GE6001: Scientific Writing, Norms and Ethics Dec 31, 2022

Notes 1 – AKS 算法以及素性测试在工程中的运用

Instructor: *guoqiang li* Scribes: *jingyao zeng yuhan li*

1 摘要

AKS 算法是由三位印度科学家 Agrawal,Kayal 以及 Saxena 共同提出的一个确定性的素性测试算法。这个算法的提出对于密码学领域产生了深远的影响，并且这个算法由于给出了一个多项式时间的确定一个数字是否是素数的确定性算法，也被认为是在 P 问题和 NP 问题领域的研究的巨大进步。

这篇文章从 AKS 具体算法以及其正确性，AKS 算法的时间复杂度分析以及对于 AKS 算法的一些优化等三个方面来介绍了 AKS 算法。并且在文章的最后一个部分，我们又介绍了在实际工程实践中被广泛使用的一些素性测试算法，并将这些算法与 AKS 算法进行了初步的比较分析。

Keywords: AKS 算法, 素性测试

2 前置条件

AKS 算法基于如下的素数恒等式，这也是对费马小定理的推广。

Lemma 2.1. *令 $a \in Z$, $n \in N$, $n \geq 2$, 且 $(a, n) = 1$. 那么 n 为素数当且仅当*

$$(X + a)^n = X^n + a \ (mod\ n). \tag{1}$$

证明. 对于 $0 < i < n$, $((X + a)^n - (X^n + a^n))$ 中 x^i 的系数为 $\binom{n}{i} a^{n-i}$.

假定 n 为素数。根据费马小定理我们有 $a^{n-1} = 1 \ (mod\ n)$, 且 $\binom{n}{i} = 0 \ (mod\ n)$, 因此 $(X + a)^n = X^n + a \ (mod\ n)$.

假定 n 为合数。那么一定存在 n 的一个素因子 q 且 $q^k \parallel n$. 于是 q^k 无法整除 $\binom{n}{q}$ 且与 a^{n-q} 互素，因此 X^q 的系数在模 n 意义下不为 0. 因此 $((X + a)^n - (X^n + a))$ 在 Z_n 上不恒为 0.(Z_n 表示模 n 意义下的整数环) □

上述恒等式给出了一个简单的素性测试：给定一个输入 n，选择一个 a 并且测试同余式 (1) 是否满足。然而，这个测试在最坏情况下需要 $\Omega(n)$ 时间，因为我们需要验证等式左边的 n 个系数。一种简单的削减系数数量的方法，是在 (1) 式两边同时约去一个形如 $X^r - 1$ 的多项式，其中 r 为一个经过恰当挑选的较小的数，这也正是 AKS 算法所使用的。换言之，我们需要测试如下等式是否成立：

$$(X + a)^n = X^n + a \ (mod\ X^r - 1, n). \tag{2}$$

我们将同样需要如下关于前 m 个正整数的最小公倍数的结论。(该结论的证明有一些复杂，超过了笔者的知识水平，因此我们在此处略去相关证明，您可以参考 [5] 给出的证明。)

Lemma 2.2. *令 $LCM(m)$ 表示前 m 个正整数的最小公倍数。对于 $m \geq 7$，我们有：*

$$LCM(m) \geq 2^m.$$

在“时间复杂度分析”这一部分中, 我们使用符号 $O\tilde{}(t(n))$ 表示 $O(t(n) \cdot poly(\log t(n)))$, 其中 $t(n)$ 是关于 n 的任意函数。例如，$O\tilde{}(\log^k n) = O(\log^k n \cdot poly(\log\log n)) = O(\log^(k + \epsilon)n)$ 对于任意 $\epsilon > 0$. 我们使用 log 表示以 2 为底的对数, ln 表示自然对数。

3 AKS 算法及其正确性

Algorithm 1 AKS(n)

Step 1:
if $n = a^b$ for $a \in N$ and $b > 1$ **then**
  output $COMPOSITE$.
end if
Step 2:Find the smallest r such that $ord_r(n) > \log^2 n$.
Step 3:
if $1 < (a, n) < n$ for some $a \leq r$ **then**
  output $COMPOSITE$.
end if
Step 4:
if $n \leq r$ **then**
  output $PRIME$.
end if
Step 5:
for $a = 1$ to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ **do**
  if $((X + a)^n \neq X^n + a \ (mod\ X^r - 1, n))$ **then**
    output $COMPOSITE$.
  end if
end for
Step 6:output $PRIME$.

Definition 3.1. 给定 $r \in N$, $a \in Z$ 且 $(a, r) = 1$, 模 r 意义下的阶指最小的整数 k 使得 $a^k = 1\ (mod\ r)$. 我们将其表示为 $ord_r(a)$.

Definition 3.2. 对于 $r \in N$, $\phi(r)$ 为欧拉函数，即所有小于等于 r 且与 r 互素的正整数的数量。

Theorem 3.3. 上述算法返回 $PRIME$ 当且仅当 n 为素数。

我们将通过一系列引理证明上述理论。以下这条引理是平凡:

Lemma 3.4. 若 n 为素数，则该算法返回 $PRIME$.

证明. 若 n 为素数，则步骤 1 和步骤 3 永远不会返回 $COMPOSITE$. 由引理 2.1, for 循环同样不会返回 $COMPOSITE$. 因此该算法将在步骤 4 或步骤 6 判定 n 为 $PRIME$. □

上述引理的逆命题则需要更多的工作。若算法在步骤 4 返回 $PRIME$ 则 n 必为素数，否则步骤 3 将会找到 n 的一个非平凡因子。故唯一遗留的情况为该算法在步骤 6 返回 $PRIME$. 此处我们略去步骤 5 的正确性证明，因为其的复杂性和笔者的能力水平，您可以参阅 [1] 以获得一个详细的证明。总之，我们说若对于 $a = 1$ 到 $\lfloor \sqrt{\phi(r)} \log n \rfloor$, $((X + a)^n = X^n + a\ (mod\ X^r - 1, n))$，则 n 一定为 p^k，$k > 0$. 若

$k > 1$, 那么该算法将会在步骤 1 返回 $COMPOSITE$, 否则该算法将会返回 $PRIME$ 且 n 当然为素数。为了表明步骤 2 中这样的 r 确实存在，且便于后续复杂性分析，我们同样需要给出这样一个恰当的 r 的界限。

Lemma 3.5. *存在一个 $r \leq max\{3, \lceil \log^5 n \rceil\}$ 使得 $ord_r(n) > \log^2 n$.*

证明. 当 $n = 2$ 时显然成立, 取 $r = 3$ 即可满足所有条件。故假定 $n > 2$. 那么 $\lceil \log^5 n \rceil > 10$, 引理 2.2 可被使用。令 r_1, r_2, ..., r_t 为所有要么 $ord_{r_i}(n) \leq \log^2 n$，要么 r_i 整除 n 的数。每一个这样的数必能整除

$$\begin{aligned} n \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor} (n^i - 1) &< n \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor} n^i \\ &= n^{1+\frac{(1+\lfloor \log^2 n \rfloor)\lfloor \log^2 n \rfloor}{2}} \\ &< n^{\log^4 n} \\ &= (2^{\log n})^{\log^4 n} \\ &= 2^{\log^5 n} \end{aligned}$$

由引理 2.2, 前 $\lceil \log^5 n \rceil$ 个数的最小公倍数至少为 $2^{\lceil \log^5 n \rceil}$，因此必然存在一个数 $s \leq \lceil log^5 n \rceil$，$s \notin \{r_1, r_2, \cdots, r_t\}$ 且 s 无法整除 $n \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor}(n^i - 1)$.(否则 $LCM(\lceil \log^5 n \rceil)$ 将小于等于 $n \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor}(n^i - 1)$, 这与引理 2.2 矛盾！) 若 $(s, n) = 1$ 那么 $ord_s(n) > \log^2 n$，s 即为所求的 r. 若 $(s, n) > 1$, 那么由于 s 不能整除 n 且 $(s, n) \in \{r_1, r_2, \cdots, r_t\}$, 可以令 $r = \frac{s}{(s,n)} \neq 1$ 且 $(r, n) = 1$. 若 $r \in \{r_1, r_2, \cdots, r_t\}$, 那么 $ord_r(n) \leq \log^2 n$, 这意味着 r 可以整除 $\prod_{i=1}^{\lfloor \log^2 n \rfloor}(n^i - 1)$. 然而, 因为 (s, n) 可以整除 n, 我们有 $s = r \cdot (s, n)$ 可以整除 $n \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor}(n^i - 1)$！这与我们之前证明的结果矛盾，因此 $r \notin \{r_1, r_2, \cdots, r_t\}$, 故 $ord_r(n) > \log^2 n$. □

4 时间复杂度分析

在计算该算法的时间复杂度时，我们使用如下关于加法、乘法和除法操作的事实：以上操作在两个 m bits 的数中可以在 $O\tilde{}(m)$ 时间内完成。类似的，在两个次数为 d 且每项系数最多为 m bits 的两个多项式之间进行如上操作，可以在 $O\tilde{}(d \cdot m)$ 时间内完成。

Lemma 4.1. *给定 $n \in N$, $n > 1$, 确定 n 是否是幂 (i.e. 是否存在 $m \in N$, $k \in N$, $k > 1$ 使得 $n = m^k$) 可以在 $O\tilde{}(\log^3 m)$ 渐进时间内完成。*

证明. 为了测试一个自然数 n 是否是幂，我们可以在整数集 $\{1, 2, \cdots, n\}$ 进行二分查找，寻找 m 使得 $n = m^k$ 对于 $k > 1$. 首先我们令 $k = 2$. 如果存在 m 和 k 的一个解，那么 m 一定位于某个区间 $[a_i, b_i]$. $i = 0$ 时，我们令 $[a_0, b_0] = [1, n]$. 为了定义 $[c_{i+1}, d_{i+1}]$, 考虑 $\alpha = \lfloor \frac{c_i + d_i}{2} \rfloor$. 若 $\alpha^k = n$ 那么我们便完成了。若 $\alpha^k > n$, 令 $[a_{i+1}, b_{i+1}] = [a_i, \alpha]$; 否则 $\alpha^k < n$，我们令 $[a_{i+1}, b_{i+1}] = [\alpha, b_i]$. 我们不断进行这样的操作直到 $|a_i - b_i| \leq 1$. 接着我们将 k 增加 1，然后再次进行循环。对所有的 $k \leq \log(n)$ 进行这样的循环，我们便完成了这个算法。

每次我们检验是否 $\alpha^k = n$ 需要花费 $O\tilde{}(k \log n)$ 渐进时间，且每次循环最多需要 $O(\log n)$ 次检查，我们最多需要进行这样的循环 $O(\log n)$ 次，因此总共的渐进时间复杂度为 $O\tilde{}(\log^3 m)$. □

Theorem 4.2. *该算法的渐进时间复杂度为* $O\tilde{}(\log^{\frac{21}{2}} n)$.

证明. 由引理 4.1，我们知道算法第一步的渐进时间复杂度为 $O\tilde{}(\log^3 m)$.

在在步骤 2，我们找了一个 r 使得 $ord_r(n) > \log^2 n$. 这可以通过尝试连续的 r 并且验证是否 $n^k \neq 1 \ (mod\ r)$ 对于每个 $k \leq \log^2 n$. 对于一个确定的 r, 该验证最多需要 $O(\log^2 n)$ 次乘法模 r 运算，因此需要 $O\tilde{}(\log^2 n \log r)$. 由引理 3.5 我们知道只有 $O(\log^5 n)$ 个不同的 r 需要尝试。因此步骤 2 总的时间复杂度为 $O\tilde{}(\log^7 n)$.

在步骤 3 我们需要计算 r 个数的最小公倍数。每一次最小公倍数计算需要 $O(log^2 n)$(将 Euclid 算法中的模运算更换为减法运算，也即更相减损术)，因此，步骤 3 的时间复杂度为 $O(r \log n) = O(\log^7 n)$.

步骤 4 中，时间复杂度仅为 $O(\log n)$.

步骤 5 中，我们需要验证 $\lfloor \sqrt{\phi(r)} \log n \rfloor$ 个等式。每个等式需要 $O(\log n)$ 次次数为 r 且系数大小为 $O(\log n)$ 的多项式乘法。所以每个等式需要在 $O\tilde{}(r \log^2 n)$ 时间步内被验证。因此步骤 5 的时间复杂度为 $O\tilde{}(r\sqrt{\phi(r)} \log^3 n) = O\tilde{}(r^{\frac{3}{2}} \log^3 n) = O\tilde{}(\log^{\frac{21}{2}} n)$. 该时间支配其他所有步骤的时间，因此也为该算法的时间复杂度。 □

更进一步,如果基于一些还未被完全证明但在实际应用中均为正确的猜想 (如 *Artin's Conjecture or Sophie−Germain Prime Density Conjecture*)，AKS 算法的时间复杂度可以被降低到 $O\tilde{}(\log^6 n)$. 不管怎么说，我们不加证明地给出如下引理，这将帮助我们把算法的时间复杂度降低到 $O\tilde{}(\log^{\frac{15}{2}} n)$:

Lemma 4.3. *令 P(m) 表示 m 的最大素因子。存在常数 $c > 0$ 和 n_0 使得对于所有 $x \geq n_0$:*

$$|\{q \ is \ prime, \ q \leq x \ and \ P(q-1) > q^{\frac{2}{3}}\}| \geq c\frac{x}{\ln x}$$

.

Theorem 4.4. *AKS 算法的渐进时间复杂度为* $O\tilde{}(\log^{\frac{15}{2}} n)$.

证明. 如前所述,具有 $P(q-1) > q^{\frac{2}{3}}$ 性质的素数 q 的密度很高,这表明步骤 2 将会找到一个 $r = O(\log^3 n)$ 且 $ord_r(n) > \log^2 n$. 这使得算法的时间复杂度降低到 $O\tilde{}(\log^{\frac{15}{2}} n)$. □

5 对于 AKS 算法的一些思考与改进

正如前文所述，AKS 算法的渐进时间复杂度为 $O\tilde{}(\log^{\frac{15}{2}} n)$，而 Lenstra 在 2005 年提出一个 AKS 算法的变种算法 [4]，其运行渐进时间复杂度为 $O\tilde{}(\log^7 n)$。然而，前述的两种 AKS 算法只具有理论意义，其极差的运行性能以及对于存储系统的极高要求使得 AKS 算法在实际中并不实用 [2]。

从 AKS 测试算法被提出以来，有很多工作都致力于从不同的角度来重新实现该算法来使得 AKS 算法具有更好的运行性能。但是这些工作最终只对 AKS 算法的运行性能产生了很微小的影响。即使这些工作的提出者没有对于非常大的输入数字运行他们自行实现的 AKS 算法，但是他们给出了对于当 n 越来越大的情况下算法的运行时间将以年为计时单位的估计结果。换句话说，无论通过修改算法使得 AKS 的算法性能得到多大的提升，都没有解决 AKS 算的固有问题。对于多项式系数的存储问题仍然是一个为解决的问题，并且值得一提的是，AKS 算法得到的多项式系数是属于 Z_n 的。Zhengjun Cao[2] 指出了对于一个输入长度为 1024 bits 的数据，其需要花费的存储空间为 $1,000,000,000$ GB。除非算法步骤 5 中的代价昂贵的对于多项式同余的验证能够被一个更好地算法替代，否则 AKS 算法可能永远都不会被应用于实际工程领域。

通过之前的 section，我们可以轻易地发现 r 的数值同时影响了 AKS 算法的时间和复杂度。r 越大，算法步骤 5 中存储全部 $2r$ 个多项式系数所需要的存储空间以及算法步骤 3,4,5 中需要花费的运算时间就

会越大。因此是否能够得到 r 的上界和下界是一个很重要的问题。同时，确定 r 的范围对于减小作为算法的另外一个瓶颈的算法步骤 2 所需要的搜索空间也具有重要的意义。Lalitha Kiran Nemana 和 V. Ch. Venkaiah[6] 通过对于大量数据的实验分析，提出了一些对于算法性能提升很有帮助的关于 r 的素性以及实际使用中用到的对于同余方程的一些 observations：

Observation 5.1. *对于输入到算法步骤 5 中的数字 n，我们可以将 $(\lceil \log n \rceil - 1)^2$ 视作其下界，将 $1.4(\lceil \log n \rceil)^2$ 视作其上界。*

回顾 AKS 算法，其正确性根本不会取决于最小的 r。因此，虽然在算法步骤 3 中大部分的合数都会满足不等式 $r \leq (\lceil \log n \rceil - 1)^2$(事实上，这也是前述 observations 中的一个)，我们仍然可以选择使得 $(\lceil \log n \rceil - 1)^2$ 成立的 r 来继续运行算法。

Observation 5.2. *对于输入到算法步骤 5 中的数字，其对应的数字 r 大多数情况下都是一个素数。*

如前所述，由于 r 并不会影响算法的正确性，所以没必要选择最小的那一个 r。所以选择一个满足 $ord_r(n) = r - 1$ 的素数 r 仍然能够使得算法成立。另一方面，由于 r 的数值总是远远小于 n 的数值，所以验证 r 是否是一个素数是很容易的。

Observation 5.3. *在算法步骤 1,3 中发现一个输入数字是一个合数的频率要大于在算法步骤 5 中发现输入数字是合数的频率。*

这表明 AKS 算法相比于作为一个素数测试器，更适合作为一个合数测试器。

Observation 5.4. *对于所有输入算法步骤 5 的合数，我们设第一个使得循环终止的 a 的数值为 a', 那么 a' 的大小是很小的，并且满足 $\forall a \in [a', \lfloor \sqrt{\phi(r)} \log n \rfloor]$, $(X + a)^n \neq x^n + a \ (mod\ X^r - 1, n)$。*

这个 observation 给出了一种在算法步骤 5 中只使用一个同余方程的可能。同时通过一些富有智慧的实现细节 (具体的细节可以参阅 [6])，我们可以显著的提升 AKS 算法的运行性能。

基于上述根据实验数据得到的 observations，这些 observations 的作者们通过修改 AKS 算法给出了满足在一个较大的 *bit* 范围内不会错判合数为素数 (这个条件很大程度上已经能够满足实际工程的要求了) 的算法变体如下：

Algorithm 2 AKS-Improvement(n)

1: **if** $n = a^b$ for $a \in N$ and $b > 1$ **then**
2: output $COMPOSITE$.
3: **end if**
4: Choose a prime $r \in ((\lceil \log n \rceil - 1)^2, 2(\lceil \log n \rceil)^2)$ and certainly $ord_r(n) \geq \lfloor \log^2 n \rfloor$.
5: **if** $1 < (a, n) < n$ for some $a < r$ **then**
6: output $COMPOSITE$.
7: **end if**
8: **if** $n \leq r$ **then**
9: output $PRIME$.
10: **end if**
11: **if** $(X + \lfloor \sqrt{\phi(r)} \log n \rfloor)^n \neq x^n + \lfloor \sqrt{\phi(r)} \log n \rfloor \ (mod\ X^r - 1, n)$ **then**
12: output $COMPOSITE$.
13: **end if**
14: output $PRIME$

由于 $r = \theta(\log^2 n)$，所以算法步骤 2,3,5 分别需要花费时间 $O\tilde{}(\log^4 n)$, $O(\log^3 n)$ 以及 $O\tilde{}(\log^4 n)$。因此总体的运行时间是 $O\tilde{}(\log^4 n)$。然而，这个算法的正确性还需要进一步的理论层面的研究。

6 实际应用中的素性测试

素性测试是密码学中很重要的一个领域。这部分将会介绍两个在实际工程中被大量使用的两个素性测试随机算法。例如，费马素性测试算法就被用于加密程序 PGP 中。我们将会依次介绍米勒-罗宾算法和费马算法两种素性测试算法。在这部分的最后，还将会把这两个算法与 AKS 算法进行比较，并指出他们的差别并分析 AKS 没有被广泛使用的原因。

6.1 在素性测试算法中使用了的一些定理

6.1.1 费马小定理

若 p 是一个素数, 那么对于任意的 $1 \leq a \leq p-1$, 都有 $a^{p-1} \equiv 1(mod\ p)$

6.1.2 二次探测定理

若 p 是一个素数, 并且有 $1 \leq x \leq p-1$, 那么方程 $x^2 \equiv 1(mod\ p)$ 的解有 $x = 1, p-1$ 两个.
简单证明:
由于 $x^2 \equiv 1(mod\ p)$, 我们可以由此得到:

$$(x+1)(x-1) \equiv 0(mod\ p)$$

同时由于 $1 \leq x \leq p-1$, x 只能取到 1, p-1 两个数值。

6.2 米勒-罗宾素性测试

6.2.1 算法内容

Algorithm 3 Miller-Robin(n)

As n is a odd number, let $n-1 = m2^q$. Then pick a random integer $1 \leq a \leq n-1$.
if $a^{n-1} \equiv 1(mod\ n)$ is FALSE **then**
    output $COMPOSITE$.
end if
Then for the following *Miller Sequence*:
$a^m(mod\ n), a^{2m}(mod\ n), \ldots, a^{m2^{q-1}}(mod\ n)$,
if $a^m \equiv 1 mod\ n$ **then**
    output $PRIME$.
end if
if there exists some $0 \leq i \leq q-1$, such that $a^{m2^i} \equiv (n-1)(mod\ n)$ **then**
    output $PRIME$.
end if

6.2.2 简单证明

假设 n 是一个素数. 我们取 $n-1 = m2^q$. 随机选取一个 a, 使得 $1 \leq a \leq n$。之后根据费马小定理 (1.1.1), 可以得到 $a^{n-1} \equiv 1(mod\ n)$.
由于 n 是一个素数, 所以根据二次探测定理 (1.1.2) 可以得到如下等式:
$a^{m2^{q-1}} \equiv 1(mod\ n)$ 或者 $a^{m2^{q-1}} \equiv n-1(mod\ n)$。

之后我们使用二次探测定理对上面的等式计算 t 次, 我们最终可以得到某个 $0 \leq r \leq q-1$ 使得:
$a^{m2^r} \equiv 1(mod\ n)$ 或者 $a^{m2^r} \equiv n-1(mod\ n)$，其中 r 应该是一个较小的数字。

6.2.3 时间复杂度分析

对于输入数据 n, 我们需要分别考虑 Miller Sequence 中的 q 个数字。并且对于其中的每一个数字, 我们都需要进行一个取模的操作。对于一个较大的数字 n, 一次取模操作耗时为 $O(log^3 n)$。同时由于 $n-1 = m2^q$, 我们可以得到 $q = log(\frac{n}{m})$,。所以总体的时间复杂度为 $O(qlog^3 n) = O(log^4 n)$, 其中 n 为输入的要进行素性测试的目标值。

6.3 费马素性测试

6.3.1 算法部分

Algorithm 4 Fermat(n, k)

1: **for** each $i \in [1, k]$ **do**
2: Pick a random int $1 \leq a \leq n-1$;
3: **if** $a^{n-1} \equiv 1(mod\ n)$ is FALSE **then**
4: output $COMPOSITE$.
5: **end if**
6: **end for**
7: output $PRIME$.

6.3.2 简单证明

我们可以很容易的根据费马小定理来证明费马素性测试的正确性。但是需要注意的是，对于一个输入数字 n, 若它不能够满足费马小定理，那么他一定是一个合数；但是如果它能够满足费马小定理，它仍有可能不是一个素数而是一个合数。

6.3.3 时间复杂度分析

在一次的费马素性测试中共有 k 次迭代。在每一次迭代中都需要进行一次相关的取模操作，所以总体的时间复杂度为 $O(klog^3 n)$, 其中 n 是输入的要进行素性测试的目标值。

6.4 小结

6.4.1 随机素性测试算法与 AKS 算法的对比

上述介绍的这两个素性测试算法是两个随机算法，也就是说最终对于一个输入得出的是否是素数的结果是不确定的。但是在实际的工程领域中，他们却获得了比 AKS 算法更加广泛地使用，导致这种现象出现的可能原因如下：

1. 上述介绍的不确定性算法有着比 AKS 算法更好的时间性能。AKS 算法在没有优化的情况下时间复杂度为 $O(log^{7.5} n)$。而根据前述分析可以知道，米勒-罗宾算法的时间复杂度为 $O(log^4 n)$，费马测试算法的时间复杂度为 $O(klog^3 n)$。

2. 虽然上述介绍的两个随机算法有概率得到错误的关于输入的素性的结论，但是这个错误概率已经被证明是很小的了。例如，米勒-罗宾测试算法单次运行错误概率的上界已经被证明是 $\frac{1}{4}$[3]。所以如果我

们对于相同的输入连续运行 k 米勒-罗宾算法后结果仍然错误的概率为 $(\frac{1}{4})^k$。所以我们可以简单的通过增大 k 的取值的方式来快速的降低最终结果出错的概率，即在实际使用中可以通过添加运行次数的限制来使得米勒-罗宾测试算法的结果更加" 确定"。同时由于 k 对于算法的 O 的时间复杂度分析不会产生很大的影响，所以我们仍然能可以认为这些随机素性测试算法比 AKS 测试算法更加高效。

3.AKS 算法对于存储系统的过高要求。在前面的 section 中已经提到，对于未经过优化的原始的 AKS 算法，其在处理一个长度为 1024 bits 的输入的时候需要花费 $1,000,000,000$ GB 的存储空间。而考虑到密码学的发展，我们总是需要一个足够长的素数，而这个存储空间大小在工程实践中是不可以被接受的。所以这也可能是一个使得 AKS 算法在实际工程领域没有被广泛应用的一个重要原因。

6.4.2 实际应用中的具体应用举例

1. 在 RSA 算法中的应用。RSA 算法是一个著名的密码学算法。他需要首先找到两个足够大的素数相乘之后得到一个新的密钥数值。所以 RSA 需要通过素性测试来判断选取的两个数字是否真的是两个素数。同时为了更好地算法运行性能，一般会选择随机算法比如米勒-罗宾算法等而不会选择 AKS 素性测试算法。

7 总结

在这篇论文中，我们着重分析了 AKS 素性测试算法。我们描述了 AKS 算法并给出了关于这个算法的正确性证明，并且分析了 AKS 测试算法运行时的时间复杂度。但是在文章的最后一个部分，我们却发现，虽然 AKS 算法是第一个有着重大意义的确定性的素性测试算法，但是他却由于很差的时间复杂度没有获得在工程领域中的广泛应用。所以通过这个我们可以初步体会到理论算法与实际应用之间的差别，同时我们也应该认识到想要将一些数学上面的理论真正付诸于实际并得到广泛应用的话，还有很长的一段路要走。

参考文献

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

[2] Zhengjun Cao. A note on the storage requirement for aks primality testing algorithm. *Cryptology ePrint Archive*, 2013.

[3] ST Ishmukhametov, R Rubtsova, and N Savelyev. The error probability of the miller–rabin primality test. *Lobachevskii Journal of Mathematics*, 39(7):1010–1015, 2018.

[4] Hendrik W Lenstra. Primality testing with gaussian periods. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 1–1. Springer, 2002.

[5] Mohan Nair. On chebyshev-type inequalities for primes. *The American Mathematical Monthly*, 89(2):126–129, 1982.

[6] Lalitha Kiran Nemana and V Ch Venkaiah. An empirical study towards refining the aks primality testing algorithm. *Cryptology ePrint Archive*, 2016.