Referat Algorithmische Anwendungen
WS 06/07

Primzahlfaktorisierung

Team C_gelb_ALA0607

Inga Feick, 11034165, inga.feick@web.de
Marc Kalmes, 11025526, ai233@gm.fh-koeln.de

23.01.2007

Problemstellung

Dieses Referat befasst sich mit der Faktorisierung, also der Zerlegung von Primzahlen.
Eine Primzahl ist eine natürliche Zahl, die außer durch die Zahl eins nur durch sich selbst ohne Rest teilbar ist, wobei die Zahlen Null und Eins weder prim noch unprim sind. Die kleinste Primzahl ist 2, die größte Primzahl ist unbekannt.

Primzahlen finden Anwendung in der Krypthografie. Viele moderne Verschlüsselungsverfahren beruhen auf der Tatsache daß es in sehr kurzer Zeit möglich ist, zwei sehr große Primzahlen, z.B. mit einer Länge von mehreren hundert Ziffern, zu multiplizieren, es auf der Gegenseite jedoch nahezu unmöglich ist, die Faktoren dieser Multiplikation in annehmbarer Zeit wieder zurückzurechnen.
Die Faktorisierung von Zahlen mit einer Länge um hundert Ziffern ist heutzutage zwar mit geringem Aufwand möglich, jedoch sind solch kurze Zahlen für die Kryptografie nicht relevant. Die aktuellen Faktorisierungsversuche bewegen sich im Feld der Zahlen von ca 200 Zeichen Länge.
Es ist nicht bewiesen, daß es kein Verfahren gibt, welches große Zahlen effizient faktorisieren kann. Möglicherweise gibt es einen noch nicht entdeckten Algorithmus zur Lösung des Problems. Würde ein solches, schnelles Verfahren gefunden, dann wären schlagartig die relevantesten heutzutage genutzten Verschlüsselungsmechanismen wertlos. Deswegen forschen Mathematiker und Krypthografieexperten aktuell und anhaltend auf diesem Gebiet und versuchen, ein schnelles Verfahren zu entwickeln. Solange keines gefunden wird, dürfen die bekannten Verschlüsselungsmethoden als sicher gelten.
Die Erfinder des Public-Key-Verschlüsselungsverfahren RSA haben zu diesem Zweck einen öffentlichen Wettbewerb ausgerufen und eine Liste von acht Zahlen bekannt gegeben, für deren Faktorisierung Preisgelder von bis zu 200,000 US-Dollar ausgesetzt sind. Bisher sind erst die beiden kleinsten Zahlen mit einer Länge von knapp unter 200 Dezimalstellen geknackt worden.

Methoden zum Erkennen von Primzahlen

Wenn eine besonders große Primzahl benötigt wird, dann ist eine zeitsparende Vorgehensweise die, daß man eine fast beliebige Zahl nimmt und überprüft, ob diese zufällig prim ist, und dann ggf. weiterverwendet. Dies funktioniert wesentlich schneller, als im sehr hohen Zahlenbereich gezielt Primzahlen ausrechnen zu lassen.
Bekannte Primzahltests sind:

- der Fermatsche Primzahltest. Er ist relativ langsam und arbeitet nur auf ungeraden Zahlen. Für diese kann er korrekt ermitteln, ob eine Zahl nicht prim ist. Er kann jedoch nicht mit 100% Sicherheit herausfinden, ob eine Zahl prim ist, da er auch Pseudoprimzahlen als Primzahlkandidaten erkennt.
- Miller-Rabin-Test. Auch dieser ermittelt zuverlässig die Nichtprimalität einer Zahl. Bezüglich der Primalität einer Zahl liefert er eine geringe Fehlerwahrscheinlichkeit, weil er anfällig für die sogenannten „starken Pseudoprimzahlen“ ist. Der Miller-Rabin-Test ist ein probalistischer Test, d.h. daß er nicht deterministisch arbeitet und dadurch wesentlich effizienter ist.
- Ein weiterer probalistischer und ähnlich funktionierender Test ist der Solovay-Strassen-Test. Genau wie der Miller-Rabin-Test gehört er zur Klasse der Monte-Carlo-Algorithmen. Dies sind Algorithmen, die mit einer Zufallskomponente arbeiten und ein erlaubtes Maß an fehlerhaften Ergebnissen liefern dürfen. Im Ausgleich dazu sind sie wesentlich effizienter als deterministische Algorithmen und oft auch einfacher zu implementieren.
  Auch der Solovay-Strassen-Test liefert unter Umständen die Aussage, daß eine Zahl prim sei, obwohl sie es nicht ist. Genau wie beim Miller-Rabin-Test kann diese Fehlerwahrscheinlichkeit durch wiederholte Anwendung des Tests mit variierender Parametrisierung auf ein für heutige praktische Zwecke tolerierbares Niveau gesenkt werden.
- Probedivision. Die Probedivision teilt die zu prüfende Zahl durch bekannte kleine Primzahlen und stellt einene Brute-Force-Ansatz dar. Sie ist zuverlässig, aber ineffizient.

Pseudoprimzahlen

Eine Gruppe zusammengesetzter Zahlen wird beim Primalitätstesten fälschlicherweise als prim erkannt, die sogenannten Pseudoprimzahlen. Sie weisen einigen oder viele, aber nicht alle mathematische Charakteristika von echten Primzahlen auf.
Es gibt verschiedene Arten von Pseudoprimzahlen mit unterschiedlichen Eigenschaften. Für Primzahltests besonders relevant sind hierbei die Fermatschen Pseudoprimzahlen, deren Untergruppe der Carmichael-Zahlen sowie die starken Pseudoprimzahlen.

Mit den durch Pseudoprimzahlen hervorgerufenen fehlerhaften Primalitätstestergebnissen sowie den daraus folgenden Implementationshürden werden wir uns im späteren Verlauf des Praktikums beschäftigen.

Faktorisierung

Eine natürliche Zahl ist ein Produkt mehrerer Primzahlen. Die Zerlegung einer nicht-primen Zahl in ihre Primzahlbestandteile wird als Faktorisierung bezeichnet und stellt die Umkehroperation der Multiplikation dar. Die Primzahlfaktoren der Zahl 12 sind zum Beispiel 2*2*3, oder 2^2*3 in der kanonischen Darstellung.

Wenn eine Zahl faktorisiert wird, dann geschieht dies in mehreren algorithmischen Schritten.
Zuerst wird überprüft, ob die Zahl prim ist. Eine prime Zahl muss nicht weiter untersucht werden. Dabei kommt einer der vorhin erwähnten Primzahltests zum Einsatz.

Als nächstes wird bei der Probedivision die zu faktorisierende Zahl n durch die Primzahlen von 2 bis Wurzel n geteilt. Stellt sich eine der Primzahlen als Teiler heraus, dann kann mit der verkleinerten Zahl weitergerechnet werden, was Effizienzgewinne bringen kann.

An dieser Stelle werden ggf. weitere Primfaktoren mit Hilfe der Methode der elliptischen Kurven ermittelt und entfernt.

Als nächstes und letztes kommt das eigentliche Faktorisierungverfahren zum Einsatz.
Folgende Möglichkeiten werden wir betrachten:

- Das quadratische Sieb eignet sich für Zahlen mit einer Länge unter 110 Dezimalstellen. Für größere Zahlen ist das Zahlkörpersieb geeignet. Davon gibt es mehrere Varianten:
    - das allgemeine Zahlkörpersieb, entwickelt um 1988
    - das spezielle Zahlkörpersieb, eine Weiterentwicklung des allgemeinen Zahlkörpersiebs.
- Die Methoden von Pollard
    - Pollards Rho-Methode
    - Pollards p-1-Methode

Wir haben uns zur Implementation der Faktorisierungsmethoden Pollard p-1 und Quadratisches Sieb entschlossen.

Primalitätstests

Fermat

Der Fermatsche Primzahltest beruht auf einer simplen mathematischen Annahme, dem Fermatschen Satz: Wenn eine natürliche Zahl n prim ist, dann gilt für alle Zahlen a aus dem Intervall [1,n-1] die Aussage:

$$a^{n-1} \equiv 1 \bmod n$$

Das Testverfahren wählt also eine zufällige Zahl a zwischen 1 und n-1 aus und potenziert diese mit n-1, modulo n. Ist das Ergebnis 1, dann liegt ein Primzahlkandidat vor. Liegt ein beliebiges anderes Ergebnis vor, dann kann die geprüfte Zahl nicht prim sein.

```
public Boolean fermat(int n) {
            Boolean result = false;
            int rand = zufallszahl(2, n - 1);
            int mod = modpow(rand, n - 1, n);
            if (mod == 1) {
                  result = true;
            }
            return result;
      }
```

Schwächen des Fermatschen Tests

1. Pseudoprimzahlen

Fermatsche Pseudoprimzahlen, darunter auch die Carmichael-Zahlen, erfüllen die oben beschriebene mathematische Aussage, obwohl sie zusammengesetzt sind. Dies führt zu fehlerhaften Ergebnissen, denn nicht-prime Zahlen werden als potentiell prim erkannt. Fermat'sch geprüfte Zahlen mit positivem Ergebnis können daher nur als Primzahlkandidaten, nicht aber als sichere Primzahlen verstanden werden.

2. Exponentiation

Die Berechnung von $a^{n-1} \bmod n$ führt schnell zu Leistungsproblemen. Berechnet man die Potenz auf nativem Wege, dann stoßen herkömmliche Rechner schnell an ihre Grenzen, z.B bei 7stelligen Exponenten. Abhilfe schafft hier die modulare Exponentiation. Sie ermöglicht die Berechnung großer Potenzen innerhalb von kürzester Zeit. Eine sehr effiziente und rechenarme Variante ist die Berechnung auf binärer Basis.
Hierbei wird zur Berechnung von x^e der Exponent e als binäre Zahl der Länge n dargestellt, und jedes e_i wird mit x_i in der Form

$$x^e = (x^{2^0})^{e_0} * (x^{2^1})^{e_1} * (x^{2^2})^{e_2} * \ldots * (x^{2^n})^{e_n}$$

multipliziert. Diese Vorgehensweise liegt in $O(2 \log_2 e)$. Eine leicht veränderte Implementation, die direkt modulo m rechnet, benutzen wir in unserer Implementation:

Implementationsbeispiel modulare Exponentiation:

```
public int modpow(int b, int e, int m) {
            int result = 1;
            while (e > 0) {
                  if ((e & 1) == 1) {
                        result = (result * b) % m;
                  }
                  e >>= 1;
                  b = (b * b) % m;
            }
            return result;
      }
```

Anwendungsbeispiel modulare Exponentiation:

```
Beispiel: Berechne modpow für 2^3 % 3
b is 2
result is 1
loop 0
e is 3
e is odd
result is ( result 1 * b 2) % m 3 => 2
b is (b 2 * b 2 ) % m 3 => 1
loop 1
e is 1
e is odd
result is ( result 2 * b 1) % m 3 => 2
b is (b 1 * b 1 ) % m 3 => 1
final result  => 2
```

Stärken des Fermatschen Tests

Der Fermat-Test erkennt zuverlässig jede Primzahl als prim. Zu Aussagen über die gegebene Primalität einer Zahl lässt er sich jedoch nicht nutzen, da unter den erkannten Primzahlen auch Pseudoprimzahlen sein können. Im Umkehrschluss kann man jedoch sagen, daß der Fermat-Test zuverlässig jede Nicht-Primzahl als nicht-prim erkennt. Er eignet sich daher besonders als Nachweis der Zusammengesetztheit einer Zahl und wird im Vorfeld einer Faktorisierung angewendet, um sicherzustellen, daß die zu zerlegende Zahl zerlegbar ist.

Solovay-Strassen

Der Solovay-Strassen-Primalitätstest ist ein nicht deterministischer, recht effizienter Algorithmus der Monte-Carlo-Klasse. Monte-Carlo-Algorithmen sind randomisierte Algorithmen, die mit einer nach oben beschränkten Wahrscheinlichkeit ein falsches Ergebnis liefern dürfen. Dafür sind sie im Vergleich zu deterministischen Algorithmen häufig effizienter. Ihr Nachteil besteht darin, dass das berechnete Ergebnis falsch sein kann. Durch Wiederholen des Algorithmus mit unabhängigen Zufallsbits kann jedoch die Fehlerwahrscheinlichkeit auf ein tolerierbares Maß gesenkt werden Solovay-Strassen ist konkret fehleranfällig gegenüber Eulerschen Pseudoprimzahlen .

Die mathematische Grundlage dieses Algorithmus bilden der Eulersche Satz und das Jacobi-Symbol.

Der Satz von Euler sagt aus, daß $a^{\varphi(n)} \equiv 1 \bmod n$ für n $\in \mathbb{N}$ und ggt(a,n) = 1. Phi (φ) ist dabei die Eulersche Funktion, die zu jeder Zahl n angibt, wieviele natürliche Zahlen zwischen 1 und n zu n teilerfremd sind.
Das Jacobi-Symbol ist eine mathematische Kurzschreibweise und gibt an, ob eine Zahl ein quadratischer Rest modulo n ist. Für prime Zahlen gilt die Aussage

$$\left(\frac{a}{n}\right) = \begin{cases} 1 & \text{wenn } a \text{ ein quadratischer Rest zu } n \text{ ist} \\ -1 & \text{wenn } a \text{ kein quadratischer Rest zu } n \text{ ist} \\ 0 & \text{wenn } a \text{ und } n \text{ nicht teilerfremd sind} \end{cases}$$

Für nicht-prime Zahlen ist das Jacobi-Symbol das Produkt der Jacobi-Symbole der Primfaktoren, und für unsere Belange nicht relevant.

Vorgehensweise:

Wir wählen eine zufällige Zahl a zwischen 1 und n.
Sei g der größte gemeinsame Teiler von a und der zu testenden Zahl n, dann ist n nicht prim falls g >

1. Wenn g = 1 ist, dann wird $b = a^{(n-1)/2}$ berechnet. Wenn $b \in \{1, -1\}$ dann könnte n prim sein.

Nun wird das Jacobi-Symbol von a,n berechnet. Falls dieses äquivalent zu b ist, dann handelt es sich um eine Primzahl oder eine Eulersche Pseudoprimzahl.

Eine zusammengesetzte Zahl wird vom Solovay-Strassen-Test zuverlässig als komposit erkannt. Eulersche Pseudoprimzahlen können jedoch fälschlich als prim erkannt werden. Ebenso können prime Zahlen als nicht-prim interpretiert werden, wenn a unglücklich gewählt ist. Daher empfiehlt sich dieser Test nur als mehrmaliger Durchlauf mit verschiedenen Zufallszahlen.

Laufzeit: O ((log n)^3)

Anwendungsbeispiel Solovay-Strassen-Test:

```
SolovayStrassen (n = 3 )
n 3 ist prim: true

SolovayStrassen (n = 4 )
random 1 < a < n =  3
ggt (a,n) = ggt(3,4) = 1
b = a^((n-1)/2) %n = 3^(1.0) % 4= 3
n 4 ist prim: false

SolovayStrassen (n = 5 )
random 1 < a < n =  4
ggt (a,n) = ggt(4,5) = 1
b = a^((n-1)/2) %n = 4^(2.0) % 5= 1
jacobi(a,n) = 1
n 5 ist prim: true

SolovayStrassen (n = 8 )
random 1 < a < n =  3
ggt (a,n) = ggt(3,8) = 1
b = a^((n-1)/2) %n = 3^(3.0) % 8= 3
n 8 ist prim: false


SolovayStrassen (n = 561 )
random 1 < a < n =  377
ggt (a,n) = ggt(377,561) = 1
b = a^((n-1)/2) %n = 377^(280.0) % 561= 67
n 561 ist prim: false
```

Miller-Rabin

Der Miller-Rabin-Primalitätstest ist ein probalistischer Test, der ein nur bedingt gültiges Ergebnis liefert, im Gegenzug aber sehr schnell arbeitet. Er gehört damit zur Klasse der Monte-Carlo-Algorithmen. Fehleranfällig ist er insbesondere für „starke Pseudoprimzahlen“.
Der Miller-Rabin-Test wird heute z.B. in Java als Primalitätstest-Standard benutzt.

Vorgehensweise

Für eine zu testende Zahl n wird zuerst eine zufällige ganze, zu n teilerfremde Zahl a > 1 und a < n

gewählt. Desweiteren berechnen wir s := max { $r \in \mathrm{N} : 2^r \% (n-1) = 0$ }, und mit 2^s die größte Potenz von zwei, die n-1 teilt. Aus d := (n-1) / 2^s ergibt sich die ungerade Zahl d.
Falls n prim ist, dann gilt eine der nachfolgenden Aussagen:

entweder ist $a^d \equiv 1 \bmod n$ oder $\exists r \in \{0,1,...s-1\} \mid a^{2^r d} \equiv -1 \bmod n$

Tritt einer dieser beiden Fälle ein, dann ist n entweder prim oder stark pseudoprim. Tritt keiner der Fälle ein, dann ist n definitiv zusammengesetzt. Man spricht dann von der Zahl a als „Zeugen“ gegen die Primalität von n.

Die Laufzeit entspricht in der Theorie der des Solovay-Strassen-Tests, in der Praxis ist der Miller-Rabin-Mechanismus jedoch oftmals deutlich schneller.

Anwendungsbeispiel Miller-Rabin-Test:

```
Miller-Rabin für N = 5 (10 Versuche)
w = 1
Random a = 3
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  3^1 mod 5 = 3
 a ist Zeuge gegen n: false
w = 2
Random a = 2
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  2^1 mod 5 = 2
 a ist Zeuge gegen n: false
w = 3
Random a = 2
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  2^1 mod 5 = 2
 a ist Zeuge gegen n: false
w = 4
Random a = 1
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  1^1 mod 5 = 1
 a ist Zeuge gegen n: false
w = 5
Random a = 3
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  3^1 mod 5 = 3
 a ist Zeuge gegen n: false
w = 6
Random a = 3
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  3^1 mod 5 = 3
 a ist Zeuge gegen n: false
w = 7
Random a = 3
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  3^1 mod 5 = 3
 a ist Zeuge gegen n: false
w = 8
Random a = 2
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  2^1 mod 5 = 2
 a ist Zeuge gegen n: false
w = 9
Random a = 1
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  1^1 mod 5 = 1
 a ist Zeuge gegen n: false
w = 10
Random a = 3
t = 2
d =  N-1 % 2^s=  4%2^2 = 1
A^d mod N =  3^1 mod 5 = 3
 a ist Zeuge gegen n: false
N 5 ist prim: true
```

Probedivision

Die Probedivision ermittelt einen Teiler der zu suchenden Zahl n. Falls ein solcher Teiler nicht existiert, ist die Zahl eine Primzahl. Folgendes Theorem dient als Grundlage:
Theorem 1 "Wenn n eine zusammengesetzte Zahl ist, dann hat n einen Primteiler p, der nicht grösser als sqrt n ist"
Wäre nach dem Theorem n eine zusammengesetzte Zahl, kann man auch n = ab schreiben. a,b > 1, a,b <= sqrt n. a und b können nicht grösser als sqrt n sein, da sonst ab > sqrt n * sqrt n ungleich n wäre.

Theorem 2 "Jede natürliche Zahl a > 1 hat einen Primteiler"
Beweis. a besitzt einen Teiler der gösser ist als 1, nämlich a selbst. Unter allen Teilern von a > 1, sei p der kleinste. Somit muss p einen Primzahl sein, da sie sonst einen Teiler b der Form 1 < b < p <= a hätte.
Da jede natürlich Zahl einen Primteiler besitzt, müssen a und b auch einen Primteiler besitzen, die wiederum n teilen.

Will man also feststellen, ob n eine Primzahl ist, braucht man nur alle Primzahlen p < sqrt n zu finden die n teilen. Wenn dies nicht gelingt, ist n eine Primzahl. Die Primzahlen p < sqrt n kann man in einer Tabelle nachsehen.

Nun nehmen wir aber an, keine Primzahlen zu kennen. Nach obiger Definition können wir nicht bestimmen ob n prim ist. Um dies auch ohne eine Liste mit kleinen Primzahlen tun zu können, muss der Algorithmus angepasst werden. Verwendet man bei der Probedivision alle Zahlen von 2 bis sqrt n erhält man das selbe Ergebnis. Hierbei werden aber unnötige Divisionen durchgeführt. Eine weitere Anpassung ermöglicht eine schnellere Berechnung. Da Primzahlen ungerade sind, machen wir uns diesen Umstand zu nutze und dividieren nur durch ungerade Zahlen.

Pseudocode Probedivision:

```
Vorbedingung:
if n = 0 or n = 1
        then nicht_prim
if n = 2
        then prim
wenn n mod 2 = 1
        then probedivision(n)

probedivision(n)
divisor <-3
probaly_prim <- true
while divisor < sqrt n and probaly_prim
        do if divisor mod 2 = 1
                then if n mod divisor = 0
                        then probaly_prim <- false
        divisor <- divisor + 1
```

Die Probedivision benötigt im schlimmsten Fall etwa sqrt n, um alle ungerade Teiler zu finden und zu dividieren.

Vergleich der Primalitäts-Tests

Laufzeitanalyse

Test	Worst-Case Laufzeit	Average-Case Laufzeit	Fehleranfällig	Math. Grundlage
Probedivision			--	
Fermat			Fermatsche Pseudoprimzahlen	kleiner Fermatscher Satz
Solovay-Strassen			Eulersche Pseudoprimzahlen	Satz nach Euler, Jacobi-Symbol
Miller-Rabin	- (implementationsabhängig)		starke Pseudoprimzahlen	Satz nach Miller

Eignung

Fermat eignet sich nur für die Erkennung von zusammengesetzten Zahlen. Er ist leicht zu implementieren und, in Verwendung mit binärer Exponentiation, sehr effizient. Daher empfiehlt er sich als Überprüfung der Nichtprimalität vor der Dekomposition einer Zahl.

Der Solovay-Strassen-Test ist der erste entwickelte Primalitätstest (1976). Nichtprime Zahlen erkennt er zuverlässig, prime Zahlen hingegen nur mit einer gewissen Wahrscheinlichkeit. Für die heutige IT ist er ohne praktische Bedeutung.

Der Miller-Rabin-Test ist der heute als Standard verwendete Primalitätstest. Er kommt z.B. in der Java.math-Klasse für die Funktion isProbablyPrime zum Einsatz. Seine Implementation ist einfacher als die des Solovay-Strassen-Tests, und seine Ergebnisse sind um Längen besser.
Wie der Solovay-S.-Test kann auch er nichtprime Zahlen zuverlässig, prime Zahlen aber nur mit einer gewissen Wahrscheinlichkeit ermitteln, dies wiederum mit so hoher Trefferquote daß wir es in unseren Tests nicht geschafft haben, einen false positive zu beobachten.

Die Probedivision ist eine sichere Methode. Mit sqrt(n) Schritten für eine Zahl n ist sie allerdings für den praktischen Einsatz wenig geeignet und bietet sich nur für kleinere Zahlen an.

Zusammenfassend lässt sich sagen:

Für die Prüfung auf Primalität empfiehlt sich der Miller-Rabin-Test.
Für die Prüfung auf Nicht-Primalität (relevant für unsere Faktorisierungszwecke) empfiehlt sich der Fermat-Test oder der Miller-Rabin-Test.

Faktorisierungsverfahren

Fermatsche Faktorisierung

Von eher historischer Bedeutung für die heutige IT ist die Fermatsche Methode zur Primzahlzerlegung. Sie eignet sich jedoch gut um die Qualitätsunterschiede und die Fortschritte in der Verfahrensentwicklung zu demonstrieren.

Das Verfahren fußt auf der Annahme, daß es für jede zusammengesetzte Zahl n zwei Zahlen a und b gibt, deren Quadrate die Differenz n haben, so daß $a^2 - n = b^2$ und damit $n = a^2 - b^2$.
Nach der dritten binomischen Formel können wir dies umformen zu n = (a+b)(a-n).

Zunächst berechnen wir x = Aufrunden(Wurzel(n))
und als nächstes $r = x^2 - n$.
Der Algorithmus führt nun solange die Anweisungen

```
r += (2 * x) + 1
x += 1
```

aus, bis r ein Quadrat ist. Wenn ein Quadrat gefunden wurde, dann wird y = Wurzel(r) berechnet und es ergeben sich mit a = x + y sowie b = x – y zwei Teiler von n. Beide Teiler können wiederum zusammengesetzt werden und müssen dann rekursiv faktorisiert werden.

Anwendungsbeispiel Fermatsche Faktorisierung:

```
fermat(n =  8) {
x = 3,  r 1
y = 1, a = 4, b = 2
8 ist teilbar durch 4 und 2 ( 0 Schritte)
4 ist nicht prim, Rekursion!
fermat(n =  4) {
x = 2,  r 0
y = 0, a = 2, b = 2
4 ist teilbar durch 2 und 2 ( 0 Schritte)
Speichere 2
Speichere 2
Speichere 2
}
```

Quadratisches Sieb

Das Quadratische Sieb ist eines der populärsten und schnellsten Verfahren zur Faktorisierung von Zahlen. Erst vor wenigen Jahren wurde damit die RSA-Zahl 129 geknackt.

Im Gegensatz zum Fermatschen Ansatz versucht man hierbei nicht, eine Differenz von Quadraten zu faktorisieren. Anstelle von $x^2 - y^2 = n$ wird ein Zahlenpaar u,v gesucht, welches $x^2 - y^2 =$ ein beliebiges Vielfaches von n, also $x^2 - y^2 = n*m$ ist. Das Bestreben dieses Algorithmus ist es also, ein Paar x,y zu finden, für welches $x^2 \equiv y^2 \bmod n$ gilt. Auf diesem Wege entstehen zwei Lösungtypen: diejenigen, für die $x \equiv \pm y \bmod n$ sowie solche, bei denen x und y nicht kongruent sind. Letztere sind für unsere Problemstellung interessant, und mit a = gcd(x-y , n) kann ein echter Teiler von n ermittelt werden.

Zuerst wird eine Faktorbasis B generiert. Dies ist eine Folge der ersten t Primzahlen, also $B = \{p_1, p_2, p_3 \ldots p_t\}$ bzw $B = \{2,3,5,7..p_t\}$.

Als nächstes treffen wir eine Auswahl relativ beliebiger Ganzzahlen, die gewissen Bedingungen genügen müssen:
Sei $A = \{a_1, a_2, .. a_x\}$ dann muss für jedes a_i die Zahl $b_i = a_i^2 \% n$ berechnet werden, welche wiederum B-glatt sein muss. Glatt gegenüber einer Zahl x bedeutet, daß alle Primfaktoren von b_i kleiner oder gleich x sein müssen. x ist in diesem Fall die größte Primzahl in der Folge B. Ein Beispiel:
Sei $B = \{2,3,5\}$ und $a_i = 3$, n = 4, dann ist $b_i = 9 \% 5 = 4$. Die Faktoren von 4 sind 2^2, also ist b_i glatt gegenüber B.
Aus den verfügbaren a_i welche der Glättebedingung entsprechen wählen wir diejenigen aus, die ein perfektes Quadrat in Z bilden. Dazu stellen wir für jedes a eine Gleichung der Form $a_i^2 \equiv x_1^{e_1} * x_2^{e_2} \ldots . \bmod n$ auf. Wir faktorisieren a_i, z.B. durch Probedivision, und erstellen mit $e_1 \ldots e_m$ einen Vektor der Exponenten der Faktoren von a_i. Die erste Position der Faktorenliste sei dabei -1, und dessen Exponent sei 1 falls $a_i < 0$ und 2 falls $a_i >= 0$. Wenn alle Exponenten in einem Vektor gerade sind, dann handelt es sich um ein Quadrat.

Für ein solches a_i können wir $x = gcd(a_i - b_i, n)$ und $y = gcd(a_i + b_i, n)$ berechnen und haben dann mit x und y zwei Teiler von n gefunden.
Die Komplexität des Algorithmus liegt in der geeigneten Auswahl der Faktorbasisgröße t sowie der Zahlen A. Für die Auswahl von A hat sich gezeigt, daß es sinnig ist, m = Abrunden(Wurzel(n)) zu berechnen und keine Werte kleiner als m zu testen. Ein Beispiel:
Sei $B = \{2,3,5\}$, n = 3071. Dann ist m = 55. Wir beginnen also mit a = 56, dessen $b = 56^2 \% 3071 = 65$. 65 ist jedoch nicht glatt gegenüber B, denn $65 = 5 * 13$ und 13 ist größer als die größte Zahl in B. Die Faktorisierung von 65 findet an dieser Stelle standardmässig durch Probedivision statt, was aber je nach Implementation unterschiedlich sein kann.
Erst bei a1 = 96 und b1 = 3 ist b glatt, allerdings kein Quadrat. Das nächste a mit glattem b tritt bei a = 157 auf, mit b = 81. Wir haben nun 9 als Wurzel von b und können mit x = gcd(157 – 9, 3071) = 37 und y = gcd(157 + 9, 3071) = 83 zwei Teiler von n festlegen.

Performanz

Es empfiehlt sich, solche Primzahlen p aus der Basis B zu entfernen, für die n % p keine Quadratzahl ist. Zum Beispiel kann für 121 (11*11) die 7 aus der Basis {2,3,5,7,11} entfernt werden weil 121%7 = 2 und 2 ist kein Quadrat. Grundsätzlich gilt: je kleiner die Basis ist, desto performanter ist der Algorithmus. Es kann jedoch auch passieren, daß t zu klein gewählt ist, was dann gegenteiliges bewirkt und den Algorithmus im schlimmsten Fall langsamer als die Probedivision werden lässt.

Desweiteren unterscheidet man bei den verschiedenen Implementation dieses Algorithmus zwischen dem nicht-siebenden Sieb und dem siebenden Sieb. Letztgenannteres ist eine Weiterentwicklung der oben beschriebenen Vorgehensweise, welches ein Siebintervall in Form eines Zahlencontainers C mit den Indizes –M <= x <= M anlegt. Zur Erinnerung: M ist die abgerundete Wurzel aus n. Für jedes Element mit Index x wird der Array mit dem Wert Abrunden(Log_2 (Betrag(q(x)))) initialisiert, wobei q der Gleichung q(x) = 0 mod p entspricht und p eine beliebige ungerade Primzahl aus der Faktorbasis ist. Nun wird für jede ungerade p in B der folgende Schritt ausgeführt:
Für alle C_x deren $x = x_1 \bmod p$ oder $x = x_2 \bmod p$: subtrahiere C_x um h, wobei h = Abrunden(lg p)

Diejenigen Containerelemente, deren Wert sich am meisten an Null annähern, haben die höchste Wahrscheinlichkeit, daß ihre Indizes x B-glatt sind, so daß diese x bevorzugt als a-Werte getestet werden können.

Die Laufzeit des Quadratischen Siebs ist bis dato nicht exakt festgelegt. Je nach Quelle und Implementation vermutet man eine Worst-Case-Performanz zwischen polynomiell und exponentiell, letzteres insbesondere wenn die Probedivision zur Ermittlung der B-Glätte eingesetzt wird (die Probedivision liegt in exponentieller Laufzeit).
Für die oben beschriebene Implementation mit echtem Siebverfahren wird eine Laufzeit von

$O(e^{(1+o(1))\sqrt{\ln n}\sqrt{\ln \ln n}})$ vermutet, wobei o(1) eine Funktion ist, deren Wert gegen Null konvergiert wenn n gegen unendlich strebt.

Anwendungsbeispiel Quadratisches Sieb (echt-siebende Implementation):

```
Beginning factorisation with candidate = 21 and t1 = 4
Choosen Prime Base S =
      -1    2     5     17
m = rootOf(21) = 4
Data collect for the first 5 of x for which q(x) is 17-smooth
i     x     q(x)  ai    vi
1     0     -5    4     (1, 0, 1, 0)
2     1     4     5     (0, 0, 0, 0)
3     -2    -17   2     (1, 0, 0, 1)
4     -3    -20   1     (1, 0, 1, 0)
5     -5    -20   -1    (1, 0, 1, 0)

By inspection:  = 0
x = 5
l1 = 0, l2 = 1, l3 = 0, l4 = 0
y = 2
gcd(x - y, n) = 3

So two non-trivial factors of 21 are 3 and 7
```

Pollards p-1-Methode

Pollard's p-1 Methode ist ein Faktorisierungsverfahren, welches auf dem kleinen Satz von Fermat basiert. Dieses Verfahren eignet sich für zusammengesetzte Zahlen n, die einen Primfaktor p besitzen, der für p-1 nur kleine Primfaktoren hat.
Ohne p zu kennen lässt sich ein Vielfaches k von p-1 bestimmen. Es gilt: $a^k \equiv 1 \pmod{p}$.

Daraus folgt, dass $a^k - 1$ einen Faktor p besitzt. Falls sich aber $a^k - 1$ nicht durch n teilen lässt, so ist der ggT(a^k-1, n) ein echter Teiler von n und somit ist n faktorisiert.
k ist das Produkt aller Primzahlpotenzen, die glatt gegenüber einem selbst gewählten, in der Regel randomisiertem B sind, also

$$k = \prod_{q \in PP, q^e \leq B} q^e$$

Wenn die Primzahlpotenzen, die p-1 teilen, alle kleiner als B sind, dann ist k ein Vielfaches von p-1. Der Algorithmus berechnet dann g = ggT(a^k -1, n) für eine Basis a.
Dabei muss a der Bedingung 1 < a < n-1 entsprechen. Falls bei der Berechnung kein Teiler von n gefunden wird, wird eine neue Schranke B verwendet.

Beispiel: Wir faktorisieren die Zahl n = 1241143 und wählen dafür B = 13. k ist somit $2^3 * 3^2 * 5^1 * 7^1 * 11^1 * 13^1 = 360360$. Der GGT von $((2^{360360})-1, 1241143)$ ist 547, somit ist 547 ein Primfaktor der Zahl 1241143. Der Kofaktor ist 2269.

Für die Laufzeit dieser Methode gibt es quellen- und implementationsabhängige Schätzungen zwischen O(B ln n / ln B) und O(x), mit x = größter Primfaktor von (p-1).

Anwendungsbeispiel P-1

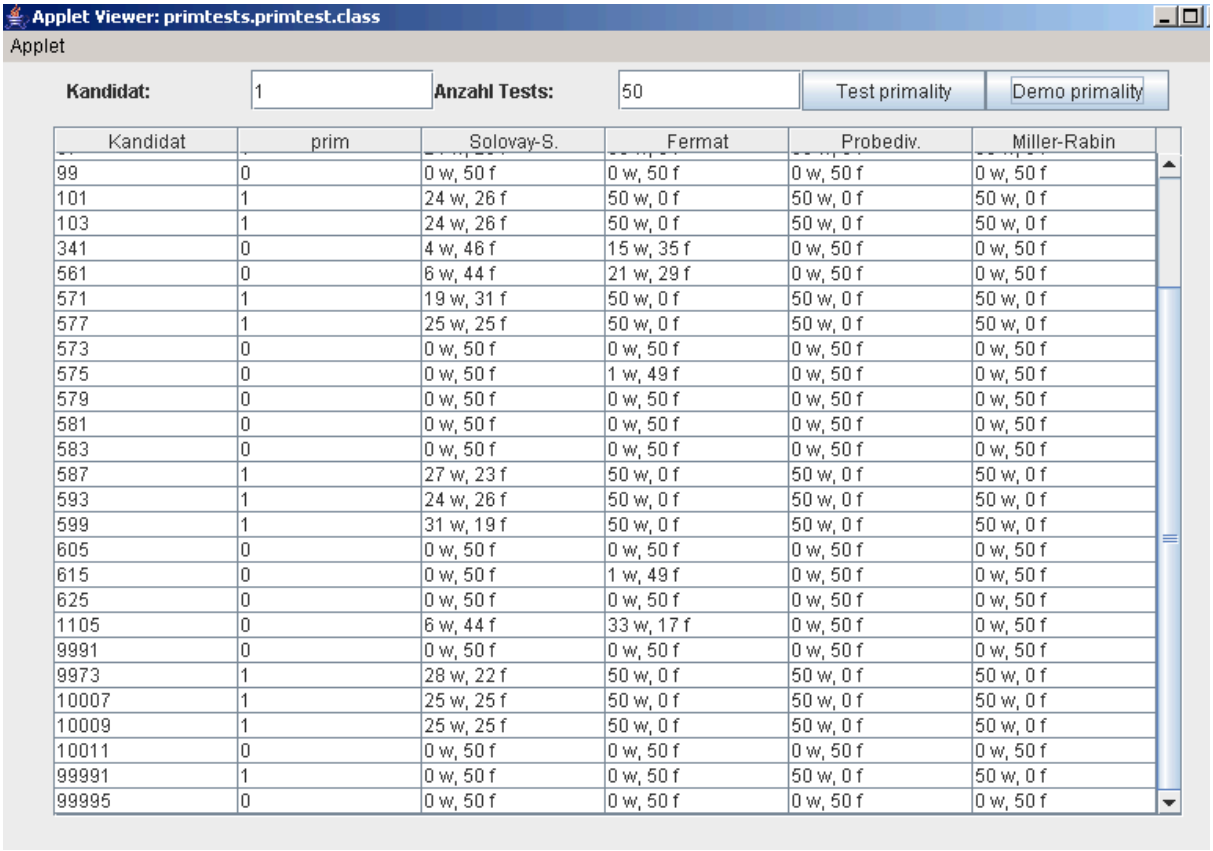
```
Pollard p-1 [Variante 2] für n = 15.0
a = 13.0, B = 2
k = 2
gcd von ( 168, 15.0) = 3.0
3.0 ist ein Teiler von 15.0
```

Performanz und Zukunft von Faktorisierungsverfahren

Bis 1988 galt die Annahme, daß die Laufzeit des Quadratischen Siebs die schnellste mögliche Problemlösung ist. John Pollard entwickelte das Quadratische Sieb zum Zahlkörpersieb weiter, dessen Laufzeit zirka L_n [1/3,$(64/9)^{1/3}$] beträgt und damit näher an polynomialer Laufzeit ist als alle anderen bekannten Methoden. Man geht davon aus, daß es grundsätzlich möglich ist, das Faktorisierungsproblem in polynomialer Zeit zu lösen (Shors Algorithmus), allerdings erst unter Zuhilfenahme von Quantencomputern.
Nach heutigem Wissensstand ist das Faktorisierungsproblem also nicht effizient lösbar und wird es voraussichtlich auch vorerst bleiben. Es kann jedoch theoretisch jederzeit ein bedeutender Durchbruch in der Mathematik erzielt werden, der eine deutlich schnellere Faktorisierung möglich macht und die gängigsten heutigen Verschlüsselungsmethoden schlagartig unwirksam werden lässt.

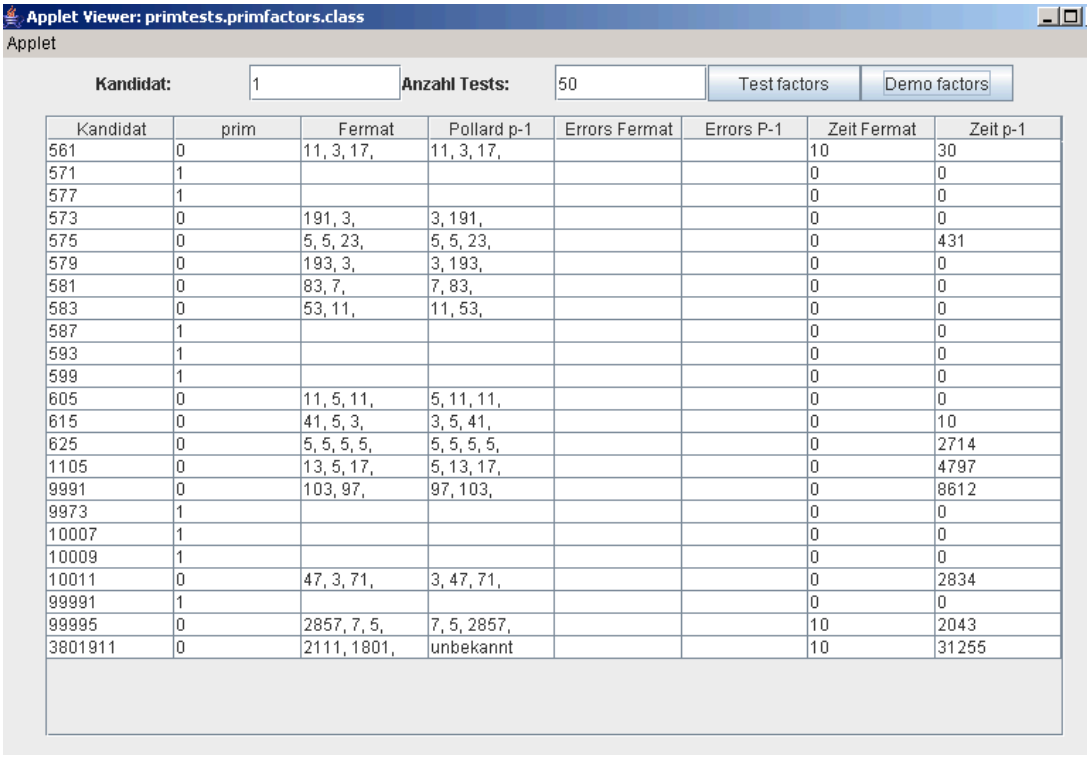
Testsoftware

Wir haben ein Applet zum Testen der Primalität sowie ein Applet zur Zerlegung der Faktoren geschrieben, welches für eine ausgewählte Reihe von primen, nichtprimen und pseudo-primen Zahlen verschiedene Tests ausführt. Die Primalitätstests werden 50x ausgeführt, um zu demonstrieren in wievielen Fällen die Monte-Carlo-Algorithmen unkorrekte Ergebnisse liefern.



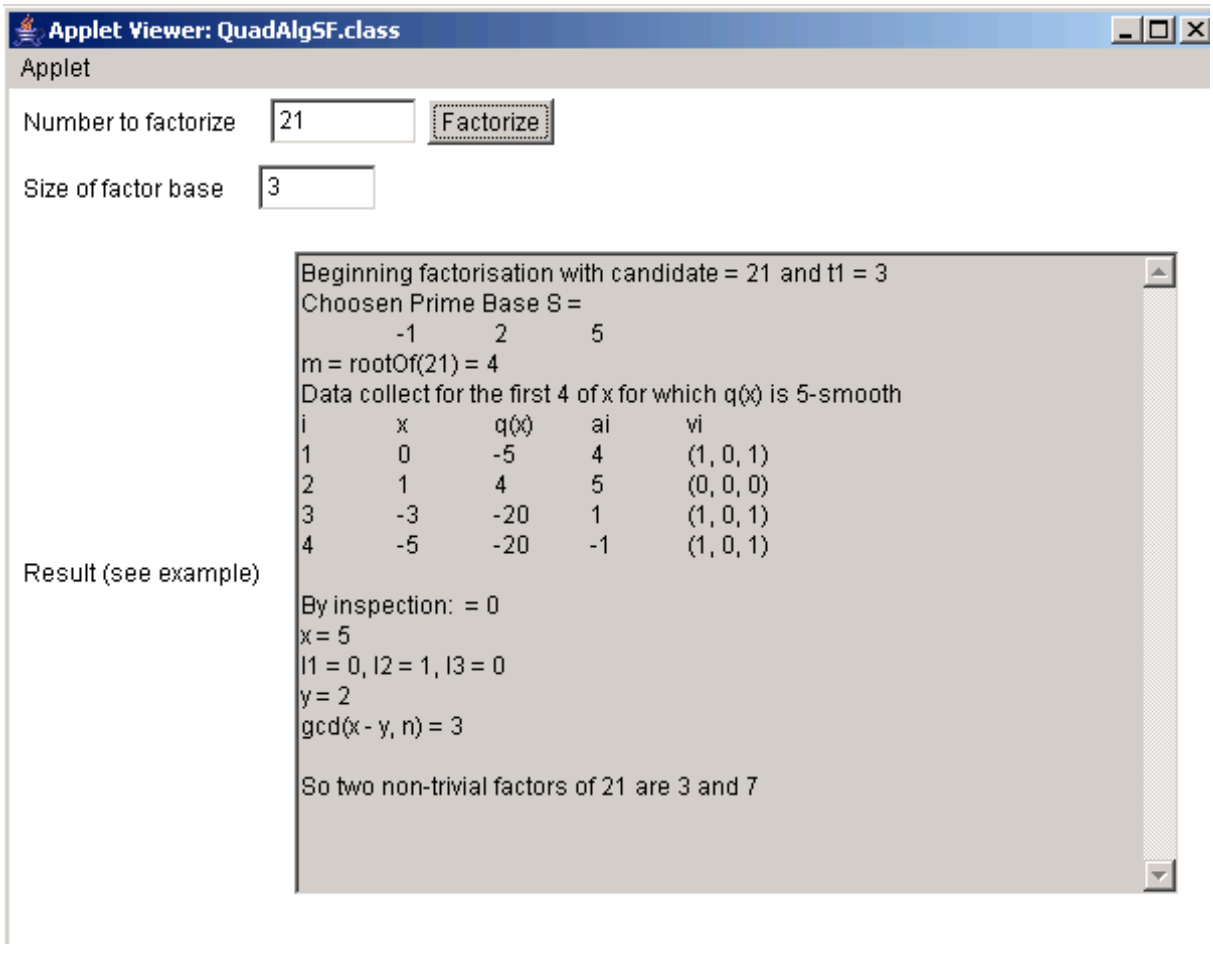
Kandidat	prim	Solovay-S.	Fermat	Probediv.	Miller-Rabin
99	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
101	1	24 w, 26 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
103	1	24 w, 26 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
341	0	4 w, 46 f	15 w, 35 f	0 w, 50 f	0 w, 50 f
561	0	6 w, 44 f	21 w, 29 f	0 w, 50 f	0 w, 50 f
571	1	19 w, 31 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
577	1	25 w, 25 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
573	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
575	0	0 w, 50 f	1 w, 49 f	0 w, 50 f	0 w, 50 f
579	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
581	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
583	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
587	1	27 w, 23 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
593	1	24 w, 26 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
599	1	31 w, 19 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
605	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
615	0	0 w, 50 f	1 w, 49 f	0 w, 50 f	0 w, 50 f
625	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
1105	0	6 w, 44 f	33 w, 17 f	0 w, 50 f	0 w, 50 f
9991	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
9973	1	28 w, 22 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
10007	1	25 w, 25 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
10009	1	25 w, 25 f	50 w, 0 f	50 w, 0 f	50 w, 0 f
10011	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f
99991	1	0 w, 50 f	0 w, 50 f	50 w, 0 f	50 w, 0 f
99995	0	0 w, 50 f	0 w, 50 f	0 w, 50 f	0 w, 50 f

Screenshot Primalitäts-Applet



Kandidat	prim	Fermat	Pollard p-1	Errors Fermat	Errors P-1	Zeit Fermat	Zeit p-1
561	0	11, 3, 17,	11, 3, 17,			10	30
571	1					0	0
577	1					0	0
573	0	191, 3,	3, 191,			0	0
575	0	5, 5, 23,	5, 5, 23,			0	431
579	0	193, 3,	3, 193,			0	0
581	0	83, 7,	7, 83,			0	0
583	0	53, 11,	11, 53,			0	0
587	1					0	0
593	1					0	0
599	1					0	0
605	0	11, 5, 11,	5, 11, 11,			0	0
615	0	41, 5, 3,	3, 5, 41,			0	10
625	0	5, 5, 5, 5,	5, 5, 5, 5,			0	2714
1105	0	13, 5, 17,	5, 13, 17,			0	4797
9991	0	103, 97,	97, 103,			0	8612
9973	1					0	0
10007	1					0	0
10009	1					0	0
10011	0	47, 3, 71,	3, 47, 71,			0	2834
99991	1					0	0
99995	0	2857, 7, 5,	7, 5, 2857,			10	2043
3801911	0	2111, 1801,	unbekannt			10	31255

Screenshot Faktorisierungs-Applet



Screenshot Faktorisierungs-Applet für Quadratisches Sieb

Quellenangaben

Bücher

- Johannes Buchmann: Faktorisierung großer Zahlen
  In: Spektrum der Wissenschaft S. 80-88 9.1996
- Thomas H. Cormen ... [et al.]
  Introduction to algorithms 2nd ed.
- Michael T. Goodrich, Roberto Tamassia
  Algorithm Design: Foundations, Analysis and Internet Examples
- Paul Garett
  Making, breaking codes
- Buchmann
  Einführung in die Kryptographie
- Douglas R. Stinson
  Cryptography, Theory and practice 2nd ed.
- Bruce Schneier
  Applied Cryptography, 2nd ed.
- Carl Pomerance
  A Tale of Two Sieves (in “NOTICES OF THE AMS” VOLUME 43, NUMBER 12)

Internet

- The RSA Factoring Challenge FAQ
  http://www.rsasecurity.com/rsalabs/node.asp?id=2094
- http://www.jjam.de/Java/Applets/Primzahlen/Miller_Rabin.html
- http://en.wikipedia.org/wiki/Modular_exponentiation#An_efficient_method:_the_right-to-left_binary_algorithm

Sourcen

- Das quadratische Sieb basiert auf einer Implementation von Kristijan Dragicevic.
- Die Implementation von Miller-Rabin basiert auf einem im Internet entdeckten Code.