# Some Basic Number Theory Algorithms and RSA Encryption

In this chapter we'll learn several algorithms that work with integers, and see how these can be used to do public key encryption by the RSA approach. This work will use the divide and conquer and the slightly different reduce and conquer algorithm design techniques for its two crucial algorithms, use $Z_n$ concepts, and will multiply large integers suggesting that Karatsuba's algorithm would be useful, so this is a reasonable time to study it.

> This chapter presents an almost complete technical description of RSA encryption. Many people present themselves as experts in "cybersecurity," but have no idea how something like RSA actually works, because they haven't been forced to endure a chain of mathematics courses up to MTH 3170, and they haven't taken a course in advanced algorithms. Of course, our focus—and specifically what you are actually expected to fully understand—will be several algorithms. You will want to understand enough of the mathematics to understand the entire process, but some of that material—-which will be clearly identified as "optional"—is presented just in case you care.

---

## Overview of RSA Encryption

In RSA encryption, the central entity that is being sent encrypted messages ("headquarters") publishes large, specially chosen integers $n$ and $e$. To send a message—which is itself a large integer $a$ that is smaller than $n$—the sender computes, in $Z_n$, the quantity $c = a^e$, and sends $c$ as the encrypted message. At headquarters, the quantity $c^d$ is computed, in $Z_n$, and this turns out to be $a$. The integer $d$ is very specially chosen relative to $n$ and $e$, and is kept secret from everyone outside headquarters.

The value $n$ is taken as the product of two large prime numbers, and the entire security of the method rests on the idea that no algorithm is known for computing the factors of $n$ in a reasonable amount of time.

To understand how to implement this process efficiently, and how to get the special number $d$ (and, for that matter, $n$ and $e$), and to prove that it all works, we will need to cover several fundamental algorithms and appreciate some theorems from number theory.

---

### Introductory Case Study

Let's begin by doing RSA naively—without the two main algorithms that we will shortly study. Suppose at headquarters we pick $p = 7$ and $q = 13$ and form $n = pq = 91$. Then we need to pick $e$, which can be any number that has no factors in common with

$$\phi = (p-1)(q-1) = 6 \cdot 12 = 72.$$

So, say we pick $e = 11$ (picking a prime number for $e$ is always safe, as long as it is big enough to not itself be a factor of $\phi$).

We publish, say on our web page, $n = 91$ and $e = 11$, along with clear instructions telling people how to send us a message $a$: simply compute $a^e$ in $Z_n$ (actually, we would give people a tool for doing this, since they, being cybersecurity experts and not algorithm and mathematics experts, are unlikely to be happy doing this for themselves).

Say, for this case study, that someone wants to send the message 2. They need to compute $2^{11}$ in $Z_{91}$, which is easy—using a calculator, they compute

$$2^{11} = 2048,$$

and then they divide 2048 by 91, obtaining 22.505494505494505. After some thought, they figure out that what they want is the remainder, which is the decimal part, so they subtract 22, obtaining .505494505494505, and then multiply by 91, obtaining 46. We label this number, which is the encrypted message, by $c$.

They send $c = 46$ to headquarters, where we need to decrypt it. To do this, we need, as will be explained shortly, a magical number $d$ such that $de = 1$ in $Z_\phi$.

We will find such a number in a naive, horribly inefficient way. Since $\phi = 72$, and $e = 11$, we want a multiple of 11 that is one more than a multiple of 72. We can simply generate integers that are one more than a multiple of 72, namely 73, 145, 217, and so on, checking each to see whether it is divisible by 11. Pretty quickly we find $649 = 9 \cdot 72 + 1 = 59 \cdot 11$. We see that $59 \cdot 11$ is equal to 1 in $Z_{72}$, because multiples of 72 are effectively 0.

To decrypt the message $c = 46$ at headquarters, we now want to compute $c^d = 46^{59}$ in $Z_{91}$. Unfortunately, this number—$46^{59}$—is too large and can't be computed to full accuracy on our calculator. We could use Mathematica or something, but we'll instead do a terrible algorithm that is still kind of clever: we simply start computing powers of 46, noting that to get each successive row of our table, we multiply the row above by 46:

| $k$ | $46^k$ in $Z_{91}$ |
|---|---|
| 1 | 46 |
| 2 | $46^2 = 2116 = 23$ |
| 3 | $23 \cdot 46 = 1058 = 57$ |
| 4 | 74 |
| 5 | 37 |
| 6 | 64 |
| 7 | 32 |
| 8 | 16 |
| 9 | 8 |
| 10 | 4 |
| 11 | 2 |
| 12 | 1 |

We stop there because now, knowing that $46^{12} = 1$ in $Z_{91}$, we can cleverly observe that

$$46^{59} = 46^{48+11} = 46^{12\cdot 4} \cdot 46^{11} = \left(46^{12}\right)^4 \cdot 46^{11} = (1)^4 \cdot 46^{11} = 2,$$

which is the original message!

This case study has identified the need for several algorithms.

> First, when computing $a^e$ in $Z_n$, and when computing $c^d$ in $Z_n$, we need an efficient algorithm for performing this *modular exponentiation* calculation. Note that the way we computed $46^{59}$ in $Z_{91}$ would be hopelessly time-consuming with large integers (even with small integers, it could have required us to keep going in our table up to 59 to get the desired answer, as far as we know—we lucked out by finding a much smaller power—12—for which we got 1 as the answer).
>
> Second, we need an efficient algorithm for finding the magical $d$—obviously the brute force approach of just trying multiples of $\phi$ until we find one that is 1 below a multiple of $e$ is hopeless.
>
> In the rest of the chapter we will develop these two algorithms, and prove that they produce the desired answers even when $n$, $e$, $\phi$, $c$, and $d$ are huge—like thousands of digits long.

Later we will prove, for anyone interested, that the method works.

So, this case study has given us a map for most of the rest of the chapter, namely developing an efficient algorithm to do modular exponentiation, developing an efficient algorithm to find the inverse of $e$ in $Z_\phi$, and proving some number theory facts.

The case study has not foreshadowed one other issue that we will discuss: how does headquarters come up with large prime numbers to use for $p$ and $q$?

---

## Modular Exponentiation

Our first new problem in this chapter is *modular exponentiation*, whose efficient solution is crucial for RSA encryption: given positive integers $a$ and $n$, where $a < n$, and any positive integer $k$, compute $a^k$ in $Z_n$.

The fairly obvious algorithm for solving this problem would be:

```
public int modExp( int a, int k, int n )
{
  int r = 1;
  for( int j=1; j<=k; j++ )
    r = (r * a) % n;
  return r;
}
```

Unfortunately, this algorithm has efficiency in $\Theta(k)$, and $k$ can be very large, so this is not acceptable.

## A More Efficient Algorithm for Modular Exponentiation

Suppose for simplicity that $k = 2^m$. Then to compute $a^k = a^{2^m}$ in $Z_n$, we can use the divide-and-conquer approach.

We realize that if we compute $a^{k/2} = a^{2^{m-1}}$ in $Z_n$, then we can simply square that value to compute the desired value, since $a^{k/2}a^{k/2} = a^k$. If we do this recursively all the way down, we have an algorithm that takes only $m$ operations to compute the value, rather than $2^m$.

In this case, it is easier to implement the recursion iteratively—we compute $a^1$, then square that to get $a^2$, square that to get $a^4$, and so on, until we reach $a^{2^m} = a^k$.

And, of course, at each step we reduce the result mod $n$ so that the numbers never get larger than $n^2$.

Once we have all the values of $a$ raised to all the relevant powers of 2, like $a^1$, $a^2$, $a^4$, $\ldots$, we can then express $k$ as a sum of powers of 2—basically just expressing it in binary notation—and then multiply together the terms that are needed to produce $a^k$.

Of course, if $k$ is not a power of 2, we need to adjust the algorithm somewhat, but it will still be very efficient.

It is obvious that this algorithm requires $m$ steps, and since $m = \log_2(k) \leq \log_2(n)$, the algorithm takes $O(\log_2(n))$ steps. And, each step involves multiplying together two integers of size at most $n$ and reducing the result mod $n$, which is manageable.

Note that for this problem (doing RSA encryption), we are not interested in the efficiency analysis to compare to alternative algorithms as $n$ gets arbitrarily large. Instead, we want to make sure that for a fairly large $n$ (one for which it is hopeless to factor $n$), all the work can be done in a reasonable amount of time. For example, if $n$ is around 1024 bits in size, then the modular exponentiation algorithm we are sketching out takes 10 steps, and each step takes less than a million single bit multiplications and a few thousand single bit additions (using Karatsuba, say), so the total work to compute $a^k$ in $Z_n$ can be done in a few seconds.

---

## Example of Efficient Modular Exponentiation Algorithm by Hand

First let's compute $a^e$ in $Z_n$ where $a = 73$, $e = 29$, and $n = 151$. The main idea is that since $29 = 16 + 8 + 4 + 1$,

$$73^{29} = 73^{16} \cdot 73^8 \cdot 73^4 \cdot 73^1,$$

and we note that $a^{2m} = (a^m)^2$, so we can compute all the values of $a$ raised to a power of 2 by repeated squaring, so in $Z_{151}$ we compute

| | | | |
|---|---|---|---|
| $73^1$ | | | $= 73$ |
| $73^2$ | | $= 5329$ | $= 44$ |
| $73^4$ | $= 44^2$ | $= 1936$ | $= 124$ |
| $73^8$ | $= 124^2$ | $= 15376$ | $= 125$ |
| $73^{16}$ | $= 125^2$ | $= 15625$ | $= 72$ |

Now that we have all the values of $a$ raised to the necessary powers of 2, we can multiply them together to obtain the answer, which is

$$73^{16} \cdot 73^8 \cdot 73^4 \cdot 73^1 = 72 \cdot 125 \cdot 124 \cdot 73 = 27.$$

---

**The Final Slick Modular Exponentiation Algorithm**

All we have to do now is remember how to convert a given decimal integer into binary and integrate that into our successive squares algorithm.

The idea of converting a given integer $k$ into binary is to note that if $k$ is even, its last (or least significant or rightmost) bit is 0, and if $k$ is odd, that bit is 1. Then we divide $k$ by 2 and use the same idea to get the rightmost bit of $k/2$, which is the 2's bit of $k$, and so on.

So, our final slick algorithm is demonstrated here for the same problem as was done previously. We can make a chart where the first column contains the successive squares of $a$, the second column holds the repeated halvings of $e$, using integer arithmetic, and the third column contains the cumulative products of the items in the first column where the second column contains an odd number.

Doing this gives us:

```
 73   29    73
 44   14
124    7   143
125    3    57
 72    1    27
```

---

## Exercise 12 (efficient modular exponentiation by hand)

Perform the efficient modular exponentiation algorithm using a calculator for $a = 29613$, $e = 8075$, working in $Z_{31861}$.

The folder `Code/RSA` at the course web site contains a class `ModExp` that does this algorithm, but only use it to check your work—remember that you will be asked to do this computation on the test with just a calculator.

---

## The Extended Euclidean Algorithm

We now need to study the Euclidean algorithm, which may be one of the oldest published algorithms. The Euclidean algorithm provides an efficient solution to the problem "find the greatest common divisor of two given positive integers."

The Euclidean algorithm will be the tool that lets us efficiently obtain the magical decrypting number $d$ when given $n = pq$ and $e$.

Recall that a positive integer $d$ divides, or is a divisor of, a positive integer $a$ if and only if there is some integer $q$ such that $a = qd$. The greatest common divisor of $a$ and $b$, written $\gcd(a, b)$, is the largest integer that is a divisor of both $a$ and $b$.

In elementary school you might have learned this concept when working with fractions—to reduce a fraction, you can find the greatest common divisor of the top and bottom and then divide both by that number. More likely you learned to completely factor the top and bottom into prime factors and then cancel matching factors. That algorithm was ridiculously inefficient, it turns out. We want to compute the `gcd` not to find the greatest common divisor, because we will only use it with numbers where the `gcd` is 1, but to find additional crucial information.

Given two positive integers $a$ and $b$, we perform integer division on them, obtaining non-negative integers $q$ and $r$, with $r < b$, such that

$$a = qb + r.$$

When you learned to do division in elementary school, you were learning an algorithm to find $q$ and $r$.

**Crucial Fact about the GCD:** With $q$ and $r$ as above,

$$\gcd(a, b) = \gcd(b, r).$$

**Proof:** We have to prove two directions, showing that any number that divides $a$ and $b$ also divides $b$ and $r$, and that any number that divides $b$ and $r$ also divides $a$ and $b$ (this of course proves that the two greatest common divisors are the same):

Suppose $d$ divides $a$ and $d$ divides $b$. Then $d$ divides $b$ (duh!), and since $r = a - qb$, $d$ clearly divides $r$.

For the other direction, suppose $d$ divides $b$ and $d$ divides $r$. Then since $a = qb + r$, $d$ divides $a$, and of course $d$ divides $b$.

Based on this fact we can easily write recursive code to compute the `gcd`:

```
public static int gcd( int a, int b )
{
  if( a == 0 )  return b;
  else if( b == 0 )  return a;
  else{
    int q = a/b, r = a%b;
    return gcd( b, r );
  }
}
```

**Example** Here is a chart showing the sequence of recursive calls (the algorithm is easily written using a loop instead of recursion) for the initial call `gcd(1861,757)`:

| $a$ | $b$ | $r$ | $q$ |
|---|---|---|---|
| 1861 | 757 | 347 | 2 |
| 757 | 347 | 63 | 2 |
| 347 | 63 | 32 | 5 |
| 63 | 32 | 31 | 1 |
| 32 | 31 | 1 | 1 |
| 31 | 1 | 0 | 31 |
| 1 | 0 | | |

## The Extension

You may have studied this algorithm before, but now we want to use it in a much more powerful way. It turns out that a simple extension to this algorithm proves the following fact and gives us what we need for getting the magical decrypting $d$ for RSA encryption.

**Fact** Given any positive integers $a$ and $b$, there exist integers $s$ and $t$ (one positive, one negative, actually) such that

$$\gcd(a, b) = sa + tb.$$

**Proof:** This fact actually follows from the earlier proof that $\gcd(a, b) = \gcd(b, r)$, and induction on the number of calls to the recursive function.

The base case is when one of the numbers, say $b$, is 0, and the other, $a$, is not. In that case, the gcd is the non-zero number, so the desired result is obtained by taking $s = 1$ and $t$ equal to any number whatsoever, say $w$, since

$$\gcd(a, 0) = a = 1 \cdot a + w \cdot 0.$$

For the inductive step, we assume that we have numbers $s'$ and $t'$ such that

$$\gcd(b, r) = s'b + t'r.$$

To find the desired multipliers $s$ and $t$, such that $\gcd(a, b) = sa + tb$, we simply note that this expression does not involve $r$, and that $\gcd(a, b) = \gcd(b, r)$, so we substitute $r = a - qb$ into the above equation and obtain

$$\gcd(a, b) = \gcd(b, r) = s'b + t'r = s'b + t'(a - qb) = t'a + (s' - t'q)b.$$

Thus, if we take $s = t'$ and $t = s' - t'q$, we have the desired result.

## Extended Euclidean Algorithm Code

Here is code for the extended Euclidean algorithm. Note that we want each call to `gcd` to return three values—the gcd, $s$, and $t$—so we use an array.

```
public static int[] gcd( int a, int b )
{
  int[] x = new int[3];
  if( a == 0 )
  {
     x[0] = b;  x[1] = 0; x[2] = 1;
     return x;
  }
  else if( b == 0 )
  {
    x[0] = a;  x[1] = 1;  x[2] = 0;
    return x;
  }
  else
  {
    int q = a/b, r = a%b;
    int[] xprime = gcd( b, r );

    x[0] = xprime[0];
    x[1] = xprime[2];  x[2] = xprime[1] - q*xprime[2];

    return x;
  }
}
```

**Example** Here is a chart showing the sequence of recursive calls (the algorithm is easily written using a loop instead of recursion) for the initial call `gcd(1861,757)`, with the returning values of $s$ and $t$ also shown:

| $a$ | $b$ | $r$ | $q$ | s | t |
|---|---|---|---|---|---|
| 1861 | 757 | 347 | 2 | 24 | -59 |
| 757 | 347 | 63 | 2 | -11 | 24 |
| 347 | 63 | 32 | 5 | 2 | -11 |
| 63 | 32 | 31 | 1 | -1 | 2 |
| 32 | 31 | 1 | 1 | 1 | -1 |
| 31 | 1 | 0 | 31 | 0 | 1 |
| 1 | 0 | | | 1 | 0 |

## Exercise 13 (performing extended Euclidean algorithm by hand)

Create the chart for the extended Euclidean algorithm on the instance of the problem "compute `gcd( 439, 211 )`."

First work downward, producing $a$, $b$, $r$, and $q$ for each row.

Then work back up, filling in the columns for $s$ and $t$. For the bottom row, which for our RSA work will always have $a = 1$ and $b = 0$, use $s = 1$ and $t = 0$. Work upward using the formulas

$$s = t'$$

$$t = s' - t'q,$$

where $s'$ and $t'$ are the values in a row, and $s$ and $t$ are the values in the row above.

Each time you compute $s$ and $t$ for a row, be sure to check that for the values in that row, $sa + tb$ is equal to the final GCD (which will always be 1 for our RSA work).

After doing this, do it again, but this time use $s = 1$ and $t = 1$ in the bottom row. Note that we can use $s = 1$ and $t =$anything on the bottom row and still have $sa + tb = 1$, since $b = 0$ on the bottom row. Note how the pattern of alternation reverses. Figure out why this algorithm always gives alternating signs in the $s$ and $t$ columns.

Note that the class `GCD` in the `RSA` folder implements this algorithm, using a choice of $s$ and $t$ for the bottom row that produces a positive value for $t$, as desired for the RSA work (also note that the column labels are different, maintaining $gf + de = 1$ on each row).

## Efficiency Analysis for the Euclidean Algorithm

As was the case with modular exponentiation, if we can't do the Euclidean algorithm quickly enough for huge numbers $a$ and $b$, this whole scheme won't be practical. The following theorem is somewhat reassuring.

**Theorem** Let the Fibonacci sequence be defined by the recurrence relation $f_{k+2} = f_{k+1} + f_k$, along with the initial conditions $f_1 = 1$ and $f_2 = 1$. If $n > m \geq 1$, and $\gcd(n, m)$ causes $k \geq 1$ recursive calls, then $n \geq f_{k+2}$ and $m \geq f_{k+1}$.

**Proof:** We use induction on $k$.

For the base case, when $k = 1$, if $\gcd(n, m)$ causes just one call, we want to prove that $n \geq f_3$ and $m \geq f_2$. Since $f_2 = 1$ and $m \geq 1$ by an assumption of this theorem, that part is easy. Since $n > m$ by assumption, $n > 1$, so $n \geq f_3 = 2$.

Now, assume that the desired results are true for $k - 1$ calls, for $k \geq 2$, and suppose that $\gcd(n, m)$ causes $k$ calls. The first call simply calls $\gcd(m, n \bmod m)$, which must cause $k - 1$ calls, so the induction hypothesis applies, and therefore $m \geq f_{k+1}$ and $n \bmod m \geq f_k$. The first inequality is one of the things we need to prove, so yay! For the other, note that $n = qm + (n \bmod m)$ for some integer $q$, and since $n > m$ by assumption, we must have $q \geq 1$, so

$$n = qm + (n \bmod m) \geq m + (n \bmod m) \geq f_{k+1} + f_k = f_{k+2}$$

as desired.

---

This theorem shows that the worst-case for the time required for the Euclidean algorithm is when two successive Fibonacci numbers are being worked on.

We mentioned earlier that one can show that the Fibonacci sequence has a growth rate in $\Theta(1.6^n)$. So, the previous theorem show that the time for the Euclidean algorithm is in $\Theta\left(\log_{1.6}(n)\right)$, meaning that this part of the RSA scheme is quite feasible.

---

## Building the RSA Scheme

Now we can recall how to create an RSA scheme and see where we stand.

First, we come up with two large prime numbers $p$ and $q$ (how we do this will be covered later) and compute $n = pq$. We also compute $\phi = (p-1)(q-1)$. Then we somehow come up with a positive integer $e$ such that $\gcd(\phi, e) = 1$, and at the same time as we check to make sure that $\gcd(\phi, e) = 1$, compute $s$ and $d$ such that

$$1 = s\phi + de,$$

with $d > 0$ (we were calling this number $t$ when discussing the Euclidean algorithm).

We proved earlier that the extended Euclidean algorithm is efficient enough to use for the purpose of getting the magical $d$.

As seen in Exercise 13, we have to choose either $s = 1$ and $t = 0$ or $s = 1$ and $t = 1$ for the bottom row to produce $d > 0$.

We have also seen that modular exponentiation is efficient enough to make the computations of $a^e$ in $Z_n$ and $c^d$ in $Z_n$ feasible.

---

**Exercise 14** (the entire RSA process by hand)

Work out by hand/calculator and write up a careful example of the entire RSA process, including computing $n$ to publish, encrypting $a$, finding the secret $d$, and decrypting $c$, with $p = 137$, $q = 241$, $e = 53$, and $a = 12345$. You will be expected to do all of this (including efficient modular exponentiation and the extended Euclidean algorithm) by hand/calculator during the test, so do the exercise that way, using just an ordinary calculator.

---

Note that various classes in the `RSA` folder implement all the parts of this process. So, with a little judicious copying and pasting into and out of email messages, we can communicate securely by email.

---

**Project 4** (breaking RSA)

Please don't share your answer to this project with anyone else, and don't work in a group—remember that the more people who complete a project, the less I will weight it in the course grades.

Suppose you intercept the message 1027795314451781443748475386257882516 and you suspect it was encrypted using RSA with $n = 1029059010426758802790503300595323911$ and $e = 2287529$. Break the encryption and determine the original message.

You might find some of the code in the `Code/RSA` folder helpful for this. You might also find that you need to use `wolfram alpha` to factor $n$ (on my machine this works in a reasonable time). Note that I have used the scheme shown in `Encode/Decode` to switch between integers and strings.

Email me your results, including the decrypted plain-text message and details of how you got it.

---

## Fermat's Little Theorem

Earlier we saw that in $Z_n$, computing quantities like $a^k$ leads to some interesting patterns. Now we want to prove a famous number theory fact that will let us understand exactly why the RSA scheme works. It will also help us to believe the method we will use for finding large prime numbers.

**Theorem** If $n$ is prime, and $a$ is any positive integer, then $a^n = a$ in $Z_n$.

**Proof:** We will prove this by mathematical induction on $a$.

The base case, where $a = 1$, is trivial: obviously $1^n = 1$ in $Z_n$.

Assume that $a^n = a$ in $Z_n$ for some $a > 1$. We want to show that $(a+1)^n = a+1$ in $Z_n$.

Let's do a specific example to motivate the upcoming simple proof. Use $n = 7$ and $a = 3$. If we compute the powers of 3 in $Z_7$, we get $3^1 = 3$, $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$, $3^6 = 1$, and $3^7 = 3$, which shows that the results holds for 3. Let's use that fact to show that it holds for 4, namely that $4^7 = 4$ in $Z_7$.

Recall the binomial theorem, which says that

$$(3+1)^7 = \binom{7}{0}3^7 + \binom{7}{1}3^6 + \binom{7}{2}3^5 + \binom{7}{3}3^4 + \binom{7}{4}3^3 + \binom{7}{5}3^2 + \binom{7}{6}3^1 + \binom{7}{7}3^0$$

We can quickly use Pascal's triangle to obtain these coefficients:

```
                  1
                1   1
              1   2   1
            1   3   3   1
          1   4   6   4   1
        1   5  10  10   5   1
      1   6  15  20  15   6   1
    1   7  21  35  35  21   7   1
```

The big observation is that all the binomial coefficients in row 7 other than the first and last ones are divisible by 7, hence equal to 0 in $Z_7$. So, we have

$$(3+1)^7 = 3^7 + 1,$$

since all the other terms are 0 in $Z_7$. But, then since we know the desired property holds—that is, $3^7 = 3$ in $Z_7$, this means that

$$(3+1)^7 = 3 + 1,$$

or that $4^7 = 4$ in $Z_7$.

In our actual situation, we know that

$$(a+1)^n = \sum_{k=0}^{n} \binom{n}{k} a^k = a^n + \left[ \sum_{k=1}^{n-1} \binom{n}{k} a^k \right] + 1 = a^n + 1$$

in $Z_n$, because we will prove below that all the quantities $\binom{n}{k}$, for $1 \le k \le n-1$, are equal to 0 in $Z_n$. Then, since the induction hypothesis says that $a^n = a$ in $Z_n$, we have our desired result, namely that

$$(a+1)^n = a+1$$

in $Z_n$.

To finish the proof, we recall that for $1 \le k \le n-1$,

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)(n-k)!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{1\cdot 2\cdots k}.$$

For example,

$$\binom{7}{4} = \frac{7!}{4!3!} = \frac{7\cdot 6\cdot 5\cdot 4\cdot 3!}{1\cdot 2\cdot 3\cdot 4\cdot 3!}$$

$$= \frac{7\cdot 6\cdot 5\cdot 4\cdot \cancel{3!}}{1\cdot 2\cdot 3\cdot 4\cdot \cancel{3!}} = \frac{7\cdot \cancel{6}\ \cdot 5\cdot \cancel{4}}{1\cdot \cancel{2}\ \cdot \cancel{3}\ \cdot \cancel{4}} = 7\cdot 5.$$

Then we observe that this fraction must actually end up being an integer after cancelling all the integers on the bottom with factors in the top. And, since all the integers on the bottom are less than $n$, and $n$ is prime, clearly the leading factor $n$ can't be cancelled.

---

Now we know that if $n$ is prime, then for any positive integer $a$, $a^n = a$ in $Z_n$, which means $n$ divides $a^n - a = a(a^{n-1} - 1)$. Then, if $a$ is between 2 and $n-1$ inclusive (which is how we usually represent numbers in $Z_n$), since $n$ is prime and can't have an integer between 2 and $n-1$ as a factor (it's only factors are 1 and $n$), that $n$ divides $a^{n-1} - 1$, or, in other words, $a^{n-1} = 1$ in $Z_n$.

---

## Getting Large Primes

In addition to providing facts that will be crucial in showing how and why RSA works, the preceding theorem gives us some context for our method of producing large prime numbers.

The preceding theorem says that for any $a < n$, if $n$ is prime, then $a^{n-1} = 1$ in $Z_n$.

The converse turns out to almost be true! It turns out that if $2^{n-1} = 1$ in $Z_n$, then $n$ is incredibly likely to be a prime number—but not always!

So, this is a reasonable process for obtaining large prime numbers: take a large integer $p$ and perform the test (compute $2^{p-1}$ in $Z_p$ and see if you get 1). If you do get 1, it is a really safe bet that $p$ is prime. If you don't get 1, move on to the next odd number and try again. In a reasonable number of steps like this, you'll find a prime number larger than or equal to the place you started.

It has been proven that the distribution of primes is thick enough that this process will hit a prime soon enough to be reasonable.

If we are really unlucky, we might pick a so-called pseudo-prime—an integer $p$ for which the test is passed, but $p$ is not prime.

---

**Smallest Fermat Psuedo-Prime**

If we apply this test to integers 2, 3, 4, and so on, the first one for which the test fails is $n = 341$.

We can use our modular exponentiation algorithm to compute $2^{340}$ in $Z_{341}$—here is the chart produced:

```
  2        340
  4        170
 16         85        16
256         42
 64         21         1
  4         10
 16          5        16
256          2
 64          1         1
```

which shows that 341 passes the test. However, 341 is not prime, because $341 = 11 \cdot 31$.

---

In 2002, Agrawal, Kayal, and Saxena found a polynomial-time algorithm (polynomial in the number of bits needed to represent $n$) for determining whether a given integer $n$ is prime. Before this result (or the fairly recent results leading up to it), this problem might have been thought to be NP-complete (we will learn what this means toward the end of this course). Note that their algorithm does not give factors if $n$ is found to be composite, it only says whether it's composite. We won't attempt to learn about this algorithm, but you should be aware of its existence.

---

The `BigInteger` class method `isProbablePrime` gives a more reliable way to determine whether a given integer is probably a prime number (probably uses the Miller-Rabin test, a more accurate but still efficient probabilistic test for primality).

---

## A Proof that RSA Encryption/Decryption Works

We are now ready to understand RSA encryption/decryption. Specifically, we want to prove that it works.

Recall that at headquarters large primes $p$ and $q$ are picked (we hope in a way that outsiders won't be able to reverse-engineer—for example, deciding to use $n$ with $M$ digits and then using primes near the square root of $10^M$ would be a bad idea), and $n = pq$ and $\phi = (p-1)(q-1)$ are computed. A number $e$ is picked, such that $\gcd(\phi, e) = 1$, and the extended Euclidean algorithm is used to find $s$ and $d$, with $s < 0$ and $d > 0$, such that $1 = s\phi + de$.

The values $n$ and $e$ are published. As far as anyone on Earth knows (or at least is admitting publicly), there is no algorithm for using $n$ to determine $\phi$, or $p$ or $q$, in a reasonable amount of time (assuming $p$ and $q$ are large enough, reasonable is something like "before the sun goes super-nova").

Someone out in the field wants to send the message $a$ to headquarters. They simply compute $c = a^e$ in $Z_n$ and send $c$ to headquarters.

Back at headquarters, they simply compute $c^d$ in $Z_n$ and get $a$ back.

We now want to show that this works. In all the following, note that since $s < 0$, $-s$ is a positive integer, so using it as an exponent makes sense without worrying about multiplicative inverses.

Since $c = a^e$ in $Z_n$, there is some integer $g$ such that $c = a^e + gn$ in ordinary arithmetic. So, using ordinary integer arithmetic (this is done for the proof, but not in the algorithm), since $c = a^e + gn$,

$$c^d = (a^e + gn)^d = (a^e)^d + hn$$

for an integer $h$ obtained by using the binomial theorem (in $(a+b)^n$, all terms except the $a^n$ term have $b$ as a factor). So,

$$c^d = a^{ed} + hn = a^{1-s\phi} + hn = a^1 a^{-s\phi} + hn = aa^{-s\phi} + hn.$$

Now, in $Z_p$, $hn = 0$ (since $p$ is a factor of $n$), and so

$$aa^{-s\phi} + hn = aa^{-s(p-1)(q-1)} = a(a^{p-1})^{-s(q-1)} = a1^{-s(q-1)} = a,$$

by Fermat's little theorem (which says that $a^{p-1} = 1$ in $Z_p$). So, in $Z_p$, $c^d = a$.

Similarly, in $Z_q$, $hn = 0$, so

$$aa^{-s\phi} + hn = aa^{-s(p-1)(q-1)} = a(a^{q-1})^{-s(p-1)} = a1^{-s(p-1)} = a,$$

so in $Z_q$, $c^d = a$.

So, we know that $c^d = a$ in $Z_p$, and $c^d = a$ in $Z_q$. Now we need to show that $c^d = a$ in $Z_n$.

Since $c^d = a$ in $Z_p$, there is some integer $\alpha$ such that $c^d - a = \alpha p$ in ordinary arithmetic. Similarly, since $c^d = a$ in $Z_q$, there is some integer $\beta$ such that $c^d - a = \beta q$ in ordinary arithmetic. So, in ordinary arithmetic $\alpha p = \beta q$, which shows that $p$ is a factor of $\beta q$. Since $p$ and $q$ are primes, this can only happen if $p$ is a factor of $\beta$, which means there is some integer $\gamma$ such that $\beta = \gamma p$. So,

$$c^d - a = \beta q = \gamma p q = \gamma n$$

in ordinary arithmetic, so $c^d = a$ in $Z_n$ as desired.