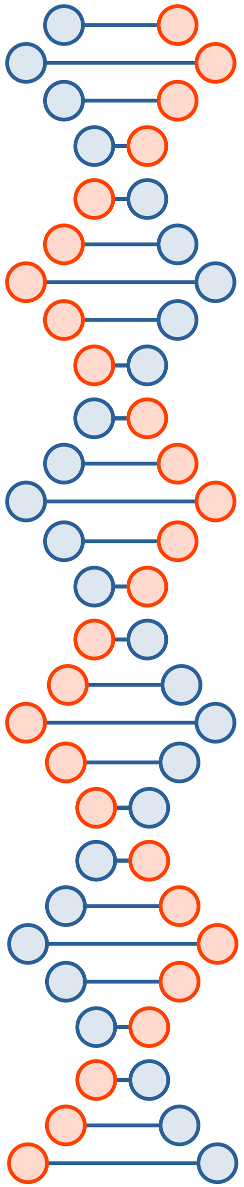
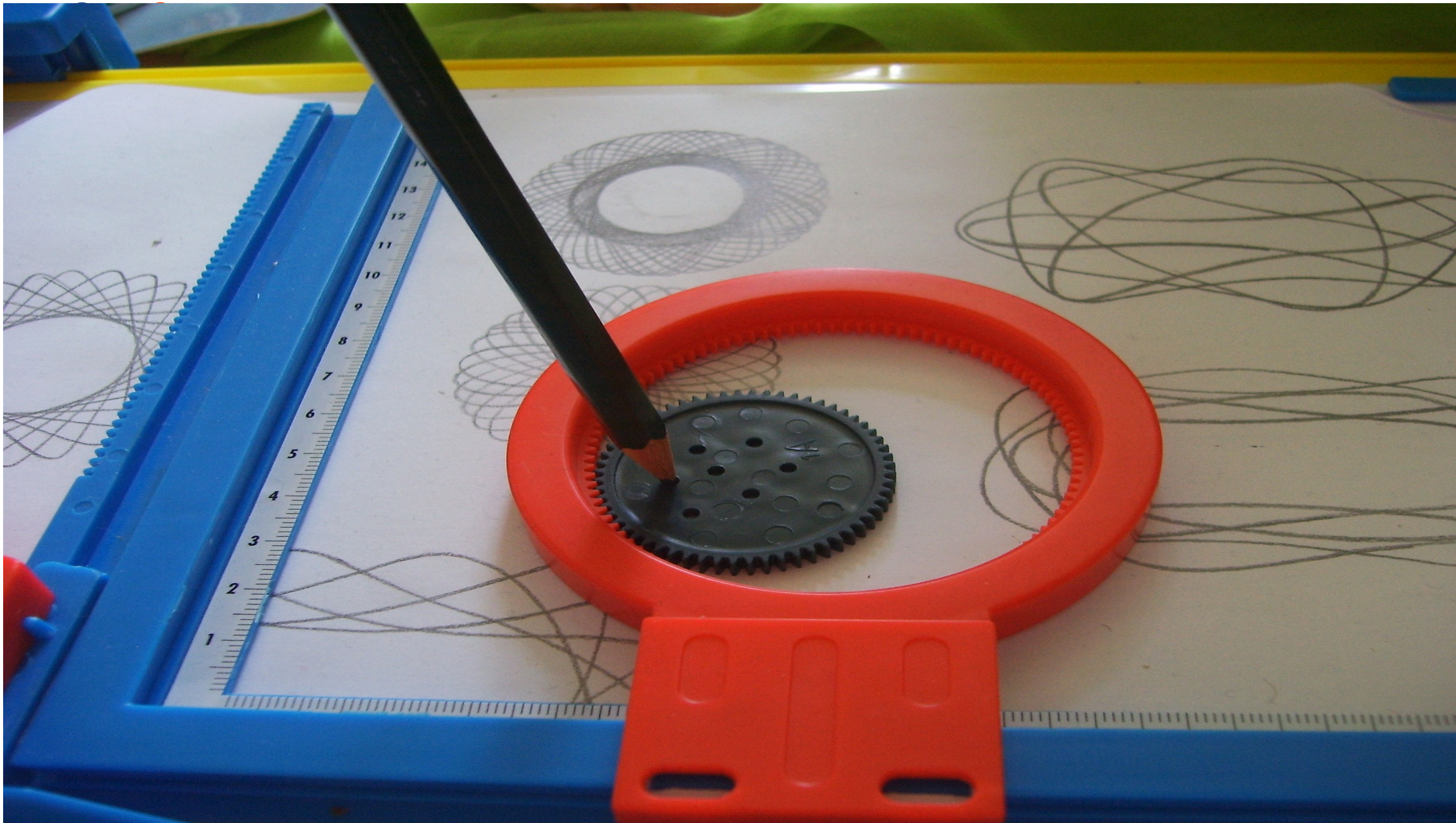
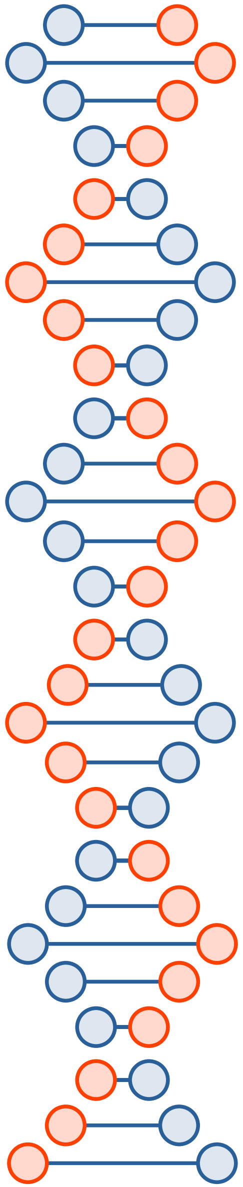


RSA

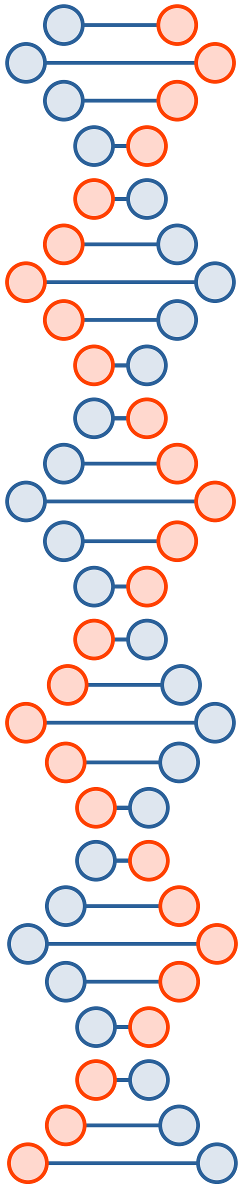
CSE 548 Spring 2025
jedimaestro@asu.edu





RSA

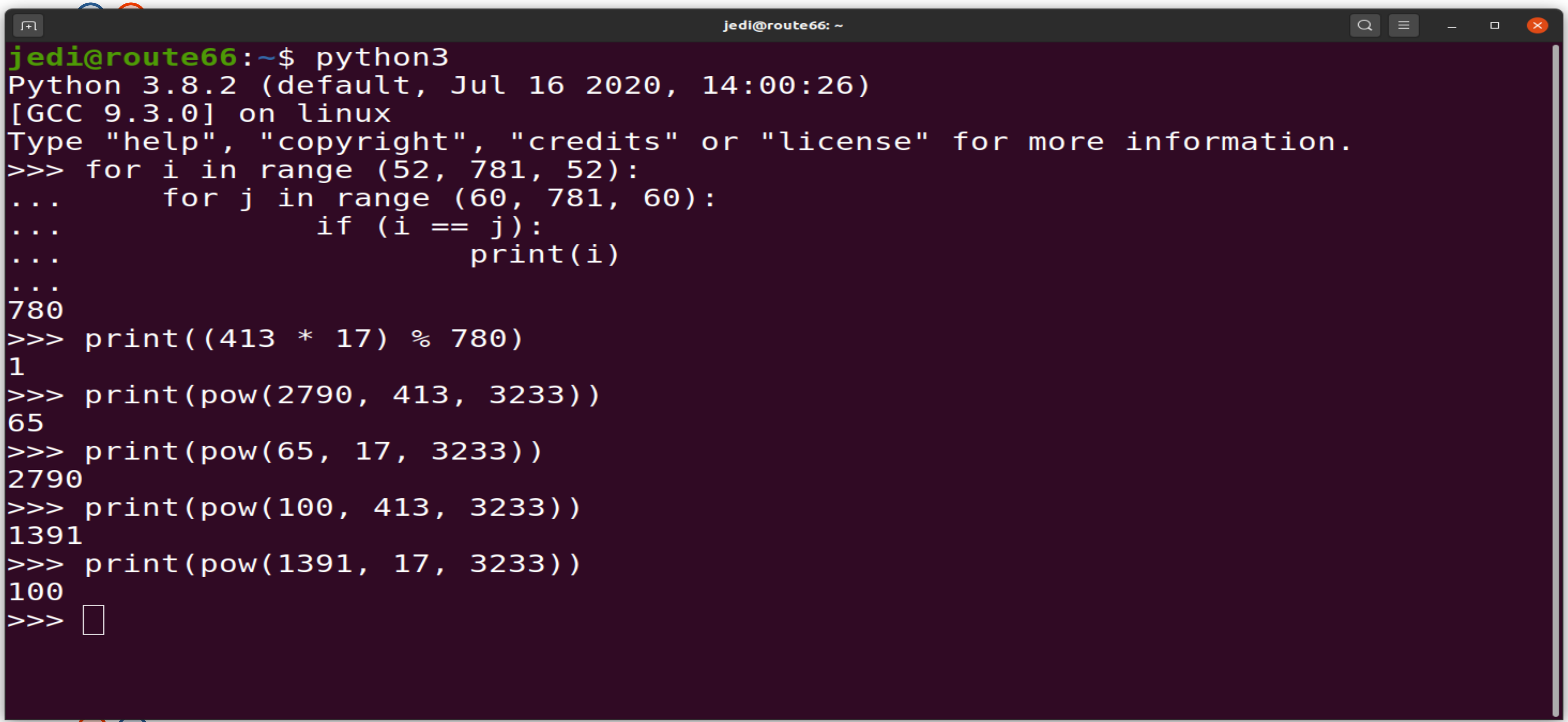
- Security is based on the hardness of integer factorization



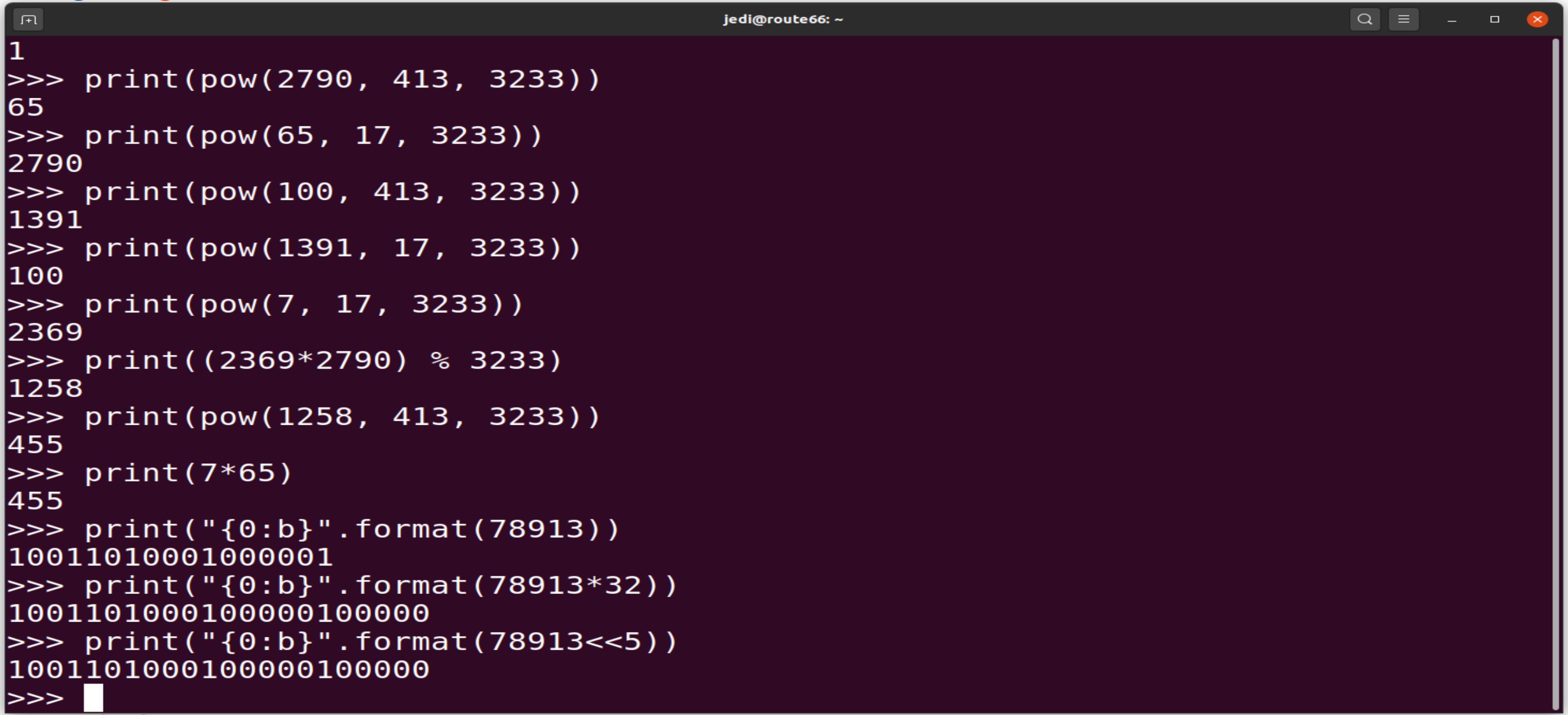
n = pq

- p and q are primes, suppose p = 61, q = 53
- n = 3233
- Euler's totient counts the positive integers up to n that are relatively prime to n
  - (61 – 1)(53 - 1) = 3120
- Carmichael’s totient function = lcm(60, 52)
  - 52,104,156,208,260,312,364,416,468,520,572,624,676,728,780
  - 60,120,180,240,300,360,420,480,540,600,660,720,780
- Choose 1 < e < 780 coprime to 780, e.g., e = 17
- d is the modular multiplicative inverse of e, d = 413
- 413 * 17 mod 780 = 1

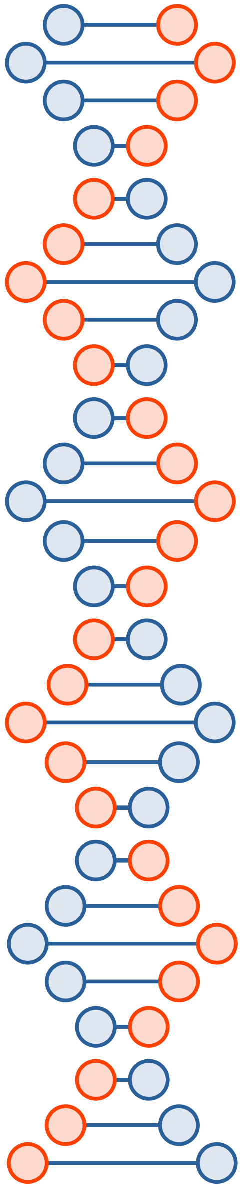
- Public key is (n = 3233, e = 17)
- Private key is (n = 3233, d = 413)
- Encryption: c(m = 65) = 65^{17} mod 3233 = 2790
- Decryption: m = 2790^{413} mod 3233 = 65
- Could also do...
  - Signature: s = 100^{413} mod 3233 = 1391
  - Verification: 100 = 1391^{17} mod 3233
- Fast modular exponentiation is the trick
  - Also need extended Euclidean algorithm
- Using RSA for key exchange or encryption is often a red flag, more commonly used for signatures



```
jedi@route66:~$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> for i in range (52, 781, 52):
...     for j in range (60, 781, 60):
...         if (i == j):
...             print(i)
...
780
>>> print((413 * 17) % 780)
1
>>> print(pow(2790, 413, 3233))
65
>>> print(pow(65, 17, 3233))
2790
>>> print(pow(100, 413, 3233))
1391
>>> print(pow(1391, 17, 3233))
100
>>> 
```



```
1
>>> print(pow(2790, 413, 3233))
65
>>> print(pow(65, 17, 3233))
2790
>>> print(pow(100, 413, 3233))
1391
>>> print(pow(1391, 17, 3233))
100
>>> print(pow(7, 17, 3233))
2369
>>> print((2369*2790) % 3233)
1258
>>> print(pow(1258, 413, 3233))
455
>>> print(7*65)
455
>>> print("{0:b}".format(78913))
10011010001000001
>>> print("{0:b}".format(78913*32))
1001101000100000100000
>>> print("{0:b}".format(78913<<5))
1001101000100000100000
>>> 
```



“Relatively prime”

- 9 is not prime, 9 = 3^2
- 13 is prime
- 10 is not prime, 10 = 5*2
- 9 and 10 are relatively prime, gcd(9,10) = 1
- 5 and 10 are not relatively prime, gcd(5,10) = 5
- Also called “coprime”

$$M^{\phi(n)} \equiv 1 \pmod{n} \ . \tag{3}$$

Here $\phi(n)$ is the Euler totient function giving number of positive integers less than n which are relatively prime to n. For prime numbers p,

$$\phi(p) = p - 1 \ .$$

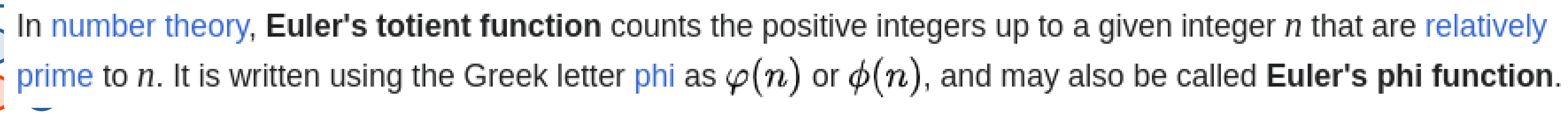
In our case, we have by elementary properties of the totient function [7]:

$$\begin{aligned} \phi(n) &= \phi(p) \cdot \phi(q) \\ &= (p-1) \cdot (q-1) \\ &= n - (p+q) + 1 \ . \end{aligned} \tag{4}$$

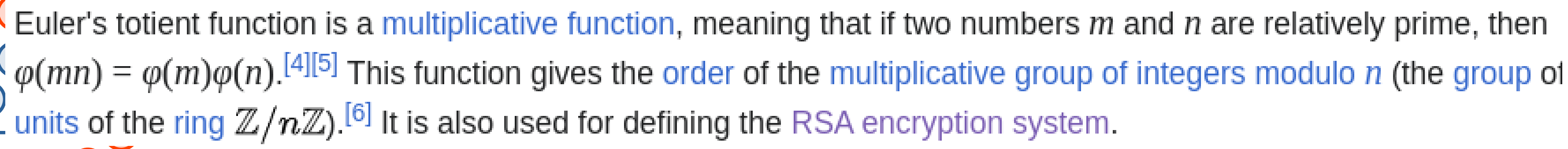
Since d is relatively prime to $\phi(n)$, it has a multiplicative inverse e in the ring of integers modulo $\phi(n)$:

Euler's totient function

- https://en.wikipedia.org/wiki/Euler%27s_totient_function



In number theory, **Euler's totient function** counts the positive integers up to a given integer n that are relatively prime to n. It is written using the Greek letter phi as $\varphi(n)$ or $\phi(n)$, and may also be called **Euler's phi function**.



Euler's totient function is a multiplicative function, meaning that if two numbers m and n are relatively prime, then $\varphi(mn) = \varphi(m)\varphi(n)$.[4][5] This function gives the order of the multiplicative group of integers modulo n (the group o units of the ring $\mathbb{Z}/n\mathbb{Z}$).[6] It is also used for defining the RSA encryption system.

Since d is relatively prime to $\phi(n)$, it has a multiplicative inverse e in the ring of integers modulo $\phi(n)$:

$$e \cdot d \equiv 1 \pmod{\phi(n)}. \tag{5}$$

$$\begin{aligned} D(E(M)) &\equiv (E(M))^d \equiv (M^e)^d \pmod{n} = M^{e \cdot d} \pmod{n} \\ E(D(M)) &\equiv (D(M))^e \equiv (M^d)^e \pmod{n} = M^{e \cdot d} \pmod{n} \end{aligned}$$

and

$$M^{e \cdot d} \equiv M^{k \cdot \phi(n)+1} \pmod{n} \text{ (for some integer } k).$$

From (3) we see that for all M such that p does not divide M

$$M^{p-1} \equiv 1 \pmod{p}$$

and since $(p-1)$ divides $\phi(n)$

$$M^{k\cdot\phi(n)+1} \equiv M \pmod{p}.$$

This is trivially true when $M \equiv 0 \pmod{p}$, so that this equality actually holds for *all* M. Arguing similarly for q yields

$$M^{k\cdot\phi(n)+1} \equiv M \pmod{q}\ .$$

Together these last two equations imply that for all M,

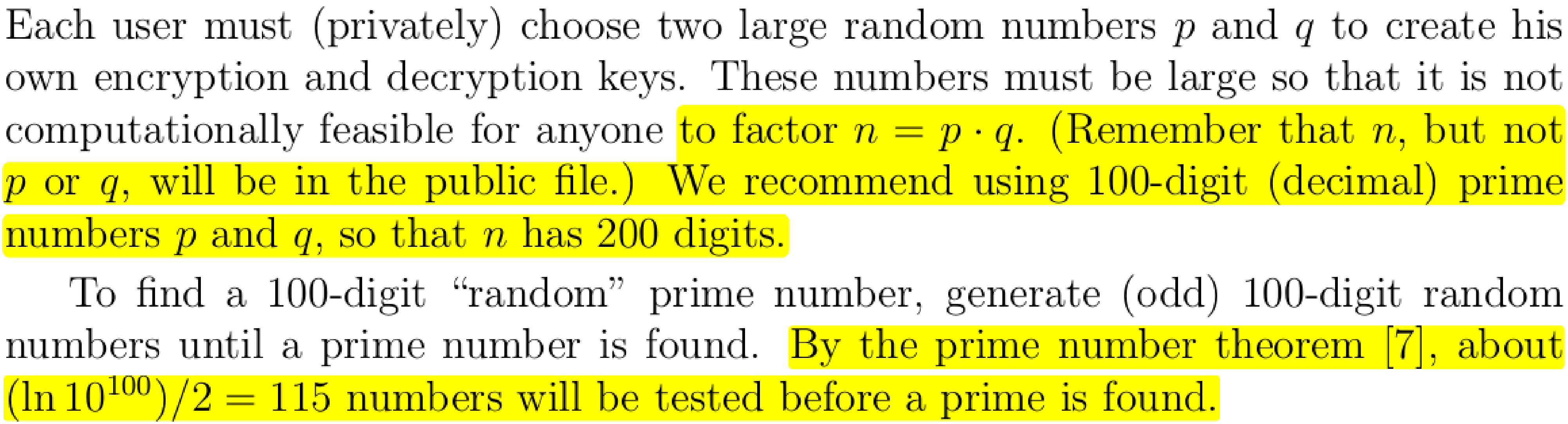
$$M^{e\cdot d} \equiv M^{k\cdot\phi(n)+1} \equiv M \pmod{n}.$$

This implies (1) and (2) for all $M, 0 \le M < n$. Therefore E and D are inverse permutations. (We thank Rich Schroeppel for suggesting the above improved version of the authors' previous proof.)

Computing $M^e \pmod{n}$ requires at most $2 \cdot \log_2(e)$ multiplications and $2 \cdot \log_2(e)$ divisions using the following procedure (decryption can be performed similarly using d instead of e):

```
Step 1. Let $e_k e_{k-1} \ldots e_1 e_0$ be the binary representation of $e$.
Step 2. Set the variable $C$ to 1.
Step 3. Repeat steps 3a and 3b for $i = k, k-1, \ldots, 0$:
    Step 3a. Set $C$ to the remainder of $C^2$ when divided by $n$.
    Step 3b. If $e_i = 1$, then set $C$ to the remainder of $C \cdot M$ when divided by $n$.
Step 4. Halt. Now $C$ is the encrypted form of $M$.
```

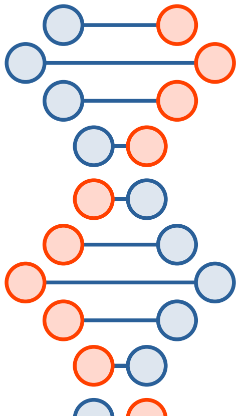
This procedure is called "exponentiation by repeated squaring and multiplication." This procedure is half as good as the best; more efficient procedures are known. Knuth [3] studies this problem in detail.



Each user must (privately) choose two large random numbers p and q to create his own encryption and decryption keys. These numbers must be large so that it is not computationally feasible for anyone to factor $n = p \cdot q$. (Remember that n, but not p or q, will be in the public file.) We recommend using 100-digit (decimal) prime numbers p and q, so that n has 200 digits.

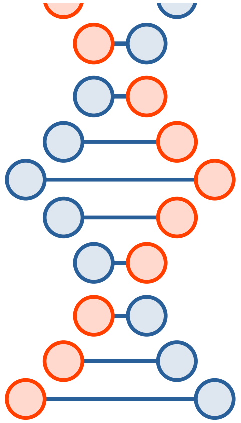
To find a 100-digit "random" prime number, generate (odd) 100-digit random numbers until a prime number is found. By the prime number theorem [7], about $(\ln 10^{100})/2 = 115$ numbers will be tested before a prime is found.

(About 665 bits, 2048 or 4096 are standard today)



To test a large number b for primality we recommend the elegant "probabilistic" algorithm due to Solovay and Strassen [12]. It picks a random number a from a uniform distribution on $\{1, \ldots, b-1\}$, and tests whether

$$\gcd(a,b) = 1 \text{ and } J(a,b) = a^{(b-1)/2} \pmod{b}, \tag{6}$$



To gain additional protection against sophisticated factoring algorithms, p and q should differ in length by a few digits, both $(p-1)$ and $(q-1)$ should contain large prime factors, and $\gcd(p-1, q-1)$ should be small. The latter condition is easily checked.

To find a prime number p such that $(p-1)$ has a large prime factor, generate a large random prime number u, then let p be the first prime in the sequence $i \cdot u + 1$, for $i = 2, 4, 6, \ldots$. (This shouldn't take too long.) Additional security is provided by ensuring that $(u-1)$ also has a large prime factor.

C How to Choose d

It is very easy to choose a number d which is relatively prime to $\phi(n)$. For example, any prime number greater than $\max(p, q)$ will do. It is important that d should be chosen from a large enough set so that a cryptanalyst cannot find it by direct search.

D How to Compute e from d and $\phi(n)$

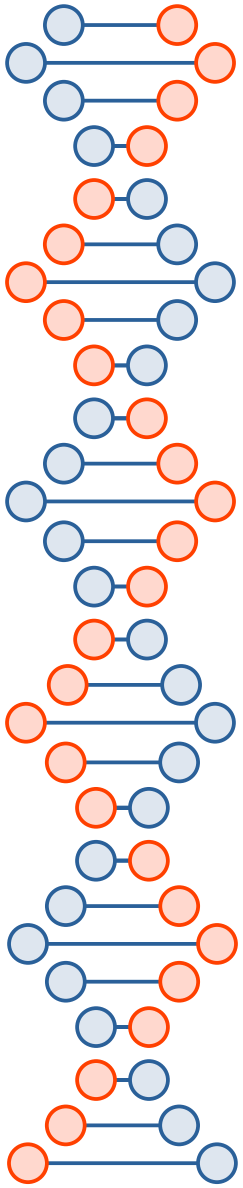
Euclid's algorithm

If e turns out to be less than $\log_2(n)$, start over by choosing another value of d. This guarantees that every encrypted message (except $M = 0$ or $M = 1$) undergoes some “wrap-around” (reduction modulo n) .

The era of "electronic mail" [10] may soon be upon us;

encryption keys. (We assume that the intruder cannot modify or insert messages into the channel.) Ralph Merkle has developed another solution [5] to this problem.

A public-key cryptosystem can be used to "bootstrap" into a standard encryption scheme such as the NBS method. Once secure communications have been established, the first message transmitted can be a key to use in the NBS scheme to encode all following messages. This may be desirable if encryption with our method is slower than with the standard scheme. (The NBS scheme is probably somewhat faster if special-purpose hardware encryption devices are used; our scheme may be faster on a general-purpose computer since multiprecision arithmetic operations are simpler to implement than complicated bit manipulations.)



200-digit message M

$\log_2(10^{200})$ = about 665 bits

Since no techniques exist to *prove* that an encryption scheme is secure, the only test available is to see whether anyone can think of a way to break it. The NBS standard was "certified" this way; seventeen man-years at IBM were spent fruitlessly trying to break that scheme. Once a method has successfully resisted such a concerted attack it may for practical purposes be considered secure. (Actually there is some controversy concerning the security of the NBS method [2].)

How can n be factored using $\phi(n)$? First, $(p+q)$ is obtained from n and $\phi(n) = n - (p+q) + 1$. Then $(p-q)$ is the square root of $(p+q)^2 - 4n$. Finally, q is half the difference of $(p+q)$ and $(p-q)$.

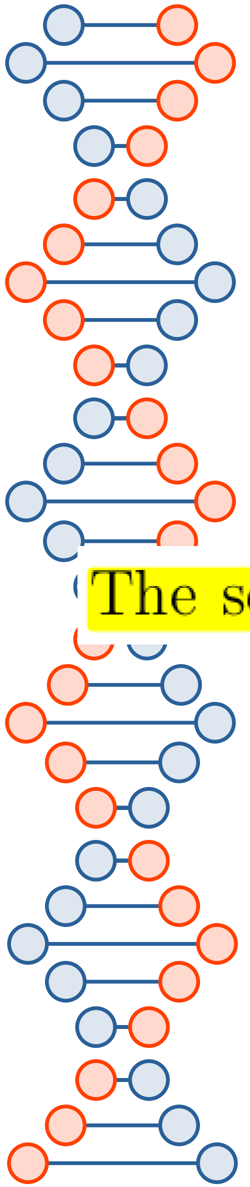
Therefore breaking our system by computing $\phi(n)$ is no easier than breaking our system by factoring n. (This is why n must be composite; $\phi(n)$ is trivial to compute if n is prime.)

A knowledge of d enables n to be factored as follows. Once a cryptanalyst knows d he can calculate $e \cdot d - 1$, which is a multiple of $\phi(n)$. Miller [6] has shown that n can be factored using any multiple of $\phi(n)$. Therefore if n is large a cryptanalyst should not be able to determine d any easier than he can factor n.

D Computing D in Some Other Way

Although this problem of “computing e-th roots modulo n without factoring n” is not a well-known difficult problem like factoring, we feel reasonably confident that it is computationally intractable. It may be possible to prove that any general method of breaking our scheme yields an efficient factoring algorithm. This would establish that any way of breaking our scheme must be as difficult as factoring. We have not been able to prove this conjecture, however.

Our method should be certified by having the above conjecture of intractability withstand a concerted attempt to disprove it. The reader is challenged to find a way to “break” our method.



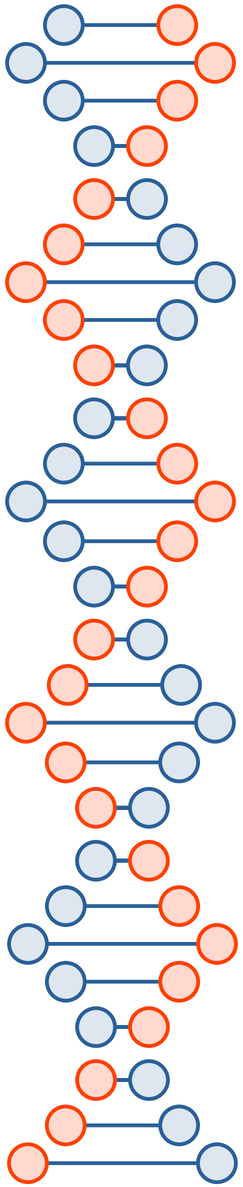
The security of this system needs to be examined in more detail.

Side notes

- GCHQ claims to have invented RSA in 1973, and declassified this info in 1997
- In my own research (*e.g.*, looking for amateurish crypto in Android apps) using RSA for key distribution is often a red flag
  - An authenticated version of Diffie-Hellman is better, most common thing these days is ECDH (Elliptic Curve Diffie-Hellman)

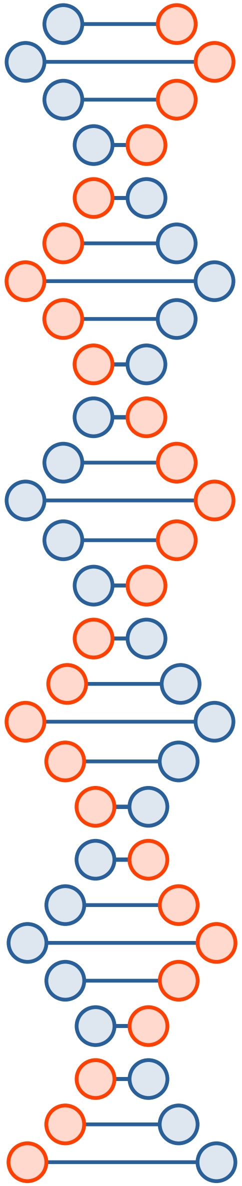
RSA in real cryptosystems

- What we just learned, and read about in the paper, is called “Textbook RSA”
  - Not secure and should not be used (padding is strictly necessary in real schemes)
  - Padding oracle attacks (same idea as for CBC)
- Side channels



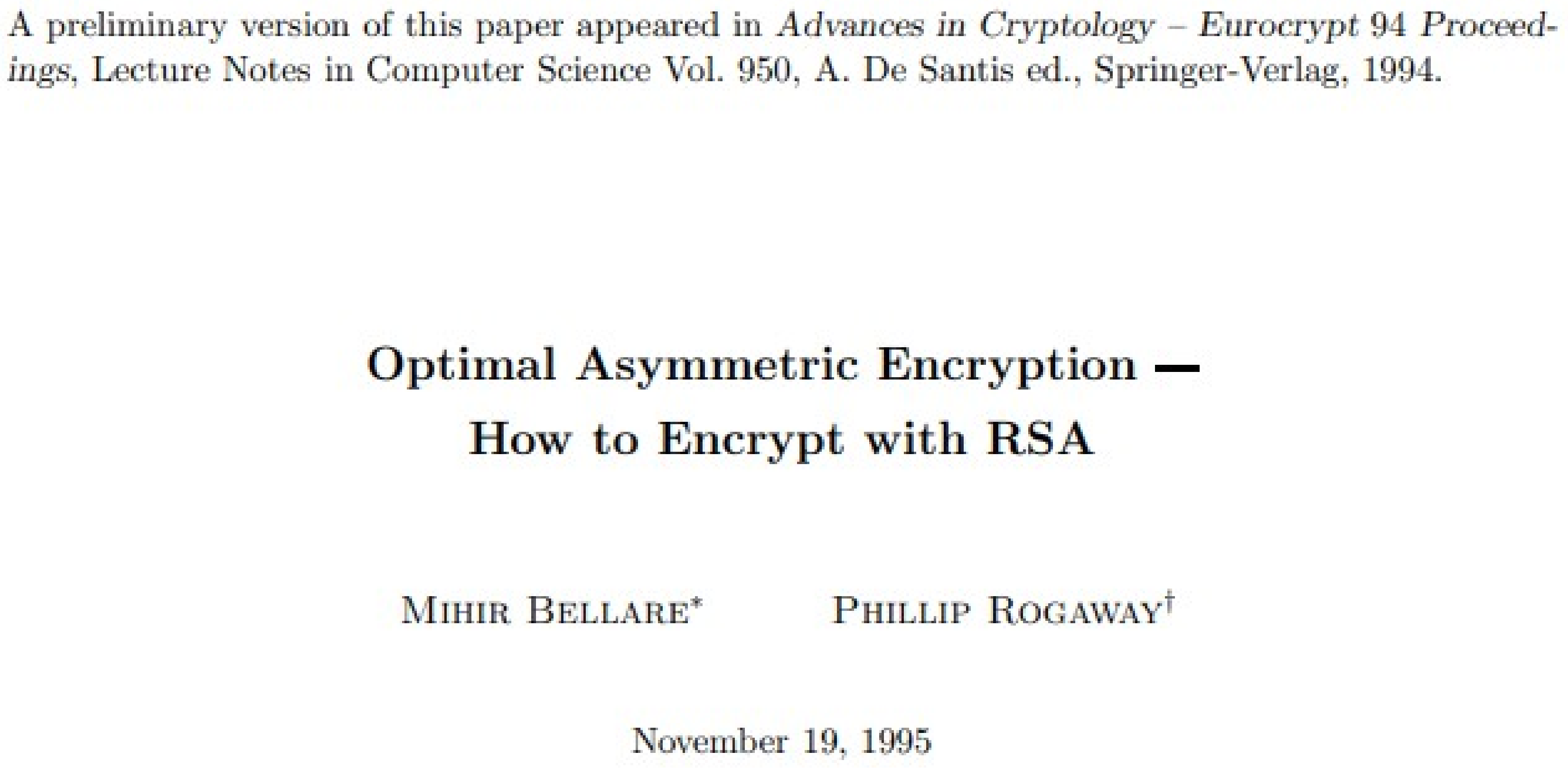
Symmetric attack types

- Ciphertext only
  - Think Caesar cipher, or Viginere cipher
- Known plaintext
  - Linear cryptanalysis
- Chosen plaintext
  - Differential cryptanalysis



Asymmetric notions of semantic security

- Now threat models are very complicated, but in a nutshell:
  - IND-CPA – Indistinguishability under chosen plaintext attack
  - IND-CCA – Indistinguishability under chosen ciphertext attack
  - IND-CCA2 – Indistinguishability under chose ciphertext attack (adaptive)



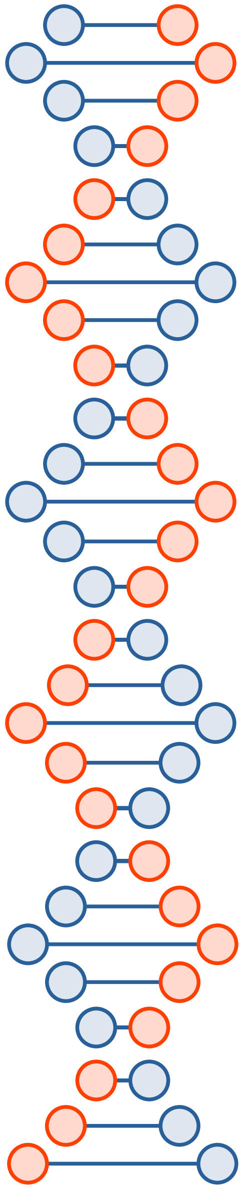
A preliminary version of this paper appeared in *Advances in Cryptology – Eurocrypt 94 Proceedings*, Lecture Notes in Computer Science Vol. 950, A. De Santis ed., Springer-Verlag, 1994.

Optimal Asymmetric Encryption — How to Encrypt with RSA

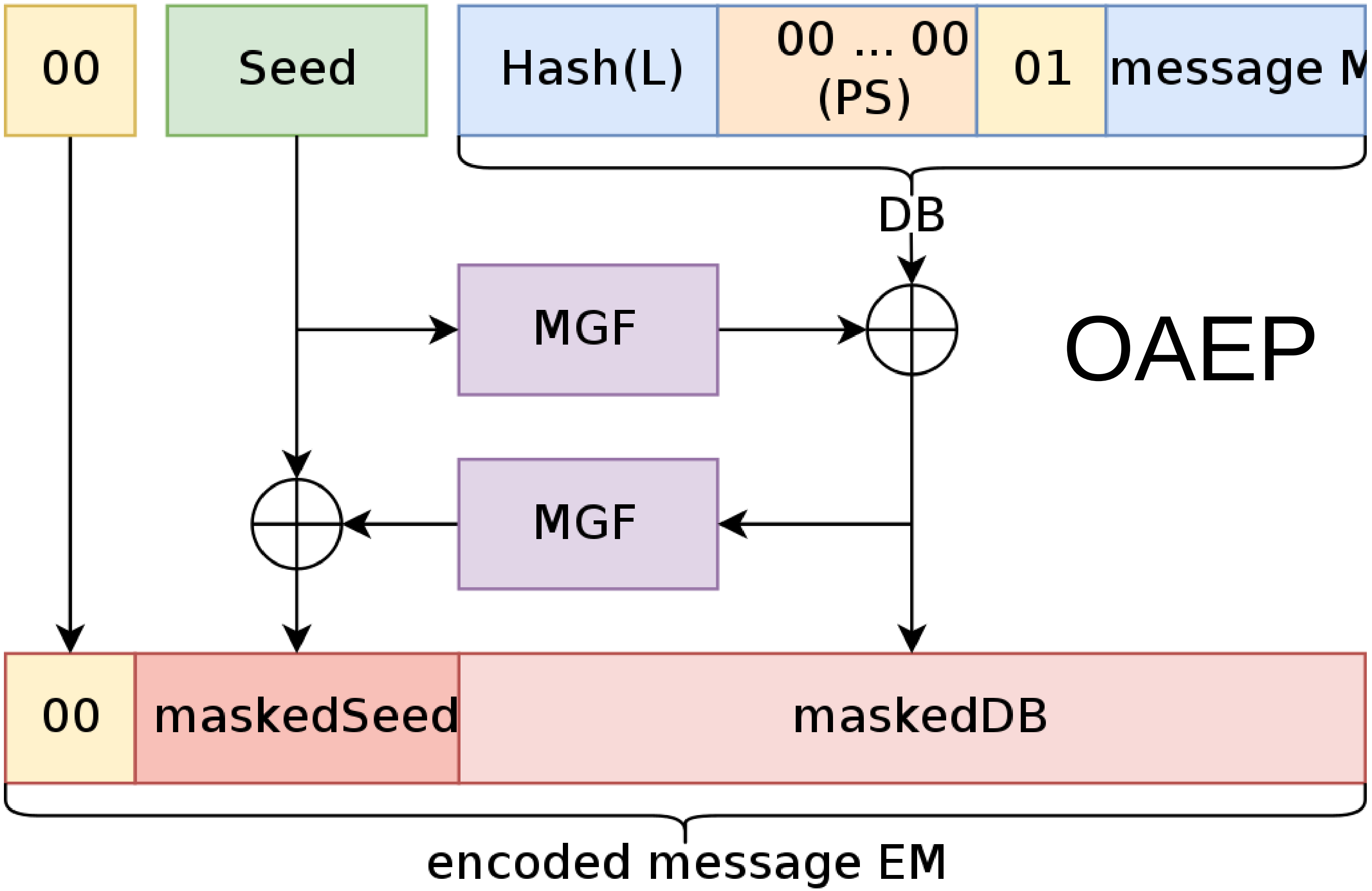
Mihir Bellare[*] Phillip Rogaway[†]

November 19, 1995

https://cseweb.ucsd.edu//~mihir/papers/oaep.pdf



OAEP



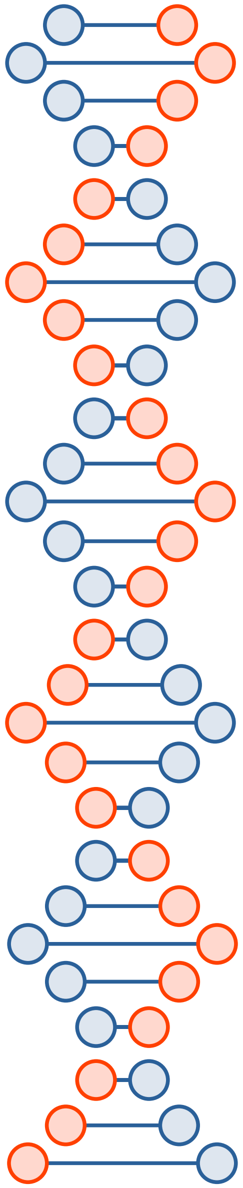
https://en.wikipedia.org/wiki/File:OAEP_encoding_schema.svg

IND-CCA2 in a nutshell

- I’ll encrypt or decrypt as many plaintexts or ciphertexts as you like
  - plaintext/ciphertext pairs
- You give me two plaintexts, I’ll flip a coin (heads or tails) and encrypt one of them (you don’t know which) to give you C
- In polynomial time, you can do more encryption and decryption, just not for C
- You guess my coin flip (heads or tails)

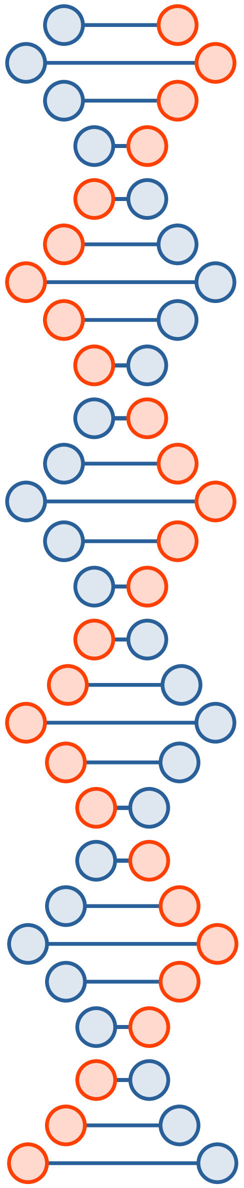
If you can’t win with >50% probability

- You can’t break my scheme (*e.g.*, OAEP) with an adaptive chosen ciphertext attack



If you **can** win with >50% probability

- You’ve potentially broken my scheme with an adaptive chosen ciphertext attack
- Let’s win the Turing award together, by publishing a paper showing how to factor large integers with a classical computer in polynomial time
    - Or, build a cybercrime cartel together?



Okay to grab the RSA paper and start coding?
Or just use a textbook, *i.e.*, textbook RSA?

Let C be the RSA encryption of 128-bit AES key k with RSA public key (n, e). Thus, we have

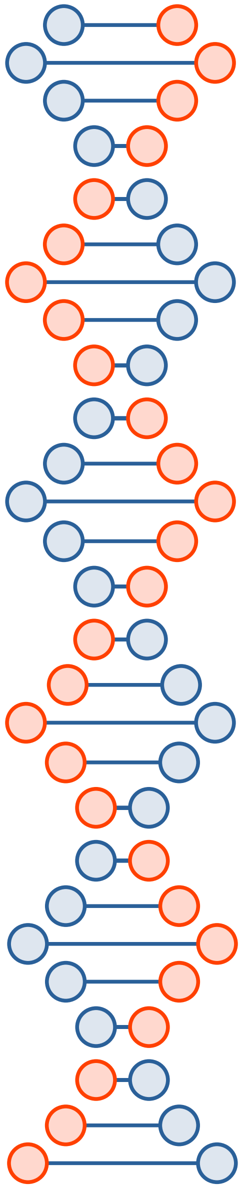
$$C \equiv k^e \pmod{n}$$

Now let C_b be the RSA encryption of the AES key

$$k_b = 2^b k$$

i.e., k bitshifted to the left by b bits. Thus, we have

$$C_b \equiv {k_b}^e \pmod{n}$$



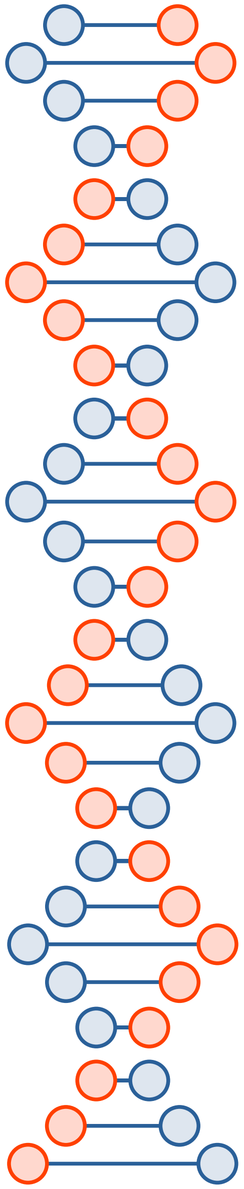
$$C_b \equiv {k_b}^e \pmod{n}$$

We can compute C_b from only C and the public key, as

$$\begin{aligned}
C_b &\equiv C(2^{be} \bmod n) \pmod{n} \\
&\equiv (k^e \bmod n)(2^{be} \bmod n) \pmod{n} \\
&\equiv k^e 2^{be} \pmod{n} \\
&\equiv (2^b k)^e \pmod{n} \\
&\equiv {k_b}^e \pmod{n}
\end{aligned}$$

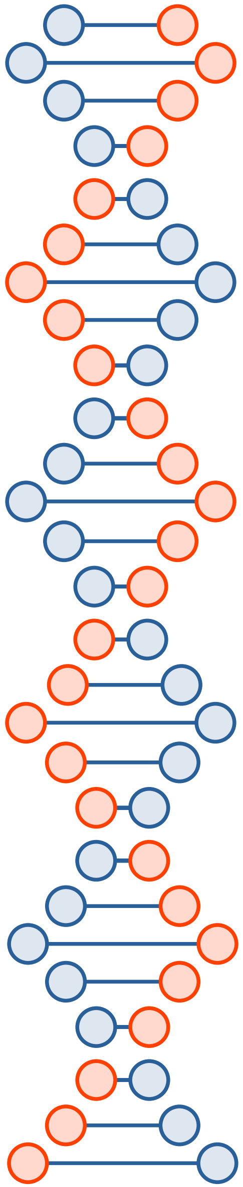
Suppose the client whose communications we want to decrypt encrypts the following 128-bit AES key with 1024-bit RSA and sends it to the server:

```
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

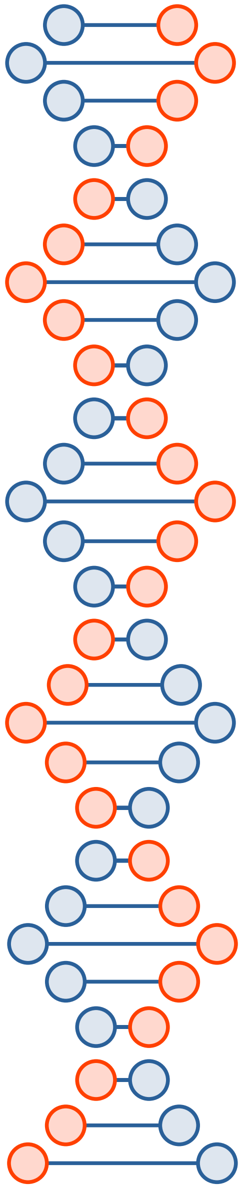


```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```

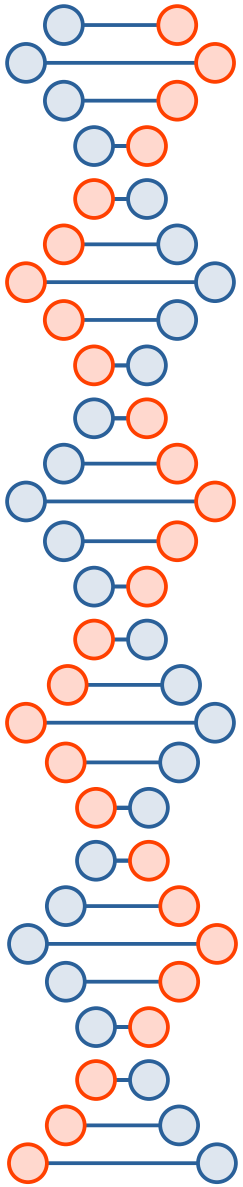
This shows what the server sees as the RSA plaintext ***after*** **it decrypts the ciphertext we sent it,** which we are trying to trick the server into leaking bits of to us. So, we as the attacker have recorded c by eavesdropping on the client and server's communications over the Internet. But without the private key, d, we don't know what m (the green part, which is the AES key to decrypt the rest of the message) is. Let's explore what happens if we open our own connection to the server, and as our ciphertext we send $c \times 2^e$. The server will decrypt that into the following plaintext:



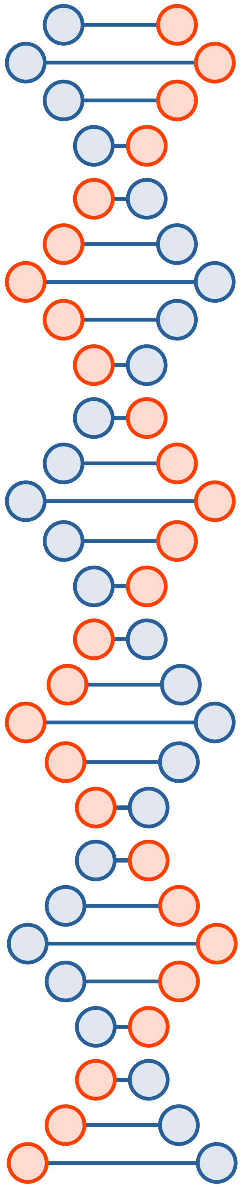
```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001
0110000100101100111011110111011001000101111110011101010111100110
0000000111001001011110010010101001000111011010100001011101110110
```



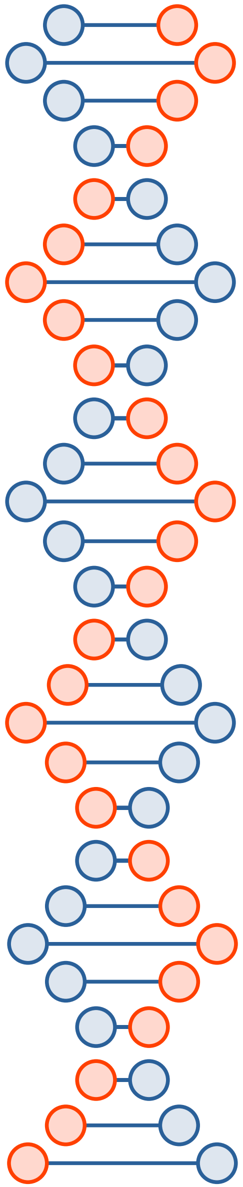
So, we know how to double plaintexts by manipulating ciphertexts. What if we double it more than once? What if we do it 16 times, by sending $c \times 2^{16e}$ as out ciphertext. When we multiplied the plaintext by 2 above, we effectively bit shifted the AES key by 1. Now we're multiplying the plaintext by 2^{16}, which is the equivalent of bit-shifting to the left 16 times:



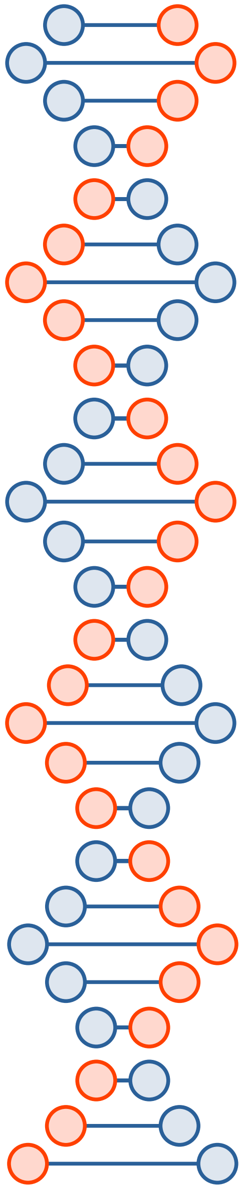
```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000001011000010010110
0111011110111011001000101111110011101010111100110000000011100100
1011110010010101001000111011010100001011101110110000000000000000
```



$$c \times 2^{127e}$$



```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0101100001001011001110111101110110010001011111100111010101111001
1000000001110010010111100100101010010001110110101000010111011101
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

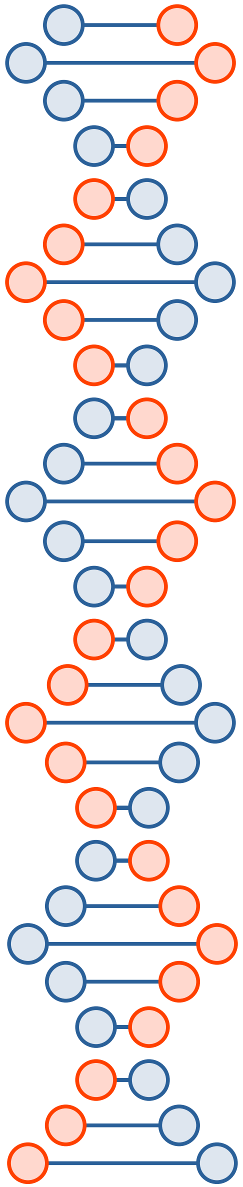


Now, we've finally figured out **step one** of our attack. By sending $c \times 2^{127e}$ as our ciphertext, we've forced the server to encrypt what it sends back to us with one of two AES keys, either...

$$\boxed{\begin{array}{l}0000000000000000000000000000000000000000000000000000000000000000\\0000000000000000000000000000000000000000000000000000000000000000\end{array}}$$

...or...

$$\boxed{\begin{array}{l}1000000000000000000000000000000000000000000000000000000000000000\\0000000000000000000000000000000000000000000000000000000000000000\end{array}}$$



```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0010110000100101100111011110111011001000101111110011101010111100
1100000000111001001011110010010101001000111011010100001011101110
1100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

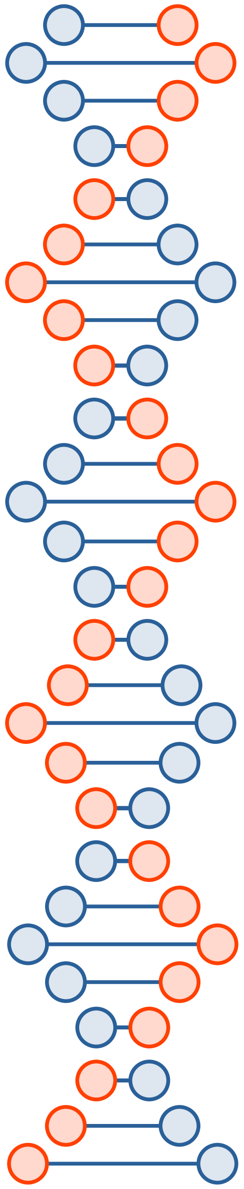
So now there are two bits in our AES key with the server that we don't know for sure are zeroes, but we don't want to try all four possibilities because that's not going to be efficient going forward. We already know the bit in blue from step 1, so really there are only two possibilities of keys to try: Either...

```
0100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

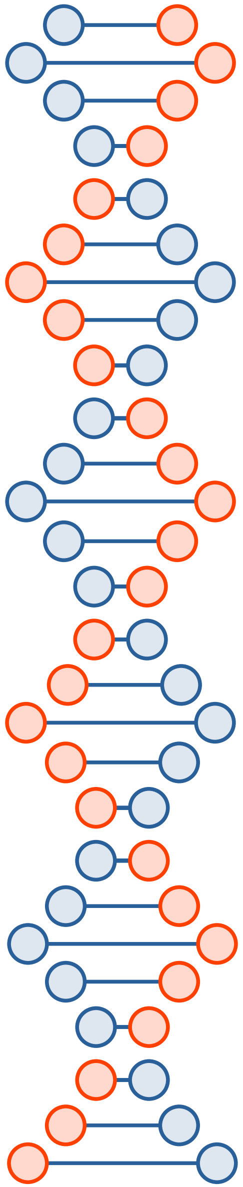
...or...

```
1100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

So then, for step 3, we would send $c \times 2^{125e}$ as our ciphertext, and the server would decrypt:



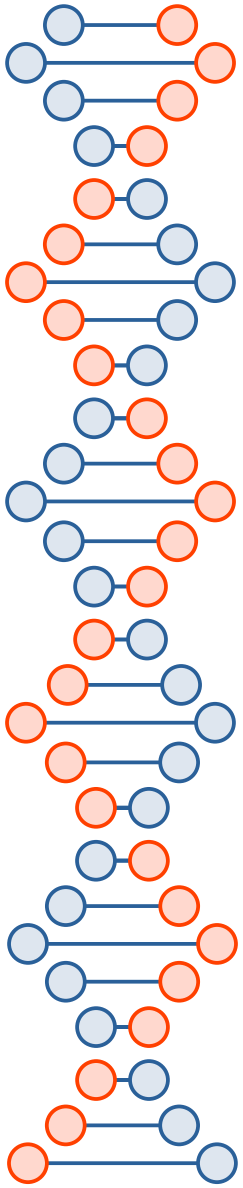
```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0001011000010010110011101111011101100100010111111001110101011110
0110000000011100100101111001001010100100011101101010000101110111
0110000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```



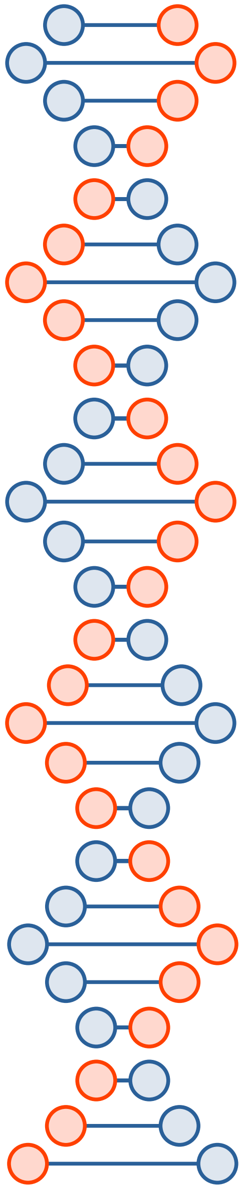
...or...

```
0110000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

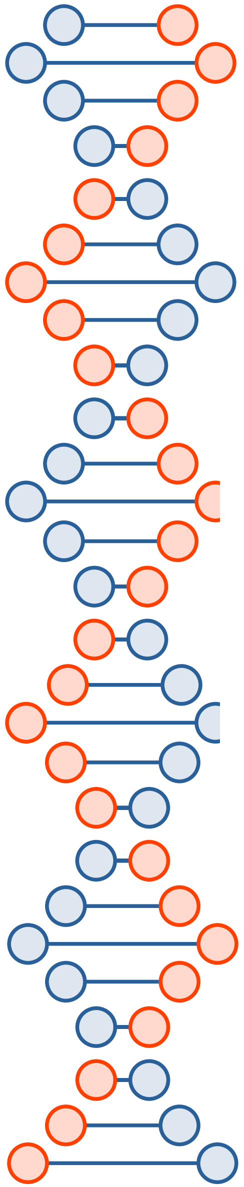
```
1110000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```



By the 128th step...



```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```



...or...

0011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011

1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011

WUP requests

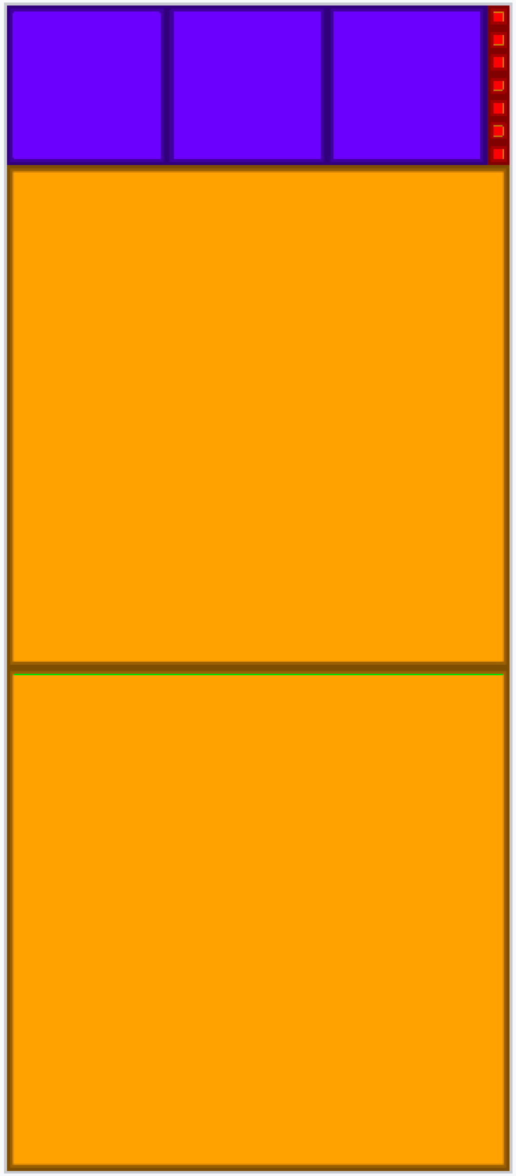
- Full attack is at: https://arxiv.org/pdf/1802.03367.pdf

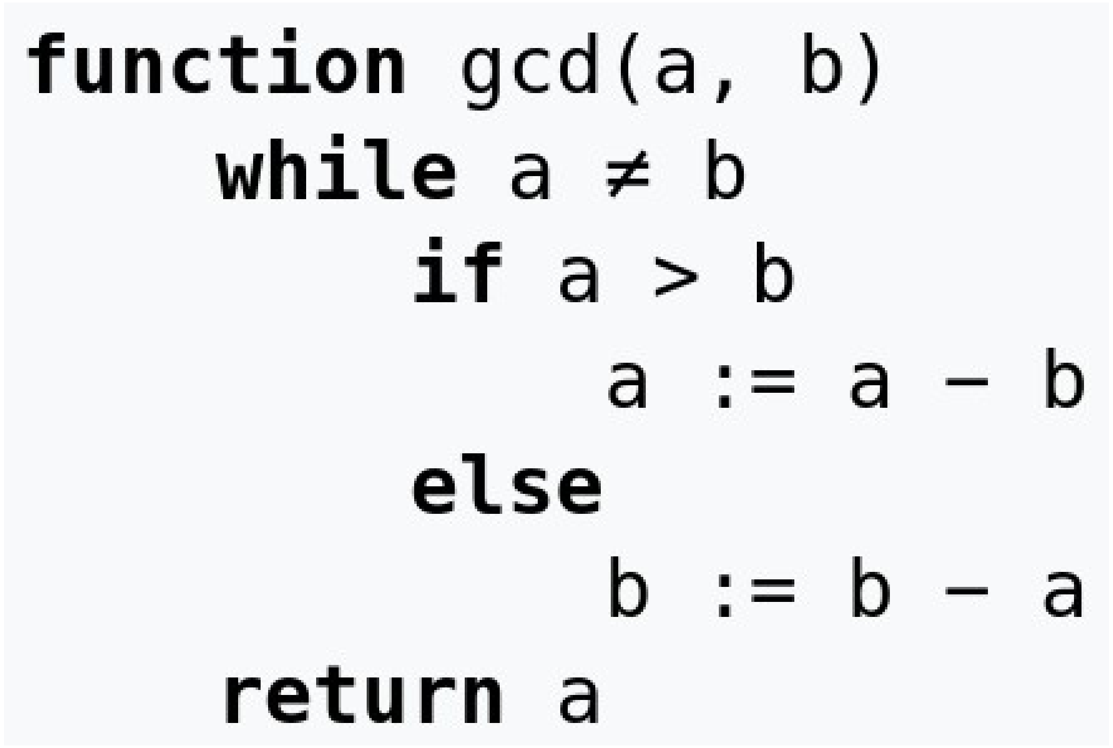
Euclidean Algorithm

For gcd (greatest common divisor)

- https://en.wikipedia.org/wiki/Euclidean_algorithm

Subtraction-based animation of the Euclidean algorithm. The initial rectangle has dimensions *a* = 1071 and *b* = 462. Squares of size 462×462 are placed within it leaving a 462×147 rectangle. This rectangle is tiled with 147×147 squares until a 21×147 rectangle is left, which in turn is tiled with 21×21 squares, leaving no uncovered area. The smallest square size, 21, is the GCD of 1071 and 462.





```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b
        else
            b := b − a
    return a
```

```
function gcd(a, b)
    if b = 0
        return a
    else
        return gcd(b, a mod b)
```

Extended Euclidean Algorithm

- https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
- https://crypto.stackexchange.com/questions/5889/calculating-rsa-private-exponent-when-given-public-exponent-and-the-modulus-fact

Your goal is to find d such that $ed \equiv 1 \pmod{\varphi(n)}$.

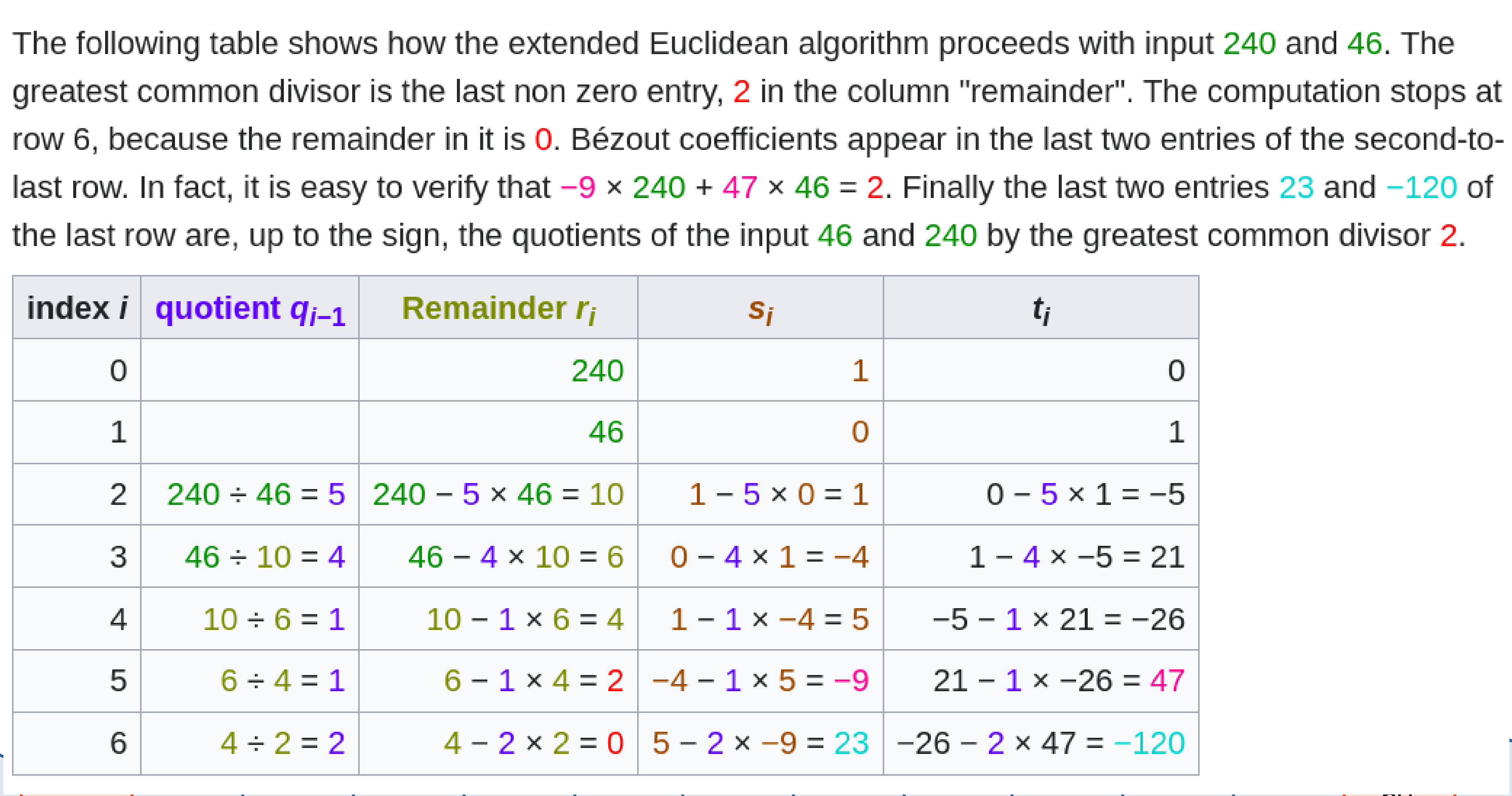
Recall the EED calculates x and y such that $ax + by = \gcd(a, b)$. Now let $a = e$, $b = \varphi(n)$, and thus $\gcd(e, \varphi(n)) = 1$ by definition (they need to be coprime for the inverse to exist). Then you have:

$$ex + \varphi(n)y = 1$$

Take this modulo $\varphi(n)$, and you get:

$$ex \equiv 1 \pmod{\varphi(n)}$$

And it's easy to see that in this case, $x = d$. The value of y does not actually matter, since it will get eliminated modulo $\varphi(n)$ regardless of its value. The EED will give you that value, but you can safely discard it.



The following table shows how the extended Euclidean algorithm proceeds with input 240 and 46. The greatest common divisor is the last non zero entry, 2 in the column "remainder". The computation stops at row 6, because the remainder in it is 0. Bézout coefficients appear in the last two entries of the second-to-last row. In fact, it is easy to verify that −9 × 240 + 47 × 46 = 2. Finally the last two entries 23 and −120 of the last row are, up to the sign, the quotients of the input 46 and 240 by the greatest common divisor 2.

index i	quotient q_{i-1}	Remainder r_i	s_i	t_i
0		240	1	0
1		46	0	1
2	240 ÷ 46 = 5	240 − 5 × 46 = 10	1 − 5 × 0 = 1	0 − 5 × 1 = −5
3	46 ÷ 10 = 4	46 − 4 × 10 = 6	0 − 4 × 1 = −4	1 − 4 × −5 = 21
4	10 ÷ 6 = 1	10 − 1 × 6 = 4	1 − 1 × −4 = 5	−5 − 1 × 21 = −26
5	6 ÷ 4 = 1	6 − 1 × 4 = 2	−4 − 1 × 5 = −9	21 − 1 × −26 = 47
6	4 ÷ 2 = 2	4 − 2 × 2 = 0	5 − 2 × −9 = 23	−26 − 2 × 47 = −120

```
function extended_gcd(a, b)
    (old_r, r) := (a, b)
    (old_s, s) := (1, 0)
    (old_t, t) := (0, 1)

    while r ≠ 0 do
        quotient := old_r div r
        (old_r, r) := (r, old_r − quotient × r)
        (old_s, s) := (s, old_s − quotient × s)
        (old_t, t) := (t, old_t − quotient × t)

    output "Bézout coefficients:", (old_s, old_t)
    output "greatest common divisor:", old_r
    output "quotients by the gcd:", (t, s)
```

See also…

- Fermat’s Little Theorem

Takeaways

- Generally, DH for key exchange and RSA for signatures
  - Alternatives, such as elliptic curves
- Symmetric cyrpto for the actual encryption
- Semantic security is the gold standard for asymetric
  - Reduction proofs
- “Textbook RSA,” like you’ll do on the exams, is dangerous