Algoritmo AKS para Verificação de Primalidade em Tempo Polinomial

FABIO ENRIQUE BROCHERO MARTINEZ
THIAGO MAIOLI CAMPOS

Universidade Federal de Minas Gerais,
Avenida Antonio Carlos 6627, Belo Horizonte, MG, Brasil
fbrocher@mat.ufmg.br
thiagomc@dcc.ufmg.br

Abstract. Este trabalho visa apresentar o resultado obtido por uma equipe de cientistas indianos, Agrawal, Kayal e Saxena onde mostram um algoritmo determinista em tempo polinomial para determinar se um dado número de entrada N é primo ou composto. Adicionalmente mostraremos algumas simplificações feitas sobre o algoritmo original, o pseudo-código de uma possível implementação e analisaremos sua complexidade algorítmica.

1 Introdução

Desde tempos remotos, problemas concernentes a números primos têm fascinado os matemáticos. De fato, Karl Friedrich Gauss (1777-1855), um dos maiores matemáticos, chegou a afirmar em seu Disquisitiones Arithmeticae (1801): "O problema de distinguir números primos de compostos e de decompor esses últimos em seus fatores primos é conhecido como sendo um dos mais importantes e úteis na aritmética... ...a dignidade da própria ciência parece requerer que todos os meios possíveis sejam explorados para a solução de um problema tão elegante e tão celebrado" (traduzido de Knuth [11]). Com o advento dos computadores, a partir de da década de 60, surgiram inúmeras tentativas de se obter uma formulação matemática que permitisse a concepção de um algoritmo eficiente para o teste de primalidade de um número. A relevância desse problema tem crescido imensamente em anos recentes devido à utilização intensa de números primos em algoritmos de criptografia, como os algoritmos RSA e El Gammal para criptografia pública. Dessa forma o problema do teste de primalidade se tornou um importante problema para a ciência da computação teórica. Sobre esse ponto de vista duas coisas são requeridas: um certificado de prova de que o algoritmo realmente produz a resposta correta; e uma medida da eficiência do algoritmo, isto é, quão bem o algoritmo faz uso dos recursos computacionais (como o tempo ou número de passos executados, espaço ou memória utilizada) em função do tamanho da entrada do problema para a obtenção da solução. O tamanho da entrada para um dado número N é o tamanho da sua codificação para a execução do algoritmo. Sabe-se que o número de bits necessários para representar um dado número N é aproximadamente $k = \log_2 N$, isto é $2^k \leq N < 2^{k+1}$. Para testar a primalidade de números pequenos, digamos todos os inteiros abaixo de 10^{10}, um dos algoritmos mais eficientes é devido ao matemático grego Eratóstenes (ca. 240 A.C.). Para testar a primalidade de N basta dividir esse número por todos os números primos menores que $\sqrt{N}$. Portanto, em termos do tamanho da entrada k, temos que o número de operações é $\mathcal{O}(\sqrt{N}) = \mathcal{O}(2^{k/2})$, ou seja, o algoritmo tem complexidade de tempo exponencial no tamanho da entrada.

Em agosto de 2002, um grupo de pesquisadores indianos, formado por um professor (Manindra Agrawal) e dois alunos de graduação (Neeraj Kayal e Nitin Saxena) do departamento de Ciência da Computação e Engenharia do Indian Institute of Technology, provou que o problema de teste de primalidade pertence à classe $\mathcal{P}$, ao obter o primeiro algoritmo determinista em tempo polinomial com relação ao tamanho da entrada, isto é, se a entrada tem k bits, o algoritmo determina a possível primalidade de N em tempo que é da ordem de um polinômio com relação a $k = \log_2 N$. Em particular o algoritmo não executa nenhuma escolha aleatória como fazem todos os algoritmos eficientes conhecidos até então. Esta descoberta deixou a comunidade de cientistas da área surpresos pelo fato de que, não apenas esse algoritmo resolve um problema de longa data, ele também o faz de uma maneira brilhantemente simples. Fica no ar a pergunta sobre o que mais tem sido deixado passar de forma semelhante.

A partir de então, esforços têm sido feitos para implementar o algoritmo de forma eficiente, acarretando no surgimento de diversas variantes, que agora são mencionados como pertencentes à classe AKS. Dentre essas variantes, pode-se destacar as de Lenstra [12], Pomerance [16], Berrizbeitia [4], Cheng [7], Bernstein [5] e Lenstra e Pomerance [13]. A tendência geral é a redução do expoente de complexidade k, cujo valor rigoroso é atualmente $k = 6 + \epsilon$, resultado obtido por [13], embora valores de até $k = 4 + \epsilon$ tenham sido obtidos para determinados valores de entrada, ou para qualquer valor de entrada assumindo hipóteses como a HGR (Hipótese Generalizada de Riemann) e/ou heurísticas. Muitos esforços têm sido concentrados para acelerar o loop de congruência do algoritmo, que é a etapa que consome maior tempo como será explicado a seguir.

2 Trabalhos Anteriores ao AKS

No século 17, Fermat provou o teorema que afirma que para qualquer número primo p, e qualquer número a não divisível por p,

$$a^{p-1} \equiv 1 \pmod{p}. \tag{1}$$

Embora o recíproco desse teorema não seja verdadeiro, esse resultado tem sido utilizado como ponto de partida para vários algoritmos eficientes de teste de primalidade. Em 1976, Miller [14] utilizou essa propriedade para a obtenção de um algoritmo determinista polinomial assumindo a HGR, um dos problema mais importantes da matemática contemporânea. Esse teste foi modificado por Rabin [17], que introduziu uma função de probabilidade e obteve um algoritmo muito mais eficiente e incondicional (isto é, sem utilizar a HGR), porém probabilístico. Entretanto, o algoritmo Miller-Rabin é em termos práticos o mais eficiente e utilizado em aplicações da vida real, como na criptografia, devido à probabilidade de erro extremamente baixa. O algoritmo de Miller-Rabin funciona pelo fato que se N não é primo então $a^{N-1} \equiv 1 \pmod{N}$ falha para aproximadamente 75% dos casos, assim se escolhemos aleatoriamente k valores de a sem fatores em comum, e eles validam a anterior equação, então a probabilibilidade que N seja primo é aproximadamente $1 - (\frac{1}{4})^k$.
Solovay e Strassen [18] obtiveram um outro algoritmo probabilístico em tempo polinomial utilizando resíduos quadráticos. Desde então, vários algoritmos probabilísticos têm sido propostos.
Um grande avanço ocorreu em 1983 com o trabalho de Adleman, Pomerance e Rumely [1] que obtiveram um algoritmo determinista e incondicional em tempo sub-exponencial $(\log n)^{\mathcal{O}(\log\log\log n)}$ (enquanto todos os outros algoritmos deterministas e incondicionais anteriores requeriam tempo exponencial), apesar de ser muito menos eficiente que o algoritmo de Miller-Rabin. Em 1986, Goldwasser e Kilian [10] propuseram um algoritmo probabilístico baseado em curvas elípticas com tempo esperado polinomial em quase qualquer entrada (ou qualquer entrada assumindo uma hipótese que se acredita ser verdadeira) e que produz um certificado de primalidade (até então, todos os algoritmos probabilísticos produziam certificados apenas de que o número era composto). Adleman e Huang [2] modificaram o algoritmo de Goldwasser-Kilian obtendo um algoritmo probabilístico em tempo polinomial que sempre produz um certificado de primalidade.

3 Fundamentação Matemática

Todos os algoritmos eficientes conhecidos até o momento, sejam deterministas ou probabilísticos se baseiam no teorema de Fermat: Um número p é um número primo se e somente se para todo número natural $a < p$ temos que $a^{p-1} - 1$ é divisível por p. Usando a notação de congruência equivale a dizer

$$p \text{ é primo} \Leftrightarrow a^{p-1} \equiv 1 \pmod{p} \text{ para todo } a < p$$

No algoritmo AKS, o fundamento matemático de fato não é diferente: Suponhamos que x é uma variável, a um inteiro e p um número primo. Usando o binômio de Newton temos que

$$(x+a)^p = \sum_{j=0}^{p} \binom{p}{j} x^{p-j} a^j,$$

mas nos casos em que j é diferente de 1 e p, o coeficiente binomial $\binom{p}{j}$ é divisível por p, logo todos os termos intermediários desta expansão são divisíveis por p, assim

$$(x+a)^p \equiv x^p + a^p \equiv x^p + a \pmod{p}$$

onde na última igualdade usamos o teorema de Fermat. Desta forma obtemos o seguinte critério de primalidade

$$N \text{ é primo } \Leftrightarrow (x+a)^N \equiv x^N + a \pmod{N}, \text{ para todo } a < N$$

Este critério, por enquanto é ineficiente, porque temos que calcular todos os coeficiente de $(x+a)^N$ e mostrar que todos os coeficientes intermediários são divisíveis por N. Outra observação importante é que se os polinômios $(x+a)^N$ e $x^N + a$ são iguais módulo N, então eles deixam o mesmo resto quando divididos por qualquer polinômio. Em particular, se dividimos por $x^r - 1$ temos que

$$N \text{ é primo } \Leftrightarrow \begin{array}{l}(x+a)^N \equiv x^N + a \pmod{x^r - 1, N}, \\ \text{para todo } a < N, r.\end{array}$$

O fato importante mostrado por Agrawal, Kayal e Saxena, é que só precisamos testar que esta congruência é valida para um valor especial de r que depende polinomialmente de $\log N$ e alguns poucos valores de a para determinar a primalidade de N. Formalmente temos

Teorema 3.1 (Agrawal, Kayal, Saxena) *Seja N um número inteiro positivo. Sejam q e r números primos e S um subconjunto com s inteiros tais que*

1. *q divide $r-1$ e $N^{(r-1)/q} \not\equiv 0$ ou $1 \pmod{r}$.*
2. *$mdc(N, a-b) = 1$ para elementos distintos a e b de S.*
3. *$\binom{s+q-1}{s} \geq N^{2\lfloor\sqrt{r}\rfloor}$*
4. *$(x+a)^N \equiv x^N + a \pmod{x^r - 1, N}$ para todo $a \in S$*

Então N é uma potência de um primo.

Em [3] se mostra, usando um teorema não elementar devido a Fouvry (ver [9]), a existência de r da ordem $\mathcal{O}((\log_2 N))^6)$ satisfazendo $q \geq 4\sqrt{r}\log N$. Assim tomando S com $2\sqrt{r}\log N$ elementos, temos que r, q e s cumprem a condição 1 e 3 do teorema. Se a condição 2 é falsa, significa que foi encontrado um fator de N, que portanto não seria primo. Assim o pseudo-código gerado pelo teorema anterior fica da seguinte forma

Algoritmo **AKS**
1. Entrada $N > 1$
2. Se $N = a^b$ com $b > 1$, retorna COMPOSTO
3. $r \leftarrow 2$.
4. Enquanto $(r < N)$ {
5. Se $mdc(r, N) \neq 1$, retorna COMPOSTO
6. Se r é primo {
7. Encontre q o maior divisor primo de $r - 1$
8. Se $q >= 2\sqrt{r}$ e $N^{(r-1)/q} \not\equiv 1 \pmod{r}$ vai para 10.}
9. $r \leftarrow r + 1$}
10. Para $a = 1$ até $2\sqrt{r}\log N$ faça
11. Se $(x + a)^N \not\equiv x^N + a \pmod{x^r - 1, N}$ retorna COMPOSTO
12. retorna PRIMO.

Lenstra mostrou a seguinte modificação do teorema anterior

Teorema 3.2 *Sejam N, r e v inteiros positivos, e $\varphi(r)$ denota o número de números menores que r e primos relativos com r. Seja S um conjunto finito com s elementos. Suponhamos que*

1. *N e r são primos relativos e a ordem de N módulo r é v, i.e. v é o mínimo tal que $N^v \equiv 1 \pmod{r}$.*
2. *$mdc(N, d_1 - d_2) = 1$ para elementos $d_1 \neq d_2 \in S$.*
3. *$\binom{s+\varphi(r)-1}{s} \geq n^{d\lfloor\sqrt{\varphi(r)/2d}\rfloor}$ para todo d divisor de $\varphi(r)/v$.*
4. *$(x+b)^N \equiv x^N + b \pmod{x^r - 1, N}$ para todo $b \in S$.*

Então N é potência de um primo.

Neste caso, empregando técnicas elementares de distribuição de primos pode-se mostrar que existe um primo r menor do que $\frac{1}{2}(\log N)^5$ tal que $v = \mathrm{Ord}_r\, N > \frac{1}{2}(\log N)^2$. De fato, se consideramos o número

$$\begin{aligned} M &= N(N-1)(N^2-1)\cdots(N^{\frac{1}{2}\lceil\log^2 n\rceil} - 1) \\ &< N^{(1+1+2+\cdots+\frac{1}{2}\lceil\log^2 n\rceil)} \approx 2^{\frac{\log^5 N}{8}}, \end{aligned}$$

temos que r é exatamente o menor primo que não divide M. Como conseqüência do teorema de Chebyshev sabemos que o produto dos primos menores que $2k$ é maior que 2^k. Portanto $r < \frac{1}{2}(\log N)^5$. Tomando o conjunto $S = \{1, 2, \ldots r\}$ com r elementos, temos que r, v e S satisfazem as condições 1 e 3 do teorema 3.2. De fato, os valores no conjunto S não têm fatores em comum com N, e a ordem v de N módulo r é maior do que $\lceil\frac{1}{2}(\log N)^2\rceil$. Assim se d é um divisor de $\varphi(r)/v$ então

$$d \leq \frac{(r-1)}{v} < \frac{(r-1)}{\log^2 n}$$

portanto

$$d\lfloor\sqrt{(r-1)/d}\rfloor \leq \sqrt{d(r-1)} < \frac{(r-1)}{\log n}$$

logo $n^{d\lfloor\sqrt{(r-1)/d}\rfloor} < 2^{r-1}$. Por outro lado como $s = r$ temos que $\binom{r+r-1-1}{r} = \binom{2r-2}{r} > 2^{r-1}$.
Observemos que não necessariamente estas são as melhores escolhas dos valores para S e r, ficando para análise futura a obtenção de valores ótimo para r e S. O pseudo-código associado a este teorema com estes valores de r e S fica da seguinte forma

Algoritmo **AKSL**
1. Entrada $N > 1$
2. Se $N = a^b$ com $b > 1$, retorna COMPOSTO
3. Encontrar o menor r tal que $\mathrm{Ord}_r\, N > \frac{1}{2}(\log N)^2$
4. Se $(mdc(a, n) > 1)$ para algum primo $a \leq r$, retorna COMPOSTO.
5. Se $(\sqrt{n} < r)$, retorna PRIMO
6. Para $a = 1$ até r faça
7. Se $((x+a)^n \not\equiv x^n + a \pmod{x^r - 1, n})$, retorna COMPOSTO.
8. Retorna PRIMO

4 Implementação e Complexidade

Nesta seção mostraremos uma possível implementação de cada passo do algoritmo anterior, assim como analisaremos a complexidade de cada passo. No que segue $\overline{\mathcal{O}}(k^n)$ significa $\mathcal{O}(k^n P(\log k))$, onde P é um polinômio. Observe que para todo $\epsilon > 0$, $\overline{\mathcal{O}}(k^n) < \mathcal{O}(k^{n+\epsilon})$.

4.1 Determinar se $N = a^b$

Observemos que se N é uma potência perfeita, isto é $N = a^b$ com $b > 2$, como $a \geq 2$ então $N = a^b \geq 2^b$, portanto $b \leq \log N$, o que gera o seguinte algoritmo que retorna 1 caso seja potência perfeita e 0 caso contrario

Algoritmo **PotenciaPerfeita**
Entrada N
1. Para $b = 2$ até $\log N$ faça {
2. $S \leftarrow \lfloor N^{1/b} \rfloor$
3. Se $S^b = N$ retorna 1}
4. Retorna 0.

Existem varias formas de implementar o passo 2.. Uma das mas simples é usando o método de Newton encontrado em qualquer livro de Cálculo elementar:"A seqüencia $\{x_n\}$ definida recorrentemente por $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ converge para uma raiz de $f(x) = 0$ para quase todo ponto inicial", além disso a velocidade de convergência é quadrática. Em nosso caso $f(x) = x^b - N$, e nosso ponto inicial é um valor a direita da raiz. Dado que estamos interessados somente em raízes inteiras, o algoritmo fica da seguinte forma

Algoritmo **RaizInteira**
Entrada N, b
1. $P = 2^{\lceil B(N)/b \rceil}$ *//*$B(N)$ *numero de Bits de* N
2. Faça{
3. $Q \leftarrow \left\lfloor \frac{(b-1)P + \lfloor N/P^{b-1} \rfloor}{b} \right\rfloor$
4. Se $(Q \geq P)$ *retorna* P
5. $P \leftarrow Q$}

Dada que a convergência do método de Newton é quadrática, em cada passo do loop obtemos o dobro de dígitos (em base dois) significativos, e no segundo passo já temos no mínimo 1 digito significativo. Isso significa que tal loop se repete no máximo $\log(\lceil B(N)/b \rceil) = \mathcal{O}(\log(\log N))$ vezes para obter a raiz. Dado que para calcular A^B preciso no máximo de $2 \log B$ multiplicações e uma divisão inteira tem complexidade equivalente à multiplicação, o algoritmo anterior tem complexidade $\mathcal{O}(\log N \log\log N)$. Portanto, podemos determinar, se um número N é potência perfeita com complexidade $\mathcal{O}((\log N)^2 \log\log N)$ no número de multiplicações de inteiros com no máximo $\log N$ dígitos. Como a multiplicação de inteiros com k bits, tem complexidade (k^2) usando o método clássico, ou $(k \log k)$ usando Transformada Rápida de Fourier (FFT), temos que a complexidade de determinar se um número é raiz perfeita é $\mathcal{O}((\log N)^3 (\log\log N)^2) = \overline{\mathcal{O}}((\log N)^3)$. Este valor está longe do ótimo. De fato, em [6], Bernstein mostra um algoritmo não elementar, usando aritmética de ponto flutuante, mas com complexidade

$$\log N \exp(\mathcal{O}(\sqrt{\log\log N \log\log\log N}) = \overline{\mathcal{O}}(\log N),$$

isto é, quase linear.

4.2 Ordem N módulo r

O seguinte é um algoritmo simples, para determinar a ordem de N módulo r, que verifica passo a passo qual é o menor inteiro j tal que $N^j \equiv 1 \pmod r$.

Algoritmo **Ordemmódulo**
Entrada N, r
1. Se $mdc(N, r) \neq 1$ *retorna* -1.
2. $A \leftarrow N \pmod r$
3. $B \leftarrow A$
4. $i \leftarrow 1$
5. Enquanto $B \neq 1$ *faça*{
6. $B \leftarrow A \cdot B \pmod r$
7. $i \leftarrow i + 1$}
8. Retorna i

Pelo teorema de Euler (generalização do teorema de Fermat para r não primo) sabemos que a ordem de N módulo r é um divisor de $\varphi(r) < r$ assim o passo 5 se repete no máximo r vezes. No passo 6. temos que fazer um produto de números com $\log r$ dígitos e reduzir módulo r, logo a complexidade do algoritmo **Ordemmódulo** usando FFT para multiplicar é $\mathcal{O}(r \log r \log\log r) = \overline{\mathcal{O}}(r)$. Assim como no passo 3. do algoritmo **AKSL** testamos todos os valores desde $\frac{1}{2}(\log N)^2$ até o menor r que cumpre a condição. Temos que a complexidade é no máximo $\mathcal{O}(r^2 \log r \log\log r) = \overline{\mathcal{O}}(r^2)$ para determinar r, que pode-se mostrar, por métodos elementares, é de fato menor do que $\frac{1}{2}(\log N)^5$.

4.3 Calculo de MDC

O seguinte algoritmo é clássico, e está baseado no algoritmo de Euclides:“ Seja r o resto ao dividir A por B, então $mdc(A, B) = mdc(B, r)$”. Usando interativamente este algoritmo até obter resto 0, obtemos o seguinte algoritmo

Algoritmo **MDC**
Entrada A, B
1. $R \leftarrow A - B\lfloor \frac{A}{B} \rfloor$
2. Enquanto $R \neq 0$ *faça* {
3. $A \leftarrow B$
4. $B \leftarrow R$
5. $R \leftarrow A - B\lfloor \frac{A}{B} \rfloor$}
6. Retorna B.

É fácil provar que o ciclo do algoritmo tem no máximo $\log_\varsigma \min\{A, B\}$ passos, onde $\varsigma = \frac{1+\sqrt{5}}{2}$ é a razão áurea, e este número é obtido exatamente quando tomamos dois termos consecutivos da seqüência de Fibonacci (Ver por exemplo [15]), assim a complexidade do passo 4 do algoritmo **AKSL** é $\mathcal{O}(r \log r)$ em número de multiplicações de números com $\log r$ dígitos, logo a complexidade usando FFT para multiplicar é

$$\mathcal{O}(r(\log r)^2 \log\log r) = \overline{\mathcal{O}}(r).$$

4.4 Calculo de $(x + a)^N$ módulo $(x^r - 1, N)$

Este passo é de fato, o mais complicado de implementar, e também tem a maior complexidade algorítmica, que ilustramos no seguinte algoritmo

Algoritmo **PotenciaPolinomio**
Entrada N, a, r *onde* $N = b_l b_{l-1} \ldots b_0$ *em base 2*
1. $P[x] \leftarrow 1$
2. Para $i = l$ *até 0 faça*{
3. $P[x] \leftarrow P[x]^2$
4. Se $b_i = 1$ *faça*
5. $P[x] \leftarrow P[x] \cdot (x + a)$.
6. $P[x] \leftarrow P[x] \pmod{x^r - 1, N}$
7. Retorna $P[x]$

Como estamos interessados em polinômios módulo $(x^r - 1, N)$ cada polinômio pode ser implementado como um arranjo com r entradas menores do que N. Assim se $P(x)$ é um polinômio de grau menor ou igual a $r-1$ então $(P(x))^2$ é um polinômio de grau menor ou igual $2r-2$, isto é, $(P(x))^2 = \sum_{j=0}^{2r-2} a_j x^j$. Observemos que

$$(P(x))^2 = \sum_{j=0}^{r-1}(a_j + a_{j+r})x^j + (x^r - 1)\sum_{j=0}^{r-1} a_{j+r}x^j$$

logo $(P(x))^2 \equiv \sum_{j=0}^{r-1}(a_j + a_{j+r})x^j \pmod{x^r - 1 - 1}$, assim aplicar módulo $x^r - 1$ é uma operação com complexidade linear com relação a r. Agora, a multiplicação de polinômios de grau r pode ser feita usando o método clássico com r^2 multiplicações e r somas, ou $r \log r$ multiplicações usando FFT, onde estamos multiplicando números com $\log N$ bits. Assim a complexidade dos passos 3., 4., 5. e 6. do algoritmo é $\mathcal{O}(r \log r \log N \log\log N) = \overline{\mathcal{O}}(r \log N)$, e portanto a complexidade do algoritmo **PotenciaPolinomio** é $\mathcal{O}(r \log r (\log N)^2 \log\log N) = \overline{\mathcal{O}}(r(\log N)^2)$.
Com isto concluímos que a complexidade dos passos 6. e 7. do algoritmo **AKSL** é

$$\mathcal{O}(r^2 \log r (\log N)^2 \log\log N) = \overline{\mathcal{O}}(r^2(\log N)^2).$$

Dado que a complexidade máxima do algoritmo **AKSL** ocorre nos passo 6. e 7. temos que a complexidade o algoritmo é $\overline{\mathcal{O}}(r^2(\log N)^2)$.
Na prática temos que $r = \mathcal{O}(\log N)^2)$, assim a complexidade do algoritmo fica igual a $\overline{\mathcal{O}}((\log N)^6)$.

5 Dados Experimentais

Os dados experimentais foram retirados de [8], onde foi usada uma maquina Apple G4 de 1GHz, obtendo para o tempo a fórmula empírica $T \sim C(\log N)^6$ onde $C \sim 1000$ clocks. Assim um número com 30 dígitos decimais levou aproximadamente um dia para a verificação. A implementação usou a biblioteca GMP para cálculo de inteiros de precisão arbitrária, mais não fica claro neste trabalho quais detalhes foram considerados na implementação do algoritmo. É conhecido que a biblioteca GMP emprega aproximadamente 100 clocks por dígito para realizar um produto entre dois números. Este resultado pode ser melhorado usando a biblioteca glucas (http://glucas.sourceforge.net) que emprega apenas 13 clocks por dígito. Em [5] D. Bernstein afirma que com os melhoramentos feitos por Lenstra, Bjorn Poonen, Felipe Voloch e Jeff Vaaler, o algoritmo fica $2 \cdot 10^6$ vezes mais rápido que o algoritmo AKS original, assim neste caso um número de 30 dígitos decimais pode ser verificado em menos de 0,04 segundos neste mesmo equipamento. Porém, existem poucos trabalhos, para não dizer nenhum, onde se encontre a análise da velocidade de tais melhoramentos. De fato na Web, encontramos unicamente implementações do algoritmo original. Por exemplo em http://perso.wanadoo.fr/yves.gallot/src/ pode-se encontrar três implementações do algoritmo AKS original usando respectivamente as bibliotecas GMP, NTL e MIRACL. Yves Gallot afirma que esta última implementação é mais eficiente em termos de tempo.

6 Conclusões

O Algoritmo AKS é interessante do ponto de vista teórico, já que mostrou que o problema de determinar a primalidade de um número está na classe $\mathcal{P}$, mas na prática o tempo de processamente é muito inferior com relação aos algoritmos probabilísticos clássicos, tais como Miller-Rabin e Solovay-Strassen que são altamente eficientes, e amplamente usados nos métodos de criptografia. De fato, escolhendo aleatoriamente 20 primos que validam os testes, temos que a probabilidade do número escolhido não ser primo é menor do que $9 \cdot 10^{-13}$. Por outro lado, caso a Hipótese Generalizada de Riemann seja verdadeira, o teste de Miller-Rabin torna-se um teste determinista, mas este problema ainda está em aberto e é considerado um dos problemas mais difíceis da matemática. De fato, o Clay Mathematics Institute oferece um prêmio de 1 milhão de dólares para a primeira solução, (ver http://www.claymath.org/millennium/). Como um possível trabalho futuro, fica aberta a possibilidade de diminuir o valor de r e o número de elementos de S do teorema 3.2, em nível de complexidade, já que neste momento sabemos que $rs = \mathcal{O}((\log N)^4)$, onde s é o número de elementos do conjunto S. Assim mesmo, um trabalho interessante pode ser a implementação de todos os melhoramentos teóricos encontrados na literatura, para realizar medições e comparações com o algoritmo original e com outros algoritmos determinísticos e/ou probabilísticos.

Referências

[1] Adleman L. Pomerance C. Rumely R. *On distinguishing prime numbers from composite numbers* Ann. Math. 117 pag. 173-206. (1983)

[2] Adleman L, Huang M. *Primality testings and two dimensional Abelian varieties over finite fields*. Lecture Notes in Mathematics. 1512 (1992)

[3] Agrawal, Kayal e Saxena, *Prime is P*, Preprint, IIT Kanpur, Agosto 2002, disponivel em http://www.cse.iitk.ac.in/news/primality.html

[4] P. Berrizbeitia, *Sharpening "Primes is in P" for a large family of numbers,* (2003) Available from http://arxiv.org/abs/math.NT/0211334.

[5] D. Berstein, *Detecting perfect powers in essentially linear time,* Math. Comp., 67:223 (1998) 1253–1283. Available from http://cr.yp.to/papers.html.

[6] D. J. Bernstein, *Proving primality in essentially quartic random time,* (2003) Disponivel em http://cr.yp.to/papers.html.

[7] Q. Cheng, *Primality proving via one round of ECPP and one iteration in AKS,* Crypto 2003, Santa Barbara, (2003) Disponivel em http://www.cs.ou.edu/ qcheng/.

[8] Crandall R., Papadopoulos J. *On the implementation of AKS-class primality tests*, Disponivel em http://www.apple.com/acg/pdf/aks3.pdf

[9] E. Fouvry, *Théorème de Brun-Titchmarsh; application au théorèm de Fermat,* Invent. Math., 79:2 (1985) 383–407

[10] Goldwasser S, Kilian J. *Almost all primes can be quickly certified*. Em Proc. of Ann. IEEE Symp. on Foundations of Computer Science. pag 316-329 (1986)

[11] Knuth, D. E. *The Art of computer Programming*, Vol 2, Seminumerical Algoritms. Addison Wesley, 1998.

[12] Lenstra, H. W. Jr. *Primality testing with cyclotomic rings*. Não publicado. exposição do argumento de Lenstra disponivel em http://cr.yp.to/paper.html#aks, Agosto 2002.

[13] Lenstra H W Jr, Pomerance C. *Primality testing with Gaussian periods*, manuscrito, exposto em XXIII rd Journées Arithmétiques Graz 2003, 6-12 julho 2003. Homepage http://ja03.math.tugraz.at/program.en.phtml?day=1

[14] Miller, G. L. *Riemann's hypothesis and tests for primality*. J. Comput. Sys. Sci. 13. pag 300-317 (1976)

[15] Nivel, Zuckerman, Montgomery,*An Introduction to the Theory of Numbers*, John Wiley and Sons(1991)

[16] Pomerance, C. *The cyclotomic ring test of Agrawal, Kayal and Saxena* contido em *Primality testing:variations on a theme of Lucas*, disponivel em http://cm.bell-labs.com/cm/ms/who/carlp/PS/primalitytalk5.ps

[17] Rabin, M. O. *Probabilistic algorithm for testing primality,* J. Number Theory, 12 (1980) 128–138.

[18] Solovay R. Strassen V. *A fast Monte-Carlo test for primality*. SIAM Journal on Computing, 6 pag 84-86, (1977)

APÊNDICE

Prova do teorema 3.1

Seja p um divisor primo de n, tal que $p^{(r-1)/q} \neq 0$ ou $1 \pmod r$. Logo por hipótese, $(x+a)^n = x^n + a$ no anel $\mathbb{F}_p[x]/(x^r-1)$ para todo $a \in S$. Substituindo x por x^{n^i} temos que $(x^{n^i}+a)^n = x^{n^{i+1}}+a$ no anel $\mathbb{F}_p[x]/(x^{n^i r}-1)$ para todo $a \in S$, e logo também no anel $\mathbb{F}_p[x]/(x^r-1)$. Indutivamente obtemos que $(x+a)^{n^i} = x^{n^i}+a$ no anel $\mathbb{F}_p[x]/(x^r-1)$ para todo $a \in S$. Pelo teorema de Fermat obtemos

$$(x+a)^{n^i p^j} = (x^{n^i}+a)^{p^j} = x^{n^i p^j} + a$$

no anel $\mathbb{F}_p[x]/(x^r-1)$ para todo $a \in S$.
Considere os produtos $n^i p^j$ com $0 \leq i,j \leq \lfloor\sqrt{r}\rfloor$. Note que

$$1 \leq n^i p^j \leq n^{2\lfloor\sqrt{r}\rfloor}.$$

Como temos $(\lfloor\sqrt{r}\rfloor+1)^2 > r$ pares (i,j), logo existem pares (i,j), (k,l) tais que $n^i p^j \equiv n^k p^l \pmod r$. Escrevendo $t = n^i p^j$ $u = n^k p^l$ temos que

$$|t-u| \leq n^{2\lfloor\sqrt{r}\rfloor}$$

e $x^t = x^u$ em $\mathbb{F}_p[x]/(x^r-1)$, logo $(x+a)^t = x^t + a = x^u + a = (x+a)^u$ em $\mathbb{F}_p[x]/(x^r-1)$.
Seja h polinômio irredutível em $\mathbb{F}_p[x]/(x^r-1)$ divisor de $(x^r-1)/(x-1)$. O grau de h é a ordem de p módulo r e portanto $\deg h \geq q$.
Agora $(x+a)^t = (x+a)^u$ no corpo $\mathbb{F}_p[x]/h$ para todo $a \in S$. Seja G o subgrupo de $(\mathbb{F}_p[x]/h)^*$ gerado por $\{x+a | a \in S\}$. Como os elementos de G são da forma

$$\prod_{a\in S}(x+a)^{e_a} \text{ com } \sum_{a\in S} e_b \leq q-1$$

portanto G tem no mínimo

$$\binom{s+q-1}{s} \geq n^{2\lfloor\sqrt{r}\rfloor} > |t-u|$$

elementos, e para todo $g \in G$ temos que $g^t = g^u$, mas esta equação pode ter no máximo $|t-u|$ soluções num corpo, logo $t=u$, isto é, $n^i p^j = n^k p^l$, mas se $i=k$ implica $j=l$, o que contradiz o fato de (i,j) e (k,l) ser diferentes, portanto n tem que ser uma potência de p.