RSA

> 计64 翁家翌 2016011446

编译运行

在ubuntu上：

```
sudo apt install libgmp-dev # 安装GMP
g++ rsa.cpp -orsa -O3 -lgmpxx -lgmp -fopenmp
```

然后输入 `./rsa [length]` 即可运行，分为两个阶段：第一阶段是运行加密指定字符串 `moderncryptography` 并解密，检查是否一致；第二阶段是随机生成长度为 `length` 的只包含数字的字符串，测量运行时间并检查正确性。

一个可能的输出结果如下：（每次生成的p和q都可能不一样）

```
➜  RSA算法实现 git:(master) ✗ ./rsa 10000000
p:
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602112011387987139335765878127060653155421926839722816928487973061993498185276553102849503488766249530931620117916880190701377176065220213316655154163856109870716342276100730661121448575942704256251922332673670 7 (10)
q:
17976931348623159077293051907890247336179769783322109203475452170808034175442330326669097946507379393791128455814405857101953804249069373080120453445087908719610441307654783606570324082560736281503595542108229192128079145141783791421737083859052508413705164815685313426745401889544444431828866000801013065342 3 (10)
e: 65537 (10)
d:
10953964521844190551567515613502514806129442856118290998849552822575408233299925342972138362317073957564741234501989500952995485989467485055748421329104215500015427906410550225706411754020134966922979580407975510818861746836922355535977488494731642927620168700793820376605757380750336020893663084219818849672699610335946473081046959550502644804049493505708601289234966750578204943428628917104844635891729961301589374421189157569944908303056982977096213505863334263485983499980524658905066409461121457840006315533192523954628235555152993903404104398464226054865101421361280723580439872481096084108910430714233900462377 (10)
n:
32317006071311007300714876688669951960444102658747836373080181117003895263607509102474341939820567027861728922551403886015866803155295334838326473784347842406793513041434601158824214780013396296355060446619136177749875857677652850218842021404484905129532861985411209418457347684443808940366795604146676328036720138365287219925607020129779837562325992797529225526772487111763076787141361798599822744013388707010542284880865628378731414453884363727892290008016733508750045019426567258411739233615839737758547466281302488035003499472575110470956948407867824486166870615145352638827983279015737085182517811962668189298061 (10)
enc length: 512
pass checking!
----------
gen strlen = 10000000 random string
enc use 0.280853s, speed 35.61M/s
dec use 17.725146s, speed 564.17K/s
pass checking!
```

实现

底层支持

我手写了一个高精度库，位于 `big.hpp` 中，只支持16进制的加减乘除、乘方、带模运算、左移右移，通过了所有自己的测试，但是和gmp比起来效率差了100倍，因此最后还是决定使用gmp。

再也不想自己造轮子了qwq

Miller-Rabin 素性检测

具体函数位于 `check_prime` 和 `witness` 中。

check_prime(p)

Miller-Rabin的入口函数，步骤大致如下：

1. 首先判断掉p为偶数的情况
2. 随机生成一个数字 $r \in [0, p-1)$，调用函数 `witness(r, p)`，如果它返回真则说明p一定不是素数，直接返回假
3. 重复步骤2若干次，如果调用 `witness(r, p)` 的结果均为假，则大概率认为p是素数，返回真

witness(a, p)

给出一个随机数a，判断在这个a下p是否是合数。具体方法为：

假设p是一个奇素数，记 $p-1=2^k q, 2 \nmid q$，设a是不被p整除的数，则下面两个条件之一必然成立：

1. $a^q \equiv 1 \pmod{p}$
2. $a^q, a^{2q}, a^{2^2 q}, \ldots, a^{2^{k-1} q} \equiv -1 \pmod{p}$

根据如下命题

$a^2 \equiv 1 \pmod{p} \Rightarrow a \equiv \pm 1 \pmod{p}$

又因为上面第二个条件中的数，每一个都是前一个的平方，又因为最后一个数的平方是 1，所以往前，如果表中一个数它模 p 不余 1，但是它的平方模 p 余 1，那么那个数一定是 -1，所以在这种情况下表中包含 -1，又或者表中全是 1，那么第一个条件就会成立。

因此得到 Miller-Rabin 素性测试的方法：如果一个数不满足上面的性质，那么它就是合数，a就成p不是合数的证据。只要随机n个数来测试，那么测试失败的概率一定小于2^{-n}。在实现中取n=50。

RSA算法实现

RSA初始化时会自动选取素数p和q，选素数步骤为：

1. 给定位宽n，选取一个随机数p位于 $[2^{n-1}, 2^n)$ 范围内
2. 使用Miller-Rabin算法判断p是否为素数，如果是则返回，否则++p

然后就是按照RSA标准进行计算：

- $n = pq$
- $\varphi = (p-1)(q-1)$
- 初始化e=65537，判断gcd(e, φ) 是否为1，不为1则再次随机e
- $d = e^{-1} \bmod \varphi$

至此初始化完成。

给出消息m，首先将其转换为16进制数字，具体而言，如果len(m)==n，则将m转换为2n长度的新的字符串，新字符串中每两个字符对应原字符串中的一个字符的ascii值；其次计算 $g = m^e \bmod n$，得出密文。

给出密文g，计算 $m = g^d \bmod n$，然后将m每两位转换为uint8之后再转换成一个字符，恢复可读性。

然而这么做无法处理消息长度过长所导致的信息缺失。我采用了 `RSA_PKCS1_PADDING` 模式，对于密钥长度为2048，理论上最多能够表示2048/8=256个字符，而在该模式下明文必须 比 n 短至少11个字节，为256-11=243。因此处理明文时，先按照243为间隔划分字符串，然后分别送入加密算法中产生密文。解密同理。

这么做有一个好处，能够并行计算，因为RSA本来就慢，如果搞成CBC的话实在是无法接受这种速度；但是弄成这样能够很方便地并行，速度能够成倍的提升。

并行计算在c++中使用openmp，能够非常简便地实现。

性能测试

在我本机上（i7-8750H，12核）的测试结果为

字符串长度	加密时间/s	解密时间/s	加密速度/(MB/s)	解密速度/(KB/s)
100	0.000042	0.002819	2.38	35.47
1000	0.000097	0.004815	10.31	207.68
10000	0.000502	0.019238	19.93	519.80
100000	0.002957	0.166209	33.82	601.65
1000000	0.032538	1.686058	30.73	593.10
10000000	0.277753	17.225266	36.00	580.54
100000000	2.868012	197.463963	34.87	506.42
1000000000	32.415718	2044.543990	30.85	489.11

大致加密速度为**33MB/s**，解密速度为**550KB/s**。

加密比解密快了不止一个数量级：因为加密使用的e很小，65537=1<<16|1，只需16+2=18次高精度乘法就能完成加密。但是解密的d的数量级为1<<2048，需要做上千次的高精度乘法和高精度取余，因此运行效率相差悬殊。

上网搜了一圈，发现Java内置的RSA加密解密效率比我的实现慢了十倍多。