Computational Mathematics: Handout 05

Curtis Bright

September 28, 2022

1 Cryptography and Coding Theory

We now discuss how modular arithmetic is very useful in the fields of coding theory and cryptography.

Coding theory deals with how to send a message over a potentially "noisy" channel. For example, imagine sending a spaceprobe to Mars. You might communicate with such a probe using radio waves but when the radio waves are sent through space they will likely be corrupted somehow by the time they are recieved. How can you guard against this? When you recieve a message you would like to know if the message has been corrupted, and if so, ideally what the original message was. The development of "error-correcting codes" makes this dream a reality!

Cryptography deals with how to send messages so that they can only be read by their intended recipient. Even if you cannot guarantee the communication channel is secure cryptography allows two parties the ability to communicate secretly. For example, your Internet Service Provider (ISP) can (and sometimes does!) look at the content of webpages that you visit. In fact, there have been cases of ISPs inserting ads into pages. Cryptographic protocols prevent them from doing this and are a part of all modern Internet browsers. This way your ISP cannot read your credit card number when you buy something using it online.

At first these applications seem almost miraculous, but we'll see how with modular arithmetic they are possible!

1.1 Check Digit Schemes

First, we will see an example of a scheme that will allow the recipient to *detect* (but not correct) errors in the transmission. Universal Product Codes (UPC) that are used to track items in stores. They are instantly recognizable since they are printed on virtually every kind of product:

What do the numbers mean? The first digit denotes the encoding systemed used, the next 5 digits typically denote the manufacturer code and the 5 digits after that typically denote the product code. The last digit is the check digit which can detect if an error is present in the code. Assuming the digits are labelled $a_1, \ldots, a_{12}$ they must satisfy the *check digit equation*

$$3a_1 + 1a_2 + 3a_3 + 1a_4 + \cdots + 3a_{11} + 1a_{12} \equiv 0 \pmod{10}.$$

Once $a_1, \ldots, a_{11}$ are known, a_{12} may be computed by

$$a_{12} := -(3a_1 + 1a_2 + 3a_3 + 1a_4 + \cdots + 3a_{11}) \bmod 10.$$

For example, in the above UPC 03600029145x (with x denoting the check digit) we have

$$x = -3(0+6+0+2+1+5) - 1(3+0+0+9+4) \bmod 10 = -58 \bmod 10 = 2.$$

A similar scheme is used in 10-digit ISBNs (International Standard Book Numbers):

The check digit equation is

$$\sum_{i=1}^{10} i a_i \equiv 0 \pmod{11}$$

and the check digit a_{10} can be computed by $a_{10} := \sum_{i=1}^{9} i a_i \bmod 11$ (since $10 \equiv -1 \pmod{11}$).

For example, the check digit x in the above ISBN $076452641x$ must be

$$x = (1 \cdot 0 + 2 \cdot 7 + 3 \cdot 6 + 4 \cdot 4 + 5 \cdot 5 + 6 \cdot 2 + 7 \cdot 6 + 8 \cdot 4 + 9 \cdot 1) \bmod 11 = 168 \bmod 11 = 3.$$

The check digit in this case might be 10, it which case the letter `X` is used as the check digit (the roman numeral for 10).

1.2 Error Correction Scheme

Now we will see a simple scheme that can detect and *correct* up to one error.

Let's say we want to send an alphabetic string such as CODEWORD. By representing each character as an integer between 1 and 26 we want to send the sequence $[3, 15, 4, 5, 23, 15, 18, 4]$. We will perform computations modulo 29 (the first prime larger than 26) and prepend the sequence with two additional integers (say a_0 and a_1) so that the entire sequence $a_0, \ldots, a_9$ satisfies the two congruences

$$\sum_{i=0}^{9} a_i \equiv 0 \pmod{29} \qquad \sum_{i=0}^{9} i a_i \equiv 0 \pmod{29}.$$

Intuitively, a_0 is used to detect the presence and magnitude of an error and a_1 is used to detect the position of an error when an error exists. They may be computed via

$$a_1 := \sum_{i=2}^{9} -i a_i \bmod 29 \qquad a_0 := \sum_{i=1}^{9} -a_i \bmod 29.$$

1.2.1 Example in Sage

[1]:
```
# Characters in word to encode represented as integers
A = [0,0,3,15,4,5,23,15,18,4]
A[1] = sum(-i*A[i] for i in (2..9)) % 29
A[0] = sum(-A[i] for i in (1..9)) % 29
# Sequence with check digits prepended
print(A)
```

```
[22, 7, 3, 15, 4, 5, 23, 15, 18, 4]
```

1.2.2 Example of error correcting

What happens if you receive a corrupted message? This scheme can correct up to one error. For example, let's say that $[22, 7, 3, 15, 4, 5, 23, 1, 18, 4]$ was recieved (literally decoding to CODEWARD).

[2]:
```
A = [22,7,3,15,4,5,23,1,18,4]
# Both computations should produce zero if the word was sent correctly
print(sum(A[i] for i in (0..9)) % 29)
print(sum(i*A[i] for i in (0..9)) % 29)
```

```
15
18
```

Because these computations are nonzero there must be at least one error; assume there is exactly one.

The first check digit computation gives the *magnitude* of the error. Let $e := \sum_{i=0}^{9} a_i$. Then some integer a_k in the sequence should be be replaced with $a_k - e$ in order for the first check digit equation to hold.

The second check digit computation can be used to determine the *position* of the error (i.e., k from above). Say that c is the "correct" value of the entry at index k and this was replaced with $a_k = c + e$ during transmission. Then

$$\sum_{i=1}^{9} ia_i - ke \equiv 0 \pmod{29}$$

since subtracting off e from the recieved (incorrect) a_k will cancel the error in the second check digit computation. Now we can rearrange this to solve for k

$$k \equiv e^{-1} \sum_{i=1}^{9} ia_i \pmod{29}$$

(using the extended Euclidean algorithm to compute $e^{-1} \bmod 29$). Finally, once we know k we can replace a_k with $c = a_k - e$.

[3]:
```
# Recieved message
A = [22,7,3,15,4,5,23,1,18,4]
# Compute error magnitude
e = sum(A[i] for i in (0..9)) % 29
# Compute error position
k = e^(-1)*sum(i*A[i] for i in (1..9)) % 29
print("Error in position {}".format(k))

# Correct message
A[k] = (A[k]-e) % 29
# Convert message to ASCII string and print it
print("".join([chr(A[i]+64) for i in (2..9)]))
```

```
Error in position 7
CODEWORD
```

1.3 Diffie–Hellman Key Exchange

Can two parties agree on a shared secret over a public channel? At first one might expect that this is impossible, but in fact cryptographers Whitfield Diffie and Martin Hellman devised such a scheme in 1976. (Documents declassified by the British government in 1997 revealed that Malcolm Williamson of the British intelligence agency GCHQ had discovered the same method in 1974.)

First, let's start with a puzzle from Caroline Calderbank:

Alice wishes to mail Bob a ring. Unfortunately, anything sent through the mail will be stolen unless it is enclosed in a padlocked box. Alice and Bob each have plenty of padlocks, but none to which the other has a key. How can Alice get the ring safely into Bob's hands?

1.3.1 One possible solution

Here is the solution Caroline had in mind in four steps:

1. Alice sends Bob a box with the ring in it and one of her padlocks on it.
2. When Bob receives the box he affixes his own padlock to the box and mails it back to Alice with both padlocks on it.
3. When Alice gets it she removes her padlock and sends the box back to Bob.
4. When Bob receives the box again he can remove his padlock and then open the box!

What does this have to do with cryptography? Surprisingly enough, the idea in this solution is the same idea that makes the Diffie–Hellman key exchange work.

1.3.2 Primitive roots mod p

A number g is a *primitive root* modulo p if the smallest solution of

$$g^x \equiv 1 \pmod{p} \qquad (x > 0) \tag{1}$$

is $x = p - 1$. Note that g must be coprime to p for solutions of (1) to exist. Fermat's little theorem implies that $x = p - 1$ is *always* a solution for any g with $\gcd(g, p) = 1$. When $x = p - 1$ is the **smallest** solution then g is a primitive root.

For example, 5 is a primitive root modulo 23 since $5^x \bmod 23$ for $x = 1, \ldots, 22$ is

$$[5, 2, 10, 4, 20, 8, 17, 16, 11, 9, 22, 18, 21, 13, 19, 3, 15, 6, 7, 12, 14, \mathbf{1}].$$

Conversely, 3 is not a primitive root modulo 23 since $3^x \bmod 23$ for $x = 1, \ldots, 22$ is

$$[3, 9, 4, 12, 13, 16, 2, 6, 18, 8, \mathbf{1}, 3, 9, 4, 12, 13, 16, 2, 6, 18, 8, \mathbf{1}].$$

[4]:
```
show([5^x % 23 for x in (1..22)])
show([3^x % 23 for x in (1..22)])
```

$$[5, 2, 10, 4, 20, 8, 17, 16, 11, 9, 22, 18, 21, 13, 19, 3, 15, 6, 7, 12, 14, 1]$$

$$[3, 9, 4, 12, 13, 16, 2, 6, 18, 8, 1, 3, 9, 4, 12, 13, 16, 2, 6, 18, 8, 1]$$

Sage provides the built-in function `primitive_root` which computes a primitive root modulo a given modulus m.

[5]:
```
primitive_root(23)
```

[5]: 5

1.3.3 Outline of Diffie–Hellman

Say Alice wants to send a secret to Bob over an insecure channel. The first step is to derive a *shared secret key* that only Alice and Bob know. They can do this as follows:

1. Alice chooses a prime p by some random process.
2. Alice chooses some g, a primitive root modulo p.
3. Alice chooses a random integer a and computes $A := g^a \bmod p$.
4. Alice sends p, g, and A to Bob, but keeps a secret.
5. Bob receives p, g, and A, chooses a random integer b and computes $B := g^b \bmod p$.
6. Bob sends B to Alice but keeps b secret.
7. Alice computes $B^a \bmod p$ and Bob computes $A^b \bmod p$. This is their shared secret number.

1.3.4 Example

1. Alice chooses the prime $p = 31337$.
2. Alice chooses $g = 3$ which is a primitive root modulo p.
3. Alice chooses the random intger $a = 11589$ and computes $A = 3^{11589} \bmod p = 14479$.
4. Alice sends p, g and A to Bob.
5. Bob chooses the random integer $b = 31239$ and computes $B = 3^{31239} \bmod p = 2879$.
6. Bob sends B to Alice.
7. Alice computes $2879^{11589} \bmod p = 27390$ and Bob computes $14479^{31239} \bmod p = 27390$.

1.3.5 Example in Sage

[6]:
```
p = 31337
g = primitive_root(p)
print("p = {}, g = {}".format(p, g))

# Alice generates a random integer between 0 and p-1
a = ZZ.random_element(p)
# Bob generates a random integer between 0 and p-1
b = ZZ.random_element(p)
print("a = {}, b = {}".format(a, b))

A = g^a % p # Alice computes A and sends it to Bob
B = g^b % p # Bob computes B and sends it to Alice
print("A = {}, B = {}".format(A, B))

B^a % p # Alice computes B^a mod p
A^b % p # Bob computes A^b mod p
print("B^a = {} mod p and A^b = {} mod p".format(B^a % p, A^b % p, p))
```

```
p = 31337, g = 3
a = 15181, b = 28095
A = 17025, B = 7570
B^a = 30228 mod p and A^b = 30228 mod p
```

1.3.6 Why does this work?

First, note that B^a and A^b produce the same number modulo p because

$$B^a \equiv (g^b)^a \equiv g^{ba} \equiv g^{ab} \equiv (g^a)^b \equiv A^b \pmod{p}.$$

Also note that an evesdropper can learn p, g, A, and B, but does not know a or b. In order to compute these they would have to solve one of the following congruences for a or b:

$$g^a \equiv A \pmod{p} \qquad\qquad g^b \equiv B \pmod{p}$$

Solving such an equation is known as the *discrete logarithm problem* because it is equivalent to computing a base-g logarithm modulo p. In other words, the congruences can be rewritten as

$$\log_g(A) \equiv a \pmod{p} \qquad\qquad \log_g(B) \equiv b \pmod{p}.$$

Perhaps surprisingly, computing discrete logarithms is a famously difficult problem. This is in stark contrast to computing logarithms such as $\log_2(x)$ or $\ln(x)$ over the real numbers which can be done efficiently.

1.3.7 Takeaway

Diffie–Hellman relies on the property that the modular exponentiation $g^x \bmod p$ can be done efficently (via repeated squaring) but no one knows any method of computing the modular logarithm $\log_g(x) \bmod p$ efficiently (i.e., polynomial in $\text{len}(p)$) even when g is small, e.g., $g = 3$.

For example, suppose you choose a 1024-bit prime p (which can fit in sixteen 64-bit words). It is estimated that computing $\log_g(x) \bmod p$ requires on the order of 100 million dollars in computing power.

While this is quite expensive, such a computation could actually be afforded by large governments—so if you *really* want your secrets to be safe, choose a 2048-bit prime instead (which can fit in thirty-two 64-bit words). It's estimated that computing $\log_g(x) \bmod p$ would now be a billion times harder, i.e., would cost 100 trillion dollars! Conversely, computing $g^x \bmod p$ is only slightly more challenging.

1.3.8 Application to communication

Note that Diffie–Hellman only enables two parties to agree on the same shared secret. At first this might seem useless for communication purposes, since neither party can control what the shared secret is. However, we will now see a way that a shared secret can enable secure communication over an insecure channel.

First, we will describe a simple method that only allows sharing a short message (shorter than the shared secret) but demonstrates the basic principle.

Suppose Alice and Bob have agreed on the shared secret in the example above (27390) and Alice wishes to send Bob the message `CAT`. First, this string can be converted to a number by treating it as a base-27 number with `A=1`, `B=2`, `C=3`, etc. Then Alice wishes to send the number $3 \cdot 27^2 + 1 \cdot 27 + 20 = 2234$.

Now Alice writes the shared secret and the message to send in binary: 27390 is `110101011111110` and 2234 is `000100010111010`. She then adds together the digits in the same position modulo 2:

$$\begin{array}{r} 110101011111110 \\ +\ 000100010111010 \\ \hline 110001001000100 \end{array}$$

The result is `110001001000100` in binary which is 25156 in decimal. This number is sent to Bob who can add the shared secret to the number in a similar manner in order to recover the original message. This works because $0 + 0 \equiv 1 + 1 \equiv 0 \pmod 2$. In other words, adding the shared secret a second time will "cancel off" the secret and Bob will be left with the original message.

Conversely, an evesdropper who does not know the secret cannot perform this cancellation; there is no way of recovering the bits of the original message or shared secret from the encoded message. However, it only works for short messages, since the message cannot be longer than the shared secret without leaking information.

1.3.9 Stream ciphers

The above communication method only works for short messages but it can be augmented in order to allow longer messages.

The idea is to use a *stream cipher* which from a secret key will produce a continuous *keystream* of "pseudorandom" numbers. Since the keystream is deterministically generated from a secret key it is not truly random—however the numbers should "behave" as if they were random.

In particular, an adversary should not be able to determine the key from the keystream and should also not be able to predict future values of the keystream.

Then Alice and Bob can use Diffie–Hellman to agree on a shared secret key which they use to seed a stream cipher (which they also agree on). Then they use the same process as above (using addition mod 2 on the bits of the message) to encode and decode their messages. However, they add the numbers from the **keystream** to the message instead of the key itself.

Because stream ciphers can have very long periods (i.e., they take a long time to start repeating) this allows Alice and Bob communicate much longer messages using a single Diffie–Hellman key exchange.

1.4 RSA Cryptosystem

Diffie–Hellman requires both parties to work together in order to exchange a message over an insecure channel. What if Alice wants to send an encrypted message to Bob who is currently not available to reply?

One solution would be for Alice and Bob to set up a shared key in advance for communicating purposes, but this has some downsides:

1. Setting up keys in advance can require a lot of work. If there are n parties that each would like to communicate between each other then there are $\binom{n}{2}$ keys (i.e., a quadratic number of keys in n) that need to be exchanged in advance.
2. It may not be possible to set this up in advance, e.g., if Alice did not know Bob before wanting to contact him.

It might seem like an impossible problem for Alice to securely send a message to someone that she's never previously communicated with—and someone who does not respond back in any way. However, we'll now see that the RSA cryptosystem makes this possible.

1.4.1 Public-key cryptography

The idea behind public-key cryptography is to introduce an asymmetry between encoding and decoding. In symmetric encryption both encoding and decoding is done with the same shared key which has to be known by both parties that are communicating.

In public-key cryptography there are separate keys for encoding and decoding. The encoding key is known as the *public key* (because it can be publicly distributed) and the decoding key is known as the *private key*. The private key has to be kept secret since anyone who has access to it can decode encrypted messages.

Using such a scheme a group of n people only need to generate $2n$ keys in order to allow anyone to securely communicate with anyone else. Parties also do not have to communicate in advance; they can publish their public keys on a "key server" or by other means such as through their webpage.

To communicate a secure message to someone you only need to look up their public key and then encode your message using that key. Only the holder of the private key associated with that public key is able to decode the message.

Sounds great in theory, but how can it be implemented in practice?

1.4.2 Enter Rivest, Shamir, and Adleman

Diffie and Hellman proposed the idea of public-key cryptography in 1976 in addition to their key distribution method. However, they were unable to determine a method for realizing such a scheme.

In 1977, the three cryptographers Ronald Rivest, Adi Shamir, and Leonard Adleman tried many different methods for designing a public-key cryptosystem. Rivest recounts that sometimes they thought such a scheme was actually impossible to achieve.

In April 1977, Rivest, Shamir, and Adleman spent Passover at a student's house, apparently drinking Manischewitz wine and leaving around midnight (or so the legend goes). Rivest couldn't sleep and developed the basics of what would later be known as the RSA cryptosystem after the three who had worked on developing a working public-key cryptosystem.

Similar to the case of Diffie and Hellman's scheme, documents declassified by the British government in 1997 revealed that Clifford Cocks of the British intelligence agency GCHQ had discovered an equivalent system to the RSA method in 1973.

1.4.3 Outline of RSA

In order to realize public-key cryptography in practice we need a "one-way function"—a function that is easy to compute but hard to invert.

The Diffie–Hellman scheme used the discrete logarithm problem and the RSA cryptosystem will use the factorization problem (given a number compute its prime factors). It is easy to multiply

two prime numbers together but difficult to go in the other direction.

The intuition behind RSA is that computations will be performed modulo a number N that is the product of two prime numbers. Encoding a message x will be done by performing a modular exponentiation $x^e \pmod N$ for some specific integer e. Decoding a message will be done by computing the eth root of a number mod N which is equivalent to performing a modular exponentation $x^d \pmod N$ for some specific integer d.

In other words, encryption is done via the process $x \mapsto x^e \bmod N$ and decryption is done via the process $x \mapsto x^d \bmod N$. Thus e must be public and d must be private.

Generation of public and private keys The following steps are done by Alice in order to create RSA public and private keys:

1. Select two prime numbers p and q by some random process and compute $N = p \cdot q$.
2. Compute $\varphi(N) = (p-1)(q-1)$. Euler's theorem tells us that $x^{\varphi(N)} \equiv 1 \pmod N$ for all x coprime to N.
3. Select an $e > 1$ that is coprime to $\varphi(N)$. This choice can be random or fixed. The number $e = 2^{16} + 1 = 65537$ is typical and even $e = 3$ works fine (though such a small choice is less secure in some settings).
4. Compute $d := e^{-1} \bmod \varphi(N)$ using the extended Euclidean algorithm. Note that the inverse must exist since e is coprime to $\varphi(N)$.
5. Publish N and e as your public key. Keep d as your private key. The numbers p, q, and $\varphi(N)$ must not be published; they can be kept private or discarded at this point.

As previously mentioned, a message x is encoded to $x^e \bmod N$ and an encoded message y is decoded to $y^d \bmod N$.

Note that computing the inverse of $e \bmod \varphi(N)$ is easy (via the EEA) if $\varphi(N) = (p-1)(q-1)$ is known. Alice does know $\varphi(N)$ because she choose p and q. However, an attacker has no known way of computing $\varphi(N)$ without knowing the factorization of N.

Why does this work? The fundamental theory that RSA relies on is Euler's theorem that $x^{\varphi(N)} \equiv 1$ for x coprime to N. Using this we can show that encryption and decryption are inverses. In other words, if you encrypt something and then decrypt it you end up with what you started with.

In symbols, we want to show that $(x^e)^d \equiv x \pmod N$ for all x.

Note that d is the inverse of $e \bmod \varphi(N)$ so we have that $ed \equiv 1 \pmod{\varphi(N)}$. Thus we have that $ed = 1 + k\varphi(N)$ for some integer k.

First suppose x is coprime to N. Then we have

$$(x^e)^d \equiv x^{ed} \equiv x^{1+k\varphi(N)} \equiv x \cdot (x^{\varphi(N)})^k \equiv x \cdot 1^k \equiv x \pmod N$$

where Euler's theorem was used to reduce $x^{\varphi(N)}$.

This is the main case, though to complete the proof we must also consider the rare case when x and N are not coprime. This happens when x is a multiple of p or q (since the only prime divisors of N are p and q), although in practice this will never occur if p and q are large primes chosen randomly.

Say x is a multiple of p and not a multiple of q. Then $x \equiv 0 \pmod{p}$, so

$$(x^e)^d \equiv 0^d \equiv 0 \equiv x \pmod{p}.$$

By Fermat's little theorem we have $x^{q-1} \equiv 1 \pmod{q}$. Then

$$(x^e)^d \equiv x^{ed} \equiv x^{1+k\varphi(N)} \equiv x \cdot (x^{q-1})^{k(p-1)} \equiv x \cdot 1^{k(p-1)} \equiv x \pmod{q}$$

where Fermat's little theorem was used to reduce x^{q-1}. Then $(x^e)^d \equiv x$ modulo p and q and the Chinese remainder theorem then implies $(x^e)^d \equiv x \pmod{p \cdot q}$.

Similar reasoning applies when x is multiple of q but not p. Incidentally, if x is a multiple of p and q then $x \equiv 0 \pmod{N}$ so the theorem still holds (but this case is not useful for encryption).

1.4.4 Example

[7]:
```
# Set up public and private keys
while True:
    p = random_prime(10^30)
    q = random_prime(10^30)
    n = p*q
    phi = (p-1)*(q-1)

    # Ensure that phi(n) is coprime to 3
    if phi % 3 != 0:
        break

e = 3
d = e^(-1) % phi

show(html("\\begin{align*}"+
          "p&={}\\\\".format(p)+
          "q&={}\\\\".format(q)+
          "n&={}\\\\".format(n)+
          "\\varphi(n)&={}\\\\".format(phi)+
          "e&={}\\\\".format(e)+
          "d&={}".format(d)+
          "\\end{align*}"))
```

$$\begin{aligned}
p &= 254720409927450222938794161191 \\
q &= 305516782275999176882684397551 \\
n &= 77821360021058069052639187980342253936597343533274219643241 \\
\varphi(n) &= 77821360021058069052639187979782016744393894133452741084500 \\
e &= 3 \\
d &= 51880906680705379368426125319854677829595929422301827389667
\end{aligned}$$

[8]:
```
# Message to send
message = "ANTIDISESTABLISHMENTARIANISM"
l = len(message)

# Convert message to list of integers in the range 1-26
A = [ord(message[i])-64 for i in range(l)]

# Convert message to an integer
x = sum(A[i]*27^i for i in range(l))

# Encode codeword
y = power_mod(x, e, n)

# Decode encrypted codeword
z = power_mod(y, d, n)

show(html("\\begin{align*}"+
          "x&={}\\\\".format(x)+
          "y=x^e\\bmod n&={}\\\\".format(y)+
          "z=y^d\\bmod n&={}".format(z)+
          "\\end{align*}"))

print("The decrypted message is " + "".join([chr(z.digits(base=27)[i]+64) for i␣
 ↪in range(l)]))
```

$$\begin{aligned} x &= 608237614112977152221882977492698035453 5 \\ y = x^e \bmod n &= 207347871368762323964517979314487889421050108722800646549 76 \\ z = y^d \bmod n &= 608237614112977152221882977492698035453 5 \end{aligned}$$

```
The decrypted message is ANTIDISESTABLISHMENTARIANISM
```

Warning Although this is the basic idea, RSA does have to be used carefully in practice as in some cases there are ways that an attacker can break the system.

For example, the choice of $e = 3$ is conveinient because it allows encoding to be done very quickly (just a cubing modulo N).

However, this is insecure if the message x to encrypt is small, namely $x < N^{1/3}$.

Why? In the case $x < N^{1/3}$ it follows that $x^3 < N$, so the encrypted message will just be the *integer* cube of x (i.e., no reducing modulo N will be done). Then x be recovered by computing the cube root of x^3 over the reals or integers which is an easy operation.

1.4.5 Takeaway

RSA provides a way of sending secure messages to anyone who has published their public key.

Generating a public/private keypair involves finding 2 large random primes p and q and computing an inverse modulo $(p-1)(q-1)$. After that, encryption and decryption is simply done via exponentation modulo pq.

Since p and q are relatively large (e.g., 1024-bits) computations modulo pq are relatively expensive, at least compared with most encryption algorithms based on stream ciphers, for example. For this reason RSA is not normally used to encrypt messages. Instead, it it normally only used to communicate a shared secret key that is then used to seed a stream cipher or some other symmetric encyption method.

RSA also enables *digital signatures*. Alice can prove that she had access and 'signed' a message x by computing $S = x^d \bmod N$.

Anyone can now verify that Alice's signature is genuine by checking that $S^e \equiv x \pmod{N}$ where (e, N) is Alice's public key. However, no one can sign a message pretending to be Alice unless they know d.

How secure is RSA? When RSA is performed correctly it provides very good security because there is no known method of efficiently computing eth roots modulo N without knowing the prime factorization of N.

Moreover, there is no known method of efficiently computing the prime factorization of an arbitrary integer N.

This is not for lack of trying: RSA Labs created an "RSA Factoring Challenge" of many numbers $N = p \cdot q$ of various lengths that they challenge anyone to try to factor.

Currently, the largest RSA challenge number that has been successfully factored has 829 bits. This was completed in 2020 using numerous computers simultaneously on a computing cluster and in total requiring about 2,700 computer years.

In particular, factoring an RSA key of 1024 bits is currently out of the realm of feasibility. (As far as anyone knows.) Thus, it is very easy for systems like Sage to produce simple numerical problems that are totally out of reach of modern computers to solve:

[9]:
```
random_prime(2^512)*random_prime(2^512)
```

[9]:
```
21807482014441466089665995062811164278420889212153514930552517180492621271161447
75862795939811985201275044367222322605882052692154666263725252734433075931110641
96018270521730396366522637101094575420394701636396206272922054831876968746765291
87208685732045937734942562028802521534642917821686690367733828608031
```

1.4.6 Final thoughts

G. H. Hardy was one of the most celebrated mathematicians of the twentieth century and a leading expert on number theory. He among anyone else was in the best position to see its amazing potential applications and yet in 1940 he wrote:

> No one has yet discovered any warlike purpose to be served by the theory of numbers or relativity, and it seems very unlikely that anyone will do so for many years.

As we saw, public-key cryptography would be designed a mere 36 years later.

Incidentally, he also cites the theory of relativity as another example of a "useless" theory. This was also proven incorrect at almost the same time—in 1973 the U.S. Department of Defense proposed the Global Positioning System (GPS) which relies on the theory of relativity to work correctly.