素数

素数与合数的定义，见 数论基础。

素数计数函数：小于或等于 x 的素数的个数，用 $\pi(x)$ 表示。随着 x 的增大，有这样的近似结果：$\pi(x) \sim \dfrac{x}{\ln(x)}$。

素性测试

素性测试（Primality test）可以用于判定所给自然数是否为素数。

素性测试有两种：

1. 确定性测试：绝对确定一个数是否为素数。常见例子包括试除法、Lucas–Lehmer 测试和椭圆曲线素性证明。
2. 概率性测试：通常比确定性测试快很多，但有可能（尽管概率很小）错误地将 合数 识别为质数（尽管反之则不会）。因此，通过概率素性测试的数字被称为 **可能素数**，直到它们的素数可以被确定性地证明。而通过测试但实际上是合数的数字则被称为 **伪素数**。有许多特定类型的伪素数，最常见的是费马伪素数，它们是满足费马小定理的合数。概率性测试的常见例子包括 Miller–Rabin 测试。

试除法

暴力做法自然可以枚举从小到大的每个数看是否能整除。

参考实现

C++

```cpp
bool isPrime(int a) {
  if (a < 2) return false;
  for (int i = 2; i < a; ++i)
    if (a % i == 0) return false;
  return true;
}
```

Python

```python
def isPrime(a):
    if a < 2:
        return False
    for i in range(2, a):
        if a % i == 0:
            return False
    return True
```

这样做是十分稳妥了，但是真的有必要每个数都去判断吗？

很容易发现这样一个事实：如果 x 是 a 的约数，那么 $\frac{a}{x}$ 也是 a 的约数。

这个结论告诉我们，对于每一对 $(x, \frac{a}{x})$，只检验其中的一个就足够了。为了方便起见，我们只考察每一对的较小数。不难发现，所有这些较小数都在 $[1, \sqrt{a}]$ 这个区间里。

由于 1 肯定是约数，所以不检验它。

参考实现

C++

```cpp
bool isPrime(int a) {
  if (a < 2) return 0;
  for (int i = 2; (long long)i * i <= a; ++i)  // 防溢出
    if (a % i == 0) return 0;
  return 1;
}
```

Python

```python
def isPrime(a):
    if a < 2:
        return False
    for i in range(2, int(sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True
```

Fermat 素性测试

Fermat 素性检验 是最简单的概率性素性检验。

我们可以根据 费马小定理 得出一种检验素数的思路：

基本思想是不断地选取在 $[2, n-1]$ 中的基底 a，并检验是否每次都有 $a^{n-1} \equiv 1 \pmod n$。

参考实现

C++

```cpp
bool fermat(int n) {
  if (n < 3) return n == 2;
  // test_time 为测试次数,建议设为不小于 8
  // 的整数以保证正确率,但也不宜过大,否则会影响效率
  for (int i = 1; i <= test_time; ++i) {
    int a = rand() % (n - 2) + 2;
    if (quickPow(a, n - 1, n) != 1) return false;
  }
  return true;
}
```

Python

```python
def fermat(n):
    if n < 3:
        return n == 2
    # test_time 为测试次数,建议设为不小于 8
    # 的整数以保证正确率,但也不宜过大,否则会影响效率
    for i in range(1, test_time + 1):
        a = random.randint(0, 32767) % (n - 2) + 2
        if quickPow(a, n - 1, n) != 1:
            return False
    return True
```

如果 $a^{n-1} \equiv 1 \pmod n$ 但 n 不是素数，则称 n 为以 a 为底的 **Fermat 伪素数**。我们在实践中观察到，如果 $a^{n-1} \equiv 1 \pmod n$，那么 n 通常是素数。但其实存在反例：对于 $n = 341$ 且 $a = 2$，虽然有 $2^{340} \equiv 1 \pmod{341}$，但是 $341 = 11 \cdot 31$ 是合数。事实上，对于任何固定的基底 a，这样的反例都有无穷多个[1]。

既然对于单个基底，Fermat 素性测试无法保证正确性，一个自然的想法就是多检查几组基底。但是，即使检查了所有可能的与 n 互素的基底 a，依然无法保证 n 是素数。也就是说，费马小定理的逆命题并不成立：即使对于所有 $a \perp n$，都有 $a^{n-1} \equiv 1 \pmod n$，n 也不一定是素数。这样的数称为 Carmichael 数。它也有无穷多个。这迫使我们寻找更为严格的素性测试。

Miller–Rabin 素性测试

Miller–Rabin 素性测试（Miller–Rabin primality test）是更好的素数判定方法。它是由 Miller 和 Rabin 二人根据 Fermat 素性测试优化得到的。和其它概率性素数测试一样，它也只能检测出伪素数。要确保是素数，需要用慢得多的确定性算法。然而，实际上没有已知的数字通过了 Miller–Rabin 测试等高级概率性测试但实际上却是合数，因此我们可以放心使用。

在不考虑乘法的复杂度时，对数 n 进行 k 轮测试的时间复杂度是 $O(k \log n)$。Miller–Rabin 素性测试常用于对高精度数进行测试，此时时间复杂度是 $O(k \log^3 n)$，利用 FFT 等技术可以优化到

$O(k\log^2 n\log\log n\log\log\log n)$。

为了解决 Carmichael 数带来的挑战，Miller–Rabin 素性测试进一步考虑了素数的如下性质：

二次探测定理

如果 p 是奇素数，则 $x^2\equiv 1\pmod p$ 的解为 $x\equiv 1\pmod p$ 或者 $x\equiv p-1\pmod p$。

证明

容易验证，p 为奇素数时，$x\equiv 1\pmod p$ 和 $x\equiv p-1\pmod p$ 都可以使得上式成立。由 Lagrange 定理 可知，这就是该方程的所有解。

将费马小定理和二次探测定理结合起来使用，就得到 Miller–Rabin 素性测试：

1. 将 $a^{n-1}\equiv 1\pmod n$ 中的指数 $n-1$ 分解为 $n-1=u\times 2^t$；
2. 在每轮测试中对随机出来的 a 先求出 $v=a^u \bmod n$，之后对这个值执行最多 t 次平方操作；
3. 在整个过程中，如果发现 1 的非平凡平方根（即除了 ± 1 之外的其他根），就可以判断该数不是素数；
4. 否则，再使用 Fermat 素性测试判断。

还有一些实现上的小细节：

- 对于一轮测试，如果某一时刻 $a^{u\times 2^s}\equiv n-1\pmod n$，则之后的平方操作全都会得到 1，则可以直接通过本轮测试。
- 如果找出了一个非平凡平方根 $a^{u\times 2^s}\not\equiv n-1\pmod n$，则之后的平方操作全都会得到 1。可以选择直接返回 `false`，也可以放到 t 次平方操作后再返回 `false`。

这样得到了较正确的 Miller Rabin：（来自 fjzzq2002）

参考实现

C++

```cpp
bool millerRabin(int n) {
  if (n < 3 || n % 2 == 0) return n == 2;
  if (n % 3 == 0) return n == 3;
  int u = n - 1, t = 0;
  while (u % 2 == 0) u /= 2, ++t;
  // test_time 为测试次数，建议设为不小于 8
  // 的整数以保证正确率，但也不宜过大，否则会影响效率
  for (int i = 0; i < test_time; ++i) {
    // 0, 1, n-1 可以直接通过测试, a 取值范围 [2, n-2]
    int a = rand() % (n - 3) + 2, v = quickPow(a, u, n);
    if (v == 1) continue;
    int s;
    for (s = 0; s < t; ++s) {
      if (v == n - 1) break;  // 得到平凡平方根 n-1，通过此轮测试
      v = (long long)v * v % n;
    }
    // 如果找到了非平凡平方根，则会由于无法提前 break; 而运行到 s == t
    // 如果 Fermat 素性测试无法通过，则一直运行到 s == t 前 v 都不会等于 -1
    if (s == t) return 0;
  }
  return 1;
}
```

Python

```python
def millerRabin(n):
    if n < 3 or n % 2 == 0:
        return n == 2
    if n % 3 == 0:
        return n == 3
    u, t = n - 1, 0
    while u % 2 == 0:
        u = u // 2
        t = t + 1
    # test_time 为测试次数,建议设为不小于 8
    # 的整数以保证正确率,但也不宜过大,否则会影响效率
    for i in range(test_time):
        # 0, 1, n-1 可以直接通过测试, a 取值范围 [2, n-2]
        a = random.randint(2, n - 2)
        v = pow(a, u, n)
        if v == 1:
            continue
        s = 0
        while s < t:
            if v == n - 1:
```

```
                    break
                v = v * v % n
                s = s + 1
            # 如果找到了非平凡平方根，则会由于无法提前 break; 而运行到 s == t
            # 如果 Fermat 素性测试无法通过，则一直运行到 s == t 前 v 都不会等于 -1
            if s == t:
                return False
        return True
```

可以证明[2]，奇合数 $n > 9$ 通过随机选取的一个基底 a 的 Miller–Rabin 素性测试的概率至多为四分之一。因此，随机选取 k 个基底后，仍将合数误判为素数的概率不超过 $1/4^k$。

证明

设 $n-1=u2^t$，其中，u 是奇数且 t 是正整数。那么，整数 n 可以通过基底为 a 的 Miller–Rabin 素性测试说明

$$a^u \equiv 1 \pmod{n}, \text{ or } a^{u2^i} \equiv -1 \pmod{n} \text{ for some } 0 \le i < t.$$

记这样的 a（的同余类）集合为 S，要说明的是

$$|S| \le \frac{1}{4}\varphi(n).$$

其中，$\varphi(n)$ 是 欧拉函数。证明分为三步。

第一步：设 ℓ 是使得 $2^\ell \mid p-1$ 对所有 n 的素因子 p 都成立的最大正整数。那么，可以证明

$$S \subseteq S' = \{a \bmod n : a^{u2^{\ell-1}} \equiv \pm 1 \pmod{n}\}.$$

集合 S 中的元素 a 只有两种可能。如果 $a^u \equiv 1 \pmod{n}$，那么，显然 $a^{u2^{\ell-1}} \equiv 1 \pmod{n}$ 也成立，亦即 $a \in S'$。如果对于 $0 \le i < t$ 成立 $a^{u2^i} \equiv -1 \pmod{n}$，那么，对于任意素因子 $p \mid n$，都有 $a^{u2^i} \equiv -1 \pmod{p}$。设 $\delta_p(a)$ 是 a 模 p 的 阶，那么，显然有 $\delta_p(a) \mid u2^{i+1}$ 但是 $\delta_p(a) \quad u2^i$，这说明，$\delta_p(a)$ 的素因数分解中，2 的指数恰为 $i+1$，因而 $2^{i+1} \mid \delta_p(a)$。由费马小定理可知，$\delta_p(a) \mid p-1$，所以，$2^{i+1} \mid p-1$。这一点对于 n 的所有素因子 p 都成立。因此，$i+1 \le \ell$。这说明 $a^{u2^{\ell-1}} = (a^{u2^i})^{2^{\ell-1-i}} \equiv \pm 1 \pmod{n}$，同样有 $a \in S'$。综合两种可能，就得到 $S \subseteq S'$。

第二步：计算 $|S'|$ 的大小。

假设 n 有素因数分解 $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$，那么，由 中国剩余定理 可知，条件 $a^{u2^{\ell-1}} \equiv 1 \pmod{n}$ 等价于 $a^{u2^{\ell-1}} \equiv 1 \pmod{p_i^{e_i}}$ 对所有 $p_i^{e_i}$ 都成立。由于模奇素数幂 $p_i^{e_i}$ 的 原根 总是存在的，所以，同余方程 $a^{u2^{\ell-1}} \equiv 1 \pmod{p_i^{e_i}}$ 的 解的数量 为

$$\gcd(u2^{\ell-1}, p_i^{e_i-1}(p_i-1)) = \gcd(u2^{\ell-1}, p_i-1) = 2^{\ell-1}\gcd(u, p_i-1).$$

第一个等号成立，是因为 u 是 $n-1$ 的因子，不可能是 p_i 的倍数；第二个等号成立，是因为 ℓ 的选取方式。所以，由中国剩余定理可知，同余方程 $a^{u2^{\ell-1}} \equiv 1 \pmod{n}$ 的解的数量为

$$\prod_{p\mid n} 2^{\ell-1}\gcd(u, p-1).$$

同理，条件 $a^{u2^{\ell-1}} \equiv -1 \pmod{n}$ 等价于 $a^{u2^{\ell-1}} \equiv -1 \pmod{p_i^{e_i}}$ 对所有 $p_i^{e_i}$ 都成立。对于任意因子 $p_i^{e_i}$，条件 $a^{u2^{\ell-1}} \equiv -1 \pmod{p_i^{e_i}}$ 都等价于 $a^{u2^{\ell-1}} \not\equiv 1 \pmod{p_i^{e_i}}$ 且 $a^{u2^\ell} \equiv 1 \pmod{p_i^{e_i}}$ 成立。类似上文，可以计算出同余方程 $a^{u2^\ell} \equiv 1 \pmod{p_i^{e_i}}$ 的解的数量为 $2^\ell \gcd(u, p_i-1)$，因此，同余方程 $a^{u2^{\ell-1}} \equiv -1 \pmod{p_i^{e_i}}$ 的解的数量也等于

$$2^\ell \gcd(u, p_i-1) - 2^{\ell-1}\gcd(u, p_i-1) = 2^{\ell-1}\gcd(u, p_i-1).$$

再次应用中国剩余定理，就得到同余方程 $a^{u2^{\ell-1}} \equiv -1 \pmod{n}$ 的解的数量等于

$$\prod_{p\mid n} 2^{\ell-1}\gcd(u, p-1).$$

因此，综合两种情形，有

$$|S'| = 2\prod_{p\mid n} 2^{\ell-1}\gcd(u, p-1).$$

第三步：证明 $|S'| \le \varphi(n)/4$。

结合欧拉函数的表达式 $\varphi(n) = \prod_i p_i^{e_i-1}(p_i-1)$ 可知

$$\frac{\varphi(n)}{|S'|} = \frac{1}{2}\prod_i p_i^{e_i-1}\frac{p_i-1}{2^{\ell-1}\gcd(u,p_i-1)}.$$

对于每一个 i，相应的因子 $p_i^{e_i-1}\dfrac{p_i-1}{2^{\ell-1}\gcd(u,p_i-1)}$ 都是一个偶数，所以，$\varphi(n)/|S'|$ 是一个整数。假设 $|S'| \le \varphi(n)/4$ 不成立。必然有 $\varphi(n)/|S'| = 1,2,3$，亦即

$$\prod_i p_i^{e_i-1}\frac{p_i-1}{2^{\ell-1}\gcd(u,p_i-1)} = 2,4,6.$$

由于连乘式中的每个因子都是偶数，所以，这个连乘式要么只有一个因子且这个因子就等于 $2,4,6$，要么就只有两个因子且都等于 2。

首先考虑有两个因子的情形。此时，两个因子都没有奇素因子，所以，$p_i^{e_i-1}=1$，亦即 n 没有平方因子。不妨设 $n=p_1p_2$ 且 $p_1<p_2$ 都是素数。两个因子都等于 2，所以，总有 $p_i-1=2^\ell\gcd(u,p_i-1)$。因此，$p_i=1+2^\ell m_i$，其中，$m_i$ 是奇数，而且 $m_i \mid u$。将 $p_1p_2=n=1+u2^t$ 对 m_1 取模就得到 $p_1p_2\equiv 1 \pmod{m_1}$，故而 $p_2 \equiv 1 \pmod{m_1}$，这说明，$m_1 \mid m_2$。反过来也成立。这就说明 $m_1=m_2$，也就是 $p_1=p_2$。这与 $p_1<p_2$ 矛盾。这一情形不成立。

最后，考虑只有一个因子的情形，亦即合数 $n=p^e$ 且 $e>1$。此时，必然有 $p^{e-1}\mid 2,4,6$。因此，唯一的情形是 $p=3,e=2$，亦即 $n=9$，与命题所设相矛盾。这一情形也不成立。

综合所有情形可知，$|S'| \le \varphi(n)/4$ 成立。

结合上述三个步骤可知，$|S| \le |S'| \le \varphi(n)/4$ 对于所有奇合数 $n>9$ 都成立。

另外，假设 广义 Riemann 猜想（generalized Riemann hypothesis, GRH）成立，则对数 n 最多只需要测试 $[2,\min\{n-2,\lfloor 2\ln^2 n\rfloor\}]$ 中的全部整数即可 **确定** 数 n 的素性。[3]

而在 OI 范围内，通常都是对 $[1,2^{64})$ 范围内的数进行素性检验。对于 $[1,2^{32})$ 范围内的数，选取 $\{2,7,61\}$ 三个数作为基底进行 Miller–Rabin 素性检验就可以确定素性；对于 $[1,2^{64})$ 范围内的数，选取 $\{2,325,9375,28178,450775,9780504,1795265022\}$ 七个数作为基底进行 Miller–Rabin 素性检验就可以确定素性。[4]

也可以选取 $\{2,3,5,7,11,13,17,19,23,29,31,37\}$（即前 12 个素数）检验 $[1,2^{64})$ 范围内的素数。

注意如果要使用上面的数列中的数 a 作为基底判断 n 的素性：

- 所有的数都要取一遍，不能只选小于 n 的；
- 把 a 换成 $a \bmod n$；
- 如果 $a\equiv 0 \pmod{n}$ 或 $a\equiv \pm 1 \pmod{n}$，则直接通过该轮测试。

反素数

顾名思义，素数就是因子只有两个的数，那么反素数，就是因子最多的数（并且因子个数相同的时候值最小），所以反素数是相对于一个集合来说的。

一种符合直觉的反素数定义是：在一个正整数集合中，因子最多并且值最小的数，就是反素数。

反素数

对于某个正整数 n，如果任何小于 n 的正数的约数个数都小于 n 的约数个数，则称为是 **反素数**（anti-prime, a.k.a., highly compositive numbers）。

注意

注意区分 emirp，它表示的是逐位反转后是不同素数的素数（如 149 和 941 均为 emirp，101 不是 emirp）。

过程

那么，如何来求解反素数呢？

首先，既然要求因子数，首先要做的就是素因子分解。把 n 分解成 $n = p_1^{k_1} p_2^{k_2} \cdots p_n^{k_n}$ 的形式，其中 p 是素数，k 为他的指数。这样的话总因子个数就是 $(k_1+1) \times (k_2+1) \times (k_3+1) \cdots \times (k_n+1)$。

但是显然质因子分解的复杂度是很高的，并且前一个数的结果不能被后面利用。所以要换个方法。

我们来观察一下反素数的特点。

1. 反素数肯定是从 2 开始的连续素数的幂次形式的乘积。
2. 数值小的素数的幂次大于等于数值大的素数，即 $n = p_1^{k_1} p_2^{k_2} \cdots p_n^{k_n}$ 中，有 $k_1 \geq k_2 \geq k_3 \geq \cdots \geq k_n$。

解释：

1. 如果不是从 2 开始的连续素数，那么如果幂次不变，把素数变成数值更小的素数，那么此时因子个数不变，但是 n 的数值变小了。交换到从 2 开始的连续素数的时候 n 值最小。
2. 如果数值小的素数的幂次小于数值大的素数的幂，那么如果把这两个素数交换位置（幂次不变），那么所得的 n 因子数量不变，但是 n 的值变小。

另外还有两个问题，

1. 对于给定的 n，要枚举到哪一个素数呢？

最极端的情况大不了就是 $n = p_1 p_2 \cdots p_n$，所以只要连续素数连乘到刚好小于等于 n 就可以的呢。再大了，连全都一次幂，都用不了，当然就是用不到的啦！

2. 我们要枚举到多少次幂呢？

   我们考虑一个极端情况，当我们最小的素数的某个幂次已经比所给的 n（的最大值）大的话，那么展开成其他的形式，最大幂次一定小于这个幂次。`unsigned long long` 的最大是 $2^{64}-1$，所以可以枚举到 $2^{64}-1$。

细节有了，那么我们具体如何具体实现呢？

我们可以把当前走到每一个素数前面的时候列举成一棵树的根节点，然后一层层的去找。找到什么时候停止呢？

1. 当前走到的数字已经大于我们想要的数字了；
2. 当前枚举的因子已经用不到了；
3. 当前因子大于我们想要的因子了；
4. 当前因子正好是我们想要的因子（此时判断是否需要更新最小 ans）。

然后 dfs 里面不断一层一层枚举次数继续往下迭代可以。

例题

Codeforces 27E. A number with a given number of divisors

求具有给定除数个数的最小自然数。答案保证不超过 10^{18}。

解题思路

对于这种题，我们只要以因子数为 dfs 的返回条件基准，不断更新找到的最小值就可以了。

参考代码

```cpp
#include <iostream>
unsigned long long p[16] = {
    2,  3,  5,  7,  11, 13, 17, 19,
    23, 29, 31, 37, 41, 43, 47, 53};  // 根据数据范围可以确定使用的素数最大为53

unsigned long long ans;
unsigned long long n;

// depth: 当前在枚举第几个素数
// temp: 当前因子数量为 num的时候的数值
// num: 当前因子数
// up：上一个素数的幂，这次应该小于等于这个幂次嘛
void dfs(unsigned long long depth, unsigned long long temp,
         unsigned long long num, unsigned long long up) {
  if (num > n || depth >= 16) return;  // 边界条件
  if (num == n && ans > temp) {        // 取最小的ans
    ans = temp;
    return;
  }
  for (int i = 1; i <= up; i++) {
    if (temp * p[depth] > ans)
      break;  // 剪枝：如果加一个这个乘数的结果比ans要大，则必不是最佳方案
    dfs(depth + 1, temp = temp * p[depth], num * (i + 1),
        i);  // 取一个该乘数，进行对下一个乘数的搜索
  }
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  ans = ~(unsigned long long)0;
  dfs(0, 1, 1, 64);
  cout << ans << '\n';
  return 0;
}
```

ZOJ 2562 More Divisors

求不超过 n 的数中，除数最多的数。

解题思路

思路同上，只不过要改改 dfs 的返回条件。注意这样的题目的数据范围，32 位整数可能溢出。

参考代码

```cpp
#include <iostream>

int p[16] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};
unsigned long long n;
unsigned long long ans,
    ans_num;  // ans 为 n 以内的最大反素数（会持续更新），ans_sum 为
              // ans的因子数。

// depth: 当前在枚举第几个素数
// temp: 当前因子数量为 num的时候的数值
// num: 当前因子数
// up：上一个素数的幂，这次应该小于等于这个幂次嘛
void dfs(int depth, unsigned long long temp, unsigned long long num, int up) {
  if (depth >= 16 || temp > n) return;
  if (num > ans_num) {  // 更新答案
    ans = temp;
    ans_num = num;
  }
  if (num == ans_num && ans > temp) ans = temp;  // 更新答案
  for (int i = 1; i <= up; i++) {
    if (temp * p[depth] > n)
      break;  // 剪枝：如果加一个这个乘数的结果比ans要大，则必不是最佳方案
    dfs(depth + 1, temp *= p[depth], num * (i + 1),
        i);  // 取一个该乘数，进行对下一个乘数的搜索
  }
  return;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> n) {
    ans_num = 0;
    dfs(0, 1, 1, 60);
    cout << ans << '\n';
  }
  return 0;
}
```

参考资料与注释

1. Rui-Juan Jing, Marc Moreno-Maza, Delaram Talaashrafi, "Complexity Estimates for Fourier-Motzkin Elimination", Journal of Functional Programming 16:2 (2006) pp 197-217.
2. 数论部分第一节：素数与素性测试
3. Miller–Rabin 与 Pollard–Rho 学习笔记 - Bill Yang's Blog
4. Primality test - Wikipedia
5. Fermat pseudoprime - Wikipedia
6. 桃子的算法笔记——反素数详解（acm/OI）
7. The Rabin-Miller Primality Test
8. Highly composite number - Wikipedia

---

1. Pomerance, Carl, John L. Selfridge, and Samuel S. Wagstaff. "The pseudoprimes to $25\cdot 10^9$." Mathematics of Computation 35, no. 151 (1980): 1003-1026. 的定理 1 说明了，对于固定的基底 a，能够通过更强的 Miller–Rabin 素性测试的合数也是无穷多的。 ↩
2. 本结论及其证明参考了 Crandall, Richard, and Carl Pomerance. Prime numbers: a computational perspective. New York, NY: Springer New York, 2005. 的第 3.5 节。 ↩
3. Bach, Eric , "Explicit bounds for primality testing and related problems", Mathematics of Computation, 55:191 (1990) pp 355–380. ↩
4. 更多类似的结果请参考 Deterministic variant of the Miller–Rabin primality test。 ↩

本页面最近更新：2025/8/30 15:23:07，更新历史

发现错误？想一起完善？ 在 GitHub 上编辑此页！

本页面贡献者：Ir1d, Tiphereth-A, c-forrest, Xeonacid, Enter-tainer, StudyingFather, iamtwz, ksyx, Marcythm, MegaOwIer, 383494, Alpacabla, HeRaNO, abc1763613206, alphagocc, Backl1ght, CCXXXI, drkelo, Early0v0, Great-designer, greyqz, GuanghaoYe, H-J-Granger, HHH2309, isdanni, kenlig, lazyasn, Menci, ouuan, r-value, shawlleyw, shopee-jin, shuzhouliu, Siger Young, TrisolarisHD, untitledunrevised, void-mian, Voileexperiments, weilycoder, xtlsoft, yusancky, YuzhenQin1