CISS451: Cryptography

Y. Liow (April 12, 2025)

Contents

Chapter 1

RSA

1.1 RSA

Now for RSA ...

We only need to work with integers. Why? Because any message M is really just a sequence of bits and you can cut your sequences of bits which can be viewed as an unsigned int, i.e, an integer ≥ 0. So we'll just think of our messages as integers.

So once again suppose Alice wants to send a secret to Bob. The secret is an integer x.

Bob selects two distinct primes p and q. He computes $N = pq$ and selected two positive integers e and d such that

$$ed \equiv 1 \pmod{\phi(N)}$$

In other words e and d and multiplicative inverses of each other in $\mathbb{Z}/\phi(N)$. Furthermore Bob publishes N and e publicly. e is called the **encryption exponent** and d is called the **decryption exponent**.

decryption exponent

We assume that the secret x is less than N because we'll be working in $\mathbb{Z}/N$. (Again if x as a bit sequence is too large for N, then we cut x up into smaller block of bits and send them separately.)

Since N and e are public, Alice can download N and e and then compute

$$E_{(N,e)}(x) = x^e \pmod{N}$$

I'll write $x^e \pmod{N}$ for the least positive remainder of $x^e \bmod N$. She then sends $x^e \pmod{N}$ to Bob.

When Bob received $x^e \pmod N$, he computes

$$D_{(N,d)}(x) = x^{ed} \pmod N$$

What we need to prove is that RSA works, i.e.,

Theorem 1.1.1. *Let p, q be primes and $N = pq$. If $ed \equiv 1 \pmod{\phi(N)}$, then*

$$(x^e)^d \equiv x \pmod N$$

for all integers x.

Proof. TODO □

Proposition 1.1.1. *Prove the following:*

(a) *If p and q are distinct primes such that $p \mid a$ and $q \mid a$, then $pq \mid a$.*
(b) *If $x \mid a$ and $y \mid a$ and $\gcd(x, y) = 1$, then $xy \mid a$.*
(c) *If $x \mid a$ and $y \mid a$, then $(xy/\gcd(x, y)) \mid a$.*

(b) is a generalization of (a) and (c) is a generalization of (b). □

Proof. (a) TODO

(b) TODO

(c) TODO □

Note that RSA (and all public key ciphers) are not meant for encrypting messages like strings (emails, sales receipts, image files, etc.) This is an unfortunate thing that's done in many cryptography textbooks. They are actually used to encrypt/decrypt keys for private ciphers since private ciphers are much faster.

For instance suppose Bob wants to send an encrypted message M to Alice. He first picks a private cipher. Say, he is going to use the substitution cipher. He creates a key for the substitution cipher: $a \mapsto h, b \mapsto z, c \mapsto t, \ldots$. The key can be described by the string `key` $= hzt\ldots$. This is of course a binary sequence of $26 \times 8 = 208$ bits. This can of course be viewed as a positive integer mod N as long as N is large enough. Bob can use RSA to send the encrypted `key` to Alice, which Alice will decrypt. After that Bob can send his M, encrypted using the substitution cipher with `key` to Alice.

Therefore RSA (and other public key cipher) is used to setup keys for private key cipher (example: AES) because AES is much faster than RSA and the message to be sent is usually way longer than the key.

The speed of private ciphers should be clear from the chapter on classical ciphers. Likewise modern private ciphers such as 3DES, AES, etc. are much faster than RSA and other private key ciphers. The current modern private cipher is AES. 3DES is considered deprecated since 2018. In particular the RSA standard (called PKCS – you can easily find lots of webpages on PKCS) does not include specification on how to break up long messages before encryption and how to reassembly them after decryption.

OK, let's summarize everything. In the following, I'll write $x \pmod N$ to be the remainder when x is divided by N. There are three steps: Bob has to generate keys, Alice has to encrypt, and Bob has to decrypt.

1. Key Generation:
   a) Bob selects distinct primes p and q.
   b) Bob computes $N = pq$.
   c) Bob computes $\phi(N) = (p-1)(q-1)$.
   d) Bob selects e such that $0 < e < \phi(N)$, $\gcd(e, \phi(n)) = 1$.
   e) Bob computes d such that $ed \equiv 1 \pmod{\phi(N)}$.
   f) Bob publishes (N, e) (the public key) but keeps (N, d) (the private key) to himself.
2. Encryption: Alice obtains the publicly available (N, e) (the public key) and computes
$$E_{(N,e)}(x) = x^e \pmod N$$
   and sends it to Bob.
3. Decryption: Bob uses (N, d) (the private key) to compute
$$D_{(N,d)}(x^e \pmod N) = x^{ed} \pmod N$$

Note that the key is made up of

- a **public key** (N, e) and — public key
- a **private key** (N, d) — private key

(N, e) is revealed to the public. (N, d) is kept private. In general

Definition 1.1.1. A **public cipher** is made up of the encryption and decryption functions $E_{\text{pubkey}}, D_{\text{privkey}}$ which depends on the key $k = (\text{pubkey}, \text{privkey})$ which is a 2-tuple made up of the public key and private key. Such a cipher is — public cipher also called an **asymmetric key cipher**.

asymmetric key cipher

Recall that in the case of private (or symmetric) key cipher the encryption and decryption keys are the same.

Exercise 1.1.1. Eve saw Alice sent the ciphertext 230539333248 to Bob. Eve checks Bob's website and found out that his public RSA key is $(N, e) = (100000016300000148701, 7)$. Help Eve compute the plaintext. In fact, compute the private key (N, d). You only need to compute d since N is known. How much time did you use? (Hint: Why is factoring N crucial?) (Go to solution, page 7) □

debug: exercises/rsa-14/question.tex

Solutions

Solution to Exercise 1.1.1.

debug: exercises/rsa-14/answer.tex

Solution not provided.

1.2 Implementation issues debug: rsa-implementation-issues.tex

For RSA to be a good cryptosystem, the operations involved (key generation, encryption, decryption) must be fast. Furthermore it must be able to sustain all types of attacks. In this section we'll talk about algorithms and performance issues.

1.3 Baby Asymptotic Analysis debug: baby-asymptotic-analysis.tex

The mathematical technology used to measure the speed of an algorithm is called asymptotic analysis. I'm not going to explain everything in asymptotic analysis. I'll only give you enough to move forward. (By the way this area of math was created by people in the area of number theory.)

Let's look at a simple problem first. Suppose you're given two 3-digit numbers to add, say you want to do $123 + 234$. This is what you would do:

```
    1 2 3
+   3 6 9
---------
    4 9 2
---------
```

The mount of work done involves reading two digits for each column, computing the digit sum, which gives two numbers (remember you need to compute the carry too, right?) the sum mod 10 and sum / 10. For instance for the first column (the leftmost one), you do

```
 3 +  9 = 12
12 % 10 = 2
12 / 10 = 1
```

Once that's done you write down:

```
      1
    1 2 3
+   3 6 9
---------
        2
---------
```

Correct? You then go on to the second column and do this:

```
1 + 2 + 6 = 9
9 % 10 = 9
9 / 10 = 0
```

and you write this:

```
        0 1
        1 2 3
+       3 6 9
---------
          9 2
---------
```

and so on.

Note that what you do is the same for each column. The first column is kind of different because you don't have to worry about any carry at all. However to make the first column like the rest, you can create an initial carry of 0 too:

```
            0
        1 2 3
+       3 6 9
---------

---------
```

Why would you do that? Well ... it make the algorithm more uniform. But in any case for each column, you basically perform the following work:

```
0 + 3 + 9 = 12
12 % 10 = 1
12 / 10 = 1
```

Of course you also have to read the digits (think of this as work done reading the piece of paper), write the digits on the piece of paper. Suppose it takes times t_1 to do all that for each column.

There are 3 columns. Therefore when you're done with all the column operations, you would have used up this amount of time:

$$3 \cdot t_1$$

There was actually one step you did by putting a zero as an initial carry:

```
            0
        1 2 3
+       3 6 9
```

```
 ---------

 ---------
```

Let's say this takes time t_0. So the total time is

$$3 \cdot t_1 + t_0$$

Don't forget that once you're done with the 3 columns, you have a carry beyond the 3rd column:

```
   0 0 1 0
     1 2 3
 +   3 6 9
 ---------
     4 9 2
 ---------
```

You can put the 0 down on the fourth column:

```
   0 0 1 0
     1 2 3
 +   3 6 9
 ---------
   0 4 9 2
 ---------
```

or just leave it blank. Suppose the time to put it down or not put it down is t_2. So the total time is

$$3 \cdot t_1 + t_0 + t_2$$

It should be clear that if you have two n–digit numbers, the time needed is

$$n \cdot t_1 + t_0 + t_2$$

Now someone who writes faster, reads faster, compute addition or quotient or mod 10 faster might do it in time

$$n \cdot t_1' + t_0' + t_2'$$

However the n doesn't go away. In the long run, it's n can controls the growth of this function. If someone were to invent a different addition algorithm that runs with this time:

$$\frac{1}{2}n \cdot t_1'' + t_0'' + t_2'' = n \cdot \frac{t_1''}{2} + t_0'' + t_2''$$

then it's really the same as the previous one. Why? Because all you need to do is to hire someone who can read and write and perform digit addition, digit mod 10, digit / 10 extremely fast so that you t_1 is much smaller than the $\frac{t_1''}{2}$ and $t_0 + t_2$ much smaller than his $t_0'' + t_2''$.

Not only that ... if n is really huge, it's clear that the $n \cdot t_1$ is going to overcome the $t_0 + t_2$ part.

And if we are concerned about the speed of our algorithm, obviously we worry about the case when n (the number of digits) is huge. After all ... who cares that much about addition of 3 digits? What we should worry about is what happens when we apply our method to the addition of two 1000000-digit numbers. If you look at for instance

$$n^2 + 1000000n$$

and

$$n^2$$

when n is small (say 3), the $1000000n$ is huge. But if you think about $n = 10^{1000}$, you see that $n^2 + 1000000n$ and n^2 are actually very close. (Take out your graphing calculator and try zooming out the graphs of $y = x^2$ and $y = x^2 + 1000000x$ to check that I'm not lying.)

This tells us that really the function to focus on when measuring algorithm runtime performance is actually

$$n$$

and not

$$n \cdot t_1 + t_0 + t_2$$

That's the whole point of asymtotic analysis. (Well ... there's a lot more ... but that's enough for *us* ...)

To say that we're ignoring the constants and focusing on the part that controls the growth of the function

$$n \cdot t_1 + t_0 + t_2$$

we write

$$n \cdot t_1 + t_0 + t_2 = O(n)$$

That's called the "big-O of n".

Let's look at what's called "high school multiplication algorithm". Suppose you're given two 3-digit numbers to multiply. Say 123 and 234. You would do this:

```
      1 2 3
  x   2 3 4
  ---------
      4 9 2
    3 6 9
+ 2 4 6
-----------
  2 8 7 8 2
-----------
```

It's easy to see that there are 9 digits to compute in the "mid-section" of the computation:

```
      1 2 3
  x   2 3 4
  ---------
      4 9 2
    3 6 9
+ 2 4 6
-----------

-----------
```

Then the 3 rows are added up.

Using this method, how much time does it take to compute the product of two n–digit integers? There are $n \times n$ numbers to compute in the mid-section. So the midsection requires

$$O(n^2)$$

Note that the numbers you get contain integers of length about $2n$. There are n such numbers. Adding two $2n$-digit numbers takes $O(2n)$ time. But remember that we ignore constants. So adding two of the $2n$-digit numbers take

$$O(n)$$

times. Given n such numbers, you need to perform $n - 1$ additions. So altogether the time is

$$O((n-1)n)$$

Now look at the function

$$(n-1)n$$

This is

$$(n-1)n = n^2 - n$$

The worse part of the function (the part that grows the fastest) is n^2. Therefore

$$O((n-1)n) = O(n^2)$$

Now the computation of the midsection takes $O(n^2)$ and the addition part takes $O(n^2)$. Together it would take $O(2n^2)$. But again we ignore constants. So the time taken is $O(2n^2) = O(n^2)$.

Hence the worse case runtime performance of the "high school multiplication algorithm" is $O(n^2)$.

This standard multiplication algorithm has been used for a very very very very long time.

Can we do better?

Before we leave this section, note that I've already said that addition runs in $O(n)$. You can't do better than that. Why? Well ... you have to at least read the two n–digit numbers, right? Reading them takes about $n + n$ time, which is $2n$, which is $O(n)$. So there's no way you can beat that. Hey ... you have to at least read what to add, right?!?!

1.4 Multiplication: Karatsuba algorithm debug: karatsuba.tex

Skip this section if you have already taken CISS358.

It turns out that there is a better way to multiply integers. This was only discovered very recently in the 1960s.

The idea is surprisingly simple. Here's a basic formula:

$$(aT + b)(cT + d) = acT^2 + (ad + bc)T + bd$$

For now you can pretend that $T = 10$ and a, b, c, d are digits. For instance to multiply 23 and 45, you can view it as

$$(2 \cdot 10 + 3)(4 \cdot 10 + 5) = (2 \cdot 4)10^2 + (2 \cdot 5 + 3 \cdot 4)10 + 3 \cdot 5$$

You can finish up this computation on your own.

But

$$(aT + b)(cT + d) = acT^2 + (ad + bc)T + bd$$

doesn't really help!!! Viewing $T = 10$ and the a, b, c, d as digits, the above multiplication of two 2-digit numbers, i.e. $(aT+b)$ and $(cT+d)$, we need *four* multiplications

$$ac, \quad ad, \quad bc, \quad bd$$

So it's still essentially an n^2 algorithm. In general if $T(n)$ is runtime of multiplying 2 integers of length n, then

$$T(n) = 4T(n/2) + An + B$$

If $n = 2^k$, then

$$\begin{aligned}
T(2^k) &= 4T(2^{k-1}) + A2^k + B \\
&= 4(4T(2^{k-2}) + A2^{k-1} + B) + A2^k + B \\
&= 4^2T(2^{k-2}) + (4 \cdot 2^{k-1} + 2^k)A + (4 + 1)B \\
&= 4^2(4T(2^{k-3}) + A2^{k-2} + B) + (4 \cdot 2^{k-1} + 2^k)A + (4 + 1)B \\
&= 4^3T(2^{k-3}) + (4^2 \cdot 2^{k-2} + 4 \cdot 2^{k-1} + 2^k)A + (4^2 + 4 + 1)B \\
&= ... \\
&= 4^kT(2^{k-k}) + (4^{k-1} \cdot 2^1 + ... + 4^2 \cdot 2^{k-2} + 4 \cdot 2^{k-1} + 2^k)A + (4^{k-1} + ... + 4^2 + 4 + 1)B
\end{aligned}$$

The coefficient for B is $\frac{4^k - 1}{4 - 1}$. The coefficient for A is

$4^{k-1}\cdot 2^1 + \ldots + 4^2 \cdot 2^{k-2} + 4 \cdot 2^{k-1} + 2^k = 2^{2k-1} + 2^{k+2} + 2^{k+1} + 2^k = 2^k(1 + 2 + \ldots + 2^{k-1}) = 2^k(2^k - 1)$

Hence

$$T(2^k) = 4^k T(1) + 2^k(2^k - 1)A + \frac{4^k - 1}{3} B$$

Hence $T(1) = C$:

$$T(2^k) = 4^k C + 2^k(2^k - 1)A + \frac{4^k - 1}{3} B$$

Since $n = 2^k$,

$$T(n) = n^2 C + n(n - 1)A + \frac{n^2 - 1}{3} B$$

Clearly $T(n) = O(n^2)$.

But wait ... here's the brilliant (but simple) idea from Karatsuba. Note that

$$\begin{aligned} & (a+b)(c+d) = ac + ad + bc + bd \\ \therefore \quad & (a+b)(c+d) - ac - bd = ad + bc \end{aligned}$$

In other words

$$(aT + b)(cT + d) = (ac)T^2 + [(a+b)(c+d) - (ac) - (bd)]T + (bd)$$

If you count all the operations on the right, you would see that there are only *three* multiplications!!!

So we can do this:

```
1. Compute A = ac          ... 1 multiplication
2. Compute B = bd          ... 1 multiplication
3. Compute C = (a+b)(c+d)  ... 2 additions, 1 multiplication
4. Compute D = C - A - B   ... 2 subtractions
5. Output AT^2 + DT + B
```

Note that the runtime for subtraction is like addition. In other words addition and subtraction are "cheap": they are both $O(n)$ where n is the length of the integers.

Practically speaking, how is Karatsuba actually used? Suppose you have to multiply two 8-digit numbers:

$$12345678 \times 13572468$$

We split them up into this (with $T = 10$)

$$(1234T^4 + 5678) \times (1357T^4 + 2468)$$

by Karatsuba

$$\begin{aligned}(aT^4 + b) \times (cT^4 + d) &= A(T^4)^2 + D(T^4) + B \\ A &= ab \\ B &= bd \\ C &= (a+b)(c+d) \\ D &= C - A - B\end{aligned}$$

Next, we *again* apply Karasuba but this time to the products $ab, bd, (a+b)(c+d)$. Etc. In other words you recursively use Karatsuba until the products involve numbers which are small enough that we can perform them quickly without Karatsuba, If we starting with the multiplication of two 8-digit numbers. We're left with three multiplications of 4-digit numbers. On applying Karasuba to the three multiplications of 4-digits numbers, each multiplication gives rise to 3 2-digit numbers. There are now 3×3 multiplications of 2-digit numbers. Going further, we get $3 \times 3 \times 3$ multiplication of 1-digit numbers.

In general you see the following. Suppose we start off with multiplying two n-digit numbers and $n = 2^m$. We have:

1 multiplication(s) of 2^m–digit numbers
3 multiplication(s) of 2^{m-1}–digit numbers
3^2 multiplication(s) of 2^{m-2}–digit numbers
3^3 multiplication(s) of 2^{m-3}–digit numbers
3^4 multiplication(s) of 2^{m-4}–digit numbers
...
3^m multiplication(s) of $2^{m-m} = 1$–digit numbers

Note that we started with n-digit numbers, i.e., $n = 2^m$, i.e., $m = \log n$ (log means $\log_2$, right?) This gives rise to $3^m = 3^{\log n}$ multiplications, i.e.

$$3^{\log n} = 3^{\frac{\log_3 n}{\log_3 2}} = \left(3^{\log_3 n}\right)^{\frac{1}{\log_3 2}} = n^{\frac{1}{\log_3 2}} = n^{\frac{1}{\log 2 / \log 3}} = n^{\log_2 3} = n^{1.58496\ldots}$$

multiplications which is a lot better than n^2 when n is huge. (The process of adding the 3 numbers for each stage is no big deal.) Some details are missing in the above analysis. You can check the details for yourself. The big-O is correct.

Here's an example. Suppose we want to multiply

$$1234 \times 1357$$

Step 1: (1234)(1357).

$$\begin{aligned}(1234)(1357) &= (12T^2 + 34) \times (13T^2 + 57)\\ &= A(T^2)^2 + DT^2 + B\end{aligned}$$

($T = 10$) where

$$\begin{aligned}&a = 12,\ \ b = 34,\ \ c = 13,\ \ d = 57\\ &A = ac = (12)(13)\\ &B = bd = (34)(57)\\ &C = (a+b)(c+d) = (46)(70)\\ &D = C - A - B\end{aligned}$$

Step 2: (12)(34).

$$\begin{aligned}(12)(13) &= (1T^1 + 2) \times (1T^1 + 3)\\ &= A(T^1)^2 + DT^1 + B\end{aligned}$$

where

$$\begin{aligned}&a = 1,\ \ b = 2,\ \ c = 1,\ \ d = 3\\ &A = ac = (1)(1) = 1\\ &B = bd = (2)(3) = 6\\ &C = (a+b)(c+d) = (3)(4) = 12\\ &D = C - A - B = 12 - 1 - 6 = 5\end{aligned}$$

So

$$(12)(34) = 1(T^1)^2 + 5T^1 + 6 = 100 + 50 + 6 = 156$$

Step 3: (34)(57)

$$(34)(57) = (3T^1 + 4) \times (5T^1 + 7) = A(T^1)^2 + DT^1 + B$$

where

$$\begin{aligned}
a &= 3, \;\; b = 4, \;\; c = 5, d = 7 \\
A &= ac = (3)(5) = 15 \\
B &= bd = (4)(7) = 28 \\
C &= (a+b)(c+d) = (7)(12) = 84 \\
D &= C - A - B = 84 - 15 - 28 = 41
\end{aligned}$$

So

$$(34)(57) = 15(T^1)^2 + 41T^1 + 28 = 1500 + 410 + 28 = 1938$$

Step 4: (46)(70).

$$(46)(70) = (4T^1 + 6) \times (7T^1 + 0) = A(T^1)^2 + DT^1 + B$$

where

$$\begin{aligned}
a &= 4, \;\; b = 6, \;\; c = 7, \;\; d = 0 \\
A &= ac = (4)(7) = 28 \\
B &= bd = (6)(0) = 0 \\
C &= (a+b)(c+d) = (10)(7) = 70 \\
D &= C - A - B = 70 - 28 - 0 = 42
\end{aligned}$$

So

$$(46)(70) = 28(T^1)^2 + 42T^1 + 0 = 2800 + 420 + 0 = 3220$$

Putting Steps 2,3,4 back into Step 1 ...

Continuing Step 1: (1234)(1357).

$$\begin{aligned}
(1234)(1357) &= (12T^2 + 34) \times (13T^2 + 57) \\
&= A(T^2)^2 + DT^2 + B
\end{aligned}$$

where

$$
\begin{aligned}
&a = 12, \;\; b = 34, \;\; c = 13, \;\; d = 57 && \\
&A = (12)(13) = 156 && \text{from Step 1} \\
&B = (34)(57) = 1938 && \text{from Step 2} \\
&C = (46)(70) = 3220 && \text{from Step 3} \\
&D = C - A - B = 3220 - 156 - 1938 = 1126 &&
\end{aligned}
$$

and

$$
\begin{aligned}
(1234)(1357) &= A(T^2)^2 + DT^2 + B \\
&= 156(T^2)^2 + 1126T^2 + 1938 \\
&= 1560000 + 112600 + 1938 \\
&= 1674538
\end{aligned}
$$

Now if you go back and look for the multiplications you see these:

$$
\begin{aligned}
(1)(1) &= 1 \\
(2)(3) &= 6 \\
(3)(4) &= 12 \\
(3)(5) &= 15 \\
(4)(7) &= 28 \\
(7)(12) &= 84 \\
(4)(7) &= 28 \\
(6)(0) &= 0 \\
(10)(7) &= 70
\end{aligned}
$$

i.e., $9 = 3^2$ multiplications if we multiply two integers of length $n = 2^2$. High-school multiplication would require $n^2 = (2^2)^2 = 16$ multiplications.

Now let us compute the runtime of Karatsuba's algorithm:

Proposition 1.4.1. *The runtime of Karatsuba is $O(n^{\lg 3}) = O(n^{1.5849\ldots})$ where n is the length of the integers to be multiplied.*

Proof. From

$$(aT + b)(cT + d) = (ac)T^2 + [(a + b)(c + d) - (ac) - (bd)]T + (bd)$$

if $T(n)$ is the runtime where n is the number of digits in the two integers to be multiplied, we have

$$T(n) = 3T(n/2) + An + B$$

Let $n = 2^k$. Then

$$\begin{aligned}
T(2^k) &= 3T(2^{k-1}) + A2^k + B \\
&= 3(3T(2^{k-2}) + A2^{k-1} + B) + A2^k + B \\
&= 3^2T(2^{k-2}) + (3 \cdot 2^{k-1} + 2^k)A + (3 + 1)B \\
&= 3^2(3T(2^{k-3}) + A2^{k-2} + B) + (3 \cdot 2^{k-1} + 2^k)A + (3 + 1)B \\
&= 3^3T(2^{k-3}) + (3^2 \cdot 2^{k-2} + 3 \cdot 2^{k-1} + 2^k)A + (3^2 + 3 + 1)B \\
&= \cdots \\
&= 3^kT(2^{k-k}) + (3^{k-1} \cdot 2 + \cdots + 3^2 \cdot 2^{k-2} + 3 \cdot 2^{k-1} + 2^k)A + (3^{k-1} + \cdots + 3^2 + 3 + 1)B
\end{aligned}$$

The coefficient of B is $\dfrac{3^k - 1}{2}$. The coeffcient of A is

$$3^{k-1} \cdot 2 + \cdots + 3^2 \cdot 2^{k-2} + 3 \cdot 2^{k-1} + 2^k = 2(3^{k-1} \cdot 2^0 + \cdots + 3^2 \cdot 2^{k-3} + 3^1 \cdot 2^{k-2} + 3^0 \cdot 2^{k-1})$$

This is a convolution and is the coeffcient of x^{k-1} of

$$\begin{aligned}
2 \cdot \sum_{n \geq 0} 3^n x^n \cdot \sum_{n \geq 0} 2^n x^n &= 2 \cdot \frac{1}{1 - 3x} \cdot \frac{1}{1 - 2x} \\
&= 2\left(\frac{3}{1 - 3x} + \frac{-2}{1 - 2x}\right) \\
&= 2\left(3\sum_{n \geq 0} 3^n x^n - 2\sum_{n \geq 0} 2^n x^n\right) \\
&= \sum_{n \geq 0} 2(3^{n+1} - 2^{n+1})x^n
\end{aligned}$$

Hence

$$T(2^k) = 3^kC + 2(3^k - 2^k)A + \frac{3^k - 1}{2}B$$

where $C = T(1)$. Since $n = 2^k$, we have $3^k = 2^{\log_2 3^k} = 2^{k \log_2 3} = (2^k)^{\log_2 3} =$

$n^{\log_2 3}$. Therefore

$$\begin{aligned}
T(n) &= n^{\log_2 3}C + 2(n^{\log_2 3} - n)A + \frac{n^{\log_2 3} - 1}{2}B \\
&= (C + 2A + B/2)n^{\log_2 3} - 2An - B/2 \\
&= O(n^{\log_2 3}) \\
&= O(n^{1.5849\ldots})
\end{aligned}$$

$\square$

The above result is the same whether you view your integer as an array of decimals or as an array of bits.

It's a good exercise to implement Karatsuba on your own. I did some number theory crunching programming during the summer of my freshman year and a prof gave me a copy of the original paper published by Karatsuba. (Karatsuba's paper has since inspired several improvements.) As you see from the above, the amount of math that you need to know is pretty minimal. Basically the ingredients are

$$(aT + b)(cT + d) = acT^2 + (ad + bc)T + bd$$

(which is not new) and this (which is new):

$$\begin{aligned}
& (a + b)(c + d) = ac + ad + bc + bd \\
\therefore \quad & (a + b)(c + d) - ac - bd = ad + bc
\end{aligned}$$

There are various obvious optimizations which only tweaks the constants of your big-O, but not the big-O itself. For instance if you look at this addition:

$$\begin{aligned}
(1234)(1357) &= A(T^2)^2 + DT^2 + B \\
&= 156(T^2)^2 + 1126T^2 + 1938 \\
&= 1560000 + 112600 + 1938 \\
&= 1674538
\end{aligned}$$

you'll see that you are adding 1560000 with another number 112600. There are some zeroes in 112600. So you might want to have a function that adds say 1126 to an integer starting at the 100s digit position. This will speed up for instance adding to an integer 123456789123456789, the number 123000000000000:

```
123456789123456789
```

```
+    123000000000000
--------------------

--------------------
      ^
      |
      start column addition here
```

Another thing to note is that our computers are mostly 32 bit machines. If you're using an array of integers to model long integers, you only want to cut up the integers up to a certain point, and not to the digit-level. For instance you might want to model your long integers with arrays of 16-bit integers. Note that the process of cutting up the integers into pieces takes time. In fact when the number of digits is small, highschool multiplication might be faster. Therefore you want to write your code in such as way that if the length of the integers is less than a certain constant, then highschool multiplication is used; if the length is greater than this constant, then Karatsuba is used. That's what I did.

(The language I used long long long time ago to do Karatsuba was Pascal. Each element of the array models 0 to 999. I found through timed testing that if the array has length ≤ 5, then highschool multiplication was actually faster. This is of course machine specific. This explains why when you install some number crunching libraries, before the installation is completed, it will tests the code to tweak such constants for maximum performance.)

OK. Enough hints. You should go ahead and write a long integer package that incorporates Karatsuba for multiplication.

Exercise 1.4.1. Compute $1122334455667788 \times 8765432187654321$ using Karasuba multiplication to continually breakdown the integers up to integers of length 2; use your calculator to perform multiplication of length 2 integers. (Go to solution, page 25) □

debug: exercises/rsa-30/question.tex

Exercise 1.4.2. Implement a long integer class where multiplication uses Karasuba. (Go to solution, page 26) □

debug: exercises/rsa-31/question.tex

Exercise 1.4.3. The algorithmic analysis above is basically correct but some details are actually missing. Write a research paper on Karatsuba, describing

debug: exercises/rsa-32/question.tex

the algorithm in detail, and analyze the runtime performance exactly. Research also on efficient implementation of Karatsuba. (Go to solution, page 27) □

Exercise 1.4.4. * Since the surprising (shocking?) discovery of Karatsuba, several improvements to his algorithm has appeared since the 60s. Write a research paper on the various new-fangled integer multiplication algorithms, analyze and comparison their runtime performance. (Go to solution, page 28) □

debug: exercises/rsa-33/question.tex

Solutions

Solution to Exercise 1.4.1.

debug: exercises/rsa-30/answer.tex

Solution not provided.

Solution to Exercise 1.4.2.

debug: exercises/rsa-31/answer.tex

Solution not provided.

Solution to Exercise 1.4.3.

debug: exercises/rsa-32/answer.tex

Solution not provided.

Solution to Exercise 1.4.4.

debug: exercises/rsa-33/answer.tex

Solution not provided.

1.5 Exponentiation: the squaring method debug: exponentials-squaring-method.tex

See CISS358. Skip this section if you have taken CISS358.

Note that for RSA we need to compute powers. Extremely large powers. The obvious method: If $n > 0$,

$$a^n = a^{n-1} \cdot a$$

has a runtime of $O(n)$ (in fact $\Theta(n)$) of multiplications. If the exponentiation is in mod N, then you take mod N after each multiplication to previous the a^k from becoming too huge, i.e.,

$$a^k = a^{k-1} \cdot a \pmod N$$

There's a much faster way to compute exponentiation.

First I will describe it using recursion:

$$a^n = \begin{cases} 1 & n = 0 \\ \left(a^{n/2}\right)^2 & n > 0 \text{ and } n \text{ is even} \\ a \cdot \left(a^{(n-1)/2}\right)^2 & n > 0 \text{ and } n \text{ is odd} \end{cases}$$

Here, $n \geq 0$.

Example 1.5.1. Here's an example to compute 2^{27}

1. $2^{27} = 2 \cdot (2^{13} \cdot 2^{13})$
2. $2^{13} = 2 \cdot (2^6 \cdot 2^6)$
3. $2^6 = (2^3 \cdot 2^3)$
4. $2^3 = 2 \cdot (2^1 \cdot 2^1)$
5. $2^1 = 2 \cdot (2^0 \cdot 2^0) = 2 \cdot (1 \cdot 1)$

It's easy to write an algorithm implementing the above:

```
ALGORITHM: power
INPUTS: a, n where n >= 0

if n == 0:
    return 1
else:
    if n is even:
```

```
        x = power(a, n / 2)
        return x * x
    else:
        x = power(a, (n - 1) / 2)
        return a * x * x
```

The number of recursions is $O(\lg n)$ where the main cost of each recursion step is either one or two multiplications.

This above recursion can be rewritten using a loop. Suppose you want to compute a^x. First you write x as a binary number:

$$x = (x_k \cdots x_0)_2 = x_k 2^k + \ldots + x_1 2^2 + x_0 2^0$$

where each x_i is 0 or 1. As an example suppose we look at

$$x = 27 = (11011)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Then

$$a^{27} \equiv a^{1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0} \equiv a^{2^4} a^{2^3} a^{2^1} a^{2^0}$$

Notice that the computation of a^{27} on the right depends on $a^{2^0}, a^{2^1}, a^{2^2}, \ldots$ In general $a^{2^{i+1}} = (a^{2^i})^2$. In the following b runs through $a^{2^0}, a^{2^1}, a^{2^2}, \ldots$. Notice that a^{2^i} is included in a^{27} if the i–th bit of x is 1. Instead of pre-computing the bits of x, we can compute the i–th bit iteratively as the exponentiation is computed. In the following p runs through the partial products of $a^{2^0}, a^{2^1}, a^{2^2}, \ldots$, picking up a^{2^i} if the i–th bit of n is 1. In this case, p and b will run through the following values:

$$\begin{aligned}
p &= 1 & b &= a = a^{2^0} \\
p &= p \cdot b = a^{2^0} & b &= b \cdot b = a^{2^1} \\
p &= p \cdot b = a^{2^0} \cdot a^{2^1} & b &= b \cdot b = a^{2^2} \\
& & b &= b \cdot b = a^{2^3} \\
p &= p \cdot b = a^{2^0} \cdot a^{2^1} \cdot a^{2^3} & b &= b \cdot b = a^{2^4} \\
p &= p \cdot b = a^{2^0} \cdot a^{2^1} \cdot a^{2^3} \cdot a^{2^4} & b &= b \cdot b = a^{2^5}
\end{aligned}$$

Here's the algorithm:

```
ALGORITHM: power
INPUTS: a, n where n >= 0
OUTPUT: a^n

p = 1
```

```
b = a

while n is not 0:
    bit = n % 2
    n = n / 2 (integer division)
    if bit == 1:
        p = p * b
    b = b * b

return p
```

And if the exponentiation is done in mod N, then we just mod by N as frequently as we can:

```
ALGORITHM: power-mod
INPUTS: a, n, N where n >= 0 and N is the modulus
OUTPUT: (a^n) % N

p = 1
b = a % N
while n is not 0:
    if n % 2 == 1:
        p = (p * b) % N
    n = n // 2
    b = (b * b) % N
return p
```

The number of iterations is the number of bits of n, i.e., the number of iterations is $\lfloor \lg n + 1 \rfloor$ ($\lg = \log_2$). Each iteration in the worse case scenario involves two mod N multiplications. Hence the runtime time is

$$O((\lg n) \cdot M)$$

where M is the runtime for multiplication of two integers in $\mathbb{Z}/N$. (In the case when we are performing exponentiation without mod N, the length of the integer `p` in the above increases. In that case the factor M for the runtime of multiplication increases.)

To include the case of negative exponent, when a is a real number

$$a^{-n} = (a^{-1})^n$$

and for mod N, if a is invertible,

$$a^{-n} = (a^{-1})^n \pmod{N}$$

Exercise 1.5.1. Leetcode 50.
https://leetcode.com/problems/powx-n/
Implement `pow(x, n)`, which calculates `x` raised to the power `n`. (Go to solution, page 33) ☐

debug: exercises/rsa-15/question.tex

Exercise 1.5.2. * In the above, the computation of a^x depends on writing x is base 2. What if you write x in base 3? (Go to solution, page 34) ☐

debug: exercises/rsa-16/question.tex

Exercise 1.5.3. Implement an exponentiation function using the squaring method. Test it. After you are done, implement an exponentiation function in $\mathbb{Z}/N$ using the squaring method. (Go to solution, page 35) ☐

debug: exercises/rsa-17/question.tex

Solutions

Solution to Exercise 1.5.1.

debug: exercises/rsa-15/answer.tex

Solution not provided.

Solution to Exercise 1.5.2.

debug: exercises/rsa-16/answer.tex

Solution not provided.

Solution to Exercise 1.5.3.

debug: exercises/rsa-17/answer.tex

Solution not provided.

1.6 Inverse in modulo arithmetic debug: inverse-in-modular-arithmetic.tex

In the key generation for RSA, Bob has to compute the multiplicative inverse of $e \bmod \phi(n)$. This is just the Extended Euclidean Algorithm. (See previous notes).

1.7 The Prime Number Theorem and finding primes debug: primality-test.tex

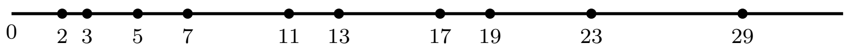
We'll need a way to create huge primes for RSA.

Of course you know Euclid's theorem that says there are infinitely many primes. So we don't have to worry about not finding them. But just because there are infinitely many primes, it does not mean they are everywhere!

Generally, we want to specify how large our primes should be. This is specified by the bit length of the primes. Given the length, one can generate a sequence of bits of that length. Of course that number should be odd. So the least significant bit is set to 1. Call this n. One can than check if n is a prime. If n is not prime, we try $n + 2$, etc.

Does this process take a long time?

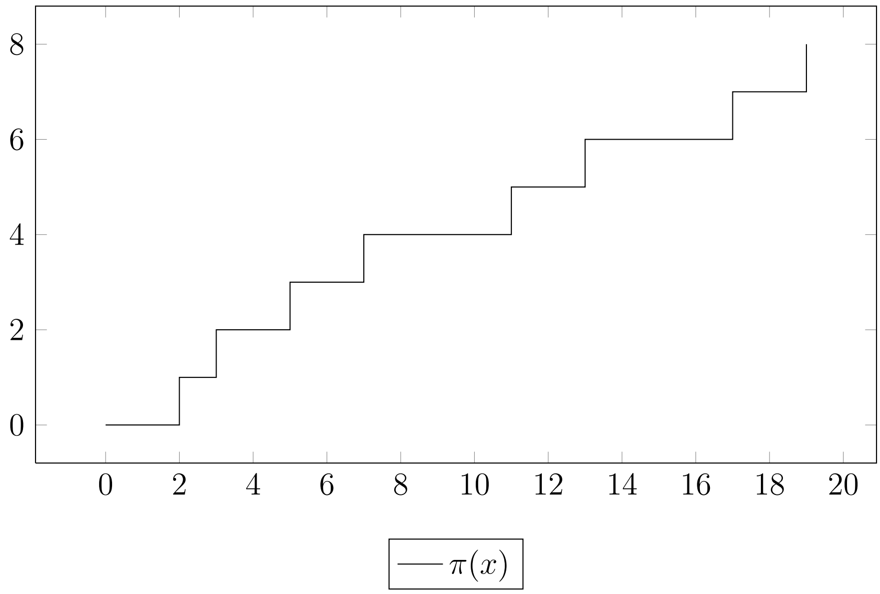
Gauss was the first to realize that even though primes seem to appear randomly on the real line,



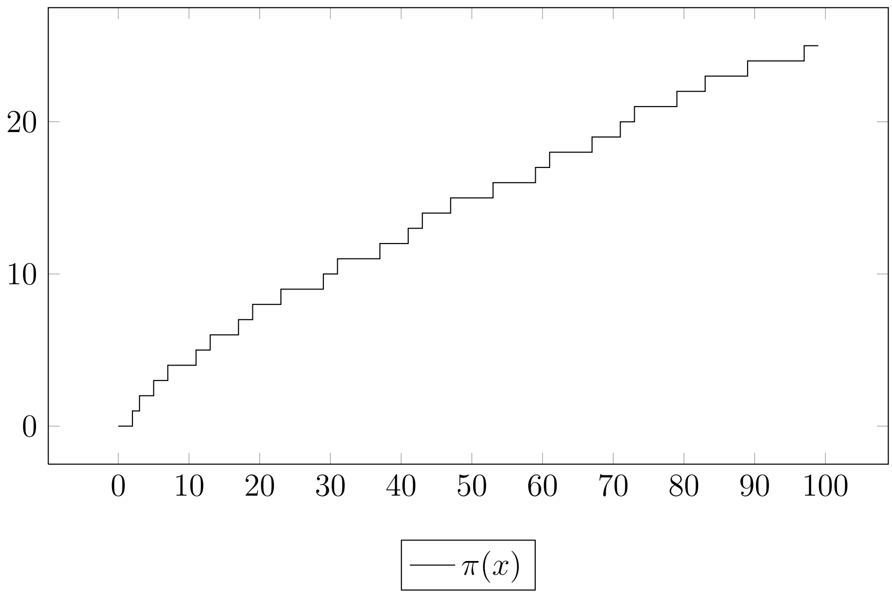
the density of primes seems to follow some law. The density of primes can be defined this way. Let $\pi(x)$ be the number of primes $\leq x$. Then the density of primes up to x is

$$\frac{\pi(x)}{x}$$

Here is the plot of $\pi(x)$ up to $x = 20$:



When the plot is up to $x = 100$ one begin to see that the rough and jagged graph begin to smooth out:



Through analyzing tables of primes, Gauss discovered that $\pi(x)/x$ is **asymptotically equivalent** to $1/\ln x$ ($\ln = \log_e$)

asymptotically equivalent

$$\frac{\pi(x)}{x} \sim \frac{1}{\ln x}$$

i.e.,

$$\lim_{x \to \infty} \frac{\pi(x)/x}{1/\ln x} = 1$$

Equivalently

$$\pi(x) \sim \frac{x}{\ln x}$$

Here a plot of $\pi(x)$ and $x/\ln x$ up to $x = 10^5$: [OMITTED]

The above was first conjectured by Gauss in 1792/3 and finally proven in 1896 by Hadamard and de la Vallée Poussin:

Theorem 1.7.1. (Prime Number Theorem)

Prime Number Theorem

$$\pi(x) \sim \frac{x}{\ln x}$$

Among number theorists and researchers in cryptography, the above deep result is known as **PNT**.

PNT

By PNT, when x is large, the density of primes up to x is approximately $1/\ln x$. Using the sieve of Eratosthenes, the number of primes up to $x = 10^5$ is 9592, i.e., $\pi(x)/x = 9592/100000 = 0.09592 = 9.592\%$ which is very close to $1/\ln x = 1/\ln 10^5 = 0.086858... = 8.6858...\%$. If we search for a prime only among *odd* integers, the chance of finding a prime is $2/\ln x = 0.173717... = 17.3717...\%$.

For modern-day RSA, primes used have approximately 1024-2048 bits. If we choose a bit length of 1024, then $2/\ln 2^{1024} = 0.1408...\%$. Therefore one might find a prime after < 1000 tries among odd integers. Usually one would begin with an integer n with a random sequence of 1024 bits, with least significant bit being 1 (so that n is odd). Then a primality test is used to check if n is prime. We'll see that a probabilistic primality test is used. If n is not a prime, one would then try $n + 2$. Etc.

Next we will look at two very important primality tests: Fermat primality test and Miller–Rabin primality test. Miller–Rabin primality test is the one that is used in the real world. However the main idea in Miller–Rabin primality test is actually Fermat primality test.

Exercise 1.7.1. Write a function `rand_odd_int` that accept L and return an

debug: exercises/rsa-05/question.tex

odd positive integer with L random bits. Hint: For python, try this:

```
n = int("0b111", 2) # the "0b" is optional
print(n)
n = int("0b100", 2)
print(n)
```

Try a few more examples to understand what is happening. (Go to solution, page 41) □

Exercise 1.7.2. Write a function `eratosthenes` that accepts an integer n and returns a bool array `isprime` of size n such that `isprime[i]` is True iff `i` is prime. (Go to solution, page 42) □

debug: exercises/rsa-06/question.tex

Exercise 1.7.3. Write a function `primes` that accepts x and returns an array of primes from 2 up to x (inclusive) in ascending order. For instance `primes(10)` return `[2, 3, 5, 7]`. (Go to solution, page 43) □

debug: exercises/rsa-07/question.tex

Exercise 1.7.4. Write a function `write_primes` that accepts `x` and a path `p` and store primes up to `x` at path `p` in comma-separated format. For instance `write_primes(10, 'primes-10.txt')` writes `"2,3,5,7"` to the file `primes-10.txt`. Write another function `read_primes` that accepts a path `p` and returns a list of primes stored at path `p`. Create a file of primes up to 10,000,000. After you are done with the above make a slight optimization by storing integer in hex. While a decimal (base-10 digit) can store 10 patterns, a hexadecimal can store 16. Try this:

debug: exercises/rsa-08/question.tex

```
i = int("0x1a", 16) # the "0x" is optional
print(i)
s = hex(i)
print(s)
```

(It's even better to store the integer directly in binary format, but that makes the file non-human readable.) (Go to solution, page 44) □

Exercise 1.7.5. Write a function `pi` that accept `x` and returns the number of primes up to `x` (inclusive). (Go to solution, page 45) □

debug: exercises/rsa-09/question.tex

Solutions

Solution to Exercise 1.7.1.

debug: exercises/rsa-05/answer.tex

Solution not provided.

Solution to Exercise 1.7.2.

debug: exercises/rsa-06/answer.tex

Solution not provided.

Solution to Exercise 1.7.3.

debug: exercises/rsa-07/answer.tex

Solution not provided.

Solution to Exercise 1.7.4.

debug: exercises/rsa-08/answer.tex

Solution not provided.

Solution to Exercise 1.7.5.

debug: exercises/rsa-09/answer.tex

Solution not provided.

1.8 Fermat primality test debug: fermat-primality-test.tex

Recall that Fermat's Little Theorem says that

$$p \text{ prime}, \ \ p \nmid a \implies a^{p-1} \equiv 1 \pmod p$$

I can also say that if $1 \leq a \leq p-1$,

$$p \text{ prime} \implies a^{p-1} \equiv 1 \pmod p$$

And of course we have: if $1 \leq a \leq p-1$,

$$p \text{ not prime} \impliedby a^{p-1} \not\equiv 1 \pmod p$$

Replacing "p" by "n", we have the following:

Proposition 1.8.1. (**Fermat compositeness test**) *If there is some a such that $1 \leq a \leq n-1$ and*

Fermat compositeness test

$$a^{n-1} \not\equiv 1 \pmod n$$

then n is composite.

Although the proposition above is true in general, when used in an algorithm, there are several cases that we want to remove.

1. Obviously $a = 1$ does not satisfy the hypothesis of the proposition. Therefore the interval for a should be $[2, n-1]$ instead of $[1, n-1]$.
2. Also, the hypothesis does not hold if $a = n-1$ and n is odd. For the case when n is even, either $n = 2$ or $n = 2k$ where $k > 1$. If the goal is to show n is composite, the case of $n = 2$ and $n = 2k$ where $k > 1$ can be checked quickly. Therefore the only useful scenario to use the proposition if now when n is odd and $a \in [2, n-2]$.
3. For the condition "$2 \leq a \leq n-2$" to have any a at at all, we need $2 \leq n-2$, i.e., $n \geq 4$.

Altogether the proposition is useful in an algorithm when $n > 3$ is odd and when $a \in [2, n-2]$.

```
ALGORITHM: Fermat-compositeness-test
INPUTS: n -- number to be tested for compositeness
        t -- number of tries
OUTPUT: "n is composite" if n is composite

if n <= 2: return "n is not composite"
```

```
if n % 2 is 0 and n / 2 > 1: return "n is composite"

for i = 1, 2, 3, ..., t:
    pick random a in [2, n - 2]
    if a^(n - 1) ≢ 1 (mod n) return "n is composite"

return "no conclusion"
```

Note that if you return `"no conclusion"`, it means either n is prime or n is composite but unfortunately your random a did not pick an a satisfying $a^{n-1} \not\equiv 1 \pmod{n}$.

Definition 1.8.1. Suppose n is composite and let $a \in [1, n-1]$. If a satisfies

$$a^{n-1} \not\equiv 1 \pmod{n}$$

then a is called a **Fermat witness** for the compositeness of n. Otherwise, if Fermat witness

$$a^{n-1} \equiv 1 \pmod{n}$$

then a is called a **Fermat liar**. The curious reason for calling such an a a liar will be clarified later. Fermat liar

Note that the above test tells you that n is composite *without* factoring n. Let's compare the above Fermat compositeness test to other compositeness test.

- Division compositeness test: Randomly pick a such that $2 \leq a \leq \lfloor \sqrt{n} \rfloor$. If a divides n, then n is a composite. If the chance of finding a is low, then one would have to run a through $[2, \lfloor \sqrt{n} \rfloor]$. In general, iteration rather than randomization is used.
- GCD compositeness test: Randomly pick a such that $2 \leq a \leq \lfloor \sqrt{n} \rfloor$, if $\gcd(a, n) > 1$, then n is composite. If the chance of finding a is low, then one would have to run a through $[2, \lfloor \sqrt{n} \rfloor]$.

Note that in both of these tests, a divisor of n is found. The speed of the above three compositeness tests depends on how fast we can find some a satisfying some condition in the test.

To better understand the condition

$$a^{n-1} \equiv 1 \pmod{n}$$

where $a \in [2, n-2]$, here's a table:

```
 3 1  None 1 ...
 4 0   0.0 0 ..0.
 5 1   1.0 1 ..11.
 6 0   0.0 0 ..000.
 7 1   1.0 1 ..1111.
 8 0   0.0 0 ..00000.
 9 0   0.0 0 ..000000.
10 0   0.0 0 ..0000000.
11 1   1.0 1 ..11111111.
12 0   0.0 0 ..000000000.
13 1   1.0 1 ..1111111111.
14 0   0.0 0 ..00000000000.
15 0  0.17 0 ..001000000100.
16 0   0.0 0 ..0000000000000.
17 1   1.0 1 ..11111111111111.
18 0   0.0 0 ..000000000000000.
19 1   1.0 1 ..1111111111111111.
20 0   0.0 0 ..00000000000000000.
21 0  0.11 0 ..000000100001000000.
22 0   0.0 0 ..0000000000000000000.
23 1   1.0 1 ..11111111111111111111.
24 0   0.0 0 ..000000000000000000000.
25 0  0.09 0 ..0000010000000000100000.
26 0   0.0 0 ..00000000000000000000000.
27 0   0.0 0 ..000000000000000000000000.
28 0  0.08 0 ..0000000100000000000000010.
29 1   1.0 1 ..11111111111111111111111111.
30 0   0.0 0 ..000000000000000000000000000.
31 1   1.0 1 ..1111111111111111111111111111.
32 0   0.0 0 ..00000000000000000000000000000.
33 0  0.07 0 ..000000001000000000000100000000.
34 0   0.0 0 ..0000000000000000000000000000000.
35 0  0.06 0 ..00001000000000000000000000010000.
36 0   0.0 0 ..000000000000000000000000000000000.
37 1   1.0 1 ..1111111111111111111111111111111111.
38 0   0.0 0 ..00000000000000000000000000000000000.
39 0  0.06 0 ..000000000000100000000001000000000000.
40 0   0.0 0 ..0000000000000000000000000000000000000.
41 1   1.0 1 ..11111111111111111111111111111111111111.
42 0   0.0 0 ..000000000000000000000000000000000000000.
43 1   1.0 1 ..1111111111111111111111111111111111111111.
44 0   0.0 0 ..00000000000000000000000000000000000000000.
45 0  0.14 0 ..000000100000000101000000101000000001000000.
46 0   0.0 0 ..0000000000000000000000000000000000000000000.
47 1   1.0 1 ..11111111111111111111111111111111111111111111.
48 0   0.0 0 ..000000000000000000000000000000000000000000000.
49 0  0.09 0 ..0000000000000000110000000000110000000000000000.
50 0   0.0 0 ..00000000000000000000000000000000000000000000000.
```

Here are the descriptions of the columns:

- First column: n
- Third column: 1–prime, 0–composite
- Fifth column: sequence of boolean values for $a^{n-1} \equiv 1 \pmod{n}$ where $a = 0, 1, 2, ..., n-1$, except that if a is not in $[2, n-2]$, the value is "." to indicate "not applicable". The "not applicable" values are $a = 0, 1, n-1$.
- Second column: The percentage of "1"s in the fourth column (not counting the "not applicable" cases).
- Third column: 1–percentage of witnesses is 0%, 0–otherwise.
- Fourth column: 1–if values in fifth column are all 1. (Ignore this for now.)

Of course when n is prime, the sequence of boolean values are all "1"s. Now let us focus on the composite n's.

For $n = 15$:

```
15 0  0.17 0 ..001000000100.
```

there are 10 Fermat witnesses for the compositeness of 15 (i.e., 2, 3, 5, 6, 7, 8, 9, 10, 12, 13) and 2 Fermat liars (i.e., 4, 11). For instance 2 is a Fermat witness since

$$2^{15-1} = 2^{14} = (2^4)^3 \cdot 2^2 = (16)^3 \cdot 4 \equiv 1^3 \cdot 4 = 4 \not\equiv 1 \pmod{16}$$

and 4 is a Fermat liar since

$$4^{15-1} = 4^{14} = (4^2)^7 = (16)^7 \equiv 1^7 = 1 \pmod{16}$$

The percentage of Fermat liars (among integers $a \in [2, 13]$) is 17%. Therefore there are more Fermat witnesses than Fermat liars. In fact, when n is composite, the fact that there are more Fermat witnesses than not seems to be very common. The first time the percentage of Fermat liars is $> 50\%$ is when $n = 561$ (at 57%).

This means that when n is composite and we randomly pick an a such that $a \in [2, n-2]$, the chance that a is a Fermat witness to n's compositeness seems be to very high. The percentage of Fermat witnesses will be quantified later.

Exercise 1.8.1. Write a program that produces the above table, say up to $n = 10000$. (Go to solution, page 62) □ debug: exercises/rsa-18/question.tex

Compared to the brute force division compositeness test, suppose $n = pq$ with $p < q$, then in $[2, \lfloor\sqrt{n}\rfloor]$, there's only one divisor. For instance if $n = 11 \cdot 13 = 143$, $[2, \lfloor\sqrt{n}\rfloor] = [2, 11]$. and the only divisor of n in this interval is 11. The percentage of finding a divisor (with one try) in this case is 10%. The chance of finding some a such that $\gcd(a, n) > 1$ is the same. If $n = pq$ where q is much larger than p, then the GCD compositeness test has a better change of finding a good a, because $a = p, 2p, 3p, ...kp$ would work where kp is the largest integer $\leq \sqrt{pq}$. For instance is $n = 2 \cdot 13$, then $a = 2, 4 \leq \left\lfloor\sqrt{2 \cdot 13}\right\rfloor = 5$ works so the number of good as is $2/4 = 50\%$. However if you look at the table above, you will see that for $n = 26$, the percentage of Fermat witness is 100%.

For Fermat compositeness test, from the above table, the entry for $n = 143$ is

```
143 0  0.01 0 ..0000000000100000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000100
00000000.
```

There's a 99% chance of finding a Fermat witness (with one try).

Exercise 1.8.2. Let $F_n = 2^{2^n} + 1$. These numbers are called **Fermat numbers** (not Fibonacci numbers). For $n = 5$, $F_5 = 2^{2^5} + 1 = 4294967297$. Fermat thought F_5 is prime, but it is not. (The first 5 Fermat numbers $F_0, F_1, ..., F_4$ are prime.) Euler was the first to realize that F_5 is not prime. Of course neither Fermat nor Euler had computers. Prove that F_5 is not a prime in two ways:

debug: exercises/rsa-19/question.tex
Fermat numbers

(a) Find the smallest Fermat witness for F_5.
(b) Find a prime divisor of F_5 by brute force division.

(Go to solution, page 63) □

Exercise 1.8.3. Is 2 a Fermat witness for $F_{14} = 2^{2^{14}} + 1$? What about 3? Can you find a divisor of F_{14}? The fact that F_{14} is composite was first discovered in 1961. It took 50 years before a nontrivial 54-digit divisor of F_{14}, 116928085873074369829035993834596371340386703423373313, was found. See https://t5k.org/mersenne/LukeMirror/lit/lit_039s.htm. I think the complete factorization of F_{14} is still unknown. See http://www.prothsearch.com/fermat.html.

debug: exercises/rsa-20/question.tex

(Go to solution, page 64) □

The above Fermat compositeness test tells you when n is a composite. It does not tell you if n is a prime. One can ask if the following is true:

$$a^{n-1} \equiv 1 \pmod{n} \implies n \text{ is prime}$$

if n is odd and $a \in [2, n-2]$. This is clearly not true. For instance in the above data for $n = 143$, you see that there are two Fermat liars, the first being $a = 12$:

$$12^{143-1} = 12^{142} = (12^2)^{71} = 144^{71} \equiv 1^{71} = 1 \pmod{143}$$

The first composite n to have a Fermat liar is $n = 15$, where the liar is $a = 4$:

```
15 0  0.17 0 ..001000000100.
```

$$4^{15-1} = 4^{14} = 16^7 \equiv 1^7 \equiv 1 \pmod{15}$$

In this case, we say that 15 is a **Fermat pseudoprime** and 4 is a **Fermat liar** for 15. (Sometimes 4 is called a **base** for 15.) Why is 15 called a pseudoprime and 4 is a liar? The fact

Fermat liar
Fermat pseudoprime
base

$$4^{15-1} \equiv 1 \pmod{15}$$

is very similar to Fermat Little Theorem where the modulus is a prime p and $p \nmid a$:

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

4 and 15 are trying to trick you into thinking that 15 is like a prime.

Note that if a is a Fermat liar, then it is invertible:

Proposition 1.8.2.

(a) *If a is a Fermat liar for composite n, then a is invertible mod n, i.e.,* $\gcd(a, n) = 1$.
(b) *Therefore if* $\gcd(a, n) > 1$, *then a cannot be a Fermat liar. In other words, if* $\gcd(a, n) > 1$, *then a is a Fermat witness.*

Proof. (a) TODO

(b) This follows from (a). □

Therefore for $a \in [0, n-1]$ (although for compositeness testing we are only interested in the case when $n > 3$ is odd and $a \in [2, n-2]$):

1. a is a Fermat witness and $\gcd(a, n) = 1$
2. a is a Fermat witness and $\gcd(a, n) > 1$
3. a is a Fermat liar (and in this case $\gcd(a, n) = 1$)

So in the above table, to be "fair" to Fermat liar, if one is interested in comparing the number of Fermat liars against Fermat witnesses for n, sometimes one can temporarily ignore the a such that $\gcd(a, n) > 1$. If we do this, then there are composite numbers n where *every* a in $[2, n-2]$ such that $\gcd(a, n) = 1$ is a Fermat liar for n. Such an n is called a **Carmichael number**. Carmichael number

Let me collect all these definitions below.

Definition 1.8.2. Let n be a composite. Then

(a) n is a **Fermat pseudoprime** if n is not a prime and if there is some $a \in [2, n-2]$ such that Fermat pseudoprime

$$a^{n-1} \equiv 1 \pmod n$$

In this case, we say that a is a **Fermat liar** for n. (a is also called a **base** for n.) Fermat liar base

(b) n is a **Carmichael number** if every $a \in [2, n-2]$ such that $\gcd(a, n) = 1$ is a Fermat liar for n, i.e., for all such a, Carmichael number

$$a^{n-1} \equiv 1 \pmod n$$

The following is similar to the earlier table:

```
  3 1  None 1 ...
  4 0  None 1 ....
  5 1   1.0 1 ..11.
  6 0  None 1 ......
  7 1   1.0 1 ..1111.
  8 0   0.0 0 ...0.0..
  9 0   0.0 0 ..0.00.0.
 10 0   0.0 0 ...0...0..
 11 1   1.0 1 ..11111111.
 12 0   0.0 0 .....0.0....
 13 1   1.0 1 ..1111111111.
 14 0   0.0 0 ...0.0...0.0..
 15 0  0.33 0 ..0.1..00..1.0.
 16 0   0.0 0 ...0.0.0.0.0.0..
 17 1   1.0 1 ..11111111111111.
```

```
  18 0   0.0 0 .....0.0...0.0....
  19 1   1.0 1 ..1111111111111111.
  20 0   0.0 0 ...0...0.0.0.0...0..
  21 0   0.2 0 ..0.00..1.00.1..00.0.
  22 0   0.0 0 ...0.0.0.0...0.0.0.0..
  23 1   1.0 1 ..11111111111111111111.
  24 0   0.0 0 .....0.0...0.0...0.0....
  25 0  0.11 0 ..000.0100.0000.0010.000.
  26 0   0.0 0 ...0.0.0.0.0...0.0.0.0.0..
  27 0   0.0 0 ..0.00.00.00.00.00.00.00.0.
  28 0   0.2 0 ...0.0...1.0.0.0.0.0...0.1..
  29 1   1.0 1 ..11111111111111111111111111.
  30 0   0.0 0 .......0...0.0...0.0...0......
  31 1   1.0 1 ..1111111111111111111111111111.
  32 0   0.0 0 ...0.0.0.0.0.0.0.0.0.0.0.0.0.0..
  33 0  0.11 0 ..0.00.00.1..00.00.00..1.00.00.0.
  34 0   0.0 0 ...0.0.0.0.0.0.0...0.0.0.0.0.0.0..
  35 0  0.09 0 ..000.1.00.000..0000..000.00.1.000.
  36 0   0.0 0 .....0.0...0.0...0.0...0.0...0.0....
  37 1   1.0 1 ..1111111111111111111111111111111111.
  38 0   0.0 0 ...0.0.0.0.0.0.0.0...0.0.0.0.0.0.0.0..
  39 0  0.09 0 ..0.00.00.00..1.00.00.00.1..00.00.00.0.
  40 0   0.0 0 ...0...0.0.0.0...0.0.0.0...0.0.0.0...0..
  41 1   1.0 1 ..11111111111111111111111111111111111111.
  42 0   0.0 0 .....0.....0.0...0.0...0.0...0.0.....0....
  43 1   1.0 1 ..1111111111111111111111111111111111111111.
  44 0   0.0 0 ...0.0.0.0...0.0.0.0.0.0.0.0.0.0...0.0.0.0..
  45 0  0.27 0 ..0.0..01..0.00.01.1..00..1.10.00.0..10..0.0.
  46 0   0.0 0 ...0.0.0.0.0.0.0.0.0.0...0.0.0.0.0.0.0.0.0.0..
  47 1   1.0 1 ..11111111111111111111111111111111111111111111.
  48 0   0.0 0 .....0.0...0.0...0.0...0.0...0.0...0.0...0.0....
  49 0   0.1 0 ..00000.000000.000110.000000.011000.000000.00000.
  50 0   0.0 0 ...0...0.0.0.0...0.0.0.0...0.0.0.0...0.0.0.0...0..
```

except that for the fifth column, an entry of "." is used to indicate "not applicable", where "not applicable" is when $a \in \{0, 1, n-1\}$ or when $\gcd(a, n) > 1$. The fourth column is 1 if all values of $a \in [2, n-1]$ satisfying $\gcd(a, n) = 1$ are Fermat liars. In this case n is a Carmichael number and the value for the third column (percentage of Fermat liars) is $1.0 = 100\%$.

Exercise 1.8.4. Modify your program to produce the above output. (Go to solution, page 65) ☐

debug: exercises/rsa-21/question.tex

The first Carmichael number is 561. Here's the row for 561 from the table:

```
561 0   1.0 1 ..1.11.11.1..11.1..11..1.11.11.11..1.11.11.1..11.11.11..1.11
```

```
.11.11.1..11.11.1..11.11..1..1.11.11.11.11.11.11.1..11.11.1...1.11.11.11.1
1..1.11.1..11.11.11..1.11.11.11.11.1..11.1..11.11.11..1.11.11.11.11.11.11.
1..11.11.11....11.11.11.11.11..1.1..11.11.11..1.11.11.11.11.11.1..1..11.11
.11..1..1.11.11.11.11.11.1..11.11.11..1.1..11.11.11.11.11....11.11.11..1.1
1.11.11.11.11.11.1..11.11.11..1.11..1.11.11.11.11.1..11.11.11..1.11.1..11.
11.11.11.1...1.11.11..1.11.11.11.11.11.11.1..1..11.11..1.11.11..1.11.11.11
.1..11.11.11..1.11.11.1..11.11.11.1..11..1.11..1.11.11.1.
```

The concept of Carmichael numbers was named after Carmichael because of a 1910 paper he wrote. However the defining property of a Carmichael number was studied as far back as at least 1885. For instance the first 6 Carmichael numbers (including 561) were first discovered by Šimerka in 1885. In his 1910 paper, Carmichael conjectured that there are infinitely many Carmichael numbers. This was not proven until 1994 (see `https://math.dartmouth.edu/~carlp/PDF/paper95.pdf`). Here's the wikipedia entry for Carmichael numbers.

Exercise 1.8.5. Šimerka was the first to discover the first 7 Carmichael numbers. Beat him by finding the first 8. To check your work, you can go to Carmichael numbers and look for the first 7 Carmichael numbers discovered by Šimerka. (Go to solution, page 66) □

debug: exercises/rsa-22/question.tex

The above gives us a probabilistic primality test: if $n > 3$ and we pick $a \in [2, n-2]$ and we have

$$a^{n-1} \equiv 1 \pmod{n}$$

then n is "probably a prime". Why? Because

$$a^{n-1} \equiv 1 \pmod{n}$$

can occur in two ways:

1. n is a prime (the fact that $n \nmid a$ is guaranteed by the fact that n is prime and $a \in [2, n-2]$)
2. n is a Fermat pseudoprime and a is Fermat liar for n

From the first table, we see that most of the "1" occurs when n is a prime. For a fixed n, the more a's we try, the higher the probability that n is prime, unless of course our n has lots of liars. In particular, in the worse scenario, if n is a Carmichael number, the test will say n is very likely a prime when in fact it's not.

Combining the Fermat compositeness test with the "probably prime test", we now have:

Fermat Primality Test

Proposition 1.8.3. (Fermat Primality Test) *If there is some a such that $1 \le a \le n-1$ and*

$$a^{n-1} \not\equiv 1 \pmod{n}$$

then n is composite. If there is some randomly chosen $a \in [2, n-2]$ such that

$$a^{n-1} \equiv 1 \pmod{n}$$

then n is "probably a prime".

Here's the Fermat primality test algorithm:

```
ALGORITHM: Fermat-primality-test
INPUTS: n -- number to be tested for compositeness/primeness. Assume n >= 2.
        t -- number of tries
OUTPUT: "n is composite" or "n is probably prime" (after t tries)

if n is 2: return "n is prime"
if n % 2 is 0 and n / 2 > 1: return "n is composite"

for i = 1, 2, 3, ..., t:
    pick random a in [2, n - 2]
    if a^(n - 1) ≢ 1 (mod n) return "n is composite"

return "n is probably prime"
```

Note that we have not included the handling of $n \le 1$. In general, primality tests are for handling cases when n is large. We have added a note that n is assumed to be ≥ 2.

Note that the first 2 checks (i.e., n is even) can be omitted:

```
ALGORITHM: Fermat-primality-test
INPUTS: n -- number to be tested for compositeness/primeness. Assume n >= 2.
        t -- number of tries
OUTPUT: "n is composite" or "n is probably prime" (after t tries)

for i = 1, 2, 3, ..., t:
    pick random a in [2, n - 2]
    if a^(n - 1) ≢ 1 (mod n) return "n is composite"

return "n is probably prime"
```

Why? Because if $n = 2$, then $[2, n-2]$ is empty, which means that `"n is probably prime"` is returned. And if $n = 2k$ where $k > 1$, then $\gcd(a, n) > 1$ whenever a is even or when $a = k$. This means that more than half of the values in $[2, n-2]$ are Fermat witnesses, so there is a strong likeliheed that `"n is composite"` will be returned especially if $t > 1$.

Note that when given an integer $n \geq 0$, either n is a prime or it is not. The statement "n is probably a prime" should actually be "there's probably a value in $[2, n-2]$ satisfying some condition". But the phrase "n is probably a prime" has been in use for a long time and it's hard to break the usage.

When comparing number of Fermat witnesses against Fermat liar, we can be a little bit more precise:

Proposition 1.8.4. *Let n be composite. If there is a Fermat witness w for n such that* $\gcd(w, n) = 1$*, then*

$$|\{a \in [1, n-1] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-1}{2}$$

Proof. TODO ☐

In the above proposition, the a is an integer in $[1, n-1]$. Remember that we usually test $a \in [2, n-2]$. Also, we usually assume $n > 3$ is odd. Note that $1^{n-1} \equiv 1 \pmod{n}$. Hence the above proposition implies

$$|\{a \in [2, n-1] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-1}{2}$$

If n is odd. Then $n-1$ is even and $(n-1)^{n-1} \equiv (-1)^{n-1} \equiv 1 \pmod{n}$. Hence

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-1}{2}$$

Also, $(n-1)/2$ is an integer. Therefore

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| \geq \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

The number of integer in $[2, n-2]$ is $n-2-1 = n-3$. Therefore $> 50\%$ of the values in $[2, n-2]$ are Fermat witnesses. If n is even. Then $n-1$ is odd and $(n-1)^{n-1} \equiv (-1)^{n-1} \equiv -1 \pmod{n}$.

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| + 1 > \frac{n-1}{2}$$

In this case $(n-1)/2$ is not an integer.

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod n\}| + 1 \geq \frac{n-1}{2} + \frac{1}{2}$$

i.e.,

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod n\}| \geq \frac{n-1}{2} - \frac{1}{2} = \frac{n-2}{2}$$

Together,

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod n\}| > \begin{cases} \dfrac{n+1}{2} & \text{if } n \text{ is odd} \\ \dfrac{n-2}{2} & \text{if } n \text{ is even} \end{cases}$$

In both of the above cases

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod n\}| > \frac{n-3}{2}$$

i.e., more than 1/2 of the values in $[2, n-2]$ are witnesses.

Suppose p is the probability of not finding a Fermat witness a in $[2, n-2]$ for n. Assuming there is a Fermat witness a with $\gcd(a, n) = 1$. From the above, more than half of the a in $[2, n-2]$ are Fermat witnesses for n. This means that $p < 0.5$. The probability of not finding a Fermat witness after t tries is

$$p^t$$

Therefore the probability of finding a Fermat witness is

$$1 - p^t$$

For instance assuming $p = 0.5$, then $t = 8$ tries, if no Fermat witness is found, the probability that n is prime is

$$1 - p^t > 1 - 0.5^8 = 0.99609375$$

If $t = 10$, we have

$$1 - p^t > 1 - 0.5^{10} = 0.9990234375$$

And if $t = 20$, we reach

$$1 - p^t > 1 - 0.5^{10} = 0.9999990463256836$$

Of course all the above is based on the assumption that there is a Fermat witness a such that $\gcd(a, n) = 1$. You are out of luck if n is a Carmichael number. You can think of Carmichael numbers as extreme failure cases of Fermat primality test.

The question is how common are Carmichael numbers? Carmichael numbers are very rare. At this point, we know that the density of Carmichael numbers is about 1 in 50 trillion $= 5 \times 10^{13}$.

Example 1.8.1. In the following, we obtain a random integer with 10 digits and found a Fermat witness with one try:

```
import random; random.seed()
d = 10
n = random.randrange(10**(d - 1), 10**d)
print(n)
for i in range(20):
    a = random.randrange(2, n - 1)
    b = (pow(a, n - 1, n) != 1)
    print(i, a, b)
    if b: break
```

The output is

```
1151731626
0 869958907 True
```

Fermat primality test says that 1151731626 is composite with only one try. In fact $1151731626 = 2 \cdot 3 \cdot 19 \cdot 10102909$.

Example 1.8.2. Here's another run of the above code where no Fermat witness was found after 20 tries:

```
3584990077
0 3295070400 False
1 3215421426 False
2 262972142 False
3 1050903352 False
4 152804132 False
5 1015451650 False
6 885960417 False
7 720561088 False
8 1694137561 False
9 2065337998 False
10 3345601994 False
```

```
11 1535607663 False
12 3183174792 False
13 1772850385 False
14 2569697199 False
15 948739551 False
16 2148646472 False
17 1640965445 False
18 2024258764 False
19 3229113680 False
```

No Fermat witness was found after 20 tries. Fermat primality test would return "probably a prime". In fact 3584990077 is prime. In this case 3584990077 is small enough that a brute force prime testing by brute force division can be used.

Exercise 1.8.6. Let $n = 18801105946394093459$. Prove that n is composite in three ways: debug: exercises/rsa-23/question.tex

(a) Use division compositeness test. First try to randomly generate a potential divisor and test it. If it fails, try to do a brute force iteration from 2 to $\sqrt{n}$ to locate a divisor.
(b) Use GCD compositeness test. First try to randomly generate a potential a and test if $\gcd(a, n) > 1$. If it fails, try to do a brute force iterate a from 2 to $\sqrt{n}$ and test if $\gcd(a, n) > 1$.
(c) Use Fermat primality test with $t = 1$.

(Go to solution, page 67) □

Exercise 1.8.7. Let $n = 58645563317564309847334478714939069495243200674793$. Prove that n is prime or probably prime in three ways: debug: exercises/rsa-24/question.tex

(a) Use division compositeness test. Do a brute force iteration from 2 to $\sqrt{n}$ to locate a potential divisor.
(b) Use GCD compositeness test. Iterate a from 2 to $\sqrt{n}$ and test if $\gcd(a, n) > 1$.
(c) Use Fermat primality test with $t = 10$.

(Go to solution, page 68) □

Exercise 1.8.8. How often is 2 a liar? Write a program to print all composite n such that 2 lies for n. Do you think 2 lie for infinitely many Fermat debug: exercises/rsa-25/question.tex pseudoprimes? Can you prove it? Compare your experiment with 3, say up to $n = 100000$, how often does 3 lie for n when compared to 2?

(Go to solution, page 69) $\square$

The following is a major theorem on Carmichael numbers:

Theorem 1.8.1. (**Koselt's Theorem** 1899) *Let n be a positive integer. Then n is a Carmichael number iff n is odd, squarefree, and if $p \mid n$, then $p-1 \mid n-1$.* Koselt's Theorem

Proof. Omitted. $\square$

The conditions on n in the above theorem is frequently called **Koselt's criterion**. Koselt's criterion

Example 1.8.3. Consider 561. From the above second table, we know that 561 is a Carmichael. Also, as mentioned earlier Šimerka was the first to discover this Carmichael number. 561 is small and can be easily factorized: $561 = 3 \cdot 11 \cdot 17$. Note that 561 is odd and squarefree. Furthermore $2 \mid 560$, $10 \mid 560$, and $16 \mid 560$. Therefore 561 satisfies Koselt's criteria and therefore 561 is a Carmichael number.

Exercise 1.8.9. Show that 1729 is a Carmichael number. (Go to solution, page 70) $\square$ debug: exercises/rsa-26/question.tex

Although Koselt proved his theorem in 1899, he never mention any Carmichael number in his publications.

Corollary 1.8.1. (Cernick 1939) *If $6k+1$, $12k+1$, and $18k+1$ are all primes, then the product $(6k+1)(12k+1)(18k+1)$ is a Carmichael number.*

Proof. Let $n = (6k+1)(12k+1)(18k+1) = 6 \cdot 12 \cdot 18 \cdot k^3 + (6 \cdot 12 + 6 \cdot 18 + 12 \cdot 18)k^2 + (6+12+18)k + 1$. n is clearly odd and since n is a product of three distinct primes, $4n$ is clearly squarefree. Note that $n-1 = (6k+1)(12k+1)(18k+1) = 6 \cdot 12 \cdot 18 \cdot k^3 + (6 \cdot 12 + 6 \cdot 18 + 12 \cdot 18)k^2 + (6+12+18)k$. Clearly $6k \mid n-1$, $12k \mid n-1$, $18k \mid n-1$. Therefore by Korselt's Theorem, n is a Carmichael number. $\square$

Exercise 1.8.10. Use Corollary 1.8.1 to show that 1729 is a Carmichael number. (Go to solution, page 71) □

debug: exercises/rsa-27/question.tex

Exercise 1.8.11. Use Corollary 1.8.1 to find your own Carmichael number. Check what you have discovered with known Carmichael numbers on the web. (Go to solution, page 72) □

debug: exercises/rsa-28/question.tex

Exercise 1.8.12. Implement Fermat's primality test. (Go to solution, page 73) □

debug: exercises/rsa-29/question.tex

Solutions

Solution to Exercise 1.8.1.

debug: exercises/rsa-18/answer.tex

Solution not provided.

Solution to Exercise 1.8.2.

debug: exercises/rsa-19/answer.tex

Solution not provided.

Solution to Exercise 1.8.3.

debug: exercises/rsa-20/answer.tex

Solution not provided.

Solution to Exercise 1.8.4.

debug: exercises/rsa-21/answer.tex

Solution not provided.

Solution to Exercise 1.8.5.

debug: exercises/rsa-22/answer.tex

Solution not provided.

Solution to Exercise 1.8.6.

debug: exercises/rsa-23/answer.tex

Solution not provided.

Solution to Exercise 1.8.7.

debug: exercises/rsa-24/answer.tex

Solution not provided.

Solution to Exercise 1.8.8.

debug: exercises/rsa-25/answer.tex

Solution not provided.

Solution to Exercise 1.8.9.

debug: exercises/rsa-26/answer.tex

Solution not provided.

Solution to Exercise 1.8.10.

debug: exercises/rsa-27/answer.tex

Solution not provided.

Solution to Exercise 1.8.11.

debug: exercises/rsa-28/answer.tex

Solution not provided.

Solution to Exercise 1.8.12.

debug: exercises/rsa-29/answer.tex

Solution not provided.

1.9 Miller-Rabin primality test debug: miller-rabin-primality-test.tex

Fermat Little Theorem says this: Let $a \in [2, n-2]$.

$$n \text{ is prime} \implies a^{n-1} \equiv 1 \pmod{n}$$

The basis of Fermat primality test is

$$n \text{ is not prime} \Longleftarrow a^{n-1} \not\equiv 1 \pmod{n}$$

We can a say bit more. Again we have

$$n \text{ is prime} \implies a^{n-1} \equiv 1 \pmod{n}$$

Suppose $n - 1 = 2^k m$ with $k \geq 0$ and $2 \nmid m$. Then we have

$$n \text{ is prime} \implies a^{2^k m} \equiv 1 \pmod{n}$$

Now note that

$$a^{2^k m} = (a^m)^{2^k}$$

This can be computed as a sequence of squares. For instance if $k = 3$, then

$$a^{2^3 m} = (a^m)^{2^3} = (((a^m)^2)^2)^2$$

Note that following fact:

Proposition 1.9.1. *If p is a prime and $x^2 \equiv 1 \pmod{p}$ then $x \equiv \pm 1 \pmod{p}$.*

Proof. TODO □

Applying this proposition to $x^2 = (a^m)^{2^k}$, we have the following fact: if n is prime, and $k > 0$, then

$$(a^m)^{2^k} \equiv 1 \pmod{n} \implies (a^m)^{2^{k-1}} \equiv \pm 1 \pmod{n}$$

And if

$$(a^m)^{2^{k-1}} \equiv 1 \pmod{n}$$

then

$$(a^m)^{2^{k-2}} \equiv \pm 1 \pmod{n}$$

Etc.

All in all, assuming n is prime, writing $n-1$ as $2^k m$ where 2^k is the highest power of 2 dividing $n-1$, then the sequence

$$\begin{aligned}
&(a^m)^{2^0} \pmod{n} \\
&(a^m)^{2^1} \pmod{n} \\
&(a^m)^{2^2} \pmod{n} \\
&\vdots \\
&(a^m)^{2^{k-2}} \pmod{n} \\
&(a^m)^{2^{k-1}} \pmod{n} \\
&(a^m)^{2^k} \pmod{n}
\end{aligned}$$

either the whole sequence is 1s or it ends with a sequence of 1s (of length ≥ 1) and before this sequence there is a -1. For instance if $k = 5$, the above sequence is a sequence of 6 numbers and here are some possibilities:

- The last three numbers might be $-1, 1, 1 \pmod{n}$ (the first three are not -1 and not 1).
- Or the last two might be $-1, 1 \pmod{n}$ (the first four are not -1 and not 1).
- Or all 6 might be $1, 1, 1, 1, 1, 1 \pmod{n}$.

Of course if $k = 0$, then there is only one number in the sequence and that number is $1 \pmod{n}$. In general, the above is a sequence of $k+1$ numbers and ends with 1s of length ≥ 1 or ends with -1 followed by 1s of length ≥ 1.

Miller-Rabin primality test is similar to Fermat prime test. For an integer n, we compute m and k such that $n - 1 = 2^k \cdot m$. We randomly pick an a in $[2, n-2]$ look at the values

$$\begin{aligned}
&a^m \pmod{n} \\
&a^{2m} \pmod{n} \\
&a^{2^2 m} \pmod{n} \\
&a^{2^3 m} \pmod{n} \\
&\vdots \\
&a^{2^{k-2} m} \pmod{n} \\
&a^{2^{k-1} m} \pmod{n} \\
&a^{2^k m} \pmod{n}
\end{aligned}$$

and if they are all 1s (i.e., the first is 1) or ends with $-1, 1, 1, ...1$, the algorithm (i.e., there's a -1) returns `"n is probably prime"`. Otherwise it returns `"n is composite"`. The difference is Fermat primality test only looks at the last value.

Example 1.9.1. As an example, note that $n = 561$ is a Carmichael number. Fermat primality test reports n as probably prime even though it is a composite. Using Miller-Rabin primality test, first $n - 1 = 560 = 35 \cdot 2^4$. Let us use $a = 2$.

$$\begin{aligned}
2^{35} &\equiv 263 \pmod{561} \\
(2^{35})^2 &\equiv (263)^2 \equiv 166 \pmod{561} \\
(2^{35})^{2^2} &\equiv (166)^2 \equiv 67 \pmod{561} \\
(2^{35})^{2^3} &\equiv (67)^2 \equiv 1 \pmod{561} \\
(2^{35})^{2^4} &\equiv (1)^2 \equiv 1 \pmod{561}
\end{aligned}$$

The sequence is $263, 166, 67, 1, 1$. And we see that n cannot be prime, because if n is prime, from line 3 above

$$(2^{35})^{2^3} \equiv 1 \pmod{561}$$

the previous line should have been $\pm 1 \pmod{561}$, but it is not. Therefore Miller-Rabin primality test will report 561 as composite.

One can define Miller-Rabin pseudoprimes (usually called **strong pseudoprimes**), Miller-Rabin witness, and Miller-Rabin liar. While the failure case of Fermat primality test is on the average less than 1/2, the failure case of Miller-Rabin is less than 1/4. strong pseudoprimes

Here's the Miller-Rabin primality test algorithm:

```
ALGORITHM: Miller-Rabin-primality-test
INPUTS: n -- integer to be tested for primeness/compositeness
        t -- number of passes

compute k and odd m such that n - 1 = 2^k * m
for pass = 1, 2, 3, ..., t:
    randomly select a in [2, n - 2]
    if Miller-Rabin-one-pass(n, k, m, a) returns "n is composite":
        return "n is a composite"
return "n is probably a prime"
```

```
ALGORITHM: Mill-Rabin-one-pass
INPUTS: n, k, m where n - 1 = 2^k * m
        a
OUTPUT: "n is composite" or "n is probably prime"

let b ≡ a^m (mod n)
if b ≡ 1 (mod n):
    return "n is probably prime"
for i = 0, 1, 2, ..., k - 1:
    if b ≡ -1 (mod n):
        return "n is probably prime"
    b ≡ b^2 (mod n)
return "n is composite"
```

Exercise 1.9.1. Are the Miller-Rabin computations for the case of $a = 50$ and $n = 561$? What does Miller-Rabin conclude in this case? (Go to solution, page 78) □

debug: exercises/rsa-00/question.tex

Exercise 1.9.2. Compare the results from Fermat primality test and Miller–Rabin primality test for $n = 1729$. (Go to solution, page 79) □

debug: exercises/rsa-01/question.tex

Exercise 1.9.3. Implement the Miller–Rabin primality test algorithm. (Go to solution, page 80) □

debug: exercises/rsa-02/question.tex

Solutions

Solution to Exercise 1.9.1.

debug: exercises/rsa-00/answer.tex

Solution not provided.

Solution to Exercise 1.9.2.

debug: exercises/rsa-01/answer.tex

Solution not provided.

Solution to Exercise 1.9.3.

debug: exercises/rsa-02/answer.tex

Solution not provided.

1.10 Monte-Carlo algorithms debug: monte-carlo.tex

As noted earlier, Fermat and Miller–Rabin prime testing algorithm are probabilitic algorithm in the sense that if the return value is `"n is composite"` then you know for sure n is a composite (i.e. not a prime), but if the return value is `"n is probably a prime"`, then n can be either be a prime or a composite. This is also called a **Monte-Carlo algorithm** because the result returned is not guaranteed to be true. It's also called a **false-biased Monte-Carlo algorithm** because the false case (i.e., `"not a prime"` return value) is always correct whereas the true case (i.e., is a prime) is only probabilistically true.

Monte-Carlo algorithm

false-biased Monte-Carlo algorithm

There are actually many primality testing algorithms. Rabin-Miller is only one of many.

In number theory, there is also a theorem called the Prime Number Theorem that gives an estimate on prime distribution. We won't go into this because this is extremely technical.

By the way for a long time it was thought that primality test is "easy". Note that Miller–Rabin prime testing (and other primality tests) is easy probabilistically. A deterministic polynomial runtime primality test was finally discovered and proven only recently (2002) by a group of computer scientists, Agrawal, Kayal and Saxena, from India. The algorithm is now called the **AKS primality test** algorithm. If you want some fancy automata notation, the AKS algorithm says that

AKS primality test

$$\textsc{Primes} \in \mathrm{P}$$

where $\textsc{Primes}$ denotes the problem (or language) of testing for primeness and P denotes the class of polynomial runtime problems. This P is the same P in the famous "P = NP" problem. AKS is however not used in real-world applications because the runtime is too slow. There is current ongoing research on improving the performance of this algorithm.

In real-world applications of Miller-Rabin prime testing algorithm, to test that a random 2048-bit odd number is a prime, using $t = 10$ rounds is usually more than enough.

In 2007, a 1039 bit integer was factored with the number field sieve using 400 computers over 11 months. Nowadays (2019), primes with 2048 bit length is definitely enough – unless someone discovered a new factoring algorithm. In RSA–speak, when you hear "RSA 1024-bit key", it means the modulus $N = pq$ has 1024 bit. That means the bit length of each of the two primes is about 512.

Exercise 1.10.1. * Good research project: Study the AKS algorithm. (Go to solution, page 83) □

debug: exercises/rsa-13/question.tex

Solutions

Solution to Exercise 1.10.1.

debug: exercises/rsa-13/answer.tex

Solution not provided.

1.11 Carmichael function debug: carmichael-function.tex

Definition 1.11.1. The **multiplicative order** of a mod n, if it exists, is the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$. multiplicative order

Some values might not have multiplicative order. For instance 0^k is not $\equiv 1 \pmod{n}$ for all k. Recall that a has a multiplicative inverse mod n iff $a^k \equiv 1 \pmod{n}$ for some $k > 0$.

Recall Euler's theorem: If $\gcd(a, n) = 1$, then

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Is $\phi(n)$ the best possible in the sense that $\phi(n)$ gives you the smallest for the above to be true?

For instance when $n = 2$, $\phi(2) = 1$ which is the smallest possible positive integer to satisfy

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

For $n = 3$, $\phi(3) = 2$ and

$$\begin{aligned} 1^1 \equiv 1,\ \ 2^1 \equiv 2 \pmod{3} \\ 1^2 \equiv 1,\ \ 2^2 \equiv 1 \pmod{3} \end{aligned}$$

So $\phi(3) = 2$ is the smallest for

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

to be true for both $a = 1$ and $a = 2$. Etc. But when you reach $n = 8$ when $\phi(8) = 4$, if $\gcd(a, 8) = 1$, then $a = 1, 3, 5, 7$ and

$$\begin{aligned} 1^2 &\equiv 1 \pmod{8} \\ 3^2 &\equiv 1 \pmod{8} \\ 5^2 &\equiv 1 \pmod{8} \\ 7^2 &\equiv 1 \pmod{8} \end{aligned}$$

and $2 < 4$ (in fact $2 \mid 4$). In other words 2 satisfies

$$a^2 \equiv 1 \pmod{8}$$

for all a such that $\gcd(a, 8) = 1$. Of course we know from Euler's theorem that

$$a^{\phi(8)} \equiv 1 \pmod{8}$$

and $\phi(8) = 4$. Clearly

$$a^2 \equiv 1 \pmod 8 \implies a^4 \equiv 1 \pmod 8$$

Let's write $\lambda(8) = 4$. In general:

Definition 1.11.2. Define the **Carmichael function** Carmichael function

$$\lambda(n)$$

to be the LCM (lowest common multiple) of the multiplicative order of a mod n for all $a \in \{1, 2, ..., n\}$ satisfying $\gcd(a, n)$. The multiplicative order of a is the smallest positive integer k such that $a^k \equiv 1 \pmod n$.

In other words, $\lambda(n)$ is "almost" $\phi(n)$.

Theorem 1.11.1.

(a) $\lambda(mn) = \lambda(m)\lambda(n)$ *if* $\gcd(m, n) = 1$.
(b) *Let p be a prime and $k \geq 0$. Then*

$$\lambda(p^k) = \begin{cases} \phi(p^k) & \textit{if } p > 2 \\ \phi(p^k) & \textit{if } p = 2, k = 0, 1, 2 \\ \frac{1}{2}\phi(p^k) & \textit{if } p = 2, k \geq 3 \end{cases}$$

Theorem 1.11.2.

(a) *Let* $\gcd(a, n) = 1$, *if* $a^k \equiv 1 \pmod n$, *then* $k \mid \lambda(n)$.
(b) *If*

$$a^k \equiv 1 \pmod n \quad \textit{for all } \gcd(a, n) = 1$$

then

$$\lambda(n) \mid k$$

(c) $\lambda(n) \mid \phi(n)$.

1.12 OpenSSL debug: openssl.tex

Do this is your bash shell:

```
openssl genpkey -algorithm RSA -out private_key.pem \
    -pkeyopt rsa_keygen_bits:2048
```

And you will get an RSA key. The key is stored in the file `private_key.pem`. The utility you are using is openssl. Here's an example of the file:

```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDD4U0gAdmz0G5I
LGmpxx5DNWclrpINB/bH1aLiFk4lxh85gE83UX3dEirn1PaVxSB4qvMr9dY0yZ8G
jd7Yj/Bubk0AYhlclWbiRERRWcGigmP/CvJWP7MSarC4sT04QGv0X6+oj64jkM55
WLApi6jHDprg4Un7LT/IJVZbr2hbmu1D6wPPYN2D1uZpav0skL6+SyYnl5U3EimQ
M9BO5FM7K7yiRDeOFXCHgfUbh5PULZLc1u/vBdr70WopRwFRTdFdGgAY3cHG3p7d
CLC+vNl7DaN7JsJFSoyPq5ynok1P16l9AdvnHwzVtkEtk2tYaQeZCjtyuM+3jzQ4
zQt1Et7tAgMBAAECggEANaii3tVC7vg9DbZk56ZtStn5PKBa0AkLeGi0qxyTIdPp
P9Y/XRcM1J+ic0mqlxKeN5AU90jr+h/1WVVJ46dipM3AeEdnTS58NaWf1W0yFzOC
8x3rjub6RiRF7wJWk+9J43LG6vUZLhMADMvXzjm87XK5yLr0imk13L0lsA4YF2eZ
d+EN/xaop2Olw76PSQkiVseVGKcvZo6lJMxZAgLMUTX5CmWu0U1z8yR/Lgj9CPo4
KO2iK0hnkePEmR9OiHZ5PRPHb4Wb3CsCniwKqH0xJpAuUVYr532r9yt6rUwotw6E
OadIcW2e5XICGRUddBhQ4Di3tN89qHVOTic6M0jQwQKBgQDvyjl7/CcyC0/86Gvj
DDXpr1m4ePSNMyY+Oy5h5S1FB2KNibNLfZp2VYnNX73R63AZQOxNNjCDfHyiBLb2
NJ7arqjqWH846N0cmSdI7Fpo38kyMKwwq95Zl1PqDhahjoZQSq00iTh2R1YXvGmM
qRSes8aLO3l3A9KRGIRnuMOTUQKBgQDRHyNzfTPw8TNSLU2QFUpmZ/6XeSjtRKxP
5dGocZ4YxLb/5FLKZeQCRZWA8ocbllwRFcuI1VgfaSclBCb8ElS76Qw/wWjaYJ/V
qoZgKCvqtrZKruax2+gB59LjJGRqGf537F3V4qB4QP2tp4glTxiQL9yFr4p8e72R
REgB05ES3QKBgAT2HjJeiUET0tfcxz6vZf4rzqNufUDeqg/nkZIc987R1EwxaTBK
rQN9yZgiPv806+DZ4wnF8UMHNFz11ANMG21S59PRePBogQqycImlukkpODR9pVJs
e/FGnEnfeMBm/ohywxqvLCfmWfWrxFNQvEh8V8NRu7Wmspil9TdgL0vBAoGBAKki
9DteUnpXu1iFx6v3bFtzVRkSJ6Xv2yYsDOyeKG6D/DbvZn7I9idYPFk0z03iyMgQ
xrP/Sezt0XlA6H8MHHh3Py75sWKer+fSqihvlUWbTckNuQy1feq8o3aPYp/mMkiw
Zhyt1XgtqH+hdp4mYQmNjGCb3/ha5LHvdgX0JewJAoGAPT3a1Zb5xPQ6RARQSX2b
Tk6AXHH7vsuYf18c0KyruUAhbQ6CUTqemz4qY5VnlWmORP277ceb9i+NtiRvm4Rd
HoyZtvZvZc0zTtIsxYaFU6pPnnexrNgRC7+jCoAqfHeShJ/fNLiHA/Ffy06S6eQV
Xo8vamqc1SMq2tQegRBEV9s=
-----END PRIVATE KEY-----
```

You can search for a website that decode PEM file data for you and see what is the data stored in this file. Here's one: https://lapo.it/asn1js/. To find out which are the primes, etc., you can check the the spec at IEFT https://tools.ietf.org/html/rfc2313#section-7.2:

```
An RSA private key shall have ASN.1 type RSAPrivateKey:

   RSAPrivateKey ::= SEQUENCE {
     version Version,
     modulus INTEGER, -- n
     publicExponent INTEGER, -- e
     privateExponent INTEGER, -- d
     prime1 INTEGER, -- p
     prime2 INTEGER, -- q
     exponent1 INTEGER, -- d mod (p-1)
     exponent2 INTEGER, -- d mod (q-1)
     coefficient INTEGER -- (inverse of q) mod p }
```

You can also extract the public key from your file:

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

which can then be sent to your friend (or a server) for communication. These files are pretty standard and can be use by encryption/decryption programs to perform RSA/AES/3DES/... encryption and decryption. Or you can execute

```
openssl rsa -in private_key.pem -noout -text
```

Either way, this will display a list of 9 integers. The command line gives this output:

```
Private-Key: (2048 bit)
modulus:
    00:c3:e1:43:a0:01:d9:b3:d0:6e:48:2c:69:a9:c7:
    1e:43:35:67:25:ae:92:0d:07:f6:c7:d5:a2:e2:16:
    4e:25:c6:1f:39:80:4f:37:51:7d:dd:12:2a:e7:d4:
    f6:95:c5:20:78:aa:f3:2b:f5:d6:34:c9:9f:06:8d:
    de:d8:8f:f0:6e:6e:4d:00:62:19:5c:95:66:e2:44:
    44:51:59:c1:a2:82:63:ff:0a:f2:56:3f:b3:12:6a:
    b0:b8:b1:3d:38:40:6b:f4:5f:af:a8:8f:ae:23:90:
    ce:79:58:b0:29:8b:a8:c7:0e:9a:e0:e1:49:fb:2d:
    3f:c8:25:56:5b:af:68:5b:9a:ed:43:eb:03:cf:60:
    dd:83:d6:e6:69:6a:f3:ac:90:be:be:4b:26:27:97:
    95:37:12:29:90:33:d0:4e:e4:53:3b:2b:bc:a2:44:
    37:8e:15:70:87:81:f5:1b:87:93:d4:2d:92:dc:d6:
    ef:ef:05:da:fb:d1:6a:29:47:01:51:4d:d1:5d:1a:
    00:18:dd:c1:c6:de:9e:dd:08:b0:be:bc:d9:7b:0d:
    a3:7b:26:c2:45:4a:8c:8f:ab:9c:a7:a2:4d:4f:d7:
    a9:7d:01:db:e7:1f:0c:d5:b6:41:2d:93:6b:58:69:
    07:99:0a:3b:72:b8:cf:b7:8f:34:38:cd:0b:75:12:
    de:ed
publicExponent: 65537 (0x10001)
privateExponent:
    35:a8:a2:de:d5:42:ee:f8:3d:0d:b6:64:e7:a6:6d:
    4a:d9:f9:3c:a0:5a:d0:09:0b:78:68:b4:ab:1c:93:
    21:d3:e9:3f:d6:3f:5d:17:0c:d4:9f:a2:73:49:aa:
    97:12:9e:37:90:14:f7:48:eb:fa:1f:f5:59:55:49:
    e3:a7:62:a4:cd:c0:78:47:67:4d:2e:7c:35:a5:9f:
    d5:6d:32:17:33:82:f3:1d:eb:8e:e6:fa:46:24:45:
    ef:02:56:93:ef:49:e3:72:c6:ea:f5:19:2e:13:00:
    0c:cb:d7:ce:39:bc:ed:72:b9:c8:ba:ce:8a:69:35:
```

```
    dc:bd:25:b0:0e:18:17:67:99:77:e1:0d:ff:16:a8:
    a7:63:a5:c3:be:8f:49:09:22:56:c7:95:18:a7:2f:
    66:8e:a5:24:cc:59:02:02:cc:51:35:f9:0a:65:ae:
    d1:4d:73:f3:24:7f:2e:08:fd:08:fa:38:28:ed:a2:
    28:e8:67:91:e3:c4:99:1f:4e:88:76:79:3d:13:c7:
    6f:85:9b:dc:2b:02:9e:2c:0a:a8:7d:31:26:90:2e:
    51:56:2b:e7:7d:ab:f7:2b:7a:ad:4c:28:b7:0e:84:
    39:a7:48:71:6d:9e:e5:72:02:19:15:1d:74:18:50:
    e0:38:b7:b4:df:3d:a8:75:4e:4e:27:3a:33:48:d0:
    c1
prime1:
    00:ef:ca:39:7b:fc:27:32:0b:4f:fc:e8:6b:e3:0c:
    35:e9:af:59:b8:78:f4:8d:33:26:3e:3b:2e:61:e5:
    2d:45:07:62:8d:89:b3:4b:7d:9a:76:55:89:cd:5f:
    bd:d1:eb:70:19:40:ec:4d:36:30:83:7c:7c:a2:04:
    b6:f6:34:9e:da:ae:a8:ea:58:7f:38:e8:dd:1c:99:
    27:48:ec:5a:68:df:c9:32:30:ac:30:ab:de:59:97:
    53:ea:0e:16:a1:8e:86:50:4a:ad:34:89:38:76:47:
    56:17:bc:69:8c:a9:14:9e:b3:c6:8b:3b:79:77:03:
    d2:91:18:84:67:b8:c3:93:51
prime2:
    00:d1:1f:23:73:7d:33:f0:f1:33:52:2d:4d:90:15:
    4a:66:67:fe:97:79:28:ed:44:ac:4f:e5:d1:a8:71:
    9e:18:c4:b6:ff:e4:52:ca:65:e4:02:45:95:80:f2:
    87:1b:96:5c:11:15:cb:88:d5:58:1f:69:27:25:04:
    26:fc:12:54:bb:e9:0c:3f:c1:68:da:60:9f:d5:aa:
    86:60:28:2b:ea:b6:b6:4a:ae:e6:b1:db:e8:01:e7:
    d2:e3:24:64:6a:19:fe:77:ec:5d:d5:e2:a0:78:40:
    fd:ad:a7:88:25:4f:18:90:2f:dc:85:af:8a:7c:7b:
    bd:91:44:48:01:d3:91:12:dd
exponent1:
    04:f6:1e:32:5e:89:41:13:d2:d7:dc:c7:3e:af:65:
    fe:2b:ce:a3:6e:7d:40:de:aa:0f:e7:91:92:1c:f7:
    ce:d1:d4:4c:31:69:30:4a:ad:03:7d:c9:98:22:3e:
    ff:34:eb:e0:d9:e3:09:c5:f1:43:07:34:5c:f5:d4:
    03:4c:1b:6d:52:e7:d3:d1:78:f0:68:81:0a:b2:70:
    89:a5:ba:49:29:38:34:7d:a5:52:6c:7b:f1:46:9c:
    49:df:78:c0:66:fe:88:72:c3:1a:af:2c:27:e6:59:
    f5:ab:c4:53:50:bc:48:7c:57:c3:51:bb:b5:a6:b2:
    98:a5:f5:37:60:2f:4b:c1
exponent2:
    00:a9:22:f4:3b:5e:52:7a:57:bb:58:85:c7:ab:f7:
    6c:5b:73:55:19:12:27:a5:ef:db:26:2c:0c:ec:9e:
    28:6e:83:fc:36:ef:66:7e:c8:f6:27:58:3c:59:34:
    cf:4d:e2:c8:c8:10:c6:b3:ff:49:ec:ed:d1:79:40:
    e8:7f:0c:1c:78:77:3f:2e:f9:b1:62:9e:af:e7:d2:
    aa:28:6f:95:45:9b:4d:c9:0d:b9:0c:b5:7d:ea:bc:
    a3:76:8f:62:9f:e6:32:48:b0:66:1c:ad:d5:78:2d:
    a8:7f:a1:76:9e:26:61:09:8d:8c:60:9b:df:f8:5a:
    e4:b1:ef:76:05:f4:25:ec:09
coefficient:
    3d:3d:da:d5:96:f9:c4:f4:3a:44:04:50:49:7d:9b:
    4e:4e:80:5c:71:fb:be:cb:98:7f:5f:1c:d0:ac:ab:
    b9:40:21:6d:0e:82:51:3a:9e:9b:3e:2a:63:95:67:
    95:69:8e:44:fd:bb:ed:c7:9b:f6:2f:8d:b6:24:6f:
    9b:84:5d:1e:8c:99:b6:f6:6f:65:cd:33:4e:d2:2c:
    c5:86:85:53:aa:4f:9e:77:b1:ac:d8:11:0b:bf:a3:
    0a:80:2a:7c:77:92:84:9f:df:34:b8:87:03:f1:5f:
    cb:4e:92:e9:e4:15:5e:8f:2f:6a:6a:9c:d5:23:2a:
    da:d4:1e:81:10:44:57:db
```

Here's a translation from the above to our notation:

$$
\begin{aligned}
\texttt{modulus:} \quad & N = pq \\
\texttt{publicExponent:} \quad & e \\
\texttt{privateExponent:} \quad & d \\
\texttt{prime1:} \quad & p \\
\texttt{prime2:} \quad & q \\
\texttt{exponent1:} \quad & d \pmod{p-1} \\
\texttt{exponent2:} \quad & e \pmod{q-1} \\
\texttt{coefficient:} \quad & q^{-1} \pmod{p}
\end{aligned}
$$

In base 10, the first prime p is

```
1683862219928816349707271349921576008244427712110187331638813164158714
6092027585090051888230585327087065337961823380208709800370390990580252
8722258697154723550189470837399623930362147945843437808281987276857460
8688959338541591277378939013616596683124563104177040476275509113499666
03867461377553068393672971089
```

while the second prime q is

```
1468502058733357263533301801667725642866211719712179742781321148035444
6332788830881798130656206247273474356027716965684873288667683801689207
9488242467105914811252659920539476366695782552188898620329151145531549
7121404006845630292860070621021218917792295007556534607746425085044723
36435831750052666953524384477
```

The modulus N is

```
2472755136588788012823283243623501709484398222718811159964669228955178
0726001518209955076203415749580212140160361880502461169830232465366367
1996218795720793055101178409956065707059295267698070887972273933143809
1099039519912687661692960509714944550110080756612347182500810089965712
5497185640786642137646341798790334361381408205787804137532050691471480
0849398396065099886053838469464732274927916105465641450996489417150244
0556732311606837485835166684077237851753421465934633294723902781417651
6983375588322606441649133504887969275866982276184004107267665903043195
072291398666071369881349881441299500925076152870541385453
```

You can verify that pq is indeed N.

Exercise 1.12.1. Write a simple python program to convert the HEX data from the above into an integer and print the integer. Combine with the openssl command, write a program that prints the integer values from the PEM file in base 10. (Go to solution, page 91) □

debug: exercises/rsa-10/question.tex

Exercise 1.12.2. You now have everything you need to write a python program to generate RSA keys (both private and public). Allow the user to specify

debug: exercises/rsa-11/question.tex

the bit length of the key (that means the bit length of N) and the number of rounds of Miller-Rabin. Default the bit length to 4096 and the number of rounds of Miller-Rabin to 128. You can choose your p and q this way: Assume p and q have $n/2$ bits each where n is the bitlength of N. Put random bits into p and q. Obvously you want p and q to be odd and if you want p to be $n/2$ bits then the $n/2$–order bit can't be 0. You then test if the p and q are prime using Miller-Rabin. You can also hard-code the testing for divisibility with a small number of primes: say the first 1,000,000 primes.

Next, test your RSA by encrypting and decrypting an integer M that is $< N$. (There are other conditions on the various quantities to make your RSA key generator more secure.) (Go to solution, page 92) □

Exercise 1.12.3. In practice, RSA does not use $e, d \pmod{\phi(pq)}$ but $e, d \pmod{\lambda(pq)}$. where $\lambda(pq) = \text{LCM}(p-1, q-1)$. (Here LCM is "largest common multiple.) Study why. (Go to solution, page 93) □

debug: exercises/rsa-12/question.tex

Solutions

Solution to Exercise 1.12.1.

debug: exercises/rsa-10/answer.tex

Solution not provided.

Solution to Exercise 1.12.2.

debug: exercises/rsa-11/answer.tex

Solution not provided.

Solution to Exercise 1.12.3.

debug: exercises/rsa-12/answer.tex

Solution not provided.

1.13 RSA security debug: rsa-security.tex

First of all Bob has to be able to select two primes. Note that the primes must be large.

Why?

First of all note that Eve has (N, e) since it is the public key. What does Eve want? She wants the decryption key (N, d) in order to decrypt messages, i.e., she wants d. Both e and d are integers in mod $\phi(N)$, i.e., $0 < d < \phi(N)$. Recall that

$$\phi(N) = N \prod_{p|N} \left(1 - \frac{1}{p}\right)$$

For the case $N = pq$,

$$\phi(N) = (p-1)(q-1)$$

Therefore if Eve has p and q, she can compute $\phi(N)$. After that she can compute the multiplicative inverse of e mod $\phi(N)$ using the Extended Euclidean algorithm. Therefore Eve might try to factor N to get p and q in order to get $\phi(N)$.

The obvious brute force method is to try to divide N by 2, 3, ... So in summary here's what Eve might want to try:

Eve's Dream 1.

1. Factor N to obtain p and q
2. Compute $\phi(N) = (p-1)(q-1)$
3. Compute the multiplicative inverse d of e in mod $\phi(N)$ using the Euclidean algorithm.

Note that the only reason why she needs p and q is so that she can compute $\phi(N)$. What if somehow Eve got a hold of $\phi(N)$? Then she can proceed onto step 3. She doesn't really care about the primes p and q in this case. The important is to compute d. And to compute d she needs to know $\phi(N)$.

Eve's Dream 2.

1. Compute $\phi(N)$ from N
2. Compute the multiplicative inverse d of e in mod $\phi(N)$ using the Euclidean algorithm.

So here's an important question: *Is there a way to compute $\phi(N)$ without factorizing N?*

There's no known fast way to computing $\phi(N)$ from N other than by definition or through the factorization N. Without the above, Eve is back to square 1, i.e., she has to factorize N. And since the naive approach is to try to divide N by $2, 3, 4, \ldots$, the goal of Bob (who's setting up the RSA parameters) is to make sure that p and q are huge primes.

Exercise 1.13.1. Make a plot of $\phi(N)$ for $1 \leq N \leq 1000000$. Then try to see if you can fit a polynomial, exponential, logarithm, etc function to your plot. debug: exercises/rsa-03/question.tex

(Go to solution, page 98) □

However ... the interesting thing is this: If Eve has $N = pq$ and $\phi(N)$, she can very quickly compute the primes p and q. Why?

Exercise 1.13.2. Suppose Eve has $N = pq$ and $\phi(N)$, but not the primes p and q. Design an algorithm to compute p and q. [HINT: Consider the quadratic polynomial $(x-p)(x-q)$. The roots are p and q.] Use your algorithm to compute p and q where $N = 968207 = pq$ and $\phi(N) = 966240$. (Go to solution, page 99) □ debug: exercises/rsa-04/question.tex

Hence Bob must be able to generate large random primes in order to not let Eve factorize N. This is the main attack on RSA: Factorization of N.

Well ... "large" is kind of vague. Just how big do we need the primes to be?

Suppose you have two primes of roughly 100 digits each. Multiplying them together you get a number n that roughly 200 digits long, i.e. 10^{200}. Suppose Eve can divide such a number by a trial divisor really fast, say she can perform such divisions at a rate of $1,000,000,000$ per second, i.e. 10^9 per second. She would need to try the numbers up to roughly the square root of 10^{200}, i.e. 10^{100}. In terms of number of years the amount of time needed is

$$10^{100}/10^9/(60 \cdot 60 \cdot 24 \cdot 365)$$

which is

317097919837645865043125317097919837645865043125317097919837645865043125317097919837

i.e., roughly 10^{85} years (give or take a century or two!) She won't be around that long. In general, this methods requires $O(n^{1/2})$ divisions.

There are many intricate factorization algorithms other than brute force testing 2, 3, 5, ...: quadratic sieve, elliptic curve factorization, number field sieve, generalized number field sieve, etc. The fastest method known is the generalized number field sieve (GNFS) with a expected runtime of

$$e^{(c+o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}}$$

where $c = (64/9)^{1/3}$. It is superpolynomial (more than polynomial) and subexponential (less than exponential). (Example: n^2 is polynomial, 2^n is exponential, but $2^{\sqrt{n}}$ is greater than polynomial but less than exponential.) The $o(1)$ means a number that becomes 0 when n grows. Formally $f(n) = o(1)$ means for any constant $c \neq 0$, $f(n)/c \to 0$ as $n \to \infty$. For instance $1/n = o(1)$.

In general public key cryptosystems (and some other security protocols) involve the solution of some complex mathematical problem. In the case of RSA, the hard problem is integer factorization. In order to gain credence, such companies usually publish challenges on their web site. For instance if you go to `http://www.rsasecurity.com/rsalabs/node.asp?id=2093` you'd find a list of number for you to factorize. The latest challenge to be factored was the RSA-576 number, a 576-bit or 174-digit number. It was broken in December 2003. The prize for factoring RSA-576 was $10,000. The next challenge is RSA-640:

31074182404900437213507500358885679300373460228427
27545720161948823206440518081504556346829671723286
78243791627283803341547107310850191954852900733772
4822783525742386454014691736602477652346609

This challenge is worth $30,000. If this is too small for you you can try the largest RSA challenge, RSA-2048:

25195908475657893494027183240048398571429282126204
03202777713783604366202070759555626401852588078440
69182906412495150821892985591491761845028084891200
72844992687392807287776735971418347270261896375014
97182469116507761337985909570009733045974880842840
17974291006424586918171951187461215151726546322822
16869987549182422433637259085141865462043576798423
38718477444792073993423658482382428119816381501067
48104516603773060562016196762561338441436038339044
14952634432190114657544454178424020924616515723350
77870774981712577246796292638635637328991215483143
81678998850404453640235273819513786365643912120103
97122822120720357

This baby is worth $200,000.

In order to defeat certain factorization techniques, researchers have suggested the following guidelines for key generation:

1. pq should have at least 1024 bits.
2. p and q should have roughly the same size
3. Both $p-1$ and $q-1$ should have a large prime factor
4. $\gcd(p-1, q-1)$ should be small
5. $d > n^{0.292}$

Integer factorization and RSA security is obviously a very hot area of research since it is used so widely. You can find many theoretical results on attacking RSA online. For instance if pq has n bits and Eve knows either the least significant or most significant $n/4$ bits, then she can factorize pq. If you want to learn more about number theory and RSA talk to me.

Solutions

Solution to Exercise 1.13.1.

debug: exercises/rsa-03/answer.tex

Solution not provided.

Solution to Exercise 1.13.2.

debug: exercises/rsa-04/answer.tex

Solution not provided.

1.14 Fermat factorization debug: fermat-factorization.tex

Suppose you want to factorize n and you know that n is a difference of two squares:

$$n = x^2 - y^2$$

then right away you know

$$n = (x + y)(x - y)$$

Of course if $x - y$ is 1, the factorization is not helpful at all.

To achieve the goal of writing n as a difference of squares, you can do the following:

- Check if $1^2 - n$ is a square.
- Check if $2^2 - n$ is a square.
- Check if $3^2 - n$ is a square.
- Etc.

Why? Because if $x^2 - n$ is a square, say y^2, then $x^2 - n = y^2$ gives us

$$n = x^2 - y^2$$

Note that if $n = ab$ is odd (which is the case for an RSA modulus), then writing n as a difference of two squares since

$$n = \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2$$

If n is odd, then a and b are odd and therefore $a + b$ and $a - b$ are both even. Hence every odd n can be expressed as the difference of two squares. What if n is even? Then you can remove 2^k from n where k is maximal to get $n = 2^k n'$ and proceed with the above idea applied to n' if $n' > 1$. In the following, I will assume n is odd.

Now we do not want the factorization $n = n \times 1$. Since we are aiming for $n = (x - y)(x + y)$, one way is to start x at around $\sqrt{n}$. Then y will be close to 0 and therefore $x + y$ and $x - y$ won't be close to 1.

Another thing to note is that if we start x at $\sqrt{n}$, if n is squarefree (which is the case for RSA), we will have $x = \lfloor \sqrt{n} \rfloor < \sqrt{n}$, which means $x^2 - n$ is negative and therefore cannot be y^2. In this case, we might as well start with $x = \lfloor \sqrt{n} \rfloor + 1$.

To find x, y such that $n = x^2 - y^2$, we try different values of x and then check if

$$x^2 - n$$

is a square:

$$x^2 - n = y^2$$

We can started with $x = \lfloor \sqrt{n} \rfloor$ (or around there) and increment x by 1 until the square is found. Note that if n is itself a square (this won't happen for the RSA case), then the above will end with $y = 0$.

```
ALGORITHM: Fermat-Factorization
INPUT: n -- an odd integer

let x = floor(sqrt(n)) + 1
while x * x - n is not a square:
    x = x + 1

y = sqrt(x * x - n)
return (x - y, x + y)
```

Here's an example execution of my code for $n = 11 \cdot 11 \cdot 37$:

```
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 67 12 3 3
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 68 147 12 3
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 69 284 16 28
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 70 423 20 23
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 71 564 23 35
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 72 707 26 31
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 73 852 29 11
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 74 999 31 38
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 75 1148 33 59
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 76 1299 36 3
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 77 1452 38 8
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 78 1607 40 7
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 79 1764 42 0
fermatfactorize: factors = 37 121
```

Note that Fermat factorization need not be faster than trial division, but it is specifically crafted to handle the product of two primes which are "close".

I'll give you one optimization:

Exercise 1.14.1. Here's a slight optimization: You need to compute x^2 and

debug: exercises/rsa-34/question.tex

then $(x+1)^2$, etc. Clearly $(x+1)^2 = x^2 + 2x + 1$. So if you have x and x^2, what's a faster way to compute $(x+1)^2$? Therefore if you have x and $x^2 - n$, how would you compute $(x+1)^2 - n$? (Go to solution, page 104) □

Exercise 1.14.2. We also need to compute $\lfloor\sqrt{n}\rfloor$, i.e., integer square root of n. How many algorithms can you come up with? (Go to solution, page 105) □

debug: exercises/rsa-35/question.tex

Exercise 1.14.3. Implement Fermat factorization and try to optimize it as much as you can. (There are some questions below on optimization.) Test for many values of n. Make a plot. What is the runtime of Fermat factorization algorithm based on your experiment? You don't have to completely factorize n. Finding a factor is good enough. Can you get roughly $O(n^{1/2})$? $O(n^{1/3})$? $O(n^{1/4})$? (Go to solution, page 106) □

debug: exercises/rsa-36/question.tex

Exercise 1.14.4. Implement brute force trial-by-division until $\sqrt{n}$ for n to find a factor of n. Compare the runtimes for this algorithm against your implementation of Fermat factorization above. Which one is faster? Remember that brute force performs $\lfloor\sqrt{n}\rfloor$ mods in the worse case scenario. (Go to solution, page 107) □

debug: exercises/rsa-37/question.tex

Exercise 1.14.5. If n is an integer, how do you handle the check "n is a square"? Can you think of several ways to do that? What are their runtimes? You might want to make a plot of their runtimes. (Go to solution, page 108) □

debug: exercises/rsa-38/question.tex

Exercise 1.14.6. Suppose z^2 is an integer square that is the largest integer square that is $\leq$ to $x^2 - n$. For instance if $x^2 - n$ is 26, then z^2 is 25 (i.e., z is 5). In other words z^2 is the square that is closest to $x^2 - n$ on the left. Note z^2 will be $x^2 - n$ if $x^2 - n$ is a square. How would you compute the the square closest to $(x+1)^2 - n$ on the left? Is it $(z+1)^2$? $(z+2)^2$? If you can prove, for instance, that it's $(z+2)^2$? then we can save on taking square root, correct? You can assume that you have the values of x and z and n. (Go to solution, page 109) □

debug: exercises/rsa-39/question.tex

Exercise 1.14.7. Base on $x^2 - n$ and $(x+1)^2 - n$, is there a way to guestimate k such that $(x+k)^2 - n$ is a square or closed to a square? (Go to solution, page 110) □

debug: exercises/rsa-40/question.tex

Exercise 1.14.8. Suppose n is an odd prime. What will happen during the execution of Fermat's factorization algorithm with n as input? Would the algorithm terminate? (If not, then it's not really an algorithm!) Can Fermat factorization algorithm be converted to a primality testing algorithm? Would it be possible? Would it be a good idea? (Go to solution, page 111) □

debug: exercises/rsa-41/question.tex

Exercise 1.14.9. Write a parallel program for the Fermat factorization algorithm. (Go to solution, page 112) □

debug: exercises/rsa-42/question.tex

Exercise 1.14.10. What if you test for the condition $n = x^3 - y^3$? Is there are version of factorization using difference of cubes? (Go to solution, page 113) □

debug: exercises/rsa-43/question.tex

Solutions

Solution to Exercise 1.14.1.

debug: exercises/rsa-34/answer.tex

Solution not provided.

Solution to Exercise 1.14.2.

debug: exercises/rsa-35/answer.tex

Solution not provided.

Solution to Exercise 1.14.3.

debug: exercises/rsa-36/answer.tex

Solution not provided.

Solution to Exercise 1.14.4.

debug: exercises/rsa-37/answer.tex

Solution not provided.

Solution to Exercise 1.14.5.

debug: exercises/rsa-38/answer.tex

Solution not provided.

Solution to Exercise 1.14.6.

debug: exercises/rsa-39/answer.tex

Solution not provided.

Solution to Exercise 1.14.7.

debug: exercises/rsa-40/answer.tex

Solution not provided.

Solution to Exercise 1.14.8.

debug: exercises/rsa-41/answer.tex

Solution not provided.

Solution to Exercise 1.14.9.

debug: exercises/rsa-42/answer.tex

Solution not provided.

Solution to Exercise 1.14.10.

debug: exercises/rsa-43/answer.tex

Solution not provided.

Index

Bibliography