Universidade Federal de Pernambuco

Centro de Informática

Ciência da Computação

2009.2

Testes de Primalidade: Uma Visão Computacional

Trabalho de Graduação

Aluno: Amirton Bezerra Chagas (abc@cin.ufpe.br)

Orientador: Profª. Drª. Liliane Rose Benning Salgado *(liliane@cin.ufpe.br*)

Recife, Novembro de 2009

Assinaturas

Este Trabalho de Graduação é resultado dos estudos sobre algoritmos de teste de primalidade desenvolvidos pelo aluno Amirton Bezerra Chagas, sob a orientação da professora Liliane Rose Benning Salgado. Todos abaixo estão de acordo com o conteúdo deste documento.

_____

Liliane Rose Benning Salgado

Orientadora

_____

Amirton Bezerra Chagas

Aluno

"$2B \vee \neg 2B$: That is the question"

William Shakespeare

Agradecimentos

A meu pai e minha mãe, os principais responsáveis por ter conseguido chegar até aqui com sucesso e determinação. Através de seus ensinamentos, repassados com muito amor e atenção, aprendi a ver o mundo de uma maneira que me permitiu superar as dificuldades que surgiram em todo o caminho que me trouxe até a conclusão deste trabalho.

A Ane, minha namorada, que agüentou todas as viradas de noite dos últimos três anos em que estamos juntos e me ajudou a superar as incertezas e os problemas que tive durante o curso.

Aos meus colegas de curso, que com seus conhecimentos, bom humor e disposição me motivaram nesta caminhada rumo à graduação. Em especial, agradeço aos bels da Proativa, sempre presentes, seja nos momentos sérios ou nos de diversão.

Aos meus professores, desde a pré-escola até a universidade. Através de suas habilidades em disseminar o conhecimento e despertar a chama da curiosidade dentro de mim, vocês, de forma incremental, criaram um imenso desejo de aprender o que já existe, de desenvolver minhas capacidades e de descobrir o novo.

A todos os amigos que sempre me ajudaram e me lembram constantemente que existe vida fora do CIn. Através de suas experiências e gostos diferentes, vocês me ajudam a não viver em um mundo isolado e a continuar conhecendo o que há de novo fora da computação.

Ao netbook em que realizei a maior parte deste projeto, que me acompanhou e possibilitou que eu trabalhasse onde quer que eu estivesse, e ao meu computador desktop, que aceitou sem questionamentos ser abandonado durante os meses que dediquei a este projeto.

Por fim, agradeço especialmente a minha orientadora, Prof[a] Liliane Salgado, e a Prof[a] Renata Souza, que me acompanham desde o segundo período, me aconselhando, ajudando e dando oportunidades das quais aproveitei muito, e por elas sou muito grato.

Resumo

Os números primos são estudados há milênios, e ainda ocultam vários mistérios que pouco a pouco são desvendados. Desde os primitivos métodos de força bruta para definir a primalidade de um número até os modernos algoritmos desenvolvidos atualmente, vários estudiosos dedicaram suas vidas a desenvolver métodos mais eficientes para realizar esta tarefa. Em 2002, um grupo de pesquisadores indianos conseguiu demonstrar através do algoritmo AKS que este problema pode ser resolvido deterministicamente e em tempo polinomial. Desde o anúncio deste algoritmo, vários outros pesquisadores refinaram o método proposto, ou criaram novos testes, baseados nesta idéia.

O objetivo deste trabalho é analisar o algoritmo AKS e as melhorias propostas desde sua publicação, focando nos fundamentos computacionais envolvidos. Também é fornecido um estudo dos algoritmos que historicamente foram e são utilizados na prática para realizar testes de primalidade, visto que o conhecimento destes métodos é um agente facilitador para a compreensão dos conceitos que baseiam o algoritmo AKS. Através de uma ferramenta de comparação com diversos testes implementados, é possível demonstrar e comparar vários destes algoritmos em função do tempo.

Palavras-chave: Teoria dos Números, Testes de primalidade, Algoritmos, Complexidade Computacional.

Sumário

CAPÍTULO 1
Introdução

Os números primos são objeto de estudo da matemática desde a antiguidade, e vários matemáticos dedicaram suas vidas a investigar suas propriedades em busca de padrões comuns. Antes mesmo dos gregos iniciarem o estudo formal da matemática, os primos já despertavam a curiosidade de povos da região mesopotâmica, por sua indivisibilidade por qualquer outro número além de 1 e ele mesmo. Alguns resultados desta curiosidade sobre os números primos influenciam nosso cotidiano até hoje, como o sistema de base 60, amplamente usado pelos Babilônios e Sumérios [1]. A base 60 tem a vantagem de ser um produto dos três menores primos (60 = 2 x 2 x 3 x 5), e é usada até hoje, na contagem das horas.

Apesar de todo o tempo que foi dedicado ao estudo dos primos desde Euclides, um uso prático para tais números foi encontrado apenas recentemente. Até os anos 1970, o estudo de características dos primos, testes de primalidade e a descoberta de grandes números primos eram atividades realizadas apenas por pesquisadores da matemática pura. Entre eles, o pacifista G. H. Hardy chegou a afirmar durante a Segunda Guerra Mundial que se orgulhava de ser um dos poucos cientistas da época trabalhando em pesquisas que nunca poderiam ser usadas para guerras. Em seu livro A Matematician Apology [2], Hardy chegou a escrever:

"I have never done anything 'useful'. No discovery of mine has made, or is likely to make, directly or indirectly, for good or ill, the least difference to the amenity of the world".

Hardy estudava Teoria dos Números, e participou de pesquisas relacionadas aos números primos, que hoje são o núcleo de vários sistemas criptográficos, considerados pela legislação de vários países como armas.

Dada a provável dificuldade do problema da fatoração, e o fato conhecido desde os antigos gregos de que todo número natural possui uma única fatoração prima, ou seja, pode ser escrito como o produto de vários primos, Rivest, Shamir e Adleman [3] publicaram o algoritmo de criptografia de chave pública RSA, que depende da geração de números primos grandes para seu funcionamento. Usando estes primos como chave privada, RSA gera uma chave pública que possui

uma probabilidade desprezível de ser quebrada sem o conhecimento dos primos usados em sua geração. Desta forma, apenas as partes legitimamente envolvidas são capazes de encriptar e decriptar as mensagem, pois tendo em mãos a chave privada, reverter o processo é simples.

A descoberta de um uso prático para números primos, principalmente numa área estratégica, de extremo interesse por parte de governos e grandes empresas foi um grande incentivo para que mais pesquisas se realizassem na área. Várias destas pesquisas se voltaram para um dos problemas mais intrigantes daquele momento: Não se sabia se PRIMO, o problema de afirmar que um número é ou não primo, estava na classe de complexidade P, os problemas resolvidos deterministicamente em tempo polinomial. Além dos métodos triviais, de tempo exponencial, existiam alguns algoritmos que resolviam ou probabilisticamente em tempo polinomial, ou deterministicamente, mas não gerais, funcionando apenas para alguns primos com determinadas características.

Vários estudos foram conduzidos para se encontrar um algoritmo determinístico e polinomial que pudesse mostrar que o problema PRIMO é um membro de P. Resultados que utilizam estruturas complexas e matemática avançada foram encontrados, possuindo custo muito perto de polinomial, mas não conseguiram alcançar a classe P, como o algoritmo APR. Para outros ainda não se sabe o custo exato, como ECPP. Outros, como o algoritmo de Miller, são determinísticos e polinomiais na dependência de teorias ainda não provadas, como a hipótese de Riemann. Encontrar um algoritmo de tempo polinomial e determinístico, utilizando uma matemática simples não parecia ser algo possível, dado todo o tempo que já havia sido dedicado ao tema.

Em 2002, três indianos, dois deles que haviam acabado de concluir a graduação em Ciência da Computação, conseguiram projetar um algoritmo que provou que PRIMO está em P, sem depender de conjecturas [4]. O algoritmo AKS, nome formado pelas iniciais de seus três idealizadores, utiliza matemática básica em sua estrutura, e apenas na sua prova original, dependia de um conceito avançado. Com as melhorias propostas por várias pessoas, os autores disponibilizaram uma versão atualizada de seu artigo [5], mostrando uma prova da correção de AKS apenas com matemática básica também.

A grande importância de AKS na área teórica, resolvendo um problema que há muito tempo povoava o imaginário daqueles que se dedicam a pesquisas em testes de primalidade, não encontra ainda grandes resultados práticos. Apesar de ser um algoritmo polinomial, seu expoente e sua constante ainda são muito altos para permitir o uso na maioria das aplicações. Os testes probabilísticos ainda são mais rápidos e promovem resultados com um grau de certeza muito além do que a maioria das aplicações necessita. Para encontrar primos titânicos, aqueles primos de 1.000

dígitos ou mais, testes focados em primos com características especiais, como os números de Mersenne, também são muito mais eficientes do que o AKS.

Este trabalho tem como objetivo expor de maneira sucinta os algoritmos de teste de primalidade desenvolvidos ao longo da história. Também apresenta um estudo aprofundado do AKS, das melhorias já propostas para este algoritmo e de algoritmos que surgiram após a publicação de Agrawal, Kayal e Saxena.

No Capítulo 2, serão definidos alguns dos principais tópicos necessários para o pleno entendimento deste trabalho. O Capítulo 3 irá descrever diversos algoritmos de teste de primalidade considerados clássicos, que foram desenvolvidos entre 300 a.C. e 1980 d.C. O Capítulo 4 trás alguns algoritmos desenvolvidos entre 1980 e 2001, que possuem características diferentes dos algoritmos clássicos em relação as suas complexidades computacionais e nível matemático. O Capítulo 5 é totalmente dedicado ao Teste AKS, explicando seu histórico, conceito, funcionamento e melhorias propostas desde sua publicação. O Capítulo 6 descreve de forma sucinta a ferramenta desenvolvida neste trabalho, que executa diferentes algoritmos para um dado número e plota um gráfico comparando os desempenhos obtidos.

CAPÍTULO 2
Preliminares

Nesta seção, serão descritos alguns tópicos importantes para a compreensão deste trabalho. O tratamento dado a estes tópicos será superficial, caso haja a necessidade de mais aprofundamento, consulte as referências citadas no texto.

2.1. Teoria dos Números

A teoria dos números teve seu início com a matemática grega, aproximadamente em 500 a.C. O principal foco desta área da matemática é o estudo das propriedades dos números em geral, com um enfoque principal nos inteiros. Apesar de ser estudada por dois milênios e meio, ainda existem algumas questões muito simples sem resposta nesta área. Alguns outros problemas aparentemente simples foram solucionados apenas há poucos anos atrás, como o problema de descobrir se um número é ou não primo deterministicamente em tempo polinomial [6].

Este ramo da matemática está diretamente ligado às propriedades dos números primos. Um dos primeiros resultados da Teoria dos Números foi o teorema fundamental da aritmética, que prova que todo número inteiro maior que 1 pode ser decomposto em um produto de números primos, e que esta decomposição é única. Outro resultado importante é o Teorema de Euclides, que prova que existem infinitos números primos.

Apenas recentemente, como já foi citado, foram encontradas aplicabilidades para números primos que resolvem problemas do mundo real. Outro tópico da Teoria dos Números que tem encontrado aplicação são os resíduos quadráticos. As ferramentas criadas a partir deste tópico são usadas em Engenharia Acústica, fatoração de números grandes e também em Criptografia.

Uma característica interessante de alguns dos tópicos abordados pela Teoria dos Números é que estes são de fácil entendimento ao público leigo, sendo até apresentados a alunos de ensino fundamental. As provas destes mesmos tópicos por sua vez podem ser tão complexas que necessitam de séculos de trabalho conjunto e contínuo de várias gerações de pesquisadores para serem formuladas, e muito tempo e conhecimento em matemática para serem entendidas posteriormente.

Um tópico da Teoria dos Números relevante para este trabalho é a aritmética modular, usada extensivamente na maioria dos testes de primalidade e apresentada na próxima seção.

2.2. Aritmética Modular

Este tópico trata da congruência de números inteiros. Dados os inteiros a, b e m, dizemos que a é congruente a b modulo m quando possuem o mesmo resto na divisão por m. Formalmente, escreve-se $a \equiv b \pmod{m}$ e diz-se que a é equivalente a b módulo m.

Esta equivalência ocorre por respeitar as seguintes propriedades:

- Reflexividade - $a \equiv a \pmod{m}$
- Simetria - $a \equiv b \pmod{m} \implies b \equiv a \pmod{m}$
- Transitividade - $a \equiv b \pmod{m};\ b \equiv c \pmod{m} \ \Rightarrow \ \ a \equiv c \pmod{m}$

Para testes de primalidade, outra propriedade importante da aritmética modular é a distributividade, a saber:

$$c \equiv a.b \pmod{m} \ = \ \ c \equiv \big(a \pmod{m}\big) . \ \big(b \pmod{m}\big) \pmod{m}.$$

A distributividade permite que o cálculo de uma potenciação modular, cujo expoente iria causar a necessidade de lidar com números muito grandes, seja realizado apenas na quantidade de bits necessários para representar m. Por exemplo:

$$a^3 \pmod{m} = \ \big(a^2 \pmod{m}\big) . \big(a \pmod{m}\big) \pmod{\mathrm{m}}$$

$$= \ \big(a \pmod{m}\big) . \big(a \pmod{m}\big) \ . \big(a \pmod{m}\big) \pmod{\mathrm{m}}$$

Por evitar a computação com números muito grandes, este método além de economizar na memória necessária para representar os números, é consideravelmente mais rápido do que simplesmente exponenciar e depois aplicar a função módulo.

2.3. Complexidade Computacional e Análise de Algoritmos

Com o surgimento dos computadores, a percepção do custo, ou seja, da quantidade de tempo e/ou espaço necessários para a resolução de um problema em uma máquina se tornou essencial. Através do cálculo do custo é possível determinar aproximadamente a quantidade de recursos necessária quando o tamanho da entrada cresce. Este custo é a complexidade computacional para a resolução de um problema.

Para representar a complexidade, utiliza-se uma função sobre o tamanho da entrada. Dependendo do custo de um problema, pode ser viável ou não resolvê-lo, ou seja, caso a função tenha uma taxa de crescimento muito alta, uma instância relativamente pequena do problema pode necessitar de uma quantidade de recursos que inviabilize o cálculo de uma resposta. Por exemplo, necessitar de vários milênios ou de alguns Yottabytes (10^{24}) para oferecer a solução.

O que define a pertinência de um problema a uma classe de complexidade é a existência de uma solução de custo compatível com a classe em questão. Cada classe descreve de maneira geral quão viável é a melhor solução existente para este problema. Aqueles que podem ser resolvidos em tempo polinomial em uma máquina de Turing determinística são classificados como pertencentes à classe **P** e são conhecidos como problemas tratáveis ou para os quais se conhece uma solução eficiente.

Uma solução para um problema é descrita por uma sequência finita de passos. Esta sequência é denominada como um algoritmo para o respectivo problema. Um algoritmo computacional é uma estratégia de resolução que executa em uma máquina de Turing fazendo-a parar em um estado final, que descreve a resposta.

A análise da complexidade de um algoritmo não necessita geralmente explicitar a exata função que descreve o seu custo. Como foi deixada implícita anteriormente, a informação que esta função deve passar é que, para uma entrada de tamanho n suficientemente grande, um algoritmo irá se comportar de uma determinada maneira. Desta forma, o algoritmo descrito pela função $f(n) = 3n^4$ irá crescer de forma suficientemente parecida com outro de custo $f(n) = 7n^4$ ou mesmo com outro algoritmo de custo $f(n) = n^4 + 2n^2 + 5$.

Para representar este comportamento assintótico, é comum utilizar a notação Big-O. Em resumo, esta notação ignora as constantes aditivas e multiplicativas e termos de menor grau em detrimento do termo que apresenta o maior expoente, de forma que represente um limitante superior para o custo do algoritmo. Desta forma, é possível comparar os algoritmos independentemente do tamanho da entrada, pois à medida que esta cresce, a quantidade de recursos necessária irá crescer de maneira similar para algoritmos de custo representado de forma idêntica pela notação Big-O. Assim, pode-se dizer, por exemplo, que os custos dos algoritmos definidos pelas três funções do parágrafo anterior possuem $O(n^4)$, onde n é o tamanho da entrada.

A notação Big-O representa um limite superior, logo, pode-se dizer que se $a = O(\sqrt{x})$, o valor a cresce na mesma medida que a raiz quadrada de x, mas não é necessariamente igual a $\sqrt{x}$, pois como dito anteriormente, a notação Big-O pode ocultar alguma constante ou termo de menor grau.

Outra notação importante no contexto de testes de primalidade é a Soft-O, escrita pela forma $\tilde{O}(f(x))$. Através desta notação é possível ocultar termos multiplicativos logarítmicos, logo, $O(x^2 \log x) = \tilde{O}(x^2)$, ou $O(\log^3 x \ . \log\log x) = \tilde{O}(\log^3 x)$.

Para maiores detalhes sobre análise de algoritmos, recomenda-se a leitura de Cormen et al. [7] e das seções 1.2.10 e 1.2.11 de Knuth [8].

2.4. Números Primos

Números primos são aqueles inteiros $p > 1$ tal que p não é divisível por qualquer outro número natural positivo além de 1 e o próprio p. Uma definição mais rigorosa cita que um número é primo se possui apenas e exatamente dois divisores distintos, desta forma, o número 1 não é primo. Números inteiros $n > 1$ que possuem divisores diferentes de ± 1 e $\pm n$ são chamados números compostos. Devido ao Teorema Fundamental da Aritmética, os primos são também conhecidos como os blocos básicos dos inteiros, dado que qualquer inteiro pode ser escrito como o produto de primos. O teorema fundamental da aritmética é outro motivo para o número 1 não ser primo, dado que este teorema define que existe apenas uma decomposição prima para qualquer número, o que seria falso caso 1 fosse primo (o número 2 poderia ser escrito como (2), (2 x 1), (2 x 1 x 1), por exemplo).

Várias tentativas de encontrar características comuns aos primos foram realizadas ao longo do tempo. A maioria delas, no entanto levou a resultados que realmente são propriedades válidas para todos os primos, mas que alguns compostos também possuem. Estes números compostos que possuem alguma característica em comum com os primos são chamados de pseudoprimos. Existem algumas classes famosas, como os pseudoprimos de Fermat, que são aqueles números compostos que satisfazem o pequeno teorema de Fermat em alguma base e os números de Carmichael, que são aqueles compostos que satisfazem o pequeno teorema de Fermat para qualquer base.

Também existem algumas propriedades que são válidas apenas para alguns primos. Estas características definem subconjuntos dentro dos primos, que podem ser muito úteis. Os números de Mersenne, por exemplo, são aqueles na forma $M_n = 2^n - 1$, onde n é um inteiro. Os primos de Mersenne são os números primos que satisfazem esta formula. Uma característica importante dos primos de Mersenne, é que quando M_n é primo, então n deve ser também primo. De acordo com [9], apesar de serem conhecidos poucos primos de Mersenne até o momento, existem algoritmos suficientemente eficientes em encontrar primos de Mersenne grandes, e isto está diretamente relacionado a esta última característica citada. Atualmente, dos 10 maiores primos conhecidos, nove são números de Mersenne.

Existem também os certificados de primalidade, que são informações geradas por alguns algoritmos de teste de primalidade que não rodam em tempo polinomial, que permitem que um primo previamente testado seja certificado como primo em tempo polinomial, sem a necessidade de rodar novamente todo o algoritmo. Hoje, com a existência do algoritmo AKS, estes certificados podem ser considerados de pouco valor, visto que o próprio número primo é seu certificado, dado que pode ser testado em tempo polinomial.

Um conceito relacionado aos números primos são os coprimos, ou números mutuamente primos. Os números inteiros de um conjunto de tamanho maior ou igual a 2 são considerados coprimos se seu máximo divisor comum é 1. Obviamente, um primo é coprimo em relação a qualquer outro primo. Para conjuntos com apenas números compostos, ou números compostos e números primos misturados, deve-se verificar o máximo divisor comum (mdc) para poder classificar os números como coprimos ou não. Outro conceito é o de números coprimos dois a dois, ou primos entre si. Neste caso, não basta que o mdc do conjunto seja 1. É necessário que o mdc de cada possível dupla de elementos do conjunto também seja igual a 1. Por exemplo, o conjunto {4,6,7} é coprimo, pois mdc(4,6,7) = 1. Este mesmo conjunto no entanto não é coprimo dois a dois, pois mdc(4,6) = 2.

2.5. O Problema da Fatoração

É um fato comum as pessoas confundirem o problema de decidir se um número é ou não primo com o problema da fatoração. Apesar disto, estes são problemas distintos, e encontrar um algoritmo que teste a primalidade de um número em tempo polinomial não leva imediatamente a um algoritmo que fatore um número eficientemente. O inverso, ou seja, um algoritmo que resolva o problema da fatoração deterministicamente em tempo polinomial, obviamente resolve o problema PRIMO, pois com o resultado deste algoritmo, bastaria verificar que o único fator primo para o número fatorado é ele próprio, retornando assim *true* para PRIMO, ou *false* caso contrário.

Atualmente, o problema da fatoração é considerado mais difícil que o do teste de primalidade, uma vez que AKS mostrou que o segundo problema está em P, enquanto ainda não se conhece um método eficiente para fatorar. Apenas por curiosidade, é valido lembrar que o algoritmo de Shor roda em tempo polinomial para resolver o problema da fatoração, no entanto, funciona apenas em computadores quânticos [10].

CAPÍTULO 3

Testes de primalidade (500 a.C – 1980 d.C.)

Este capítulo descreve vários dos métodos usados para testar se um dado número é ou não primo, criados desde a Grécia antiga até 1980. É importante perceber que com exceção ao teste de Miller-Rabin, todos estes métodos foram desenvolvidos antes de alguma aplicação real para os primos ser descoberta. Estes são métodos clássicos de teste de primalidade. Contudo, apesar de antigos, neste capítulo são descritos os testes mais eficientes na prática, logo, os mais usados atualmente em situações reais.

3.1. Divisão por Tentativa

A divisão por tentativa é o método mais trivial para resolver o problema de determinar se um número é ou não primo. Através da força bruta, este método varre todo o espaço de 2 a n, sendo n o número a ser testado, em busca de algum divisor para n diferente dele próprio. Uma rápida observação diminui o espaço da busca para $\frac{n}{2}$. É possível ainda reduzir o espaço da busca para apenas $\sqrt{n}$. Ainda assim, este espaço para números muito grandes é inviável de ser testado por completo.

Obviamente, este método é determinístico, porém sua complexidade é $O(2^{\sqrt{n}})$. É necessário lembrar que a complexidade de algoritmos é calculada com base no tamanho da entrada, que neste caso, é a quantidade de dígitos da representação binária do número em questão, e não o próprio número. Esta dúvida faz algumas pessoas acreditarem inicialmente que a complexidade deste método é $O(\sqrt{n})$. Isto é obviamente falso, pois para um número de tamanho 1 (um dígito decimal), temos $\sqrt{10}$ números a serem testados. Um número de tamanho 2 terá $\sqrt{100}$ = $\sqrt{10^2}$ valores a serem testados, um de tamanho 3 terá $\sqrt{1000} = \sqrt{10^3}$, e um de n terá que testar $\sqrt{10^n}$ valores, logo, o crescimento é exponencial. Embora neste exemplo tenha sido utilizada a base decimal por questões didáticas, a base utilizada no cálculo da complexidade é 2, pois é considerado o tamanho na representação binária, e não decimal.

Outro problema considerável para o método da divisão por tentativa é que ele não fornece um certificado verificável em tempo polinomial para um primo que tenha passado no teste, logo, para verificar o seu resultado, seria necessário rodar novamente todo o algoritmo.

Apesar de seu alto custo, a simplicidade da divisão por tentativa faz com que ela seja uma boa opção para testar a primalidade de um número relativamente pequeno. Esta simplicidade também faz com que este seja o método de teste de primalidade ensinado a crianças no ensino fundamental, junto com os conceitos básicos de primalidade.

É possível ainda aumentar um pouco a velocidade da divisão por tentativa, diminuindo em três vezes o número de divisões a serem realizadas. Outra otimização considera a vantagem de que os inteiros maiores que três podem ser escritos da forma $m = 6k + i$, onde m e k são inteiros quaisquer e i é um inteiro pertencente ao intervalo [-1..4]. Quando i = 0, 2, ou 4, m é divisível por 2, e quando i = 3, m é divisível por 3, logo, sobram apenas os inteiros da forma $6k \pm 1$. A otimização consiste em dividir apenas por 2, 3, e pelos inteiros da forma $6k \pm 1$ até $\sqrt{n}$, sendo n o número testado. A complexidade é decrescida apenas de uma constante, logo, o custo continua sendo exponencial para este algoritmo.

3.2. Crivo de Eratóstenes

O Crivo de Eratóstenes é um método muito antigo para encontrar todos os primos em um dado intervalo. Seu princípio básico é a eliminação dos múltiplos de cada primo, fazendo com que, ao final do algoritmo, sobrem apenas os primos.

O primeiro passo deste algoritmo é escrever os números de 2 a n, sendo n o final do intervalo. Depois, deve-se marcar como primo o primeiro número não marcado, e marcar como cortado todos os seus múltiplos até n, até não existirem mais números não marcados. Sua saída são os números marcados como primo.

Observa-se que ao usar o Crivo de Eratóstenes, assim como a divisão por tentativa só precisa dividir até $\sqrt{n}$ para decidir se o número é ou não primo, este algoritmo está com todos os números marcados como primos ou compostos ao chegar em $\sqrt{n}$. O Crivo de Eratóstenes é determinístico e tem custo exponencial de $O(2^n \log n)$, maior que o da divisão por tentativa, mas para encontrar todos os primos em intervalos cujo limite superior não seja muito alto, sua simplicidade o torna uma escolha aceitável. Isto faz com que ainda hoje este crivo seja alvo de estudos, como em [11], onde se desenvolvem versões do Crivo de Eratóstenes usando arquiteturas paralelas e o modelo BSP (*Bulk Synchronous Parallelism*) ou mesmo na contribuição para a criação de novos algoritmos, como o Crivo de Atkin [12].

3.3. Teste de Primalidade de Fermat

As contribuições de Pierre de Fermat mais lembradas na matemática são àquelas realizadas na área do cálculo, tanto geométrico como infinitesimal. Este grande matemático do século XVII também é lembrado pelo último teorema de Fermat, que levou 358 anos para ser provado.

Um número é perfeito se a soma de seus divisores positivos excluindo a si mesmo é igual ao próprio número. Enquanto estudava os números perfeitos, Fermat chegou àquele que hoje é conhecido como o Pequeno Teorema de Fermat:

Se p é um número primo, então para qualquer número inteiro a, temos que,

$$a^p \equiv a \pmod{p} \qquad \text{Equação 3.1}$$

Uma versão equivalente deste teorema é considerar que se p é primo e a é coprimo em relação à p, então,

$$a^{p-1} \equiv 1 \pmod{p} \qquad \text{Equação 3.2}$$

Fermat não provou seu pequeno teorema, o que ficou a cargo de Euler, que publicou uma prova formal apenas em 1736. Este teorema se mostrou de grande utilidade para a teoria dos números, sendo usado até hoje como uma etapa em diversos algoritmos, inclusive no AKS. Uma generalização do Pequeno Teorema de Fermat é utilizada em [3], para demonstrar o funcionamento do método de criptografia RSA. O primeiro produto deste teorema, no entanto foi o Teste de Primalidade de Fermat.

Um tópico que geralmente é levantado ao se abordar o Pequeno Teorema de Fermat é a chamada "Hipótese Chinesa", que teria sido proposta em 500 A.C. e afirmaria que se um número é primo, $2^p = 2 \pmod{p}$. A Hipótese Chinesa é encontrada em várias referências sobre testes de primalidade, no entanto, nunca existiu. Ribenboim esclarece como este mito foi criado em [13], explicando que o ocorrido foi um erro de compreensão do assunto por parte de um tradutor. Apesar deste erro já ter sido exposto, a hipótese continua sendo referenciada até os dias atuais.

É importante frisar que o Pequeno Teorema de Fermat é válido apenas em um sentido: Se p é primo, ele satisfaz a Equação 3.1 para qualquer a inteiro. O inverso não é verdadeiro, pois existem números n tais que para todo a inteiro, $a^n \equiv a \pmod{n}$. Estes números são chamados de Números de Carmichael, e é provado em [14] que apesar de serem raros, existem infinitos destes números.

O Teste de Primalidade de Fermat é um teste probabilístico, dada a existência dos números de Carmichael, que se resume a testar a equação do Pequeno Teorema de Fermat uma quantidade k de

vezes com valores aleatoriamente escolhidos para a. Quanto maior k, maior a confiança que se pode ter no resultado, no entanto, esta confiança nunca será de 100% devido aos números de Carmichael.

Quando um a testado não confirma a igualdade da equação, este a é uma testemunha de que o número n testado é composto. O teste pode parar e o algoritmo responder com certeza que o número é composto. Se a confirma a igualdade, e o teste ainda não foi realizado k vezes, deve-se tentar um novo valor de a. Se após a execução do teste k vezes não for encontrada uma testemunha para comprovar que o número sendo testado é composto, o algoritmo retorna que o número provavelmente é primo.

É comum as implementações práticas deste algoritmo usarem um pequeno valor para k, ou mesmo não escolher aleatoriamente os valores, dando preferência à escolha prévia de valores fixos para a. De acordo com [7], se o número a ser testado é escolhido ao acaso, basta rodar este teste uma vez, fixando $a = 2$. A probabilidade de erro do algoritmo nessas condições, para um número aleatoriamente escolhido de 1024 bits é de apenas uma em 10^{41}, o que para a maioria dos propósitos práticos, é uma chance de erro aceitável, logo, o valor de k não precisa necessariamente ser alto para atingir um nível de confiança aceitável com este algoritmo. Alguns programas vastamente usados se baseiam em Testes de Primalidade de Fermat para a geração de números primos. A segunda versão do PGP, um programa vastamente usado para criptografar emails, usa o Teste de Primalidade de Fermat com os valores fixos de a = 2, 3, 5 e 7 para encontrar números primos de no mínimo 128 bits utilizados para gerar as chaves necessárias para a criptografia [15].

Um dos principais motivos para a adoção deste teste em vários programas é que juntamente ao bom nível de confiança nos seus resultados, este teste possui um tempo de execução polinomial de $\tilde{O}(k \cdot \log^2 n)$ quando se usa exponenciação modular, e é muito rápido na prática.

3.4. Teste de Lucas-Lehmer

O teste de Lucas-Lehmer é um método determinístico e polinomial para estabelecer a primalidade de números de Mersenne. Este algoritmo é fruto do trabalho de Lucas, que em 1876 desenvolveu um teorema posteriormente melhorado por Lehmer em 1930.

O algoritmo é de simples compreensão, usando em seu corpo apenas aritmética modular e, para um inteiro n qualquer, uma equação recursiva da forma:

$$\begin{cases} s_0 = 4 \\ s_n \equiv s_{n-1}^2 - 2 \pmod{M_p} \end{cases}$$ Equação 3.3

No entanto, a prova original encontrada por Lucas é não-trivial, e mesmo utilizando-se de melhoramentos mais recentes, a prova de correção deste teste ainda exige conhecimentos em Teoria dos Grupos e alguns tópicos em Teoria dos Números. Uma prova suficientemente simplificada para este teorema pode ser encontrada em [16].

Considere uma sequência s_k, com $k = 0, 1, \ldots$, definida recursivamente pela Equação 3.3. Temos que $M_p = 2^p - 1$ será primo se somente se $s_{p-2} \equiv 0 \pmod{M_p}$. A prova para este teorema pode ser encontrada em [10]. O teste de Lucas-Lehmer é a aplicação deste resultado, e se resume a calcular s_{p-2} e retornar *true* se $s_{p-2} = 0$ e *false* caso contrário.

Utilizando uma otimização que permite efetuar adições no lugar do módulo da equação recursiva [10] e computando a multiplicação ($s^2 = s.s$) usando a Transformada Rápida de Fourier (FFT), o custo deste algoritmo é de $\tilde{O}(p^2 \log p)$ [17]. Este baixo custo em comparação a outros algoritmos de teste de primalidade, juntamente às características dos números de Mersenne são os principais responsáveis pelo fato já mencionado de que os maiores primos conhecidos atualmente são primos de Mersenne.

É relativamente rápido testar a primalidade de números de Mersenne usando o teste de Lucas-Lehmer. O segundo maior primo conhecido até o momento possui quase 13 milhões de dígitos e foi testado em "apenas" 29 dias em um desktop comum (Intel Core2 3.0GHz), que faz parte da GIMPS – Great Internet Mersenne Prime Search, uma rede que distribui números de Mersenne a serem testados para computadores de voluntários ao redor do mundo. Cada computador executa o teste completo para o número que lhe foi designado, e comunica a um servidor central, chamado PrimeNet, o resultado para este número. O projeto segue em busca de primos titânicos, e usa uma política de distribuição dos prêmios ganhos com os voluntários envolvidos. Mais informações podem ser encontradas na página do projeto, em [18].

Uma curiosidade, diretamente ligada à baixa complexidade deste método, é que o último grande primo descoberto totalmente sem o uso de computadores é M_{127} (um número de 29 dígitos), fato este demonstrado pelo próprio Lucas em 1876, utilizando este teste [13].

3.5. Teste de Solovay-Strassen

O Teste de Solovay-Strassen [19] é um teste probabilístico de tempo polinomial do tipo Monte-Carlo. Este teste escolhe aleatoriamente parte dos dados da entrada para realizar a computação e fornecer um resultado provavelmente correto. Este método possui uma importância histórica, pois foi o teste utilizado no artigo seminal de RSA para demonstrar que é possível encontrar primos grandes aleatoriamente, necessidade básica do algoritmo de criptografia proposto

no artigo. Atualmente, ele praticamente não é usado, devido ao surgimento do teste de Miller-Rabin, mais rápido e com menor probabilidade de erro do que Solovay-Strassen.

Este método utiliza o Critério de Euler, que afirma que para um primo p ímpar, e qualquer inteiro a,

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \pmod{p} \qquad \text{Equação 3.4}$$

Este critério pode ser considerado uma versão mais forte do Pequeno Teorema de Fermat [20]. Neste caso, $\left(\frac{a}{p}\right)$ é o Símbolo de Legendre, que exige que p seja primo e assume os seguintes valores:

$$\begin{cases} 0, & \text{caso } p \text{ divida } a \\ 1, & \text{caso } a \text{ seja um resíduo quadrático módulo } p \\ -1, & \text{caso } a \text{ não seja um resíduo quadrático mod } p \end{cases}$$

Um inteiro a é um resíduo quadrático, se existe algum valor x que satisfaça $x^2 \equiv a \pmod{p}$. Este critério foi provado por Euler, e leva ao teste de Solovay-Strassen, que verifica para valores aleatórios de a, se um número inteiro n é primo com a seguinte congruência:

$$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n} \qquad \text{Equação 3.5}$$

Neste caso, $\left(\frac{a}{n}\right)$ é o Símbolo de Jacobi, onde n não precisa ser primo, e é uma generalização do Símbolo de Legendre. O Símbolo de Jacobi é o produto dos Símbolos de Legendre onde p_i é cada um dos fatores primos de n, assim: $\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right).\left(\frac{a}{p_2}\right). \ldots .\left(\frac{a}{p_n}\right)$.

Como Euler provou que para todo a seu critério é válido, se testarmos todos os valores de 1 a $n-1$, e o critério for satisfeito, n é primo. Isto é uma vantagem sobre o Teste de Fermat, pois no Teste de Solovay-Strassen não existem números como os de Carmichael, e caso todos os valores possíveis para a sejam testados, o teste passa a ser determinístico. Testar todos os valores, no entanto levaria a um custo exponencial, dependente do tamanho de n, o que não é desejável.

No algoritmo de Solovay-Strassen, ao escolher aleatoriamente um valor para a, obtêm-se dois possíveis resultados: O número é composto com certeza, ou o número é provavelmente primo. No caso de ser provavelmente primo, a chance de o número ser composto é de no máximo ½, como está demonstrado em [21]. Executando-se estas operações k vezes, se após o k-ésimo teste a resposta do algoritmo ainda for provavelmente primo, a chance de o algoritmo estar fornecendo um

pseudoprimo é de $\frac{1}{2^k}$. Logo, testando-se 100 vezes, a probabilidade de erro é desprezível, e o provável primo pode ser usado para quase todas as aplicações.

Outra inovação deste algoritmo é que ele foi o primeiro a fornecer uma resposta com baixa probabilidade de erro para qualquer inteiro com uma execução suficientemente rápido. Seu custo computacional é de $O(k\log^3 n)$ [22], sendo k a quantidade de vezes que o teste será executado. Apesar de ter sido revolucionário em seu surgimento, o Teste de Solovay-Strassen foi superado poucos anos depois pelo Teste de Miller-Rabin, que será descrito na próxima seção.

3.6. Teste de Miller-Rabin

O Teste de Miller-Rabin é um método probabilístico de tempo polinomial para determinar a primalidade de um número. Assim como o teste de Solovay-Strassen, este é um teste de Monte Carlo. Sua principal vantagem sobre Solovay-Strassen é que sua probabilidade de erro é consideravelmente menor. Logo, este algoritmo exige um número k de repetições menor do que o teste de Solovay-Strassen para fornecer uma resposta com a mesma confiança no resultado.

Este método foi proposto por Rabin [23], como uma modificação de outro teste lançado poucos anos antes por Miller [24]. Miller propôs um método polinomial para decidir primalidade, e determinístico caso a Hipótese Estendida de Riemann seja verdadeira. A alteração de Rabin foi tornar o algoritmo probabilístico, demonstrando que a probabilidade de erro a cada execução do teste é de no máximo uma em quatro, apesar de que na média, este número é bem menor [25].

O algoritmo consiste em, dado um inteiro ímpar n a ser testado, primeiramente escreve-se $n-1$ na forma de $2^s.d$, sendo s um inteiro qualquer e d um inteiro ímpar. Por exemplo, para $n = 345$, $n-1$ seria reescrito como $2^3 . 43 = 344$. Com estes dois valores s e d, escolhe-se um inteiro $a < n$ aleatoriamente. Executam-se então os seguintes testes:

$$a^d \equiv 1 \pmod{n} \qquad \text{Equação 3.6}$$

e para i variando de 0 a $s-1$,

$$a^{2^i d} \equiv -1 \pmod{n} \qquad \text{Equação 3.7}$$

Caso qualquer um destes testes seja verdadeiro, n é declarado pseudoprimo. Caso todos os testes sejam executados e as igualdades nunca forem satisfeitas, o algoritmo retorna com certeza que n é composto.

O teste de Miller-Rabin possui algumas vantagens sobre o método de Solovay-Strassen. Além da menor probabilidade de erro, seu custo também é menor, dado que não existe o cálculo do Símbolo de Jacobi neste algoritmo. Usando Transformada Rápida de Fourier para realizar as multiplicações, seu custo é de $O(k \,.\log^2 n)$.

Outra vantagem que encerra qualquer possibilidade de utilização do teste de Solovay-Strassen é a demonstração de que sempre que o algoritmo de Solovay-Strassen retorna COMPOSTO, o teste de Miller-Rabin irá fornecer este mesmo resultado, logo, sempre que Solovay-Strassen está certo, Miller-Rabin também estará [26].

CAPÍTULO 4

Testes de primalidade (1980 – 2002)

Após a publicação do algoritmo RSA, e do surgimento de outras aplicações para números primos no contexto da criptografia, a questão de aprimorar os testes de primalidade passou a ter uma relevância prática. Iniciou-se uma pesquisa por novos algoritmos, mais rápidos tanto na teoria como na prática. Observa-se nesta fase o uso de uma matemática mais avançada, e testes probabilísticos com chances de erro cada vez menores.

4.1. Teste de Baillie-PSW (B-PSW)

Este teste, proposto por Baillie [27] e melhorado por Pomerance et al. [28], é simplesmente a proposta de executar dois testes probabilísticos de primalidade em sequência. Primeiramente, é usado um teste de primalidade probabilístico forte. O teste usado na proposta em [28] é o teste de Miller-Rabin, com a base $a = 2$. Caso o número sendo testado passe neste teste, é realizado um teste forte de Lucas, que em resumo, utiliza o Símbolo de Jacobi e Sequências de Lucas para verificar a primalidade de um dado número. Se o número n passar para este teste também, declara-se que n é provável primo.

A maior importância deste teste é que até a escrita deste trabalho, nenhum contra-exemplo, ou seja, um n que passe neste teste e seja na verdade composto, foi encontrado. Os autores inclusive oferecem um prêmio, simbólico (US$ 620), para quem fornecer um contra-exemplo ou uma prova de que tal número não existe. Pomerance [29] apresentou um argumento heurístico, mostrando que podem existir infinitos contra-exemplos para o teste B-PSW. Ninguém, entretanto, conseguiu até o momento encontrar uma prova formal de tal fato [13].

A probabilidade de erro deste teste é, portanto, desconhecida, mas é sabido que não existem pseudoprimos para o teste B-PSW até 10^{17}. Marcel Martin, autor de um software de teste de primalidade que utiliza o algoritmo B-PSW, estima que não existam números compostos com menos que 10.000 dígitos capazes de “enganar” o teste de Baillie-PSW [30]. Entre os programas que utilizam

implementações deste algoritmo, destacam-se o Maple, que possui um método baseado neste teste [13], e o Primo, que utiliza este teste como passo intermediário.

Quanto ao custo computacional, este algoritmo apresenta o mesmo custo de Miller-Rabin, de $O(\log^2 n)$. Sua constante no entanto é mais alta, e nos piores casos, este teste pode ser 4 vezes mais lento que o teste de Miller-Rabin.

4.2. Teste de Adleman-Pomerance-Rumely (APR)

O teste APR é um marco na história dos testes de primalidade. Ele foi o primeiro teste a realmente necessitar de tópicos avançados da teoria dos números, na sua definição e na prova de correção [13]. Seu custo não é polinomial, mas é baixo o suficiente para na prática, ser um dos mais rápidos entre os testes determinísticos [13].

Devido a sua complexidade, este trabalho não irá entrar em detalhes sobre este teste. Recomenda-se a leitura de [31] para aqueles com maturidade matemática suficiente para compreender este método. Cohen e Lenstra [32] demonstraram uma versão mais simples deste algoritmo.

O APR é um teste determinístico, não dependente de nenhuma hipótese ainda não provada como a Hipótese Estendida de Riemann. Apesar de ser possível provar sua correção utilizando esta hipótese, este teste, ao contrário do teste determinístico de Miller, consegue ser provado sem o uso de tal hipótese. Um problema deste método é que este não gera um certificado de primalidade, logo, para a verificação de um resultado, é necessário executar todo o algoritmo novamente.

O custo deste algoritmo, subexponencial na ordem de $O((\log n)^{c \log\log\log n})$ [31], é considerado como "quase polinomial". Este custo cresce assintoticamente de maneira lenta, se assemelhando a um polinômio. Para fins práticos, é um valor extremamente interessante, mas estritamente este ainda não é um teste polinomial, logo, a questão de se PRIMO está em P se manteve aberta até a publicação do algoritmo AKS.

4.3. Elliptic Curve Primality Proving (ECPP)

Durante a década de 1980, existia uma linha de pesquisa forte sobre métodos de teste de primalidade que fizessem uso de matemática avançada. Esta linha era motivada pelo pensamento de que a resolução deste problema não viria de conceitos simples, visto que várias tentativas foram realizadas nos últimos 2.300 anos. Os algoritmos APR e ECPP são exemplos dos resultados desta linha de pensamento.

O teste ECPP, como o próprio nome já diz, utiliza Curvas Elípticas para efetuar o teste de primalidade. Em 1985, Lenstra iniciou o uso destas estruturas para o problema da fatoração [13]. Em 1986, Goldwasser e Killian [33] publicaram um teste de primalidade baseado em Curvas Elípticas, cuja prova é dependente da conjectura de Cramer e da Hipótese Estendida de Riemann. Em 1986, de maneira independente, Atkin demonstrou um algoritmo determinístico, que usa Curvas Elípticas para decidir se o número é ou não primo, sem dependências de conjecturas em sua prova [34]. Uma característica importante deste teste é que é gerado um certificado de primalidade. Logo, o resultado pode ser verificado com eficiência [13].

Não é conhecido com exatidão o custo computacional deste método. Conjectura-se que o custo deste algoritmo é da ordem de $O(\log^5 n)$ [35], e com algumas melhorias propostas no mesmo artigo, este tempo pode ser diminuído para $O(\log^4 n)$.

4.4. Considerações Finais

Todos os algoritmos vistos até o momento têm algum tipo de relevância, seja ela histórica ou prática. Embora nem todos sejam vastamente usados na prática, como Solovay-Strassen ou mesmo o Crivo de Eratóstenes, a maioria dos softwares que necessitam verificar primalidade utilizam algum dos outros algoritmos citados previamente neste trabalho, por vezes utilizando mais de um. É uma prática comum, por exemplo, utilizar Divisão por Tentativa com alguns primos pequenos antes de executar qualquer outro teste [13].

A prática de utilizar outros testes como etapas para um novo algoritmo é comum. Alguns dos algoritmos mostrados neste trabalho, como Baillie-PSW, são exemplos da utilização em sequência de diferentes testes, ou ao menos da reutilização de outro teste, comumente o Teste de Primalidade de Fermat.

Os métodos vistos até o momento foram se tornando mais difíceis de entender e provar de acordo com a cronologia de seu surgimento. É obviamente mais difícil entender ECPP do que o Crivo de Eratóstenes, ou mesmo o Teste de Fermat. A tendência aparentava que os pesquisadores haviam desistido de testes mais simples, buscando em elementos complexos da matemática a resposta para definir se PRIMO está ou não em P. Contudo, o algoritmo AKS, apresentado em detalhes no próximo capítulo, retoma a simplicidade para os testes de primalidade.

CAPÍTULO 5
O Algoritmo AKS

Este capítulo irá focar em descrever o algoritmo AKS, o primeiro teste de primalidade a ser provado determinístico e de tempo polinomial, inserindo o problema PRIMO na classe P.

5.1. Histórico

O nome AKS vem das iniciais de seus três autores, os indianos Manindra Agrawal, Neeraj Kayal e Nitin Saxena. Agrawal é doutor em Ciência da Computação, na área de Complexidade Computacional, e professor do Indian Institute of Technology Kanpur (IITK). Kayal e Saxena haviam participado em 1997 do time indiano na Olimpíada Internacional de Matemática e haviam acabado de concluir o curso de Ciência da Computação no IITK quando publicaram o algoritmo [36].

O algoritmo AKS se aproveita do trabalho de conclusão de curso de Kayal e Saxena [37], cujo título, *Towards a determinstc polynomial-time Primality Test*, já indicava que eles vislumbravam o que estava por vir. Este trabalho mostra a correção do algoritmo proposto por Agrawal e Biswas em [38]. Este algoritmo é probabilístico, porém diferente dos outros testes citados anteriormente, pois sempre acerta caso a resposta seja que o número é primo. Se a saída determinar que o número é composto, o teste tem 1/3 de chance de estar errado. O algoritmo de Agrawal e Biswas é baseado no Pequeno Teorema de Fermat, e foi utilizado como base para a idéia do AKS.

A importância do resultado mostrado por Agrawal, Kayal e Saxena [4] foi comprovada pela rapidez com que a comunidade de pesquisadores em Teoria dos Números revisou, comentou e melhorou o algoritmo AKS. Bornemann [36] menciona que poucos dias depois da distribuição[1] do artigo seminal que descreve o algoritmo, vários especialistas como Pomerance, Lenstra, Goldwasser e Bernstein já haviam se pronunciado sobre o artigo, ou mesmo proposto simplificações para a prova. O Capítulo 4 de [39] mostra a cronologia das principais contribuições mostradas até 4 meses após o AKS se tornar público, e que conseguiram um ganho na ordem de 2.10^6 em relação ao algoritmo originalmente proposto.

Uma prova da relevância prática do AKS foi a submissão de uma proposta de patente ao Escritório Americano de Patentes em 2003, realizada pelos autores. Esta patente foi concedida em

[1] É usado o termo distribuição, pois inicialmente o artigo não foi publicado, mas apenas distribuído em forma eletrônica para especialistas e posteriormente disponibilizado online.

2008. No documento da patente [40] com a descrição do algoritmo, além do detalhamento realizado de maneira similar ao artigo [4], os autores referenciaram as aplicações de testes de primalidade no contexto de criptografia e apresentaram como algumas operações devem ser realizadas para alcançar custos mais baixos. Além disto, propuseram um gerador de números primos baseado no teste AKS.

5.2. Visão Geral

O algoritmo AKS é simples e não faz uso de nenhum conceito avançado de matemática, podendo ser executado manualmente para valores pequenos.

Cada etapa do AKS elimina números compostos com alguma característica em comum. Desta forma, apenas números compostos de uma determinada forma são capazes de chegar à última etapa, que por sua vez, consegue identificar definitivamente se o número é primo ou composto.

O passo principal do AKS consiste em um teste baseado na seguinte generalização do Pequeno Teorema de Fermat:

Para $a \in \mathbb{Z}, n \in \mathbb{N}, n \geq 2, (a, n) = 1$, n é primo se somente se:

$$(x + a)^n = x^n + a \pmod{n} \qquad \text{Equação 5.1}$$

Para a prova formal desta generalização, consultar o Capítulo 2 de [4]. Esta prova se baseia no fato de que, no triângulo de Pascal, caso n seja primo, a n-ésima linha do triângulo irá conter apenas o número 1 e múltiplos de n. Se n for composto, obrigatoriamente a n-ésima linha irá conter algum valor não-múltiplo de n [41]. Isto faz com que, ao expandir o polinômio do lado esquerdo da Equação 5.1, se n for primo, apenas os coeficientes de x^n e x^0 não serão zerados ao ser aplicado o módulo em n. Logo, a Equação 5.1 será válida sempre que n for primo. Para n composto, como algum dos fatores $\binom{n}{i}$ não será divisível por n, algum dos coeficientes não será zerado e a igualdade da Equação 5.1 não será verdadeira.

Observando o lado esquerdo da equação, percebe-se que o custo de realizar a potenciação em n é muito alto, tanto em termos de tempo para calcular, como em espaço para armazenar os coeficientes. Para resolver este problema, ambos os lados são calculados módulo um polinômio na forma $x^r - 1$, com r suficientemente pequeno. O formato deste polinômio permite uma computação muito rápida da operação de módulo. Para mais detalhes, consulte Implementação do AKS, na Seção 6.1 deste trabalho. A Equação 5.2 é escrita, então, da seguinte forma:

$$(x+a)^n = x^n + a \pmod{x^r - 1,\quad n)} \qquad \text{Equação 5.2}$$

O previamente citado algoritmo probabilístico proposto por Agrawal e Biswas utiliza uma idéia similar. Pode-se afirmar grosseiramente que o teste proposto em [38] fixa o valor de a e escolhe aleatoriamente o valor de r. A mudança proposta em [4], de calcular um valor fixo para r e variar a, sendo r cuidadosamente escolhido de forma a crescer polinomialmente com a entrada n, possibilitou que o AKS seja determinístico e polinomial.

O cálculo módulo $(x^r - 1)$, no entanto, traz a possibilidade de que números compostos passem neste teste. Agrawal et al. [4] mostra que, calculando-se o valor de r da maneira descrita no artigo (este cálculo será descrito a seguir), caso a equação seja verdadeira para todos os valores de a tal que $a \geq 1$ e $a \leq \lfloor 2\sqrt{r-1}\log n \rfloor$, n será uma potência prima, ou seja, $n = p^k$, com p primo e $k > 0$ inteiro.

Para o cálculo de r, é necessário o conceito de ordem de a módulo r, para $r \in \mathbb{N}$ e $a \in \mathbb{Z}$ e $\mathrm{mdc}(a,r) = 1$. A ordem de a módulo r, ou simplesmente $o_r(a)$, é o menor valor k tal que $a^k \equiv 1 \pmod{r}$. A prova de que $o_r(a)$ existe para todo a e r tal que $\mathrm{mdc}(a,r) = 1$ é simples: os valores a^k podem assumir os valores $0 \leq k \leq r-1$, logo, existem valores s e t tais que $a^s \equiv a^t \pmod{r}$. Sendo a e r coprimos, $a^{s-t} \equiv 1 \pmod{\mathrm{r}}$.

Um fato importante para o cálculo da complexidade computacional do AKS é a existência de um valor de $r \leq O(\log^6 n)$. Agrawal et al. [4] afirmam que a prova da existência de r sob esta restrição se encontra em [42], e tais provas não se encontram no escopo deste trabalho e por esta razão não serão detalhadas aqui.

5.3. O Algoritmo AKS

Nesta seção, apresenta-se o pseudocódigo do Algoritmo AKS originalmente proposto, sendo detalhados o funcionamento e o custo de cada etapa.

```
Entrada: inteiro n > 1

1. Se n é uma potência perfeita a^b com b > 1
       retorne COMPOSTO
2. Faça r := 2
3. Enquanto r < n
4.     Se (mdc(r,n) != 1)
           retorne COMPOSTO
5.     Se r é primo:
6.         Faça q := maior fator primo de (r-1)
7.         Se (q > 4 * sqrt(r) * log n) e
               n^((r-1)/q) != 1 (mod r))
8.             break
9.     r++
10. Se n=r
       retorne PRIMO
11. De a := 1 até 2 * sqrt(r) * log n
12.    Se (x - a)^n != (x^n – a) (mod (x^r – 1), n)
           retorne COMPOSTO
13. retorne PRIMO
```

Figura 1 - O Algoritmo AKS original

O primeiro passo é a verificação de que o número sendo testado não é uma raiz perfeita. Um número inteiro é uma raiz perfeita se puder ser escrito na forma a^b para a inteiro e $b > 1$. Obviamente, caso o número possua esta característica ele é composto, logo, o teste retorna COMPOSTO. Esta verificação é realizada com o objetivo de filtrar as potências primas, de forma que apenas aquelas com $b = 1$ possam chegar ao passo 11.

O valor ideal para r é calculado pelos Passos de 2 a 9. Como queremos encontrar o menor r possível, iniciamos $r = 2$ no Passo 2. O Passo 3 inicia a iteração que encontrará o valor de r até n. O Passo 4 verifica se o máximo divisor comum entre r e n é diferente de 1. Caso afirmativo, obviamente $\mathrm{mdc}(r, n)$ divide n, logo, o algoritmo retorna COMPOSTO. Caso contrário, o algoritmo segue para o Passo 5. Este passo necessita verificar a primalidade de um número, e o mesmo é válido para o Passo 6. Observe que, caso sejam usados algoritmos probabilísticos nestes passos, todo o teste AKS passa a ser probabilístico. O uso de algoritmos probabilísticos, no entanto, não é

necessário. Podemos usar um método por força-bruta como a Divisão por Tentativa para esta verificação, pois $r \leq O(\log^6 n)$, logo, assintoticamente $r \ll n$.

A verificação de que o r escolhido satisfaz as condições necessárias para que seja válido ocorre no Passo 7. O maior fator primo de (r-1) deve ser um $q \geq 4\sqrt{r}\log n$ e $o_r(n) = r - 1,\ q \mid o_r(n)$. Caso ambas as condições sejam verdadeiras, o valor de r foi encontrado, e o Passo 8 sai do loop. Caso contrário, o algoritmo segue sua execução, chegando ao Passo 9, onde r é incrementado, e volta ao Passo 3 para continuar a iteração.

O Passo 10, que não consta no artigo original, garante que o algoritmo retorne corretamente seu resultado para valores $n \leq 11807$. Para tais valores, um $r < n$ não é encontrado, e para todos os valores de r testados, $\mathrm{mdc}(r, n) = 1$, logo n é primo.

A etapa final inicia-se no Passo 11. Através de uma iteração, será buscado algum $1 \leq a \leq 2\sqrt{r}\log n$ que não satisfaça a Equação 5.2 no Passo 12. Se este valor for encontrado, obviamente n é COMPOSTO e o algoritmo para. Caso todos os valores possíveis para a sejam testados e todos satisfaçam o Passo 12, o algoritmo segue, retornando PRIMO no Passo 13. Observe que, como já foi dito, para a Equação 5.2 ser satisfeita, n deve ser uma potencia prima a^b, e como todas as potencias primas com $b > 1$ foram excluídas no Passo 1, restam apenas os $a^1 = a$, logo, a é primo.

O custo computacional demonstrado por Agrawal et al. [4] é de $\tilde{O}(\log^{12} n)$, mostrando que o passo do teste para verificar que n não é uma potência perfeita tem um custo de $O(\log^3 n)$, no entanto, Bernstein [43] apresenta um algoritmo para esta finalidade possui um custo de $O(\log^{1+o(1)} n)$.

O Passo 3 pode deixar a falsa impressão de que é executado n vezes, devido à condição $r < n$. O fato previamente citado de que $r \leq O(\log^6 n)$, no entanto, nos fornece a garantia de que os passos dentro deste loop sejam executados no máximo apenas $c \,.\log^6 n$, onde c é uma constante não dependente de n, oculta na notação Big-O, e que pode ser desprezada nos cálculos de complexidade.

O cálculo de $\mathrm{mdc}(r, n)$, pode ser realizado em $\tilde{O}(\log\log r)$ [4]. Utilizando força bruta para decidir se o r sendo testado é primo, o custo do Passo 5 é $\tilde{O}(\sqrt{r})$. Encontrar o maior fator primo de r também possui este custo utilizando um algoritmo de força bruta. Os Passos 7, 8 e 9 possuem custos baixos ($\tilde{O}(\log\log n)$) quando comparados ao do Passo 5 ou 6, logo, o custo do loop inteiro é $\tilde{O}(\log^6 n \,.\sqrt{r})$. Como $\sqrt{r} \leq c.\log^3 n$, o custo deste loop é $\tilde{O}(\log^9 n)$.

O Passo 11 é a etapa mais custosa do algoritmo. Como fica claro no algoritmo, este loop se repete $2.\sqrt{r}.\log n$ vezes. Caso seja utilizada potenciação binária para a exponenciação e Transformada Rápida de Fourier para calcular as multiplicações, cada iteração tem o custo de $\tilde{O}(\log n.r\log n)$. Desta forma, novamente utilizando $\sqrt{r} \leq c.\log^3 n$, o loop se repete $O(\log^4 n)$, tendo cada iteração o custo de $\tilde{O}(\log^8 n)$. Assim, este loop tem um custo de $\tilde{O}(\log^{12} n)$, que por dominar os custos anteriormente citados, é o custo de todo o algoritmo.

Desta forma, está demonstrado que o algoritmo AKS retorna uma resposta correta e em tempo polinomial. O seu principal problema é que tanto o seu expoente, visível na notação Big-O, como sua constante não mostrada nesta notação, são muito altos. Enquanto para números grandes, testes probabilísticos como Miller-Rabin conseguem fornecer uma resposta com uma probabilidade de erro baixíssima em segundos, o teste AKS pode levar horas, ou até mesmo dias para fornecer a resposta para o mesmo número [44].

O principal desafio nas melhorias de AKS, portanto, é diminuir o expoente de seu custo. Este objetivo foi alcançado até o momento limitando-se o valor máximo de r, logo, diminuindo o custo da fase final do algoritmo, e do próprio cálculo de r. As melhorias em AKS que serão demonstradas a seguir já conseguiram baixar o custo do algoritmo para $\tilde{O}(\log^6 n)$.

Outros algoritmos também foram desenvolvidos utilizando AKS em algum dos seus passos. Um deles, proposto em [45], utiliza ECPP e AKS de uma maneira que atinge um custo calculado heuristicamente em $\tilde{O}(\log^4 n)$, por não ser conhecido até o momento uma maneira de avaliá-lo rigorosamente.

5.4. Melhorias e testes baseados no algoritmo AKS

O alto custo do AKS original, juntamente com sua importância teórica por estar em P, motivou vários pesquisadores da área em torno do problema de reduzir o custo do AKS proposto inicialmente. Além disto, foram desenvolvidas novas maneiras de determinar a primalidade de um número se baseando nas técnicas desenvolvidas para este algoritmo.

5.4.1. A última versão do AKS

Uma das primeiras melhorias foi proposta por Lenstra, em um artigo não publicado, cujo conteúdo é utilizado e detalhado em [5] e [39]. O avanço trazido por esta modificação é significativo, visto que além de conseguir demonstrar através de uma prova simples um limite de $\tilde{O}(\log^{10.5} n)$, com um pouco de matemática avançada é capaz de mostrar que AKS tem o custo de $\tilde{O}(\log^{7.5} n)$.

Esta melhoria de Lenstra consiste em mudar o cálculo do valor de r, de forma que é fácil mostrar que um r válido existe tal que $r = \tilde{O}(\log^5 n)$, enquanto na versão original, $r = \tilde{O}(\log^6 n)$. Este valor de r também é mais fácil de ser calculado, não exigindo que r seja primo ou que a fatoração prima de $r - 1$ seja conhecida. Na Figura 2, apresenta-se o algoritmo AKS proposto na 6ª versão do artigo de Agrawal et al. [5], que utiliza esta melhoria.

```
Entrada: inteiro n > 1

1.    Se n é uma potência perfeita a^b com b > 1
          retorne COMPOSTO
2.    Faça r := O menor valor r tal que o_r(n) > log^2 n
3.    De a := 1 até r
4.         Se mdc(a,n) > 1 E mdc(a,n) < n
               retorne COMPOSTO
5.    Se n <= r
          retorne COMPOSTO
6.    De a := 1 até sqrt(φ(r)) * log n
7.         Se (x - a)^n != (x^n - a) (mod (x^r - 1), n)
               retorne COMPOSTO
8.    retorne PRIMO
```

Figura 2 - O Algoritmo AKSv6

A definição do valor de r no Passo 2 necessita do cálculo de $o_r(n)$, de forma a encontrar o valor q da maneira que o teste necessita. Este cálculo pode ser realizado em tempo $\tilde{O}(\log^7 n)$, através do algoritmo da Figura 3.

```
Entrada: inteiro n > 1

1.    De r := 2 até n
2.         De k := log^2 n até r-1
3.              Se n^k = 1 (mod r)
                     retorne k e r
```

Figura 3 - Algoritmo para calcular um valor q apropriado

Como já foi dito, Agrawal et al. [5] prova que existe um valor r em até $\tilde{O}(\log^5 n)$. O Passo 2 executa $O(\log^2 n)$, logo, o custo deste algoritmo será, como dito anteriormente, de $\tilde{O}(\log^7 n)$.

Os Passos 3, 4 e 5 da Figura 2 são triviais, e seus custos já foram explicados ao descrever o método original, logo, voltaremos nossa atenção para o Passo 6, que inicia a última e novamente mais custosa parte do teste. Este passo está limitado agora por $\sqrt{\varphi(r)}\,.\log n$, que é menor do que o $2.\sqrt{r}\log n$ usado no paper original. Assintoticamente, no entanto, não há ganho, visto que, para r primo, $\varphi(r) = r - 1$. Observando que o Passo 7 do algoritmo na Figura 2 é idêntico ao Passo 12 da Figura 1, pode-se afirmar então que o passo final do algoritmo da Figura 2 possui ganho apenas pois o valor máximo de r agora é menor. Cada iteração continua tendo o custo de $\tilde{O}(\log n\,.\,r\log n)$, mas agora, com $r = \tilde{O}(\log^5 n)$, cada iteração tem o custo de $\tilde{O}(\log^7 n)$. O loop continua se repetindo $O(\sqrt{r}\log n)$, mas com o novo valor de r, se repete $O(\log^{3.5} n)$, desta forma, o custo do algoritmo passa a ser $\tilde{O}(\log^{10.5} n)$.

Com o mesmo algoritmo da Figura 2, Agrawal et al. mostram em [5] que, através do uso de uma demonstração de Fouvry [42] valida para o intervalo em que o teste AKS opera, o valor máximo de r é $O(\log^3 n)$. Uma pequena manipulação algébrica nos cálculos realizados anteriormente mostra que o custo deste algoritmo então é $\tilde{O}(\log^{7.5} n)$.

5.4.2. Lenstra-Pomerance

A melhoria proposta por Lenstra em conjunto com Pomerance em [46] se baseia na troca do polinômio $(x^r - 1)$ utilizado para as operações de módulo realizadas no Passo 12 do algoritmo AKS, por outro polinômio cuidadosamente escolhido, que permite um custo de $\tilde{O}(\log^6 n)$ para qualquer inteiro n testado.

As restrições para que este polinômio possa ser utilizado neste algoritmo, assim como os passos necessários para encontrá-lo estão descritos em [46], mas como os próprios autores mencionam, o teste não é trivial e exige o conhecimento de assuntos como períodos Gaussianos e diversos teoremas sobre a distribuição dos números primos, vários deles fazendo uso da função zeta de Riemann.

O esboço do algoritmo em si é similar ao AKS, sendo o cálculo do valor r substituído pelas etapas que escolhem o polinômio a ser utilizado. Estes passos possuem custo menor do que os utilizados no AKS original e na versão atualizada de AKS para encontrar o valor de r (a saber, $\tilde{O}(\log^9 n)$ e $\tilde{O}(\log^7 n)$ respectivamente). De acordo com [46], este custo é $O\left(\log^{\frac{24}{11}} n\right) + \tilde{O}(\log^3 n) + \tilde{O}\left(\log^{\frac{21}{5}} n\right) = \tilde{O}(\log^{4.2} n)$.

Tendo este polinômio, digamos $f(x)$, em mãos, a Equação 5.2 é modificada para ser escrita da seguinte forma:

$$(x+a)^n = x^n + a \pmod{f(x), n} \qquad \text{Equação 5.3}$$

Lenstra e Pomerance mostram que $f(x)$ tem grau $d = \mathrm{O}(\log^2 n)$, e que o Passo 12 de AKS original deve ser verificado utilizando a Equação 5.3 de 1 até $\sqrt{d}.\log n$. Assim como nas versões anteriores, mas substituindo r por d, o custo de cada iteração do ultimo loop é de $\tilde{\mathrm{O}}(\log n \,.\, d \log n)$, logo, o custo do loop é $\tilde{\mathrm{O}}(d^{\frac{3}{2}}.\log^3 n)$. Substituindo d por $\mathrm{O}(\log^2 n)$, chegamos ao custo de $\tilde{\mathrm{O}}(\log^6 n)$.

5.4.3. Berrizbeitia

Assim como a modificação proposta por Lenstra e Pomerance, esta melhoria proposta por Berrizbeitia em [47] altera o polinômio sobre o qual é efetuado o módulo no Passo 12 do algoritmo original. Esta modificação, no entanto, é dependente de algumas propriedades da entrada para poder funcionar.

De maneira similar ao algoritmo de Lucas-Lehmer, que funciona apenas para números de Mersenne, o algoritmo de Berrizbeitia necessita de uma restrição em sua entrada, no caso, este teste precisa que a entrada n seja de alguma das seguintes formas[2]:

- $n \equiv 1 \pmod 4$ para os quais existe um a tal que $\left(\frac{a}{n}\right) = -1$
- $n \equiv -1 \pmod 4$ para os quais existe um a tal que $\left(\frac{a}{n}\right) = \left(\frac{1-a}{n}\right) = -1$

O custo deste algoritmo é pior do que o custo da melhoria de Lenstra e Pomerance, mas para algumas entradas n com $n \equiv 1 \pmod 4$ e o valor $n-1$, onde algum dos fatores seja uma potência de 2 maior que $2.\log\log n$, ou o $n \equiv -1 \pmod 4$ e o valor $n+1$ possuindo um fator que seja uma potência de 2 maior que $2.\log\log n$, este algoritmo possui um custo de $\tilde{\mathrm{O}}(\log^4 n)$.

A melhoria realizada por este algoritmo é a troca da Equação 5.2 pela seguinte:

$$(mx+1)^n = mx^n + 1 \pmod{x^{2^s} - a, n} \qquad \text{Equação 5.4}$$

Nesta equação, a é um valor que torna $\left(\frac{a}{n}\right) = -1$ ou $\left(\frac{a}{n}\right) = \left(\frac{1-a}{n}\right) = -1$, de acordo com qual das condições faz com que n possa ser verificado por este algoritmo, $s = 2\log\log n$ e m varia de 1 a $2^{\max(s-k,0)}$, onde k é a maior potência de 2 que pode dividir $n-1$ ou $n+1$, novamente de acordo com a condição que faz n ser uma entrada válida para este algoritmo.

A prova de correção e os cálculos da complexidade deste algoritmo se encontram em [47].

[2] Para este método, $\left(\frac{a}{n}\right)$ é o valor do Símbolo de Jacobi, previamente descrito na seção Teste de Solovay-Strassen, na página 13 deste trabalho

5.4.4. ECPP + AKS

Este algoritmo proposto por Cheng [45], dependendo do número dado como entrada executa uma versão modificada do AKS e partes do algoritmo ECPP, além de algum teste probabilístico que sempre esteja certo ao retornar COMPOSTO.

Como já foi citado, o método ECPP não possui até o momento um cálculo formal de sua complexidade, tendo sido este custo heuristicamente determinado como $\tilde{O}(\log^4 n)$ utilizando as melhorias em [35]. A versão do AKS utilizada neste algoritmo se baseia em uma melhoria sobre a proposta de Berrizbeitia [47], de forma que mais números possam ser verificados através deste teste. Apenas os valores de n que o algoritmo de Berrizbeitia pode determinar a primalidade em $\tilde{O}(\log^4 n)$ são testados utilizando o mesmo.

Este algoritmo, obviamente, executa heuristicamente em tempo $\tilde{O}(\log^4 n)$. Ele é capaz de determinar a primalidade da entrada, mas não pode ser considerado deterministicamente polinomial devido ao uso do ECPP. Sua maior importância, entretanto, é o uso de passos do AKS em conjunto a outros testes, de uma maneira que é tão diferente do AKS original que pode ser considerado um algoritmo a parte, como se pode dizer sobre o AKS em relação ao teste de Fermat por exemplo.

O funcionamento deste algoritmo, de maneira resumida, consiste em verificar se a entrada n é uma potência perfeita. Se o número passar por este teste, é executado um algoritmo probabilístico como Miller-Rabin. Se a entrada ainda não foi detectada como composto, verifica-se se n possui algum fator primo entre $\log^2 n$ e $2\log^2 n$ e executa-se a operação de redução do algoritmo ECPP [34] caso esta condição não seja verdadeira, substituindo n pelo valor retornado por esta parte do ECPP. Executa-se então, com o valor atual de n, uma versão modificada do algoritmo AKS melhorado por Berrizbeitia. Caso não seja retornado composto durante esta execução e o ECPP não tiver sido executado, retorna-se primo. Caso não seja retornado composto, mas o passo do ECPP foi executado anteriormente, executa-se o final do método ECPP para se obter a prova de que n é primo e é retornado primo. É importante observar que sempre que o algoritmo retorna composto, apresenta-se alguma testemunha que facilita a verificação. Quando se retorna primo através da modificação de AKS, esta prova não é fornecida (nem necessária, como já foi discutido anteriormente).

É interessante citar que de maneira similar ao trabalho de Cheng neste algoritmo, Bernstein [48] também desenvolveu um método baseado nas modificações de Berrizbeitia e que também roda em tempo $\tilde{O}(\log^4 n)$, mas funciona para qualquer entrada n, no entanto, este teste de Bernstein é aleatório (classe RP, aleatoriamente polinomial), e possui ½ de probabilidade de retornar um resultado [49]. Assim como Miller-Rabin ou Solovay-Strassen, cada execução do método de Bernstein

é independente, logo, a probabilidade deste teste não retornar uma resposta após 100 execuções é $\frac{1}{2^{100}}$, uma probabilidade completamente desprezível. Vale ressaltar que quando o algoritmo de Bernstein retorna uma resposta, esta é correta.

CAPÍTULO 6
Ferramenta de Comparação

Para concluir os estudos sobre o Algoritmo AKS, foi implementada uma ferramenta para comparações, usando diversos dos algoritmos citados neste trabalho. Esta ferramenta possui uma finalidade didática, de maneira que seja possível expor às pessoas sem afinidade com o tema Testes de Primalidade, as diferenças nos principais algoritmos que foram utilizados ao longo da história. É notável, como já foi citado no Capítulo 3 ao explicar o método de Divisão por Tentativa, que muitas pessoas acreditam que a única maneira de testar a primalidade de um número n é o método da força bruta, e outras ainda acreditam, a partir de uma compreensão incorreta de complexidade computacional, que este método de força bruta é rápido e possui custo $O(n)$. Desta forma, uma ferramenta que ajude a visualizar os tempos de execução dos diversos algoritmos é de grande valor para elucidar estes pontos, e pode ser um facilitador para professores ao abordar esta área.

O software foi implementado sobre o .NET Framework, utilizando a linguagem C#. Os principais motivos para esta escolha foram a familiaridade do autor deste trabalho com o ambiente e a linguagem, além da facilidade de criação de interfaces gráficas que este ambiente de programação oferece. Até o momento, o autor desconhece uma implementação de código aberto do algoritmo AKS em C#.

Para agilizar o desenvolvimento deste software, foi utilizado como base a classe BigInteger de Chew Keong Tan, disponível livremente para uso, alterações e redistribuição em http://www.codeproject.com/KB/cs/biginteger.aspx. Esta implementação para tratamento de números muito grandes (números acima de 64 bits) foi escolhida por ter sido construída com foco em testes de primalidade, logo, ao contrário de outras classes e bibliotecas que lidam com números muito grandes como [50] e [51], esta possui prontos métodos usados com freqüência em algoritmos de teste de primalidade, como radiciação destes números e computação do Símbolo de Jacobi.

Outra vantagem desta Classe é que além dos métodos comumente usados, ela possui alguns testes de primalidade prontos. Os testes já implementados na classe BigInteger de Tan são o Teste de Primalidade de Fermat, o teste de Solovay-Strassen, o teste de Miller-Rabin e uma versão do teste Baillie-PSW. Além da refatoração destes métodos para as necessidades da ferramenta, inclusão de

métricas de tempo e adequações para os padrões de projeto utilizados na arquitetura do software, foram implementados neste trabalho os testes de Divisão por tentativa, o teste de Lucas-Lehmer e o algoritmo AKS.

A principal desvantagem desta classe é o uso de um algoritmo lento para executar multiplicações, da ordem de $O(n^2)$, enquanto as bibliotecas [50] e [51] utilizam métodos rápidos para multiplicações. Como o objetivo deste método é simplesmente realizar uma comparação, e todos os algoritmos serão executados utilizando o mesmo método de multiplicação, o uso desta classe será justo para todos os testes, não havendo interferência no resultado comparativo.

Outra desvantagem da classe BigInteger é que, por trabalhar apenas com inteiros, não existe um método capaz de calcular o logaritmo de um número representado como BigInteger, o que inviabiliza o uso desta classe para a implementação de AKS. Como o teste AKS não é eficiente para números suficientemente grandes, foi decidido usar o tipo System.Int64 do .NET Framework, que representa um inteiro de 64 bits. Ao utilizar este tipo para representar os números a serem testados em AKS, é possível utilizar o método $Math.Log()$, também parte do Framework, para calcular os logaritmos necessários no algoritmo.

Com os algoritmos prontos, foi criada uma interface gráfica (GUI) para facilitar o uso da ferramenta, e a compreensão dos dados gerados. Nesta GUI, o usuário pode observar e comparar os tempos gastos para o teste usando cada um dos algoritmos escolhidos. Esta característica é desejável para ilustrar o fato de que mesmo algoritmos lentos, como Divisão por tentativa, podem ser a melhor opção dependendo do tamanho da entrada.

6.1. Implementação do AKS

A implementação do algoritmo AKS em si é uma atividade que não envolve uma dificuldade muito grande. Para esta ferramenta, foi escolhida a versão original do algoritmo, já explicada no Capítulo 0 deste trabalho, e descrita pela Figura 1. Codificar, testar e depurar o algoritmo é uma atividade que contribui para o entendimento do teste e dos conceitos utilizados, além de facilitar a compreensão do cálculo do custo do algoritmo.

Para a verificação de que a entrada é ou não uma potência perfeita é utilizado um teste baseado no método de Newton. O código para esta verificação foi baseado na implementação escrita em C++ existente em [52], com algumas modificações para o código em C# e para lidar com a imprecisão da operação Math.Pow resultada pelo uso do tipo *double*.

O cálculo de r na implementação deste trabalho foi levemente otimizada, através do uso de uma lista de números primos até 10^6, logo, economiza-se a verificação por força bruta na busca por um r primo para vários casos. A busca pelos fatores primos de r foi realizada utilizando-se força bruta, escolha aceitável de acordo com [4]. De resto, a implementação seguiu as orientações dadas no algoritmo original.

Para o último passo do algoritmo, foi utilizada uma técnica proposta em [53] para facilitar a operação de módulo entre um polinômio $f(x)$ qualquer de grau s e o polinômio $x^r - 1$. Esta técnica consiste em, para cada i tal que $0 \leq i \leq s$, o coeficiente do termo de expoente $i \pmod r$ do polinômio referente ao resto da divisão de $f(x)$ por $x^r - 1$ será a soma dos coeficientes de todos os termos cujos expoentes sejam equivalentes a $i \pmod r$. O código da Figura 4 deixa esta técnica mais clara.

```
Entrada: Polinômios a e b, de graus grauA e grauB respectivamente.

Poly ret = new Poly(grauB);

for (int i = grauA; i >= 0; i--)
{
      ret.coeficientes[i % grauB] += a.coeficientes[i];
}

return CortarZeros(ret);
```

Figura 4 - Técnica para calcular *f(x)* mod x^r-1

Além de simples de implementar, esta técnica obviamente é mais rápida do que efetuar a divisão dos polinômios e obter seu resto, e foi usada sempre que possível neste teste.

O código da ferramenta, assim como executáveis compilados foram disponibilizados pelo autor deste trabalho em http://primalidade.codeplex.com/. O código é de fácil leitura e foi projetado de forma a utilizar alguns padrões de projeto que facilitam a inclusão de novos testes. No caso de dúvidas, sugestões ou informações de erros, é possível entrar em contato com o autor deste trabalho através do endereço previamente citado.

CAPÍTULO 7
Conclusão e Trabalhos Futuros

Como foi possível observar durante a realização desta pesquisa, o problema de decidir se um dado número é ou não primo foi alvo constante de estudos, e mesmo após o desenvolvimento de um algoritmo polinomial e determinístico para resolvê-lo, os esforços continuaram no sentido de melhorar sua eficiência e utilizar este novo algoritmo como um modelo para novos testes.

O custo ainda alto do algoritmo AKS e de suas variações faz com que na prática, os testes probabilísticos como Miller-Rabin continuem dominando as aplicações. Pesquisas continuam sendo desenvolvidas em busca de algoritmos que possam reduzir ainda mais o custo das soluções para PRIMO e talvez tornar um algoritmo determinístico e polinomial uma escolha mais atraente para fins práticos.

É válido lembrar também que além de melhorias e variações baseadas no teste AKS, outras linhas de pesquisa podem causar o surgimento de tal algoritmo, tal como a comprovação da Hipótese Estendida de Riemann, que tornaria o teste de Miller [24] polinomial e determinístico.

A Ferramenta de Comparação desenvolvida confirma a superioridade prática dos algoritmos amplamente usados em relação ao AKS, e através da visualização dos gráficos gerados pelo software, fica bem claro que mesmo tendo custo assintoticamente menor do que a Divisão por Tentativa, por exemplo, o teste AKS na prática é mais lento do que o método de força bruta para valores relativamente grandes.

Pretende-se melhorar o software desenvolvido após a conclusão deste trabalho. A inclusão do projeto da ferramenta em uma rede de projetos open-source como o CodePlex tem como objetivos tanto a disseminação do conhecimento através do código como a prospecção de possíveis colaboradores que possam melhorar a ferramenta. As principais melhorias que o autor deseja realizar na ferramenta são:

- Utilização de threads para executar os algoritmos para melhorar a usabilidade, não deixando a aparência de que a aplicação está travada, enquanto ela está na realidade executando os algoritmos;

- Desenvolvimento de versões que façam uso de processamento em paralelo de algoritmos que permitam este tipo de processamento, como o método da Divisão por Tentativa e o próprio AKS, cuja etapa final presente em todos os algoritmos baseados em AKS pode ser dividido entre vários processadores [54];
- Uso de alguma biblioteca que permita trabalhar com números não-inteiros arbitrariamente grandes, permitindo que a implementação de AKS possa testar inteiros de tamanho maior do que a implementação atual admite.

Além de melhorias na ferramenta, pretende-se estudar a combinação de testes de primalidade já conhecidos, com o objetivo de desenvolver um novo algoritmo como B-PSW ou ECPP+AKS, que possa ter alguma vantagem prática ou teórica sobre os testes atualmente conhecidos.

Bibliografia

[1] HOYRUP, J. **Lenghts, Widths and Surfaces. A portrait of Old Babylonian Algebra and Its Kin**. Boca Raton - Florida - USA: Springer-Verlag, 2001.

[2] HARDY, G. H. **A Mathematician's Apology**. Londres: Cambridge University Press, 1940.

[3] RIVEST, R.; SHAMIR, A.; ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. **Communications of the ACM**, Nova York, v. 21, n. 2, p. 120-126, Fevereiro 1978.

[4] AGRAWAL, M.; KAYAL, N.; SAXENA, N. PRIMES is in P, Kanpur, 6 Agosto 2002.

[5] AGRAWAL, M.; KAYAL, N.; SAXENA, N. PRIMES is in P, version 6, Kanpur, Agosto 2005.

[6] LOVÁSZ, L.; PELIKÁN, J.; VESZTERGOMBI, K. **Discrete Mathematics. Elementary and Beyond**. Springer, 2003.

[7] CORMEN, T. H. et al. **Algoritmos - Teoria e Prática**. 1ª. ed. Rio de Janeiro: Elsevier, 2002.

[8] KNUTH, D. E. **The Art of Computer Programming**. 1ª. ed.: Addison-Wesley, v. 1 - Fundamental Algorithms, 1969.

[9] WEISSTEIN, E. W. Lucas-Lehmer Test. **Wolfram Mathworld**. Disponivel em: <http://mathworld.wolfram.com/Lucas-LehmerTest.html>. Acesso em: 05 Maio 2008.

[10] CRANDALL, R.; POMERANCE, C. **Prime Numbers:**A Computational Perspective. 2ª. ed. Nova York: Springer, 2005.

[11] KERSTEN, R. Parallelizing the Sieve of Erathostenes using the BSP model, Utrecht, 1 Novembro 2007.

[12] ATKIN, A.; BERNSTEIN, D. Prime Sieves using Binary Quadratic Forms. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 73, p. 1023-1030, 2004.

[13] RIBENBOIM, P. **The New Book of Prime Number Records**. 3ª. ed. Nova York: Springer, 1996.

[14] ALFORD, W. R.; GRANVILLE, A.; POMERANCE, C. There are infinitely many Carmichael numbers. **The Annals of Mathematics**, v. 139, n. 3, p. 703-722, Maio 1994.

[15] PINCH, R. G. E. **On Using Carmichael Numbers for Public Key Encryption Systems**. Proceedings of the 6th IMA International Conference on Cryptography and Coding. Londres: Springer-Verlag. Dezembro 1997. p. 265-269.

[16] BRUCE, J. W. A Really Trivial Proof of the Lucas-Lehmer Test. **The American Mathematical Monthly**, v. 100, n. 4, p. 370-371, Abril 1993.

[17] COLQUITT, W. N.; WELSH JR., L. A New Mersenne Prime. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 56, n. 194, p. 867-870, Abril 1991.

[18] **Great Internet Mersenne Prime Search**. Disponivel em: <http://www.mersenne.org/>. Acesso em: 12 Novembro 2009.

[19] SOLOVAY, R.; STRASSEN, V. A Fast Monte-Carlo Test for Primality. **SIAM Journal on Computing**, v. 6, n. 1, p. 84-85, 1977.

[20] LENSTRA JR, H. W. Primality Testing Algorithms [after Adleman, Rumely and Willams]. **Séminaire Bourbaki**, Junho 1981. 243-257.

[21] ROSENBERG, B. **The Solovay-Strassen Primality Test**. University of Miami. Department of Computer Science. Miami. 1993.

[22] JAO, D. **Public Key Cryptography**. University of Waterloo - Faculty of Mathematics. Waterloo - Ontario - Canada. 2008.

[23] RABIN, M. O. Probabilistic algorithm for testing primality. **Journal of Number Theory**, v. 12, n. 1, p. 128-138, 1980.

[24] MILLER, G. L. Riemann's Hypothesis and Tests for Primality. **Journal of Computer and System Sciences**, 1976, v. 13, n. 3, p. 300-317.

[25] SCHOOF, R. Four primality testing algorithms. **Algorithmic Number Theory**, Portland, v. 44, 2006.

[26] LEMMERMEYER, F. **Introduction to Cryptography (Lecture Notes)**. Bilkent University, Faculty of Science. Ankara. 2006.

[27] BAILLIE, R.; WAGSTAFF JR., S. S. Lucas Pseudoprimes. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 35, n. 152, p. 1391-1417, Outubro 1980.

[28] POMERANCE, C.; SELFRIDGE, J. L.; WAGSTAFF JR., S. S. The Pseudoprimes to 25. 10^9. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 35, n. 151, p. 1003-1026, Julho 1980.

[29] POMERANCE, C. **Are there counter-examples to the Baillie-PSW primality test?** Amsterdam. 1984.

[30] WEISSTEIN, E. W. Baillie-PSW Primality Test. **Wolfram Math**. Disponivel em: <http://mathworld.wolfram.com/Baillie-PSWPrimalityTest.html>. Acesso em: 19 out. 2009.

[31] ADLEMAN, L. M.; POMERANCE, C.; RUMELY, R. S. On Distinguishing Prime Numbers from Composite Numbers. **The Annals of Mathematics**, v. 117, n. 1, p. 173-206, Janeiro 1983.

[32] COHEN, H.; LENSTRA JR, H. W. Primality Testing and Jacobi Sums. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 42, n. 165, p. 297-330, Janeiro 1984.

[33] GOLDWASSER, S.; KILIAN, J. Almost all primes can be quickly certified. **Annual ACM Symposium on Theory of Computing. Proceedings of the eighteenth annual ACM symposium on Theory of computing**, Berkeley - California - USA, 1986. 316-329.

[34] ATKIN, A. O. L.; MORAIN, F. Ellipctic Curves and Primality Proving. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 61, n. 203, p. 26-68, Julho 1993.

[35] MORAIN, F. Implementing the Asymptotically Fast Version of the Elliptic Curve Primality Proving Algorithm. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 76, n. 257, p. 493-505, Janeiro 2007.

[36] BORNEMANN, F. Primes is in P - Breakthrough for “Everyman”. **Notices of the AMS**, v. 50, n. 5, p. 545-552, Maio 2003.

[37] KAYAL, N.; SAXENA, N. **Towards a deterministic polynomial-time Primality Test (Trabalho de Graduação)**. Indian Institute of Technology. Kanpur. 2002.

[38] AGRAWAL, M.; BISWAS, S. Primality and Identity Testing via Chinese Remaindering. **Journal of the ACM**, Nova York, 4 Julho 2003. 429-443.

[39] BERNSTEIN, D. **Proving primality after Agrawal-Kayal-Saxena**. University of Illinois. Chicago. 2003.

[40] AGRAWAL, M.; KAYAL, N.; SAXENA, N. **Polynomial time deterministic method for testing primality of numbers**. US Patent 7346637, 18 Março 2008.

[41] AARONSON, S. **The Prime Facts: From Euclid to AKS (Manuscript)**. 2003.

[42] FOUVRY, E. Theoreme de Brun-Titchmarsh; application au theoreme de Fermat. **Inventiones Mathematicae**, v. 79, n. 2, p. 383-407, Junho 1985.

[43] BERNSTEIN, D. J. Detecting Perfect Powers in Essentially Linear time. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 67, n. 223, p. 1253-1283, Julho 1998.

[44] SALEMBIER, R. G.; SOUTHERINGTON, P. **An Implementation of the AKS Primality Test**. 2005.

[45] CHENG, Q. Primality Proving via One Round in ECPP and One Iteration in AKS. **Journal of Cryptology**, Nova York, v. 20, n. 3, p. 375-387, Julho 2007.

[46] LENSTA JR., H. W.; POMERANCE, C. Primality testing with Gaussian periods (preliminary version), 2005.

[47] BERRIZBEITIA, P. Sharpening Primes is in P for a large family of numbers. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 74, p. 2043-2059, Abril 2005.

[48] BERNSTEIN, D. Proving Primality in Essentially Quartic Random Time. **Mathematics of Computation**, Providence - Rhode Island - USA, v. 76, n. 257, p. 389-403, Janeiro 2007.

[49] GRANVILLE, A. It`s Easy to Determine Whether a Given Integer is Prime. **Bulletin (New Series) of the American Mathematical Society**, v. 24, n. 1, p. 3-38, 2004.

[50] MICROSOFT CORP. Microsoft Solver Foundation. **MSDN Code Gallery**, 2009. Disponivel em: <http://code.msdn.microsoft.com/solverfoundation>. Acesso em: 07 nov. 2009.

[51] OYSTER. IntX. **CodePlex Open Source Community**, 2008. Disponivel em: <http://www.codeplex.com/IntX>. Acesso em: 07 nov. 2009.

[52] LI, H. **The Analysis and Implementation of the AKS Algorithm and Its Improvement Algorithms**. University of Bath. Bath - UK. 2007.

[53] ROTELLA, C. **An Efficient Implementation of the AKS Polynomial-Time Primality Proving Algorithm**. School of Computer Science - Carnegie Mellon University. Pittsburgh - Pennsylvania - USA. 2005.

[54] CRANDALL, R.; PAPADOPOULOS, J. **On the implementation of AKS-class primality tests**. Apple Computer and the University of Maryland College Park. [S.l.]. 2003.