Algoritmo AKS
Primalidade de um Número em Tempo Polinomial

Bruno da Rocha Braga
Ravel / COPPE / UFRJ
brunorb@ravel.ufrj.br
http://www.ravel.ufrj.br/

11 de Setembro, 2002

Resumo

Os algoritmos para determinação de primalidade são importantes na obtenção de números primos muito grandes, usados na confecção de chaves privadas de encriptação. Atualmente dispõe-se de algoritmos probabilísticos que executam em tempo polinomial e acusam se um número é primo com baixíssimo percentual de erro. O AKS é o primeiro algoritmo determinístico a executar este teste em tempo polinomial. Neste artigo, discutimos as bases matemáticas deste algoritmo e apresentamos uma implementação na linguagem do Maple.

1. Introdução

Os primeiros algoritmos criados para testar a primalidade de um número remontam a Grécia antiga. Até 2002, os principais algoritmos desenvolvidos para esta finalidade enquadravam-se em duas grandes classes:

- De tempo *não-polinimial* e *determinísticos*: Afirmam com 100% de certeza a primalidade de um número, mas o cálculo é realizado em tempo exponencial. Exemplos: Crivo de Eratóstenes, Adleman-Rumely ($(\log n)^{O(\log \log \log n)}$).
- De tempo *polinomial* mas *não-determinísticos*: A complexidade do algoritmo é em função de um polinômio — o tempo de cálculo não 'explode' quando o número testado é muito grande — mas *não* dão certeza absoluta quanto à primalidade. Exemplos: Testes de Monte Carlo.

O grande desafio era, portanto, obter um algoritmo de complexidade polinomial e determinístico. Caso este algoritmo não existisse, o problema do teste de primalidade seria da classe NP-completo. No entanto, o algoritmo AKS (Agrawal-Kayal-Saxena), apresentado no artigo "Primes is in P" é da classe P.

2. Pequeno Teorema de Fermat

Se p é primo, então para todo a inteiro em (1, p-1) temos:

$$a^p \equiv a \pmod p \therefore a^{p-1} \equiv 1 \pmod p$$

No entanto, este teste falha se *p* for um número de Carmichael (falsos primos).

2.1. Aritmética Modular com Polinômios

O objetivo desta seção é demonstrar, não rigorosamente, um teorema de congruência entre polinômios em aritmética modular, a saber: $(x-a)^p \equiv (x^p-a) \pmod p$. Este é a idéia básica do algoritmo.

Seja f(x) e p(x) polinômios tal que f(x) é divisível por p(x). Então, temos que f(x) ≡ 0 (mod p(x)) tal como faríamos em aritmética modular de inteiros. Vejamos um exemplo que será usado no algoritmo a seguir:

Seja p=7, a=2 e r=3 em

$(x^p - a) \pmod{x^r - 1} \therefore (x^7 - 2)/(x^3 - 1) = (x^4 + x)$ resto $(x - 2)$
$(x - 2) \pmod 7 = \underline{5+x}$

Agora vejamos o outro polinômio:

$(x - a)^p \pmod{x^r - 1} \therefore$ expansão binomial: $1 + px^1/1! + p(p-1)x^2/2! + \ldots$
$\therefore (-128+448x-672x^2+560x^3-280x^4+84x^5-14x^6+x^7)/(x^3-1) = x^4-14x^3+84x^2-279x+546$
resto $(418 + 169x - 588x^2) \pmod 7 = \underline{5+x}$

Este resultado é válido sempre que *p* é primo.

Assim, para se testar a primalidade de um número *p*, bastaria demonstrar a congruência a cima para todo *a* que não divide *p*. No entanto, tal operação pode consumir um tempo exponencial e, para contornar este problema, é proposto realizar o teste de congruência primeiro em módulo (x^r-1) e depois em módulo *p* — denotado por $(x-a)^p \equiv (x^p-a) \pmod{x^r-1,\ p}$ — graças ao resultado experimental obtido a cima.

Esta segunda congruência é válida para todos os primos *p* e valores de *a* e *r*. Porém, é também satisfeita para alguns *p* não-primos para alguns valores de *a* e *r*. Por isso, ainda temos que testar muitos valores para *a*. Os autores do AKS afirmaram em seu artigo que a escolha adequada do valor de *r* permite a validação da primalidade de *p* com a menor quantidade possível de testes da congruência. E propuseram que um valor de *r* será adequado se *r*-1 contiver um fator primo maior ou igual a $r^{½+\delta}$ para algum $\delta > 0$.

Battacharjee [BP01] diz que se $(z+1)^n \equiv (z^n+1) \pmod{z^r+1,\ n}$ onde *r* é primo e não divide *n*, então $n^2 \equiv 1 \pmod r$. Os autores do AKS também mostraram que para um grande número de pares (n, r), sempre que r é primo e n é composto então $n^2 \equiv 1 \pmod r$. Se esta conjectura for verdadeira, então este é um algoritmo de teste de primalidade com tempo polinomial muito simples.

Veremos adiante como estes resultados foram aplicados no algoritmo AKS.

3. Teste de Monte Carlo (Miller-Rabin)

É um algoritmo para teste de primalidade em tempo polinomial mas não-determinístico, ou seja, para determinar se um número *p* grande tem uma probabilidade P de ser primo. Para tanto, escolhe-se aleatoriamente um número *r* no intervalo [2, p-1] e aplica-se dois testes:

1. $r^{p-1} \neq 1 \pmod{p}$
2. para algum inteiro k, $1 < mdc(r^{(p-1)/2k} - 1, p) < p$

O primeiro teste é baseado no Pequeno Teorema de Fermat, enquanto o segundo procura achar um fator comum entre dois números, sendo um deles *p*, o que implicaria *p* ser composto. Se *r* for aprovado em ambos os testes, então *p* é composto. Mais da metade dos números do intervalo estão neste caso.

Caso contrário, se um número *r* não passa no teste, então *p* pode ser primo com probabilidade P=50%. Neste caso, escolhe-se um novo *r* e aplica-se o teste também. Se o novo *r* também não passa no teste, a probabilidade de que *p* seja primo sobre para 75%; ou seja, $P(m) = Sum(1/2^i, i=1..m)$ onde m é a quantidade de valores de *r* que não passaram no teste até então.

Por exemplo, em um teste para o número 5991, este é indicado como primo com apenas uma iteração (50% de certeza). Ao fazermos uma nova iteração, descobrimos que ele na verdade se trata de um composto. Em outro teste para o mesmo número, com valores diferentes de *r*, precisamos de 5 iterações para concluirmos que 5991 é composto. Isso ocorre porque *r* é escolhido aleatoriamente, e é possível escolhermos 1 ou mais valores que nos forneça um falso resultado. Felizmente, para 10 iterações em que um número *p* não passe no teste, temos 99,9% de certeza de que *p* é primo.

Após a publicação deste teste, diversos outros testes polinomiais mas não-determinísticos foram inventados e tornando esta classe de testes o principal meio de verificar a primalidade de um número.

4. O algoritmo AKS

No teste de Monte Carlo, se testarmos todas as possibilidades de *r*, poderemos afirmar se um número é primo. No entanto, isso é inviável para *p* grande. Logo, se fosse possível achar

um sub-conjunto de valores para *r* tal que aplicados ao teste nos fornecesse uma resposta certa, então teríamos um teste determinístico em P. É isso que faz o algoritmo AKS.

```
Input: Integer n > 1
 1. if (n = a^b with b > 1) then output COMPOSITE;
 2. r := 2;
 3. while (r < n) {
 4.      if (gcd(n,r) is not 1) then output COMPOSITE;
 5.      if (r is prime greater than 2) then {
 6.            let q be the largest factor of r-1;
 7.            if (q > 4sqrt(r)log n) and not(n^((r-1)/q) ≡ 1 (mod r)) then
 8.                  break;
 9.      r := r + 1;
10. }
11. for a := 1 to 2sqrt(r)log n {
12.      if not( (x-a)^n ≡ (x^n-a) (mod x^r-1,n) ) then output COMPOSITE;
13. }
14. output PRIME;
```

As linhas de 1 a 10 funcionam como um filtro para descartar muitos valores que não influem na decisão de primalidade, dados certos princípios algébricos já conhecidos.

Na primeira linha, é necessário aplicar um algoritmo que detecte potências perfeitas em tempo polinomial.

A linha 4, verifica se o máximo divisor comum de dois inteiros *n* e *r* é diferente de 1. Note que se *n* for primo, mdc(*n*, *r*) = 1 para qualquer $r < n$. Já na linha 5, é preciso determinar se o número *r* é primo por força bruta (todos os valores até sua raiz quadrada), pois se for usado algum algoritmo probabilístico (comum nas linguagens matemáticas como Maple e Matlab), então estaremos tornando o AKS também probabilístico.

A linha 6 determina o maior fator primo de *r-1* (q) sendo *r* primo.

Na linha 12, vemos a aplicação do Pequeno Teorema de Fermat. Considere C(n, p) = n! / p!(n-p)!, *n* primo diferente de 2 e *a* co-primo de *n.* Então, temos que:

$$(x-a)^n = -C(n,0)x^0a^n + C(n,1)x^1a^{n-1} + \ldots - C(n,n-1)x^{n-1}a^1 + C(n,n)x^na^0 =$$
$$= -a^n + C(n,1)x^1a^{n-1} + \ldots - C(n,n-1)x^{n-1}a^1 + x^n$$

Como C(n, i) ≡ 0 (mod n) (divisível por n) para $i \in [1, n-1]$, temos em módulo n que:

$$(x - a)^n \equiv -a^n + x^n \equiv (x^n - a^n) \equiv (x^n - a) \pmod n$$

Já se *n* for composto, então ele terá um fator primo *q*. Considere q^k a maior potência de *q* que divide *n* ($n = rq^k$). Então, temos que o binômio de x^i quando $i = q^k$:

$$C(n, i) = n!/i!(n-i)! = (q^kr)!/q^k!(q^kr-q^k)! =$$
$$= (q^kr)(q^kr-1)!/(q^k)(q^k-1)!(q^kr-q^k)! = (r)(q^kr-1)!/(q^k-1)!(q^kn-1)!$$

E este binômio não é divisível por $n=rq^k$ (igual a zero módulo n) e, por isso, *n* é composto. Isto é facilmente verificável dividindo-se os binômios de uma linha *n* do Triângulo de Pascal por *n* e constatando que todos são divisíveis se e somente se *n* é primo.

```
0: 1
1: 1  1
2: 1  2   1
3: 1  3   3   1
4: 1  4   6   4   1
5: 1  5  10  10   5   1
6: 1  6  15  20  15   6   1
7: 1  7  21  35  35  21   7   1
8: 1  8  28  56  70  56  28   8  1
9: 1  9  36  84 126 126  84  36  9  1
10:1 10  45 120 210 252 210 120 45 10 1
```

Os binômios da linha n são divisíveis por n quando este é primo.

O problema desta abordagem é que quando *n* for muito grande, calcular todos os *n*-1 binômios — o pior caso, $C(n, n-1)\neq 0 \pmod n$ — será muito custoso. Alterando a congruência para módulo x^r-1, implicitamente, faz-se o teste apenas para os *r* últimos termos de $(x - a)^n$, ou seja, $i \in [n-r, n-1]$.

No entanto, agora, alguns compostos *n* podem equivocadamente satisfazer a congruência da linha 12 para alguns valores de (a, r) e, por isso, são testados separadamente na linha 2.

O algoritmo primeiro escolhe um *r* primo para obter q, o maior fator primo de *r*-1, tal que este r delimita um intervalo onde certamente haverá um fator primo de *n* se este for composto. Em seguida, o algoritmo testa a congruência para $a \in [1, 2r^{1/2}\log(n)]$, uma quantidade de testes realizável em tempo polinomial no pior caso. Este é justamente o avanço feito por este algoritmo, explicado no artigo original.

5. Conclusão

A descoberta do algoritmo AKS tem apenas relevância matemática visto que, apesar de ser um algoritmo de tempo polinomial e determinístico (classe P), na prática, leva muito mais tempo para retornar a resposta que os testes de Monte Carlo — tal como o Miller-Rabin que precisa de no máximo 10 iterações para fornecer um parecer 99% confiável. Com isso, a única conseqüência desta descoberta foi provar que o problema de determinação da primalidade de um número pertence à classe polinomial já que, com certeza, testes de Monte Carlo continuarão a ser aplicados em softwares criptográficos por terem um desempenho superior.

6. Implementação no Maple

O objetivo desta implementação é apenas demonstrar o algoritmo AKS em ação, não tendo sido realizada com o rigor necessário, já que este código possui pelo menos duas falhas:

- Não foi aplicado o teste de potência perfeita da linha 1 do algoritmo original.
- Não foi aplicado um teste de primalidade por força bruta na linha 5 do algoritmo original, pois *isprime()* do Maple é probabilístico.

```
with(numtheory):

aks := proc (n)
 local r, fs, q, a;

 r := 2;
 while r < n do
  if igcd(n, r) <> 1 then RETURN(false); fi; # n is composite
  if isprime(r) then
   if r = 2 then q := 2; else fs := factorset(r-1); q := fs[-1]; fi;
   if q >= simplify(4.0*sqrt(r)*log(n)) and (n^((r-1)/q) mod r) <> 1 then
    break;
   fi;
  fi;
  r := r + 1;
 od;

 for a from 1 to simplify(2.0*sqrt(r)*log(n)) do
  if ((rem((x-a)^n, x^r-1, x) mod n)<>(rem(x^n-a, x^r-1, x) mod n)) then
   RETURN(false); # n is composite
  fi;
 od;

 RETURN(true); # n is prime
end:

# Código para chamar o algoritmo e comparar com o probabilístico
# nativo da linguagem do Maple:

y:=194;
t0 := time(): aks(y); time() - t0;
t0 := time(): isprime(y); time() - t0;
```