Ísis Binder

Algoritmo AKS : Teoria e Implementação

Monografia apresentada à disciplina Trabalho de Graduação II como requisito à conclusão do Curso de Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: André Luiz Pires Guedes

Curitiba – PR

2008

Agradeço ao professor André pela paciência, por ter emprestado livros sobre criptografia, não ter posto pressão na escolha do tema quando eu não tinha a mínima noção de que assunto escolher e pelas conversas.

Ao professor Jair pelos e-mails com algoritmos,artigos e notas de aula da matéria optativa de criptografia e por dizer que a desigualdade que ignorei conscientemente a pedido do André era um teorema.

Ao professor Renato, mas não somente a ele, por apontar meus erros e ajudar a organizar as idéias.

Resumo

Números primos são aqueles que possuem somente dois divisores diferentes. Parece simples, mas por vários anos a busca por um algoritmo de primalidade determinístico, se não falhou, fez com que matemáticos e cientistas da computação utilizassem abordagens diferentes para conseguir uma solução ótima com as ferramentas disponíveis ou métodos nos quais se podia pensar. Fermat e Riemann foram as bases para um teste de primalidade desenvolvido por Gary L. Miller. Algoritmos baseados no teste probabilístico de Fermat foram escritos usando-se aleatorização. No ano de 2002 esta busca parece ter chegado ao fim: 2 estudantes indianos encontraram um algoritmo determinístico e polinomial em $\log_2 n$ para se determinar a primalidade de um dado número n. Utilizando conceitos básicos de Teoria dos Números, seu algoritmo quebra a barreira dos números de Carmichael através de uma identidade que somente os números primos satisfazem.

Abstract

Prime numbers are those with only two different divisors. Sounds simple, but for many years the quest for an efficient polynomial deterministic primality algorithm, if it hasn't failed, has led mathematicians and computer scientists to take different approaches in order to reach an optimum solution using available tools or methods they could think of. Fermat and Riemann were the basis for a primality test devised by Gary L. Miller. Aalgorithms based on Fermat's primality test were written and made use of randomness. In the year 2002 this quest has likelly come to an end: 2 indian students found a deterministic and $\log_2 n$ polynomial algorithm to determine wether a given number n is prime or composite. Using basic concepts of Number Theory their algorithm overcomes the Carmichael numbers barrier using an identity that only prime numbers satisfy.

Sumário

1 Introdução

O problema tratado por este trabalho é o teste de primalidade de um número inteiro, ou seja, decidir se um inteiro é primo ou composto. Segundo a definição, um número $p > 1$ é primo se no conjunto de seus divisores figurarem somente dois números distintos: 1 e p. Se o conjunto possuir tamanho maior que 2, o número é dito composto. Existem dois tipos de algoritmos envolvendo primos: os crivos, que enumeram primos dentro de um intervalo e os algoritmos que, dado um número maior que 1, verificam se ele é primo. Este trabalho tem foco em um algoritmo que pertence ao segundo conjunto: o AKS.

A aplicação mais conhecida para os números primos é a criptografia de chave pública, que permite transações bancárias seguras e encriptação de emails ou de dados num disco rígido. Tudo para que terceiros não tenham acesso a informações que devem ser protegidas. Um dos algoritmos mais conhecidos é o RSA, que envolve a multiplicação de dois números primos p e q. Alguém pode usar um gerador aleatório e executar um teste de primalidade até que o número produzido seja reconhecido como primo, mas existem algoritmos próprios para gerar números primos.

O que torna o AKS interessante é o seguinte:

- É um algoritmo determinístico.
- Não está restrito a uma classe de números primos.
- É polinomial no tamanho em bits da entrada.

O objetivo deste trabalho é explicar a teoria envolvida no algoritmo. Divide-se em 8 seções: a primeira fornece uma explicação em alto nível do funcionamento do AKS; a segunda apresenta a complexidade do algoritmo; a terceira mostra os testes básicos utilizados para identificar um número composto e a seção sobre o Teorema de Fermat introduz uma congruência necessária para a sexta seção, que demonstra a identidade utilizada para o último teste no algoritmo. A sétima seção tenta explicar algumas partes do

algoritmo que não parecem tão óbvias, seguida pela implementação do AKS em Python. No apêndice consta uma implementação de um gerador de números de Carmichael, que, embora não muito eficiente, pode ser utilizado para gerar casos de teste para a implementação do AKS.

2 AKS em Pseudo-Código

A tradução do algoritmo apresentado no trabalho de Kayal, Saxena e Agrawal, de 2004, é transcrita a seguir. Optou-se por explicitar a notação da função logarítmica na base 2 a fim de se evitar confusões.

Algoritmo 1 AKS - Pseudocódigo

Entrada: $n \in \mathbf{N} : n > 1$

1: **Se** $n = a^b$ **para algum** $a \in \mathbf{Z}, b > 1$ **então**
2: Imprima COMPOSTO
3: **fim Se**
4: **Encontre o menor** r **tal que** $o_r(n) > \log_2^2 n$
5: **Se** $1 < (a, n) < n$ **para algum** $a \leq r$ **então**
6: Imprima COMPOSTO
7: **fim Se**
8: **Se** $n \leq r$ **então**
9: Imprima PRIMO
10: **fim Se**
11: **Para** $a \leftarrow 1$ **até** $\lfloor \sqrt[2]{\Phi(r)} \times \log_2 n \rfloor$ **faça**
12: **Se** $(X + a)^n \neq X^n + a \mod X^r - 1, n$ **então**
13: Imprima COMPOSTO
14: **fim Se**
15: **fim Para**
16: Imprima PRIMO

- $o_r(n)$: ordem de n módulo r. É o menor inteiro k tal que $n^k \equiv 1(mod\ r)$.
- *(a,n)*: máximo divisor comum de a e n.
- $\Phi(r)$: quantidade de números primos menores que r.
- $(X+a)^n \neq X^n+a \mod X^r-1, n$: a expansão de $(X+a)^n$ será dividida por X^r-1 e em seguida, por n, sendo que a congruência é verificada após a divisão por n.

A linha 1 verifica se a entrada n possui somente um fator primo, tendo como base o Teorema Fundamental da Aritmética, sendo explicada com maiores detalhes na seção intitulada "Números Compostos". Se for encontrado um fator, pela definição de número primo, n é um número composto e o algoritmo termina.

Na linha 4 encontra-se um inteiro r que fará parte dos testes seguintes, funcionando como limite superior do intervalo onde é realizado o cálculo do máximo divisor comum entre dois inteiros, sendo também usado na composição de um limite da quantidade de inteiros a serem substituidos em a na linha 12, limitando a quantidade de polinômios avaliados para que esse teste identifique os primos.

O teste da linha 5 visa encontrar um divisor comum entre n e um inteiro menor ou igual a r, calculado anteriormente,o que caracteriza n como número composto. Após termos verificado que n não é potência exata de um fator primo e nem possui um divisor comum no intervalo (1,r], temos uma condição que, segundo os autores, é válida apenas quando $n \leq 5.690.034$, graças a um lema demonstrado em [AKS04].

O último teste tem origem no Pequeno Teorema de Fermat,explicado na seção 5, e uma relação com o Teorema Binomial, demonstrado na seção 6. Sem entrar nos detalhes, a congruência utilizada é válida se, e somente se, n for um número primo. Se existir algum valor de a que torne a identidade falsa, então n é composto, caso contrário, o algoritmo segue e indica que n é primo.

De modo geral, para um dado inteiro $n \geq 2$, tal que a ordem de n módulo r maior que $\log_2^2 n$, n é primo se, e somente se

- n não for potência exata.
- n não possuir fator primo $\leq r$.
- $(X+a)^n \neq X^n+a \mod X^r-1, n$ para todo inteiro a, $0 < a \leq \lfloor \sqrt{\Phi(r)} \times \log_2 n \rfloor$.

3 Complexidade do Algoritmo

O número de operações necessárias para se executar um algoritmo é chamado de *complexidade computacional*.Nesta seção desejamos explicitar que, por "tamanho da entrada", queremos dizer a quantidade de dígitos necessários para se representar o número p (a entrada do algoritmo) na base binária. Para um inteiro p essa quantidade vale $\lceil \log p \rceil$. A complexidade do AKS é $O\tilde{(}\log^{21/2} p)$. Seja $m(p)$ o tempo de execução do algoritmo. Utilizando a notação assintótica, um algoritmo possui execução polinomial se $m(p) = O(p^k)$, sendo k uma constante.

- Verificar a existência de potências próprias tem complexidade $O\tilde{(}\log^3 p)$.
- Encontrar um r que satisfaça a condição da linha 4 do algoritmo envolve testar no máximo $\log_2^2 p$ expoentes. Isso envolve $O(\log^2 p)$ multiplicações módulo r, cada uma custando $O(\log r)$. Segundo um lema demonstrado em [AKS04], é necessário testar somente $O(\log^5 p)$ valores de r. A complexidade total do segundo passo é $O\tilde{(}\log^7 p)$.
- O terceiro passo consiste em encontrar o máximo divisor de r números, resultando em $O(r \log p) = O(\log^5 p \log p) = O(\log^6 p)$.
- O teste da congruência consiste na verificação de $\lfloor \sqrt[2]{\phi(r)} \log p \rfloor$ equações. Cada uma delas possui r coeficientes de tamanho $O(\log p)$, totalizando $O\tilde{(}r\sqrt{\phi(r)} \log^3 p) = O\tilde{(}r^{3/2} \log^3 p) = O\tilde{(}\log^{15/2} p \log^3 p) = O\tilde{(}\log^{21/2} p)$.

Como a última complexidade domina as outras, temos que $O\tilde{(}\log^{21/2} p)$ é a complexidade do algoritmo, sendo polinomial em $\log_2 p$.

A título de comparação: o algoritmo mais conhecido que verifica a primalidade de um determinado número p é o Crivo de Eratóstenes. Ele consiste em verificar se existe algum divisor de p no intervalo $[2, \sqrt{p}]$, sendo que, no pior caso, realiza $\sqrt{p}$ operações. Isso equivale a dizer que o algoritmo tem complexidade $O(p^{1/2})$, ou seja, polinomial em p. Mas sendo $k = \log_2 p$, $p = 2^k$ e substituindo, temos $O(2^{k/2})$, ou seja, tempo de execução exponencial em $\log_2 p$.

4 Números Compostos

Segundo o Teorema Fundamental da Aritmética, todo número natural maior do que 1 pode ser escrito como produto de números primos, sendo esse produto único a menos da ordem dos fatores.

Sendo p um número composto, ele pode ser escrito como $f_1^{\alpha_1} f_2^{\alpha_2} \dots f_i^{\alpha_i}$, onde, para $i, \alpha \geq 1$ e $1 < f < p$, f_i é seu fator primo e α_i seu respectivo expoente.

Uma potência de primo é um número que possui somente um fator primo, ou seja, $p = f_1^{\alpha_1}$. Note que números primos também se encaixam nessa definição. Temos aí o primeiro passo do algoritmo : verificar se a entrada é uma potência de primo, de tal forma que o expoente do fator seja maior do que 1. Os números de Carmichael falham nesse teste, já que possuem pelo menos 3 divisores.

Se um inteiro positivo p não é potência de primo e nem número primo, possui outros divisores no seu conjunto de divisores próprios (divisores menores que p). Aqui podemos esbarrar em um dos problemas da Teoria dos Números: a fatoração. Exemplo: $p = 18 = 2 \times 3^2$. Mas e quanto a $p = 8374283698277717171$? Utilizando um computador a fatoração pode nos parecer extremamente rápida ($8374283698277717171 = 7 \times 29 \times 97 \times 239 \times 2927 \times 607937977$), mas os algoritmos de criptografia utilizam números muito maiores do que esse, o que torna a fatoração um processo extremamente custoso mesmo para alguém realmente interessado em saber o número do seu cartão de crédito internacional.

Sendo falso o teste das potências próprias, o algoritmo utiliza o máximo divisor comum de dois inteiros para verificar se p é composto.

5 O Pequeno Teorema de Fermat

Esta seção explica o Pequeno Teorema de Fermat, do qual a idéia central do AKS é derivada e demonstra uma identidade utilizada na próxima seção. Fermat foi um matemático francês, e em uma carta endereçada a Frénicle [2], discorre sobre progressões geométricas. Um trecho da carta é reproduzido abaixo.

> Tout nombre premier mesure infailliblement une des puissances - 1 de quelque progression que ce soit, et l'exposant de la dite puissance est sous-multiple du nombre premier donnée - 1; et, après quon'a trouvée la première puissance qui satisfait à la question, toutes celles dont les exposants sont multiples de l'exposant de la première satisfont tout de même à la question. [1]

Segundo Fermat, se p é primo, então existe um expoente mínimo $d \in \mathbf{Z}$ tal que $b^d - 1 \equiv 0 \pmod{p}$ e $d \mid p-1$. Além disso, $b^{p-1} - 1 \equiv 0 \pmod{p}$. Embora não seja a identidade utilizada diretamente no AKS, ela será necessária para demonstrar a congruência entre $(x+a)^p$ e $x^p + a$.

PEQUENO TEOREMA DE FERMAT. Seja p um primo > 0 e b um inteiro. Se $p \nmid b$ então $b^{p-1} \equiv 1 \pmod{p}$.

DEMONSTRACÃO. Vide [JPOS05], página 41.

O Pequeno Teorema de Fermat é um teste necessário mas não suficiente para se determinar a primalidade de um número: R. D. Carmichael encontrou, em 1910, números compostos que satisfazem a congruência dada por Fermat. Esses números satisfazem o teorema de Fermat para qualquer b escolhido.

A partir desse teorema temos um corolário: se p é um primo e b um inteiro positivo, então $b^p \equiv b \pmod{p}$.

[1] Todo número primo divide uma das potências - 1 de alguma progressão geométrica, e o expoente da respectiva potência é submúltiplo do número primo dado - 1; e, após termos encontrado a primeira potência que satisfaz à questão, todas aquelas que possuem expoentes múltiplos do expoente da primeira satisfazem igualmente à questão.

DEMONSTRACÃO. Se p divide b, então p divide $b(b^{p-1}-1) = b^p - b$. Logo, $b^p \equiv b \pmod{p}$. Se p não divide b, pelo pequeno teorema de Fermat, p divide $b^{p-1}-1$. Multiplicando por b temos $b^p - b$, que também é divisível por p. Logo, $b^p \equiv b \pmod{p}$.

6 Teorema Binomial

Um dos passos do algoritmo envolve a expansão de $(x+a)^p$, para a e p inteiros. O Teorema Binomial nos diz que $(x+a)^p = \sum_{i=0}^{p} \binom{p}{i} x^{p-i}a^i$, sendo $\binom{p}{i} = \frac{p!}{i!(p-i)!}$. A base do algoritmo é a seguinte identidade que somente números primos satisfazem, sendo $a \in \mathbf{Z}, p \in \mathbf{N}$ tal que $p \geq 2$ e $MDC(a,p) = 1$:

$$(x+a)^p \equiv x^p + a \pmod{p} \qquad (1)$$

Somente se p for primo, a expansão de $(x+a)^p$ dividida por p possui resto $x^p + a$.

Exemplos:

p = 3

$(x+a)^3 = a^3 + 3xa^2 + 3x^2a + x^3 = 3(xa^2 + x^2a) + (a^3 + x^3)$

$(x+a)^3 \equiv x^3 + a^3 \pmod{3}$

p = 4

$(x+a)^4 = a^4 + 4xa^3 + 6x^2a^2 + 4x^3a + x^4 = 4(xa^3 + x^3a) + (a^4 + 6x^2a^2 + x^4)$

$(x+a)^4 \not\equiv x^4 + a^4 \pmod{4}$

Por enquanto ignore o expoente em a. Isso será demonstrado posteriormente. Agora iremos demonstrar porque a congruência só é válida quando p é primo. Os termos em a^p e x^p sempre terão coeficientes iguais a 1, de modo que consideraremos $1 < i < p$ para o cálculo de $\binom{p}{i}$, ou seja, iremos provar que o restante do polinômio contém somente termos múltiplos de p quando p for primo e apenas quando essa condição for verdadeira.

Assumindo $\binom{p}{i}$ como um número inteiro, temos que, considerando um inteiro c,

$$\frac{p!}{i!(p-i)!} = c = \frac{p(p-1)(p-2)...(p-(i-1))(p-i)!}{i!(p-i)!} = \frac{p(p-1)(p-2)...(p-(i-1))}{i!}$$

Suponha que p seja primo. Para $1 < i < p$, o denominador não divide p, mas como o lado direito é um inteiro, $i!$ deve dividir $(p-1)(p-2)\ldots(p-(i-1))$. Seja q o quociente dessa divisão. Então $pq = c$ e $pq \equiv 0 \pmod{p}$.

Agora suponha que p seja composto. Então existe um fator primo q tal que q^e seja a maior potência desse fator que divide p, para um inteiro $e > 0$.

$$\begin{aligned}\binom{p}{q} &= \frac{p!}{q!(p-q)!} = \frac{p(p-1)(p-2)...(p-q)!}{q!(p-q)!} \\ &= \frac{p(p-1)(p-2)...(p-q+1)}{q!} \\ &= \frac{q^e d(p-1)(p-2)...(p-q+1)}{q(q-1)!} \\ &= \frac{q^{e-1} d(p-1)(p-2)...(p-q+1)}{(q-1)!}\end{aligned}$$

Como o produto de $q-1$ inteiros consecutivos é divisível por $(q-1)!$ (veja demonstração em [JPOS05], página 12), temos que

$$\binom{p}{q} a^{p-q} x^q \;=\; q^{e-1} dc a^{p-q} x^q, \text{ sendo } c = \frac{(p-1)(p-2)...(p-q+1)}{(q-1)!}$$

Exemplo com $n = 4$, sendo $q = 2$: $\binom{4}{2} = \frac{2^2 \times 3}{2 \times 1!} = \frac{2 \times 3}{1} = 6$.

Assim, o termo $\binom{p}{q} a^{n-q} x^q$ não é múltiplo de p se p for composto e o polinômio não é congruente a $a^p + x^p$ quando dividido por p.

Mas a congruência utilizada é $(x+a)^p \equiv x^p + a \pmod{p}$. Isso segue a partir do corolário do Pequeno Teorema de Fermat e das seguintes propriedades:

- Se a, b, d e m são inteiros, $m > 0$ e se $a \equiv b \pmod{m}$ e $b \equiv d \pmod{m}$, então $a \equiv d \pmod{m}$.
- Se a, b, c e m são inteiros tais que $a \equiv b \pmod{m}$, então $a + c \equiv b + c \pmod{m}$.

O corolário nos diz que $a^p \equiv a \pmod{p}$. Pelo segundo item acima, $x^p + a^p \equiv x^p + a \pmod{p}$. Como $(x+a)^p \equiv x^p + a^p \pmod{p}$ e $x^p + a^p \equiv x^p + a \pmod{p}$, pelo primeiro item (propriedade transitiva), temos que $x^p + a^p \equiv x^p + a \pmod{p}$.

Um breve histórico sobre esse teste: em [AB03] a identidade utilizada na versão aleatória do algoritmo é $(x+1)^p \equiv x^p + 1 \pmod{p}$. Em 2001, num relatório técnico, Rajat Bhatacharjee e Prashant Pandey fixaram $a = 1$ e decidiram verificar os requisitos para vários valores de r. Em 2002, Nitin Saxena e Neeraj Kayal, juntamente com Manindra Agrawal, fizeram o inverso: fixaram r e variaram a.

7 Divisibilidade do Produtório

A parte que ainda resta entender é o papel desempenhado pelo r no algoritmo.Esta seção exibe a demonstração de uma observação em [AKS04]. Iniciamos com a definição de ordem multiplicativa:

Para $r \in \mathbf{N}$, $i \in \mathbf{Z}$, com $MDC(p,r) = 1$, $o_r(p)$ é o menor inteiro $i > 1$ tal que $p^i \equiv 1 \pmod{r}$.

Agrawal, Kayal e Saxena utilizam o seguinte produto para demonstrar que a ordem de p módulo r é maior do que $\log[2]^2 p$.

$$p^{\lfloor \log_2 B \rfloor} \prod_{i=1}^{\lfloor \log_2^2 p \rfloor} (p^i - 1), \text{ sendo } B = \lceil \log_2^5 p \rceil \quad (2)$$

Seja r o menor inteiro que não divide o produto acima. Agrawal, Kayal e Saxena afirmam que o mdc(r,p) não pode ser divisível por todos os fatores primos de r, caso contrário, r divide $p^{\lfloor \log_2 B \rfloor}$.

DEMONSTRACÃO. Seja $r = f_1^{\alpha_1} f_2^{\alpha_2} \dots f_j^{\alpha_j}$ e $f_k^{\alpha_k}$ um fator primo de r para $0 < k \leq j$. O máximo divisor comum entre dois inteiros p e r é um inteiro tal que $MDC(p,r) = pb + rc$. Se $MDC(p,r)$ for divisível por todos os fatores primos de r, então ele é da forma $f_1 f_2 \dots f_j a$. Nesse ponto, temos:

- [1].$MDC(p,r) = f_1 f_2 \dots f_j a$
- [2].$r = f_1^{\alpha_1} f_2^{\alpha_2} \dots f_j^{\alpha_j} = f_1 f_2 \dots f_j (f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1}) = \frac{MDC(p,r)}{a} (f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1})$

Mas $MDC(p,r) = pb + rc$. Substituindo:

$$f_1 f_2 \dots f_j a = pb + rc$$

$$f_1 f_2 \dots f_j a = pb + f_1^{\alpha_1} f_2^{\alpha_2} \dots f_j^{\alpha_j} c$$

$$f_1 f_2 \dots f_j a - f_1^{\alpha_1} f_2^{\alpha_2} \dots f_j^{\alpha_j} c = pb$$

$$f_1 f_2 \dots f_j (a - f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1} c) = pb$$

$$\frac{MDC(p,r)}{a} (a - f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1} c) = pb$$

$$\frac{r}{f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1}} (a - f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1} c) = pb$$

$$r(\frac{a}{f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1}} - c) = pb$$

Como o lado direito é um inteiro e r é inteiro, $\frac{a}{f_1^{\alpha_1 - 1} f_2^{\alpha_2 - 1} \dots f_j^{\alpha_j - 1}} - c$ deve ser inteiro. Logo, r divide pb ($r \mid p^{\lfloor \log_2 B \rfloor} = p \times p^{\lfloor \log_2 B \rfloor - 1}$).

Como r é o menor inteiro que não divide (2), então $MDC(r, p) = 1$. Além disso, como r não divide nenhum dos termos $(p^i - 1)$ para $1 \leq i \leq \lfloor \log_2^2 p \rfloor$, a ordem de p módulo r, $o_r(p)$, é maior que $\log_2^2 p$.

8 Implementação

Inicialmente decidi implementar o algoritmo usando Groovy, uma linguagem de script para a plataforma Java, mas uma propriedade da linguagem terminou por ser considerada um obstáculo devido à biblioteca utilizada para realizar os cálculos dos polinômios. Mas ao ler a documentação da API notei que as definições das variáveis tornariam o código complicado para ser lido. Levando isso em conta, decidi escrever o algoritmo em Python (versão 2.5.1).

Em Java tem-se tanto os tipos primitivos – int, char, float,double, byte – como os *wrappers* – Integer, Character, Float, Double, Byte. Em Groovy, apesar de ser permitido declarar uma variável como int, apenas o *wrapper* é usado. Utilizando a biblioteca JScience, alguns métodos necessitam de tipos primitivos como argumentos, o que inevitavelmente acarreta em erro de compilação. A integração do código em Groovy com o código em Java seria mais trabalhoso do que implementar em outra linguagem que possuísse um tipo de dado para polinômios.

A implementação em Python é limitada a inteiros até 10^7. Os inteiros possuem um alcance ilimitado, sujeitos somente à disponibilidade de memória virtual, mas os números em ponto flutuante, implementados como *double* de C, estão sujeitos à limitação do hardware.

Algoritmo 2 AKS

```
1
  from math import ceil,floor,log10,sqrt
  from numpy import poly1d,polydiv,zeros
4 import sys
  import os

7 def mdc(Numero,r):
      shift = 0
```

```
    tmp1,tmp2 = Numero,r

    while tmp1 > 0 and tmp2 > 0:
        if tmp1%2 == 0 and tmp2%2 == 0:
            tmp1,tmp2,shift = tmp1>>1,tmp2>>1,shift+1
        elif tmp1%2 == 0:
            tmp1 = tmp1>>1
        elif tmp2%2 == 0:
            tmp2 = tmp2>>1
        elif tmp1 > tmp2:
            tmp1 = tmp1-tmp2
        else:
            tmp2=tmp2-tmp1
    return max(tmp1,tmp2)*(2**shift)


def phi(r):
    contadorPhi = 0
    for indexPhi in range(2,int(r)):
        if mdc(indexPhi,int(r)) == 1:
            contadorPhi+=1
    return contadorPhi


def ordem(Numero,logaritmo):
  q = logaritmo+1
  indexOrdem = 1
  while indexOrdem <= logaritmo:
    if pow(Numero,indexOrdem,q) == 1:
      q+=1
      indexOrdem=1
    else:
      indexOrdem+=1
  return q

```

```
46 def main(argv):
       Numero = int(argv[1])
       # Calcula potencia exata
49     for base in range(2,Numero):
           logaritmo = log10(Numero)/log10(base)
           if ceil(logaritmo) == floor(logaritmo):
52             print "COMPOSTO"
               return


55
       logaritmo = int((log10(Numero)/log10(2))**2)

58     #o_r(n)
       r = ordem(Numero,logaritmo)

61     print r
       a = 1
       while a <= r :
64         k = mdc(a,Numero)
           if 1 < k and k < Numero:
               print "COMPOSTO"
67             return
           a+=1

70     if Numero <= r:
           print "PRIMO"
           return
73     else:
           limite = int(sqrt(phi(r))*(log10(Numero)/log10(2)))
           redutor = zeros(r+1,int)
76         redutor[0],redutor[-1] = 1,-1
           redutor = poly1d(redutor)
```

```
79          for a in range(1,limite+1):
                LS = zeros(2,int)
                LS[0],LS[-1] = 1,a
82              LS = poly1d(LS)**Numero

                RS = zeros(Numero+1,int)
85              RS[0],RS[-1] = 1,a
                RS = poly1d(RS)


88
                resto1 = polydiv(LS,redutor)[1]
                resto1 = polydiv(resto1,poly1d([Numero]))[1]
91
                resto2 = polydiv(RS,redutor)[1]
                resto2 = polydiv(resto2,poly1d([Numero]))[1]
94              if resto1 != resto2:
                    print "COMPOSTO"
                    return
97
            print "PRIMO"

100 if (__name__ == "__main__"):
        main(sys.argv)
```

Para aumentar a legibilidade do programa, podemos substituir *log10(Numero)/log10(base)* por *log(Numero,base)*.

8.1 Máximo Divisor Comum

Em substituição ao algoritmo tradicional para o cálculo do MDC, utilizou-se a versão binária. Existem outros modos de se calcular o máximo divisor comum entre dois inteiros, como o algoritmo de Lehmer ou o algoritmo de Schönhage.

O algoritmo de Euclides tradicional baseia-se no seguinte teorema: para qualquer inteiro a não-negativo e qualquer inteiro b positivo, MDC(a,b) = MDC(a, a mod b).*"A mod b"* denota o resto da divisão de a por b.A demonstração pode ser encontrada em

[CLR01], no capítulo 31.

Algoritmo 3 Pseudo-código do Algoritmo de Euclides

```
    EUCLIDES(a,b):Se b = 0 então
1:      retorne a
3:  Senão
4:      retorne EUCLIDES(a, a mod b)
5:  fim Se
```

Se ambos os inteiros, *tmp1* e *tmp2* forem pares, temos 2 como um fator comum. Sendo $MDC(tmp1, tmp2) = tmp1 \times n + tmp2 \times m$, com n e m inteiros, $tmp1 = 2d$ e $tmp2 = 2e$ (d e e inteiros), substituindo, temos $MDC(tmp1, tmp2) = 2d \times n + 2e \times m = 2(d \times n + e \times m) = 2MDC(d, e) = 2MDC(\frac{tmp1}{2}, \frac{tmp2}{2})$. Isso justifica o deslocamento à direita e a contagem desses deslocamentos utilizados no algoritmo. Note que nenhum dos dois inteiros passados como argumentos podem ser potências de 2, pois isso é reconhecido no primeiro passo do algoritmo.

Se um dos inteiros for ímpar, precisamos retirar o fator 2 do inteiro par, já que não é um divisor comum. Supondo que *tmp1* seja par e *tmp2* ímpar, temos $MDC(tmp1, tmp2) = m \times tmp1 + n \times tmp2 = m \times 2c + n \times tmp2$. Fazendo $d = m \times 2$, $MDC(tmp1, tmp2) = d \times c + n \times tmp2 = MDC(c, tmp2) = MDC(\frac{tmp1}{2}, tmp2)$.

As duas últimas condicionais lidam com a hipótese de ambos os inteiros serem ímpares. Nesse caso, calculamos o resto da divisão utilizando subtrações sucessivas de *tmp1* ou *tmp2*, sendo o quociente igual a 1.

O valor retornado é o máximo divisor comum, sendo que, quando *tmp1* e *tmp2* são pares, é calculado o valor do expoente da potência de 2, totalizando o maior fator par entre os números. A iteração termina quando um dos inteiros for nulo, sendo que o fator restante do MDC é obtido calculando-se o máximo entre tmp1 e tmp2 ($MDC(0, x) = x$).

8.2 Potências Próprias

Na identificação de potências próprias, a implementação utiliza uma identidade que é válida somente quando o número n for inteiro: $\lfloor n \rfloor = \lceil n \rceil$. Calcula-se $\log_b n$ e verifica-se esta igualdade. Em [CP03], é mencionado o algoritmo de Newton para inteiros, usado para calcular $\lfloor p^{1/b} \rfloor$. O valor retornado é então elevado a b e verifica-se se essa exponenciação produz p como resultado. O teste é realizado para todo $2 \leq b < \log_2 p$.

Algoritmo 4 Algoritmo de Newton para calcular $\lfloor p^{1/b} \rfloor$

```
x = 2^⌈B(p)/b⌉; /* B(p) = número de bits em p */
Enquanto 1 faça
    y = ⌊((b − 1)x + ⌊p/x^(k−1)⌋)/b⌋;
    Se x > y então
    Retorne x;
  fim Se
  x = y;
fim Enquanto
```

8.3 Congruência Polinomial

Para o cálculo da congruência, utilizou-se as operações fornecidas nos módulos *builtins* e *Numpy*, mas a exponenciação usada em $(x+a)^p$ pode ser calculada como segue.

Algoritmo 5 Algoritmo para cálculo de $(x + a)^p$

```
F(X) = 1
B(P) = b_{l−1} ... b_0 /* Representação binária de P */
Para K ← l − 1 até 0 faça
  F(X) = F(X)^2
  Se b_k = 1 então
    F(X) = F(X) * (X+a)
  fim Se
  F(X) = F(X) (mod X)^r − 1
fim Para
Retorna F(X)
```

8.4 Ordem Multiplicativa

Finalmente, uma explicação sobre a implementação do cálculo da ordem de n módulo r. Inicialmente, $q = \lfloor \log_2^2 n \rfloor + 1$. Essa variável armazena o menor expoente tal que $\prod_{i=1}^{\lfloor \log_2^2 p \rfloor} (p^i - 1)$ não seja divisível por r. O cálculo da ordem é interrompido quando $indexOrdem > \lfloor \log_2^2 n \rfloor$, com *indexOrdem* correspondendo ao i do produto mencionado.

O primeiro *if* verifica se $n^{indexOrdem} \equiv 1 \pmod{q}$. Se isso for verdade, então q divide $\prod_{i=1}^{\lfloor \log_2^2 p \rfloor} (p^i - 1)$ e teremos que incrementar q, pois ele não satisfaz a definição da ordem multiplicativa e redefinir *indexOrdem*. Se a congruência testada for falsa, então prossegue-se para o próximo *indexOrdem*. O teste executa até que *indexOrdem* seja maior

que $\lfloor \log_2^2 p \rfloor$, indicando que nenhum termo do produto é divisível por q e o algoritmo retorna a ordem de n módulo q.

9 Conclusão

Sobre a implementação, posso dizer que mesmo utilizando uma linguagem que dizem ser fácil de se aprender tive muitas surpresas. A primeira delas foi quanto ao limite máximo de uma das versões: se $p > 1086$ o algoritmo entrava em loop, aparentemente numa função lambda utilizada para zerar o polinômio, gerando a mesma mensagem de erro que um interpretador de Prolog exibiria se a recursão excedesse a profundidade máxima. A segunda delas tinha a ver com os tipos em Python e que fazia o programa concluir que 37 é número composto. Só escolhi a linguagem por causa do módulo Numpy, já que a biblioteca JScience,que seria utilizada caso o algoritmo fosse implementado em Java, deixaria o código muito poluído, pois abusa de Java Generics (construção parecida com os templates em C++).

É difícil identificar as idéias presentes no algoritmo, principalmente no segundo passo, onde se obtém um limite que é utilizado nos passos seguintes. Por causa disso a interpretação não ultrapassou o básico.Se uma pessoa compreender melhor a função da ordem multiplicativa calculada no passo 2, talvez fique mais claro o motivo da limitação do máximo divisor comum ser r, bem como o expoente do polinômio e o limite superior da iteração onde se utiliza o Teorema Binomial.

Referências

AARONSON, S. *The Prime Facts : From Euclid to AKS*. 2003. Disponível em: <http://www.scottaaronson.com/writings/prime.pdf>. Acesso em: 10 Nov. 2007.

AGRAWAL, M.; BISWAS, S. *Primality and Identity Testing via Chinese Remaindering*. [S.l.], 2003. Disponível em: <http://www.cs.berkeley.edu/ luca/cs174/ab.ps>. Acesso em: 23 Set. 2007.

AGRAWAL, M.; KAYAL, N.; SAXENA, N. *Primes is in P*. [S.l.], 2004. Disponível em: <http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf>. Acesso em: 20 Set. 2007.

AGRAWAL MANINDRA E KAYAL, N. e. S. N. *Towards a Deterministic Polynomial-Time Primality Test*. [S.l.], 2002. Disponível em: <http://www.cse.iitk.ac.in/research/btp2002/primality.ps.gz>. Acesso em: 25 Set. 2007.

BERNSTEIN, D. J. *Proving Primality After Agrawal-Kayal-Saxena*. 2003. Disponível em: <http://cr.yp.to/papers/aks.pd f>. Acesso em: 10 Nov. 2007.

BORNEMANN, F. Primes is in p: A breakthrough for "everyman ". *American Mathematical Society*, 2003. Disponível em: <http://www.ams.org/notices/200305/fea-bornemann.pdf>. Acesso em: 25 Nov. 2007.

FERMAT, P. de. *Carta a Frénicle de Bessy*. Disponível em: <http://www.cs.utexas.edu/users/wzhao/fermat2.pdf>. Acesso em: 23 Set. 2007.

GRANVILLE, A. It is easy to determine whether a given integer is prime. *American Mathematical Society*, v. 42, n. 1, Abr. 2004. Disponível em: <http://math.stanford.edu/ brubaker/granville.pdf>. Acesso em: 12 Dez. 2007.

JR, H. L.; POMERANCE, C. *Primality Testing with Gaussian Periods*. 2005. Disponível em: <http://gauss.dartmouth.edu/ carlp/PDF/complexity12.pdf>. Acesso em: 20 Nov. 2007.

KIMING, I. *The Agrawal-Kayal-Saxena Primality Test*. 2006. Disponível em: <http://www.math.ku.dk/ kiming/lecturenotes/2006-cryptography/aks.pdf>. Acesso em: 12 Nov. 2007.

STEHLÃ©, D.; ZIMMERMANN, P. A binary recursive gcd algorithm. In: SPRINGER (Ed.). *Algorithmic Number Theory - 6th International Algorithmic Number Theory Symposium*. [s.n.], 2004. Disponível em: <http://books.google.com/books?id=XplX3pNGtYC& printsec=frontcover& hl=pt-BR>. Acesso em: 15 Jan. 2008.

TOU, C.-S.; ALEXANDER, T. *AKS Algorithm*. 2005. Disponível em: <http://padic.mathstat.uottawa.ca/ MAT3166/reports/AKS.pdf>. Acesso em: 10 Dez. 2007.

APÊNDICE A – Geração de Números de Carmichael

Devido ao meu ceticismo em relação à minha implementação resolvi escrever um script em Bash que gerasse números de Carmichael para que pudessem ser utilizados como entrada. Foram encontrados dois critérios que podem ser utilizados para isso: o critério de Korselt e o problema da função Φ *deLehmer*.

Critério de Korselt. Seja $n \geq 2$. Então as seguintes afirmações são equivalentes:

- $a^n \equiv a(mod\ n) \forall a \in \mathbf{Z}$
- $a^{n-1} \equiv 1(mod\ n) \forall a \in \mathbf{Z}$ tal que $(a, n) = 1$
- n não possui fatores repetidos e $p - 1 \mid n - 1$ para todos os fatores primos p de n

Problema de Lehmer. O problema envolve encontrar números compostos tais que $\Phi(n) \mid n - 1$. As soluções para esse problema devem ser números de Carmichael.

À primeira vista, o problema de Lehmer me pareceu mais complicado por me relembrar do Pequeno Teorema de Fermat e fazer menção ao $\Phi(n)$, então decidi implementar o critério de Korselt. Além da terceira afirmação do critério de Korselt, foi utilizada a seguinte propriedade dos números de Carmichael:

- Todos os números de Carmichael são ímpares

Algoritmo 6 Gerador de Números de Carmichael

```
#!/bin/bash
archive="carmichael.txt"
touch $archive
echo "Limite superior:"
read n
i=561

while (( $i <= $n )); do
  i=$(($i))
  echo "Numero : $i"


  # Verifica se possui fatores repetidos
  divisors=`factor $i|cut -d ':' -f 2`
  duplicate=$(echo "$divisors"|tr ' ' '\n'|uniq -d|wc -l)
  # $duplicate : quantas linhas duplicadas existem

  if (( $duplicate == 0 )); then
    line_count=$(echo "$divisors"|tr ' ' '\n'|wc -l)

    # Se factor retornar somente um numero, entao i e primo.
    # Como estamos interessados somente nos impares compostos,
       precisamos filtrar estes casos.
    if (( $line_count > 1 )); then
      ALL_DIVIDES="true"


      for p in $divisors; do
        if (( $(($i-1))%$(($p-1)) != 0)); then
          ALL_DIVIDES="false"
          break
        fi
      done

      if [ $ALL_DIVIDES = "true" ]; then
```

```
36              # TODOS os numeros de Carmichael possuem pelo menos 3
                    divisores
                n_divisors=$(echo $divisors|tr ' ' '\n'|wc -l)
                if [ $n_divisors -ge 3 ]; then
39                  echo "$i" >> $archive
                fi
            fi
        fi
42  fi

45  i=$i+2
done
```

Factor é um programa GNU que recebe um inteiro e retorna sua fatoração. Um detalhe que pode ser levado em conta é que o primeiro número de Carmichael é 561. Logo, fazendo $i = 561$ evitamos várias iterações desnecessárias quando o limite superior é inferior a 561.