Primes in P - Der AKS Algorithmus

Christian Kühl

30. Juni 2011

1 Einleitung

Die Menge der Primzahlen spielt in vielen Bereichen eine wichtige Rolle. Deshalb werden die Eigenschaften von Primzahlen seit vielen Jahrzehnten genau analysiert. Insbesondere die Eigenschaften, welche es ermöglichen schnell zu berechnen ob eine gegebene Zahl eine Primzahl ist. So gibt die Definition der Primzahlen bereits einen einfachen Ansatz zur Bestimmung vor. Man teilt eine gegebene Zahl n durch alle Zahlen $m \leq \sqrt{n}$ und wenn eine solche Zahl m ein Teiler von n ist kann es sich um keine Primzahl handeln. Jedoch ist dieser Primzahlentest nicht effizient, denn der Test benötigt $\Omega(\sqrt{n})$ Zeit. Da das n exponentiell groß in der Eingabe sein kann, würde dies eine exponentielle Laufzeit ergeben. Man benötigt also einen deterministischen Algorithmus mit logarithmischer Laufzeit.
Seit den Anfängen der Komplexitätstheorien wurde dieses Problem intensiv untersucht. Jedoch gab es lange Zeit nur randomisierte Algorithmen für dieses Problem.
Bei dem später vorgestellten Algorithmus handelt es sich um einen deterministischen und polynomiellen Algorithmus, wodurch das Problem in der Komplexitätsklasse P liegt.

2 Grundlagen

2.1 Fermats kleiner Satz

Der kleine fermatsche Satz, kurz der kleine Fermat, ist ein Lehrsatz der Zahlentheorie. Er macht eine Aussage über die Eigenschaften von Primzahlen.

$a^p \equiv a(\text{mod p})$

Wobei a eine ganze Zahl, p eine Primzahl und ggT(a,p)=1 ist.

Der Algorithmus basiert auf einer Verallgemeinerung des kleinen fermatischen Satzes für polynomielle Ringe über endliche Felder.

2.2 Lemma

Sei $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$ und ggT(a,n)=1, dann ist n eine Primzahl, wenn folgende Gleichheit erfüllt ist:

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

Beweis:
Für $0 < i < n$ ist der Koeffizient von x^i in $((X + a)^n - (X^n + a)$ $\binom{n}{i} a^{n-i}$.
Angenommen n ist eine Primzahl, dann gilt $\binom{n}{i} = 0$ und somit sind alle Koeffizienten gleich 0 und die Gleichung ist erfüllt.
Angenommen n ist eine zusammengesetzte Zahl, dann wählen wir eine Primzahl q, die ein Teiler von n ist. Sei q^k die größte Zahl, die n teilt, dann kann q^k nicht $\binom{n}{q}$ teilen und ist teilerfremd zu a^{n-q}. Folglich ist der Koeffizient von $X^q \neq 0$(mod n), wodurch auch $((X + a)^n - (X^n + a) \neq 0$ ist.

Mit Hilfe dieser Identität kann ebenfalls ein Primzahlentest implementiert werden. Dafür müssen im worst Case jedoch n Koeffizienten ausgewertet werden, dies benötigt $\Omega(n)$ Zeit. Um die Anzahl der Koeffizienten zu reduzieren kann man auf beiden Seiten zusätzlich modulo eines Polynoms der Form $X^r - 1$ rechnen. Dadurch entsteht folgende Gleichung:

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

Aus der Gleichung (1) ist ersichtlich, dass alle Primzahlen, für alle Werte von a und r, die Gleichung erfüllen. Jedoch gibt es jetzt auch einige zusammengesetzte Zahlen, die ebenfalls diese Gleichung für einige Werte von a erfüllen.

3 Der Algorithmus

Input: integer $n > 1$.

1. If ($n = a^b$ for $a \in \mathbb{N}$ and $b > 1$), output COMPOSITE.
2. Find the smallest r such that $o_r(n) > (\log n)^2$. [1]
3. If $1 < ggT(a, n) < n$ for some $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME.
5. For $a = 1$ to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ do
   if $((X + a)^n \neq X^n + a(mod X^r - 1, n))$, output COMPOSITE;
6. Output PRIME;

3.1 Theorem

Algorithmus gibt genau dann PRIME zurück, wenn n eine Primzahl ist.

3.2 Lemma

Wenn n eine Primzahl ist gibt der Algorithmus PRIME aus.

Beweis:
Wenn n eine Primzahl ist, dann kann der Algorithmus in Zeile 1 und Zeile 3

[1] Ordnung $o_r(n)$: Kleinste Zahl k, sodass $n^k = 1$(mod r) ist

nicht COMPOSITE ausgeben. Durch das Lemma 2.1 wissen wir, dass auch in Zeile 5 nicht COMPOSITE ausgegeben wird.

Nun muss noch das Gegenteil des Lemmas gezeigt werden.
Wenn Schritt 4 PRIME ausgibt, dann handelt es sich bei n um eine Primzahl, da ansonsten Schritt 3 einen nicht trivialen Faktor von n gefunden hätte. Deshalb muss nur der Fall betrachtet werden, dass der Algorithmus PRIME in Schritt 6 ausgibt.
Dafür müssen die Schritte 2 und 5 genauer betrachtet werden. Schritt 2 findet ein entsprechendes r und Schritt 5 wendet die Gleichung (2) $a = 1$ bis $a = \lfloor\sqrt{\phi(r)}\log n\rfloor$ an.
Somit muss gezeigt werden, das auch das r logarithmisch beschränkt ist.

3.3 Lemma

Es existiert ein $r \leq \max 3, \lceil(\log n)^5\rceil$ **sodass** $o_r(n) > (\log n)^2$

Beweis:
Für n=2 und r=3 sind alle Bedingungen trivialerweise erfüllt.
Also wählen wir $n > 2$. Auerdem definieren wir ein $B = \lceil(\log n)^5\rceil$ und $k = \lfloor(\log n)^2\rfloor$. Zusätzlich bestimmen wir r als die kleinste Zahl, welche das Produkt

$$n^{\lfloor\log B\rfloor} * \prod_{i=1}^{\lfloor(\log n)^2\rfloor}(n^i - 1)$$

teilt. Nun benötigen wir ein zusätzliches Lemma.

3.3.1 Lemma

Sei KGV(m) das kleinste gemeinsame Vielfache von 1. . . m. Dann gilt für $m \geq 7$ **folgende Gleichung:**

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

Mit Hilfe dieses Lemmas können wir nun fortfahren und die Zahl r etwas abschätzen.

$$\prod \geq KGV(r-1) > 2^{r-1}$$

Da die Aussage allein uns nicht weiterhilft benötigen wir noch folgende Aussage:

$$n^{\lfloor\log B\rfloor} * \prod_{i=1}^{\lfloor(\log n)^2\rfloor}(n^i - 1) < n^{\lfloor\log B\rfloor + \frac{1}{2}(\log n)^2 * ((\log n)^2 - 1)} \leq n^{(\log n)^4}$$
$$\leq 2^{(\log n)^5} \leq 2^B$$

Fasst man nun beide Aussagen zusammen erhält man folgenden Zusammenhang:

$$2^B > \prod > 2^{r-1} \rightarrow B > r - 1 \rightarrow B \geq r$$
$$r \leq (\log n)^5$$

4 Laufzeitanalyse

Einschub:
Die Schreibweise $\mathcal{O}\tilde{}(t(n))$ bedeutet $\mathcal{O}(t(n) * poly(\log t(n))$, wobei t(n) irgendeine Funktion in Abhängigkeit von n ist. Zum Beispiel: $\mathcal{O}\tilde{}(log^k n)$= $\mathcal{O}(log^k n * poly(\log\log n))$

Bei der Berechnung der Laufzeit wird angenommen, dass Addition, Multiplikation und Division zwischen zwei m Bit großen zahlen in $\mathcal{O}(m)$ durchführbar ist. Die benötigte Zeit um diese Operationen zwischen 2 Polynomen mit Grad d durchzuführen beträgt $\mathcal{O}\tilde{}$(dm)

4.1 Theorem

Die asymptotische Laufzeit des Algorithmus beträgt $\mathcal{O}\tilde{}((\log n)^{\frac{21}{2}})$

Beweis:
Die 1.Zeile des Algorithmus benötigt $\mathcal{O}\tilde{}((\log n)^3)$ Zeit.
In Zeile 2 wird ein r mit $o_r(n) > (\log n)^2$ gesucht. Dies kann gefunden werden indem man nacheinander für alle r's überprüft, ob $n^k \neq 1$ (mod r) gilt. Das bedeutet für ein r müssen maximal $\mathcal{O}((logn)^2)$ viele Multiplikationen modulo r durchgeführt werden. Dies benötigt somit $\mathcal{O}\tilde{}((\log n)^2 \log r)$ Zeit. Nach Lemma 3.2 wissen wir, dass es maximal $\mathcal{O}((\log n)^5)$ unterschiedliche r's geben kann. Somit ergibt sich eine Laufzeit von $\mathcal{O}\tilde{}((\log n)^7)$ für Schritt 2.
Die 3. Zeile des Algorithmus berechnet den ggT für r Zahlen. Eine ggT Berechnung benötigt $\mathcal{O}(\log n)$ Zeit, wodurch die Gesamtzeit $\mathcal{O}(r \log n) = \mathcal{O}((\log n)^6)$ beträgt.
Die 4. Zeile benötigt lediglich $\mathcal{O}(\log n)$ Zeit.
Die 5. Zeile berechnet $\lfloor\sqrt{\phi(r)}\log n\rfloor$ Gleichungen. Jede Gleichung benötigt $\mathcal{O}(\log n)$ Multiplikationen von Polynomen mit Grad r und Koeffizienten der Größe $\mathcal{O}(\log n)$. Somit kann jede Gleichung in $\mathcal{O}\tilde{}(r(\log n)^2)$ berechnet werden. Daraus ergibt sich eine Laufzeit von $\mathcal{O}\tilde{}(r\sqrt{\phi(r)}(\log n)^3) = \mathcal{O}\tilde{}(r^{3/2}(\log n)^3))$ $= \mathcal{O}\tilde{}((\log n)^{\frac{21}{2}})$.

Die letzte Zeit dominiert alle anderen Zeiten wodurch die Gesamtlaufzeit des Algorithmus $\mathcal{O}\tilde{}((\log n)^{\frac{21}{2}})$ beträgt.

Literatur

[AB03] M. Agrawal and S. Biswas. Primality and identity testing via chinese remaindering. http://www.cse.iitk.ac.in/users/manindra/algebra/identity.pdf, 2003.

[MAS03] N. Kayal M. Agrawal and N. Saxena. Primes is in p. http://www.cse.iitk.ac.in/users/manindra/algebra/primalityv6.pdf, 2003.