Computer Science

Dr. Y. Liow (April 2, 2023)

Contents

Chapter 204

RSA

`File: chap.tex`

`File: rsa.tex`

204.1 RSA

We only need to work with integers. Why? Because any message is really just a sequence of bits and you can cut your sequences of bits into blocks of fixed size say 1024 bits. Of course binary numbers can be converted into integers. 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 has already selected 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. Furthermore Bob publishes N and e publicly. We assume that the secret x is less than N.

Since N and e are public, Alice can compute

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

i.e., she sends the least positive remainder of x^e mod N.

Now Bob receives $x^e \pmod{N}$ and computes $D_{(N,d)}(x) = x^{ed} \pmod{N}$. What we need to prove is that RSA works, i.e.,

Theorem 204.1.1. *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 M.

Proof. Here's the proof. Now since $ed \equiv 1 \pmod{\phi(N)}$,

$$ed = k\phi(N) + 1$$

for some integer k. Right? Therefore

$$x^{ed} = x^{k\phi(N)+1} = (x^{\phi(N)})^k \cdot x$$

Now let's assume that $\gcd(x, N) = 1$. Recall that Euler's theorem says that if $\gcd(x, N) = 1$, then

$$x^{\phi(N)} \equiv 1 \pmod{N}$$

Hence continuing the above computation we get

$$\begin{aligned} x^{ed} &= (x^{\phi(N)})^k \cdot x \\ &\equiv 1^k \cdot x \pmod{N} \\ &= x \end{aligned}$$

Voila!

Is the above still true when $\gcd(x, N) \neq 1$? Yes it is!!!

Suppose that $\gcd(x, N) \neq 1$. Since $N = pq$, then $\gcd(x, N)$ is p or q or pq since the only possible divisors of $N = pq$ are $1, p, q, pq$.

If $\gcd(x, N) = N$, then N divides x. Then clearly N also divides x^{ed} and hence $x^{ed} \equiv 0 \equiv x \pmod{N}$. So I'm left with the cases $\gcd(x, N)$ is p or q.

Without loss of generality, suppose $\gcd(x, N) = p$. Since $\gcd(x, N) = p$, we have $p \mid x$. This also implies that $p \mid x^{ed}$. Hence

$$p \mid x^{ed} - x$$

Now since $\gcd(x, N) = p$, we have $q \nmid x$. By Fermat's Little Theorem or Euler's Theorem:

$$x^{q-1} \equiv 1 \pmod{q}$$

Therefore

$$\begin{aligned} (x^{q-1})^{(p-1)} &\equiv 1^{p-1} \pmod{q} \\ &= 1 \\ \therefore \quad x^{\phi(N)} &\equiv 1 \pmod{q} \\ \therefore \quad x^{ed} &= x^{k\phi(N)+1} = \left(x^{\phi(N)}\right)^k x \equiv 1^k x \equiv x \pmod{q} \end{aligned}$$

Hence

$$q \mid x^{ed} - x$$

Altogether I have

$$p \mid x^{ed} - x$$
$$q \mid x^{ed} - x$$

This implies that

$$pq \mid x^{k\phi(N)+1} - x$$

(see exercise below) and hence

$$x^{ed} \equiv x \pmod{N}$$

□

Exercise 204.1.1.

1. In the above, I used the fact that if p and q are distinct primes such that $p \mid a$ and $q \mid a$, then $pq \mid a$. (This is sort of the opposite of Euclid's lemma.) Prove this fact.
2. Once you are done with the above, prove this: if $x \mid a$ and $y \mid a$ and $\gcd(x, y) = 1$, then $xy \mid a$. This is a generalization of the above. So you could have proven this first and deduce the above from this result.
3. Generalizing the above one more step, prove that if $x \mid a$ and $y \mid a$, then $(xy/\gcd(x, y)) \mid a$.

$\square$

Exercise 204.1.2. Get a partner. Pick p, q, e, d and let $N = pq$ and e be public. Get your partner to select a number less than N, encrypt it and send it to you. You then decrypt it. Check that the number you get is the same as what your partner sent you. $\square$

Of course in general, the data you want to send might not be an integer. But clearly all data can be converted to bits using some form of encoding. For instance a string of ASCII characters can be converted to a string of bits, 8 bits per character using ASCII codes. And then the bits can be converted to integers. What if the bit string is too long? Because, don't forget for mod N, the remainders are from 0 to $N-1$. No problem: You just break up your bit string into chunks. You also need to decide on how to reassemble these chunks. However, 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 (3DES, AES, etc.) 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 $0 < e < \phi(N)$, $\gcd(\phi(n), e) = 1$.
   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 (N, e) and a private key (N, d). (N, e) is revealed to the public. (N, d) is kept private. In general a public (or asymmetric) key cipher is made up of the encryption and decryption functions $E_{\text{pubkey}}, D_{\text{privkey}}$ which depends on the key $k = (\text{pubkey}, \text{privkey})$ where the key k is a 2-tuple made up of the public and private key.

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

`File: carmichael-function.tex`

204.2 Carmichael function

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

Definition 204.2.1. 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 $a \in \{1, 2, ..., n\}$ satisyfing $\gcd(a, n)$. 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 204.2.1.

(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 204.2.2.

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

File: rsa-implementation-issues.tex

204.3 Implementation issues

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.

```
File:  baby-asymptotic-analysis.tex
```

204.4 Baby Asymptotic Analysis

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?!?!

`File: karatsuba.tex`

204.5 Karatsuba algorithm

It turns out that there is a better way to multiply integers. This was only discovered very recently in the 60s.

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.

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 mult
2. Compute B = bd          ... 1 mult
```

```
3. Compute C = (a+b)(c+d) ... 2 add, 1 mult
4. Compute D = C - A - B  ... 2 minuses
5. Output AT^2 + DT + B
```

Note that the runtime for subtraction is like addition. In otherwords addition and subtraction is "cheap": they are both $O(n)$ where n is the length of the integers. The high school way of doing subtraction using the borrow method when a column is too small is actually not the best way to do it either. But in any case subtraction can be done in $O(n)$ time.

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:

$$\begin{aligned}
&1 \text{ multiplication(s) of } 2^m\text{–digit numbers} \\
&3 \text{ multiplication(s) of } 2^{m-1}\text{–digit numbers} \\
&3^2 \text{ multiplication(s) of } 2^{m-2}\text{–digit numbers} \\
&3^3 \text{ multiplication(s) of } 2^{m-3}\text{–digit numbers} \\
&3^4 \text{ multiplication(s) of } 2^{m-4}\text{–digit numbers} \\
&\ldots \\
&3^m \text{ multiplication(s) of } 2^{m-m} = 1\text{–digit numbers}
\end{aligned}$$

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.

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

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

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

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 ...

<u>Continuing Step 1: (1234)(1357)</u>.

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

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.

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 204.5.1. 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.

Exercise 204.5.2. Implement a long integer class where multiplication uses Karasuba.

Exercise 204.5.3. The algorithmic analysis above is basically correct but some details are actually missing. Write a research paper on Karatsuba, describing the algorithm in detail, and analyze the runtime performance exactly. Research also on efficient implementation of Karatsuba.

Exercise 204.5.4. 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.

File: primality-test.tex

204.6 Primality testing

We'll need a way to create huge primes.

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!

[Short intro to PNT?]

We can start with a random integer n and check if n is a prime. Obviously n is odd. If n is not prime, we try $n+2$. If $n+2$ is not a prime, we try $n+4$. Etc. Obviously, it would be a good idea to know how many integers we need to try. And of course we still need to know how to test if n is a prime.

File: fermat-primality-test.tex

204.7 Fermat primality test

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 $1 \leq a \leq p-1$,

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

Obviously $a = 1$ does not apply. It's the same for $a = p-1$ if p is odd. Therefore: if $2 \leq a \leq p-2$,

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

This means that given an odd integer n, if $2 \leq a \leq n-2$,

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

It won't tell us if n is prime, but at least it tells us to move on to another odd integer, say $n+2$.

But it won't be too surprising that there's some n and some $a \in [2, n-2]$ such that

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

but n is not a prime. For instance if $n = 15$ (clearly not a prime!) and $a = 4$,

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

I'll say that 4 is a **Fermat liar** for 15. We say that 15 is a **Fermat pseudoprime** and 4 is a liar for 15. Sometimes 4 is called a **base** for the Fermat pseudoprime 15 – I prefer to call 4 a liar.

Fermat liar
Fermat pseudoprime
base

The Fermat primality test is just this: Given n, you randomly pick one $a \in [2, n-2]$ and check if

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

If it is true, you return "n might be a prime". Otherwise you return "n is definitely a composite".

Exercise 204.7.1. Suppose $n = pq$ is a Fermat pseudoprime. Can p be a (Fermat) liar for n? In other words, is the following possible:

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

More generally, if $p \mid n$, can p liar for n? [HINT: No. Now prove it.] Even more generally, suppose $\gcd(a, n) \neq 1$, can a lie for n? Prove it. $\square$

Exercise 204.7.2. A small research experiment: Write a program that attempts to find n such that n is not a prime and

$$a \in [2, n-2],\ \gcd(a, n) = 1 \implies a^{n-1} \equiv 1 \pmod{n}$$

$\square$

Exercise 204.7.3. Is 2 a liar for some Fermat pseudoprime? Write a program to print all n where 2 lies for n. Do you think 2 lie for infinitely many Fermat pseudoprimes? Can you prove it? $\square$

But what if you tried *all* a's in $[2, n-2]$ which are coprime to n and they all lie for n:

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

If this is the case, I will call n a **strong Fermat pseudoprime**. Does it mean that n is a prime?

strong Fermat pseudoprime

Exercise 204.7.4. Write a program to print strong Fermat pseudoprimes. Ignore 6 – the set of possible liars happens to be $\emptyset$. [HINT: The first one is $561 = 3 \cdot 11 \cdot 17$.] Do you think there are infinitely many such numbers? You'll see that Fermat pseudoprimes are kind of rare – more rare than primes.

File: miller-rabin-primality-test.tex

204.8 Miller-Rabin primality test

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

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

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

Proof. From $x^2 \equiv 1 \pmod{p}$, we have $p \mid (x^2 - 1) = (x-1)(x+1)$. By Euclid's Lemma, either $p \mid x - 1$ or $p \mid x + 1$, i.e. $x \equiv \pm 1 \pmod{p}$ □

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

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

ends with a sequence of 1s and before this sequence, there might be a -1. For instance if $k = 5$, the above sequence is a sequence of 6 numbers. The last three numbers might be $\equiv -1, 1, 1 \pmod{n}$ (the first three are not -1 and not 1). Or the last three might be $\equiv 1, 1, 1 \pmod{p}$ (the first three are not -1 and not 1). Or the last two might be $\equiv -1, 1 \pmod{p}$ (the first four are not -1 and not 1). Or all 6 might be $\equiv 1, 1, 1, 1, 1, 1 \pmod{p}$. Of course if $k = 0$, then there is only one number in the sequence and that number is $\equiv 1 \pmod{n}$. In general, the above is a sequence of $k+1$ numbers

If any of the conditions on the left holds and we conclude that a is is a Fermat liar and n is probably prime.

Here's the famous Miller-Rabin algorithm for probabilistic primality testing:

```
ALGORITHM: MILLER_RABIN
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: MILLER_RABIN_ONE_PASS
INPUTS: n, k, m where n = 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"
```

Note that if the return value is `"n is composite"`,

If `"n is probably a prime"` is returned, then the probability that n is prime is $1 - 4^{-t}$. (The point is that there's a probability of $1/4$ that a is a liar for n.) You probably want different `a`s for each test. But if n is large then the chance of picking the same a is very small.

Of course the point is that if n is a prime, then from Fermat's Little Theorem

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

As noted earlier, this is a 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). However 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 guarantee 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

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 (and other primality tests) only shows that it is easy probabilistically. This was finally proven only recently (2002) by a group of computer scientists in India, Agrawal, Kayal and Saxena, that you can do it in polynomial runtime. 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

$$\text{Primes} \in \text{P}$$

where 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, to test that a random 2048-bit odd number is a prime, using 10 rounds is already way 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 the primes are about 512.

Exercise 204.8.1. Good research project: Study the AKS algorithm.

Exercise 204.8.2. 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+9J43LG6vUZLhMADMvXzjm87XK5yLrOimk13L0lsA4YF2eZ
d+EN/xaop2Olw76PSQkiVseVGKcvZo6lJMxZAgLMUTX5CmWu0U1z8yR/Lgj9CPo4
KO2iKOhnkePEmR9OiHZ5PRPHb4Wb3CsCniwKqH0xJpAuUVYr532r9yt6rUwotw6E
OadIcW2e5XICGRUddBhQ4Di3tN89qHVOTic6M0jQwQKBgQDvyjl7/CcyC0/86Gvj
DDXpr1m4ePSNMyY+Oy5h5S1FB2KNibNLfZp2VYnNX73R63AZQOxNNjCDfHyiBLb2
NJ7arqjqWH846N0cmSdI7Fpo38kyMKwwq95Zl1PqDhahjoZQSq00iTh2R1YXvGmM
qRSes8aLO3l3A9KRGIRnuMOTUQKBgQDRHyNzfTPw8TNSLU2QFUpmZ/6XeSjtRKxP
5dGocZ4YxLb/5FLKZeQCRZWA8ocbllwRFcuI1VgfaSclBCb8ElS76Qw/wWjaYJ/V
qoZgKCvqtrZKruax2+gB59LjJGRqGf537F3V4qB4QP2tp4glTxiQL9yFr4p8e72R
REgBO5ES3QKBgAT2HjJeiUET0tfcxz6vZf4rzqNufUDeqg/nkZIc987R1EwxaTBK
rQN9yZgiPv8O6+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:

`modulus:`	$N = pq$
`publicExponent:`	e
`privateExponent:`	d
`prime1:`	p
`prime2:`	q
`exponent1:`	$d \pmod{p-1}$
`exponent2:`	$e \pmod{q-1}$
`coefficient:`	$q^{-1} \pmod{p}$

In base 10, the first prime p is

```
1683862219928816349707271349921576008244427712110187331638813164158714
6092027585090051888230585327087065337961823380208709800370390990580252
8722258697154723550189470837399623930362147945843437808281987276857460
8688959338541591277378939013616596683124563104177040476275509113499666
038674613775530683936729710 89
```

while the second prime q is

```
1468502058733357263533301801667725642866211719712179742781321148035444
6332788830881798130656206247273474356027716965684873288667683801689207
9488242467105914811252659920539476366695782552188898620329151145531549
7121404006845630292860070621021218917792295007556534607746425085044723
364358317500526669535243844 77
```

The modulus N is

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

You can verify that pq is indeed N.

Exercise 204.8.3. 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.

Exercise 204.8.4. You now have everything you need to write a python program to generate RSA keys (both private and public). Allow the user to specify 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.)

Exercise 204.8.5. 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.

`File: exponentials-squaring-method.tex`

204.9 Exponentials in Modulo Arithmetic: Squaring method

Note that for RSA we need to compute powers. Extremely large powers. Here's a very common technique for doing it. Suppose you need to compute a^x (mod N). First you write x as a binary number:

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

As an example suppose we look at $x = (11011)_2 = 27$. Then

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

Now everything is clear: a^{2^1} is the square of a^{2^0}, a^{2^2} is the square of a^{2^1}, etc. In general $a^{2^{n+1}} = (a^{2^n})^2$. You now compute the following: Initialize p to 1 and b to a and compute

$$\begin{array}{ll}
p \equiv p \cdot b & b \equiv b \cdot b \\
p \equiv p \cdot b & b \equiv b \cdot b \\
 & b \equiv b \cdot b \\
p \equiv p \cdot b & b \equiv b \cdot b \\
p \equiv p \cdot b &
\end{array}$$

all congruences are in mod n, i.e. take mod n after every multiplication to cut down on the size of the product. Notice that b runs through a^{2^0}, a^{2^1}, ... The final result is of course stored in p. Of course you can write an algorithm for this exponentiation-by-squaring method. Right?

Exercise 204.9.1. Write a function `pow_by_sq(a, b, n)` which computes a^b (mod n) by using the squaring method. □

This takes care of exponentiation, i.e. encryption and decryption.

File: inverse-in-modular-arithmetic.tex

204.10 Inverse in Modulo Arithmetic

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).

File: rsa-security.tex

204.11 Security

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 and e since (N, e) is the public key. What does Eve want? She wants d in order to decrypt messages. 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, then 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 $\phi(N)$. So ... 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 204.11.1. 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. $\square$

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 204.11.2. 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 $\phi(N) = 966240$.

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

$$3170979198376458650431253170979198376458650431253170979198376458650431253170979198 37$$

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:

$$\begin{aligned}&3107418240490043721350750035888567930037346022842\,7\\&2754572016194882320644051808150455634682967172328\,6\\&7824379162728380334154710731085019195485290073377\,2\\&4822783525742386454014691736602477652346609\end{aligned}$$

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.

File: rsa-padding.tex

204.12 RSA padding – OAEP

https://crypto.stackexchange.com/questions/83463/how-rsa-oaep-decrypt-a-symmetric-key

TODO

The optimal assymmetric encryption padding (OAEP) is a method that adds extra bits to the plaintext before the encryption process. It also involves removing these extra bits after the decryption process.

If I write F to be the "padded with bits", then the encryption

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

is replaced by first preprocess m by padded with bits:

$$m \mapsto F(m) \mapsto E_{(N,e)}(F(m)) = F(m)^e \pmod N$$

Why? RSA cipher is homomorphic in the sense that

$$\begin{aligned} E_{(N,e)}(m_1 m_2) &= (m_1 m_2)^e \pmod N \\ &= m_1^e m_2^e \pmod N \\ &= E_{(N,e)}(m_1) E_{(N,e)}(m_2) \end{aligned}$$

Recall that $N = pq$ is the RSA modulus. Suppose N contains n bits. Let k_0, k_1 be positive integers and all messages have bit length $n - k_0 - k_1$. Let

$$G : \{0,1\}^{k_0} \to \{0,1\}^{n-k_0}$$

and

$$H : \{0,1\}^{n-k_0} \to \{0,1\}^{k_0}$$

be cryptographic hash functions. Generate r which is made up of $n - k_0 - k_1$

random bits. Then m (the plaintext) is padded to become

$$\begin{aligned} L &= \underbrace{(m||0^{k_1})}_{n-k_0 \text{ bits}} \oplus \underbrace{G(r)}_{n-k_0 \text{ bits}} \\ R &= \underbrace{H(L)}_{k_0 \text{ bits}} \oplus \underbrace{r}_{k_0 \text{ bits}} \\ F_{k_0,k_1,r,G,H}(m) &= L||R \end{aligned}$$

(OAEP is a Feistel network – see symmetric ciphers.) You can recover r and $m||0^{k_1}$ by doing

$$\begin{aligned} r &= R \oplus H(L) \\ m||0^{k_1} &= L \oplus G(r) \end{aligned}$$

The padded RSA encryption of m is then

$$E_{(N,e)}(F_{k_0,k_1,r,G,H}(m)) = F_{k_0,k_1,r,G,H}(m)^e \pmod N$$

i.e., you pad m and then encryption.

```
File:  rsa-digital-certificate.tex
```

204.13 RSA digital signatures: authenticity, integrity

Ciphers can be used to hide information – they solve the confidentiality problem. As I mentioned at the beginning of the class, there are other security issues. Another security problem is authentication. Obviously if Alice published a public RSA key, Eve can pretend to be Bob and send a message (Example: “Hi Alice: I’m the real Unabomber. Feel free to report me to the authorities and arrest me. - Bob”) to Alice.

RSA can be used to solve the authentication problem.

Suppose Alice’s RSA key is $(p_A, q_A, N_A = p_A q_A, \phi_A = \phi(p_A q_A), e_A, d_A)$ and Bob’s RSA key is $(p_B, q_B, N_B = p_B q_B, \phi_B = \phi(p_B q_B), e_B, d_B)$. Now suppose Bob wants to send a message M to Alice. Alice and Bob has to agree on a cryptographic hash function which is like a hash function H (see CISS350) in the sense that it generates a value such that hash collisions are rare, i.e., if $x \neq y$, then $H(x) \neq H(y)$ with a very high probability. There are other crucial conditions to be satisfied such as given $H(x)$, it is very difficult to compute x and it is also very difficult to compute y such that $H(x) = H(y)$.

There’s a family of hash functions called SHA. SHA-1 is now considered unsafe. SHA-2 and SHA-3 are considered OK. SHA-2 is a family of hash functions (SHA-224, SHA-256, ...) Try this in python:

```
import hashlib
s = "hello world"
h = hashlib.sha256(s.encode())
print(h)
print(h.hexdigest())
print(int(h.hexdigest(), 16)
```

The output is

```
<sha256 HASH object @ 0x7f0fe741f810>
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
83814198383102558219731078260892729932246618004265700685467928187377105751529
```

The second line of output is the hash in base-16 (in hexidecimal as a string). The third line of output is the hash in base-10 as an int value.

Here's how the RSA digital signature works: Bob wants to send M to Alice and (obviously) Alice needs to get M and verify that it was from Bob.

1. Bob downloads Alice's public key (N_A, e_A).
2. Bob computes the ciphertext (the usual RSA encryption)

$$C = M^{e_A} \bmod N_A$$

3. Bob computes his digital signature for this message:

$$S = H(M)^{d_B} \bmod N_B$$

4. Bob sends the following to Alice:

$$(C, S) = \big(M^{e_A} \bmod N_A,\ \ H(M)^{d_B} \bmod N_B\big)$$

Alice receives (C, S) and does the following:

4. She computes

$$X = C^{d_A} \bmod N_A = (M^{e_A})^{d_A} \bmod N_A$$

   Alice now has X. If M was encrypted correctly using Bob's public key, $X = M$.
5. Alice downloads Bob's public key (N_B, e_B) and computes

$$Y = S^{e_B} \bmod N_B = (H(M)^{d_B})^{e_B} \bmod N_B$$

   She now has $H(M)$.
6. Alice checks that the digital signature is from Bob by computing

$$H(X)$$

   If $H(X)$ is Y, then (C, S) is (extremely likely) from Bob.

If Eve wants to masquerade as Bob, she would need to be able to sign her message M. This means that she need to compute

$$S = H(M)^{d_B} \bmod N_B$$

But this requires d_B which she does not have.

Exercise 204.13.1. Is the cryptographic hash function necessary? □

Exercise 204.13.2. Suppose Bob wants to send a message to Alice. How can Alice checks that the message from Bob was not changed, i.e., how does Alice check for message integrity? [HINT: Use the RSA digital signature. What crucial property of the hash function H is used to ensure that the integrity check actually works?] Suppose Eve intercepts the following message that Bob sends to Alice:

$$(C, S) = \left(M^{e_A} \bmod N_A,\ \ H(M)^{d_B} \bmod N_B\right)$$

and changes the message to M', and sends

$$(C', S) = \left(M'^{e_A} \bmod N_A,\ \ H(M)^{d_B} \bmod N_B\right)$$

(For instance say Eve has access to the mail server of the company that Alice works for.) Alice needs to check the integrity of the message (i.e., she is interested in whether the message is changed or not). How would she check that? What crucial property about H is used?

Index