# 一、非对称加密算法

1、乙方生成两把密钥（公钥和私钥）。公钥是公开的，任何人都可以获得，私钥则是保密的。

2、甲方获取乙方的公钥，然后用它对信息加密。

3、乙方得到加密后的信息，用私钥解密。

# 二、RSA算法

1977年，三位数学家Rivest、Shamir 和 Adleman 设计了一种算法，可以实现非对称加密。这种算法用他们三个人的名字命名，叫做RSA算法。从那时直到现在，RSA算法一直是最广为使用的"非对称加密算法"。毫不夸张地说，只要有计算机网络的地方，就有RSA算法。

这种算法非常可靠，密钥越长，它就越难破解。根据已经披露的文献，目前被破解的最长RSA密钥是768个二进制位。也就是说，长度超过768位的密钥，还无法破解（至少没人公开宣布）。因此可以认为，1024位的RSA密钥基本安全，2048位的密钥极其安全。

# 数学基础

## 互质关系

如果两个正整数，除了1以外，没有其他公因子，我们就称这两个数是互质关系（coprime）。比如，15和32没有公因子，所以它们是互质关系。这说明，不是质数也可以构成互质关系。

关于互质关系，不难得到以下结论：

1. 任意两个质数构成互质关系，比如13和61。

2. 一个数是质数，另一个数只要不是前者的倍数，两者就构成互质关系，比如3和10。

3. 如果两个数之中，较大的那个数是质数，则两者构成互质关系，比如97和57。

4. 1和任意一个自然数是都是互质关系，比如1和99。

5. p是大于1的整数，则p和p-1构成互质关系，比如57和56。

6. p是大于1的奇数，则p和p-2构成互质关系，比如17和15。

## 2、欧拉函数

请思考以下问题：

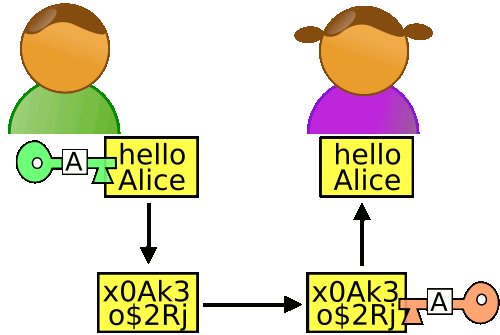
任意给定正整数n，请问在小于等于n的正整数之中，有多少个与n构成互质关系？（比如，在1到8之中，有多少个数与8构成互质关系？）

计算这个值的方法就叫做欧拉函数，以φ(n)表示。在1到8之中，与8形成互质关系的是1、3、5、7，所以 φ(n) = 4。

φ(n) 的计算方法并不复杂，但是为了得到最后那个公式，需要一步步讨论。

# 密钥生成

我们通过一个例子，来理解RSA算法。假设爱丽丝要与鲍勃进行加密通信，她该怎么生成公钥和私钥呢？



* 第一步，随机选择两个不相等的质数p和q。

爱丽丝选择了61和53。（实际应用中，这两个质数越大，就越难破解。）

* 第二步，计算p和q的乘积n。

爱丽丝就把61和53相乘。

|  |
| --- |
| n = 61×53 = 3233 |

n的长度就是密钥长度。3233写成二进制是110010100001，一共有12位，所以这个密钥就是12位。实际应用中，RSA密钥一般是1024位，重要场合则为2048位。

* 第三步，计算n的欧拉函数φ(n)。

根据公式：

|  |
| --- |
| φ(n) = (p-1)(q-1) |

爱丽丝算出φ(3233)等于60×52，即3120。

* 第四步，随机选择一个整数e，条件是1< e < φ(n)，且e与φ(n) 互质。

爱丽丝就在1到3120之间，随机选择了17。（实际应用中，常常选择65537。）

* 第五步，计算e对于φ(n)的模反元素d。

所谓"模反元素"就是指有一个整数d，可以使得ed被φ(n)除的余数为1。

|  |
| --- |
| ed ≡ 1 (mod φ(n)) |

这个式子等价于

|  |
| --- |
| ed - 1 = kφ(n) (k∈Z) |

于是，找到模反元素d，实质上就是对下面这个二元一次方程求解。

|  |
| --- |
| ex + φ(n)y = 1 |

已知 e=17, φ(n)=3120，

|  |
| --- |
| 17x + 3120y = 1 |

这个方程可以用"扩展欧几里得算法"求解，此处省略具体过程。总之，爱丽丝算出一组整数解为 (x,y)=(2753,-15)，即 d=2753。

至此所有计算完成。

* 第六步，将n和e封装成公钥，n和d封装成私钥。

在爱丽丝的例子中，n=3233，e=17，d=2753，所以公钥就是 (3233,17)，私钥就是（3233, 2753）。

* 总结，实际上就是计算n,e,d的过程

pq的作用用于求n==pq，再用 (p-1)(q-1)求φ(n)，在φ(n)范围内随机选择即为e，d==e对于φ(n)的模反元素

# 验证RSA算法的可靠性

公钥公开，私钥不公开，故d被破解即RSA算法被破解。

回顾上面的密钥生成步骤，一共出现六个数字：

|  |
| --- |
| p,q,n,φ(n),e,d |

这六个数字之中，公钥用到了两个（n和e），其余四个数字都是不公开的。其中最关键的是d，因为n和d组成了私钥，一旦d泄漏，就等于私钥泄漏。

那么，有无可能在已知n和e的情况下，推导出d？

* ed=1 (mod φ(n))。只有知道e和φ(n)，才能算出d。
* φ(n)=(p-1)(q-1)。只有知道p和q，才能算出φ(n)。
* n=pq。只有将n因数分解，才能算出p和q。

结论：如果n可以被因数分解，d就可以算出，也就意味着私钥被破解。

可是，大整数的因数分解，是一件非常困难的事情。目前，除了暴力破解，还没有发现别的有效方法。维基百科这样写道："对极大整数做因数分解的难度决定了RSA算法的可靠性。换言之，对一极大整数做因数分解愈困难，RSA算法愈可靠。

假如有人找到一种快速因数分解的算法，那么RSA的可靠性就会极度下降。但找到这样的算法的可能性是非常小的。今天只有短的RSA密钥才可能被暴力破解。到2008年为止，世界上还没有任何可靠的攻击RSA算法的方式。

只要密钥长度足够长，用RSA加密的信息实际上是不能被解破的。"

举例来说，你可以对3233进行因数分解（61×53），但是你没法对下面这个整数进行因数分解。

|  |
| --- |
| 1230186684530117755130494958384962720772853569595334792197322452151726400507263657518745202199786469389956474942774063845925192557326303453731548268507917026122142913461670429214311602221240479274737794080665351419597459856902143413 |

它等于这样两个质数的乘积：

|  |
| --- |
| 3347807169895689878604416984821269081770479498371376856892431388982883793878002287614711652531743087737814467999489 |

　　　　×

|  |
| --- |
| 36746043666799590428244633799627952632279158164343087642676032283815739666511279233373417143396810270092798736308917 |

事实上，这大概是人类已经分解的最大整数（232个十进制位，768个二进制位）。比它更大的因数分解，还没有被报道过，因此目前被破解的最长RSA密钥就是768位。

# 加密与解密

有了公钥和密钥，就能进行加密和解密了。

## 1、加密要用公钥 (n,e)

假设鲍勃要向爱丽丝发送加密信息m，他就要用爱丽丝的公钥 (n,e) 对m进行加密。这里需要注意，m必须是整数（字符串可以取ascii值或unicode值），且m必须小于n。

所谓"加密"，就是算出下式的c：

|  |
| --- |
| m^e ≡ c (mod n) |

爱丽丝的公钥是 (3233, 17)，鲍勃的m假设是65，那么可以算出下面的等式：

65^17 ≡ 2790 (mod 3233)

于是，c等于2790，鲍勃就把2790发给了爱丽丝。

## 2、解密要用私钥(n,d)

爱丽丝拿到鲍勃发来的2790以后，就用自己的私钥(3233, 2753) 进行解密。可以证明，下面的等式一定成立：

|  |
| --- |
| c^d ≡ m (mod n) |

也就是说，c的d次方除以n的余数为m。现在，c等于2790，私钥是(3233, 2753)，那么，爱丽丝算出

2790^2753 ≡ 65 (mod 3233)

因此，爱丽丝知道了鲍勃加密前的原文就是65。

至此，"加密--解密"的整个过程全部完成。

我们可以看到，如果不知道d，就没有办法从c求出m。而前面已经说过，要知道d就必须分解n，这是极难做到的，所以RSA算法保证了通信安全。

你可能会问，公钥(n,e) 只能加密小于n的整数m，那么如果要加密大于n的整数，该怎么办？有两种解决方法：一种是把长信息分割成若干段短消息，每段分别加密；另一种是先选择一种"对称性加密算法"（比如DES），用这种算法的密钥加密信息，再用RSA公钥加密DES密钥。

# 私钥解密的证明

a=b(mod c)等价于a/c的余数是b,a mod c ==b

最后，我们来证明，为什么用私钥解密，一定可以正确地得到m。也就是证明下面这个式子：

|  |
| --- |
| c^d ≡ m (mod n) |

因为，根据加密规则

|  |
| --- |
| ｍ^e ≡ c (mod n) |

于是，c可以写成下面的形式：

|  |
| --- |
| c = m^e - kn(h∈Z) |

将c代入要我们要证明的那个解密规则：

|  |
| --- |
|  |

由于(a-b)^n=a^n-C1n a^(n-1)b+C2n a^(n-2)b^2+...+(-b)^n

它等同于求证

|  |
| --- |
| m^(ed) ≡ m (mod n) |

由于

|  |
| --- |
| ed ≡ 1 (mod φ(n)) |

所以

|  |
| --- |
| ed = hφ(n)+1 |

将ed代入：

|  |
| --- |
| m^(hφ(n)+1) ≡ m (mod n) |

接下来，分成两种情况证明上面这个式子。

* 当m与n互质。

根据欧拉定理，此时

|  |
| --- |
| =====> m^φ(n)=kn+1 (k∈Z) |

得到

|  |
| --- |
| 证明 (kn+1)^h\*m=m(mod n)展开即可 |

原式得到证明。

* 当m与n不是互质关系。

此时，由于n等于质数p和q的乘积，所以m必然等于kp或kq。

以 m = kp为例，考虑到这时k与q必然互质，则根据欧拉定理，下面的式子成立：

|  |
| --- |
| (kp)^q-1 ≡ 1 (mod q) |

进一步得到

|  |
| --- |
|  |

即

|  |
| --- |
|  |

将它改写成下面的等式

|  |
| --- |
|  |

这时t必然能被p整除，即 t=t'p

|  |
| --- |
|  |

因为 m=kp，n=pq，所以

|  |
| --- |
|  |

原式得到证明。

# 快速幂模算法

在讲解快速幂取模算法之前，我们先将几个必备的知识

1.对于取模运算：

(a\*b)%c=(a%c)\*(b%c)%c

这个是成立的：也是我们实现快速幂的基础

核心思想在于：

将大数的幂运算拆解成了相对应的乘法运算，利用上面的式子，始终将我们的运算的数据量控制在c的范围以下，这样我们可以客服朴素的算法的缺点二，我们将计算的数据量压缩了很大一部分，当指数非常大的时候这个优化是更加显著的，我们用Python来做一个实验来看看就知道我们优化的效率有多高了

## 算法实现：

|  |
| --- |
| # 快速幂模运算，把b拆分为二进制，遍历b的二进制，当二进制位为0时不计入计算  def quick\_pow\_mod(a, b, c):  a = a % c  ans = 1  # 这里我们不需要考虑b<0，因为分数没有取模运算  while b != 0:  # 判断b的二进制最后一位数是不是1，是则参与计算  if b & 1:  ans = (ans \* a) % c  # ans = (ans \* a) % c,理论上等价于 ans = (ans % c) \* (a % c)但是不知道为什么这样写会出错。已解决，因为可能最后一次相乘的时候返回一个未除尽的数  # 相当于遍历二进制的b  b >>= 1  # A(n) == A(n-1)^2，% c可以提高效率  a = (a % c) \* (a % c)  return ans |

我们现在来看核心原理：

对于任何一个整数的模幂运算

|  |
| --- |
| a^b%c |

对于b我们可以拆成二进制的形式

|  |
| --- |
| b=b0+b1\*2+b2\*2^2+...+bn\*2^n |

这里我们的b0对应的是b二进制的第一位（倒数第一位），那么我们的a^b运算就可以拆解成

|  |
| --- |
| a^b0\*a^b1\*2\*...\*a^(bn\*2^n) |

对于b来说，二进制位不是0就是1，那么对于bx为0的项我们的计算结果是1就不用考虑了，我们真正想要的其实是b的非0二进制位，那么假设除去了b的0的二进制位之后我们得到的式子是

|  |
| --- |
| a^(bx\*2^x)\*...\*a(bn\*2^n) |

这里我们再应用我们一开始提到的公式，那么我们的a^b%c运算就可以转化为

|  |
| --- |
| (a^(bx\*2^x)%c）\*...\*(a^(bn\*2^n)%c) |

这样的话，我们就很接近快速幂的本质了。

|  |
| --- |
| (a^(bx\*2^x)%c)\*...\*(a^(bn\*2^n)%c) |

我们会发现令

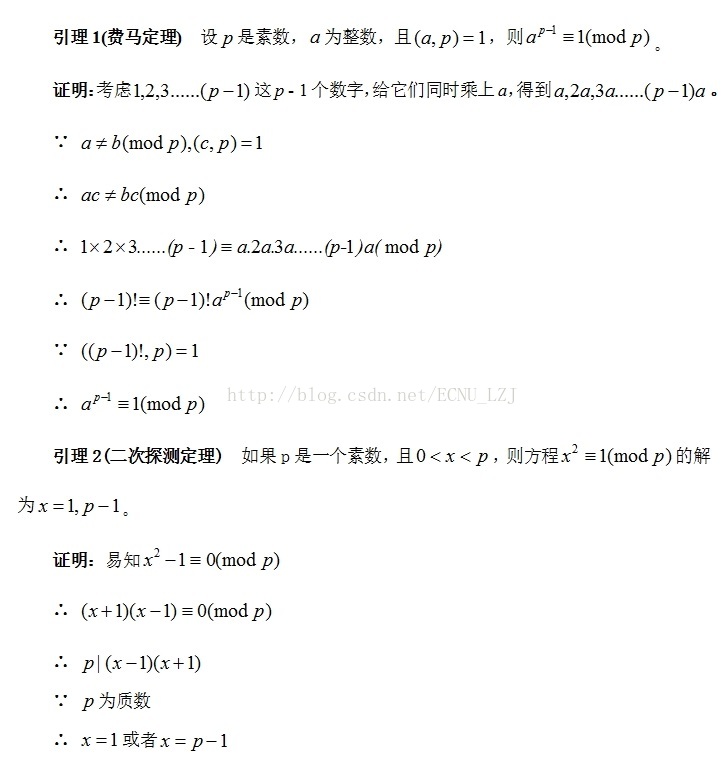
|  |
| --- |
| A1=(a^(bx\*2^x)%c) |
| ... |
| An=(a^(bn\*2^n)%c) |

这样的话，假设bx都=1，An始终等于A(n-1)的平方，依次递推。

首先，我们会观察到，我们每次都是将b的规模缩小了2倍。

那么很显然，原本的朴素的时间复杂度是O(n)。快速幂的时间复杂度就是O(logn)。在数据量越大的时候，者中优化效果越明显。

# Miller-Rabin素性测试算法



素性测试(即测试给定的数是否为素数)是近代密码学中的一个非常重要的课题。虽然Wilson定理(对于给定的正整数n，n是素数的充要条件为)给出了一个数是素数的充要条件，但根据它来素性测试所需的计算量太大，无法实现对较大整数的测试。目前，尽管高效的确定性的素性算法尚未找到，但已有一些随机算法可用于素性测试及大整数的因数分解。下面描述的Miller-Rabin素性测试算法就是一个这样的算法。

算法：

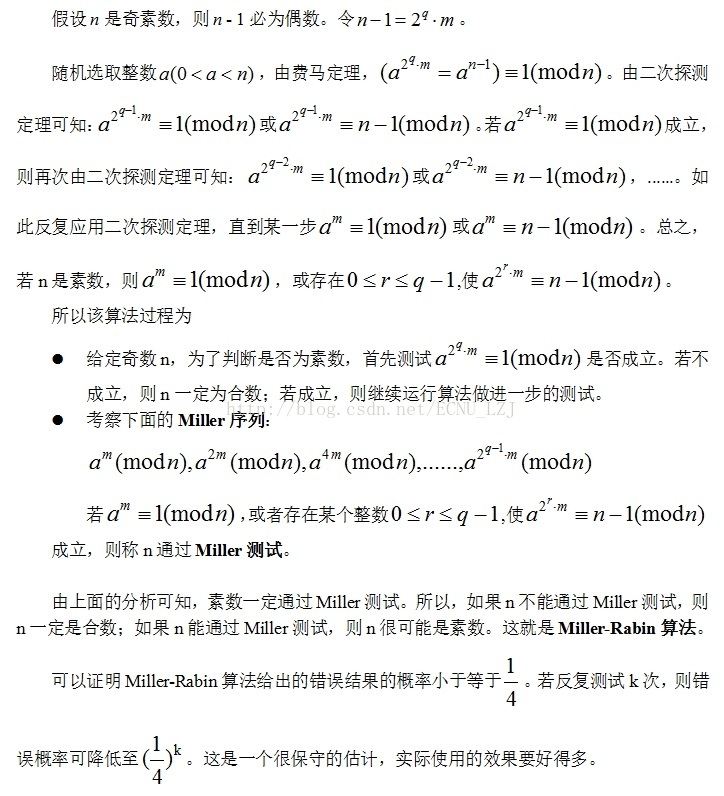
首先要知道费马定理只是n是素数的必要条件。即费马定理不成立，n一定是合数；费马定理成立，n可能是素数。接下来请看Miller-Rabin算法的分析过程。

x^2 = 1(mod p),p为质数，x小于p

x = 1或 p -1

x的偶数次方对p取余数，结果可能是1^x \* (p-1)^y对p取余数，即结果有可能是1，p-1，或(p-1)^k对p取模，当k为偶数时=1，当k为奇数时=p-1

若有解，3/4概率是质数



## 算法实现：

|  |
| --- |
| # n为要检验的大质数，a < n,k = n - 1  def miller\_rabin\_witness(a, n):  if n == 1:  return False  if n == 2:  return True  # n - 1 = m \* 2^q 求解 m, q,因为n为偶数，所以必有解  k = n - 1  # 2为 底数，n为N  q = int(math.floor(math.log(k, 2)))  while q > 0:  m = k / 2 \*\* q  # 必须同时满足两个条件，因为m有可能是未除尽的数  if k % 2 \*\* q == 0 and m % 2 == 1:  break  q = q - 1  # 先计算 a ^ (n-1) == 1 mod(n) 是否成立，不成立必定为合数  if quick\_pow\_mod(a, n - 1, n) != 1:  return False  # 计算第一项  b1 = quick\_pow\_mod(a, m, n)  for i in range(1, q + 1):  if b1 == n - 1 or b1 == 1:  return True  # 后一项等于前一项的平方mod n  b2 = b1 \*\* 2 % n  b1 = b2  if b1 == 1:  return True  return False  # Miller-Rabin素性检验算法,检验8次  def prime\_test\_miller\_rabin(p, k):  while k > 0:  a = random.randint(1, p - 1)  if not miller\_rabin\_witness(a, p):  return False  k = k - 1  return True |

# 扩展欧几里德算法

扩展欧几里德算法是用来在已知a, b求解一组x，y，使它们满足贝祖等式： ax+by = gcd(a, b) =d（解一定存在，根据数论中的相关定理）。

e取65537，故list[0] \* s + list[1] \* e = 1，list[1]为(e)mod(s)的乘法逆元，也就是e对于φ(n)的模反元素d，此方程必有解。

欧几里德算法停止的状态是： a= gcd ， b = 0 ，那么，这是否能给我们求解 x y 提供一种思路呢？因为，这时候，只要 a = gcd 的系数是 1 ，那么只要 b 的系数是 0 或者其他值（无所谓是多少，反正任何数乘以 0 都等于 0 但是a 的系数一定要是 1），这时，我们就会有： a\*1 + b\*0 = gcd

当然这是最终状态，但是我们是否可以从最终状态反推到最初的状态呢？

假设当前我们要处理的是求出 a 和 b的最大公约数，并求出 x 和 y 使得 a\*x + b\*y= gcd ，而我们已经求出了下一个状态：b 和 a%b 的最大公约数，并且求出了一组x1 和y1 使得： b\*x1 + (a%b)\*y1 = gcd ， 那么这两个相邻的状态之间是否存在一种关系呢？

我们知道： a%b = a - (a/b)\*b（这里的 “/” 指的是整除，例如 5/2=2 , 1/3=0）， 代入b\*x1 + (a%b)\*y1 = gcd 那么，我们可以进一步得到：

gcd = b\*x1 + (a-(a/b)\*b)\*y1

= b\*x1 + a\*y1 – (a/b)\*b\*y1

= a\*y1 + b\*(x1 – a/b\*y1)

对比之前我们的状态：求一组 x 和 y 使得：a\*x + b\*y = gcd ，是否发现了什么？

这里：

x = y1

y = x1 – a/b\*y1

## 算法实现：

|  |
| --- |
| # 这里的逻辑很复杂  # 扩展欧几里得算法，得到结果list[0]是a的系数，list[1]是b的系数 list[0] \* a + list[1] \* b = 1,但是有可能得到的list[1]是负数  def ex\_euclid(a, b, list):  # b==0时 a 为先求出最大公约数  if b == 0:  list[0] = 1L  list[1] = 0L  list[2] = a  else:  # 把b作为a传入函数，会形成一个交替的过程以 8,3为例，以次为[8,3],[3,2],[2,1],[1,0],即函数入栈时，第二个参数的值为入栈后第一个参数的值  # 对应着出栈时，函数的第一个参数的值等于出栈后第二个参数  # [8, 3], [3, 2], [2, 1], [1, 0]对应的list取值为[-1,3,1][1,-1,1][0,1,1][1,0,1]  ex\_euclid(b, a % b, list)  temp = list[0]  # a%b = a - (a/b)\*b ,b\*x1 + (a%b)\*y1 = gcd  # gcd = b\*x1 + (a-(a/b)\*b)\*y1 = b\*x1 + a\*y1–(a/b)\*b\*y1 = a\*y1 + b\*(x1 – a/b\*y1)  # 故出栈时，函数的第二个参数的系数等于出栈后第一个参数的系数，出栈后第二个参数b的系数=x1 – a/b\*y1  list[0] = list[1]  # 算法 1  list[1] = temp - a / b \* list[1]  # 3 \* x1 + 2 \* y1 = 1,x1 已知= 1,y1 = (1 - 3 \* x1 )/2  # 算法2，结果一致，使用时注释temp = list[0]  # list[1] = (list[2] - a \* list[0]) / b  # 求模反元素  def mod\_inverse(a, b):  # x = list[0],y = list[1],q = list[2]  list = [0L, 0L, 0L]  if a < b:  temp = a;a = b;b = temp;  ex\_euclid(a, b, list)  # 改进，将负的模反元素变为正的模反元素，根据公式 ed ≡ 1 (mod (s)),ed + s \* k ≡ 1 (mod (s)),k为任意整数，令 k = m \* e,即e的整数倍  # e(d + s \* m) ≡ 1 (mod (s)), abs(b) < s，所以只加一次即可  # 此处只对list[1]进行修改  if list[1] < 0:  list[1] = a + list[1]  return list[1] |

# RSA实现流程

1.先创建一个包含有接近一万小质数的数组，随机获得一个30-31位数的十进制数字num，判断是否与数组元素都互质，若不互质则+2，直到获得一个都互质的整数

2.对num进行Miller-Rabin素性检验8次或者更多次。如果num没有通过检验，重新随机生成大整数重复之前步骤，否则认为num是素数。Miller-Rabin素性检验有一定概率会失败。