Faktorisierung ohne Probedivisionen

Rainer Glaschick, Paderborn
rainer@glaschick.de
2017-04-19 2022-09-11

1. Einleitung

Etliche Verfahren der Kryptographie stützen sich darauf, dass ein Produkt von zwei Primzahlen einfach berechnet werden kann (selbst bei Faktoren mit 100 Dezimalstellen ist das ohne Computer möglich). Jedoch ist bislang kein Verfahren bekannt, mit dem man aus einer solchen Zahl die Primfaktoren systematisch in praktisch brauchbarer Zeit berechnen kann, sofern die Zahl mehrere hundert Dezimalstellen umfasst.

Auf einem von Fermat angegebenen Verfahren, dass die fragliche Zahl als Differenz von Quadratzahlen darstellt, basieren die meisten heute bekannten Verfahren mit der größten Effizienz. Um den Rechenaufwand gering zu halten, wird ein Großteil der Rechnungen in einem Restklassenring, also modulo einer Zahl, durchgeführt, und die dadurch entstehenden Kandidaten verprobt. Ein wesentlicher Teil dieser Verfahren ist eine Methode, um Kandidaten für Teiler anders als durch lineares Ausprobieren zu finden, also durch Einsatz von mathematischen Methoden den Rechenaufwand zu vermindern.

Sowohl das Fermat'sche Verfahren der Quadratzahlen als auch ein zweites (von mir als Verfahren der absteigenden Basen bezeichnet, da ich es in der Literatur bislang nicht finden konnte) lassen sich so gestalten, dass sie eine lineare Suche verwenden, dafür aber ohne Multiplikationen und Divisionen auskommen. Sie wären damit für eine Hardware-Lösung geeignet, da sie zudem leicht parallelisierbar sind.

Beide stellen keine Bedrohung für kryptographische Verfahren dar, weil der Aufwand ungefähr proportional zur Wurzel der Ausgangszahl ist, also exponentiell mit Anzahl der Stellen zunimmt.

Mit der Methode der absteigenden Basen und acht parallelen Prozessen konnte ich auf einem einfachen Desktop-PC mit vier Kernen den kleineren Faktor 1073075395319 der 80-Bit Zahl 1179132915127157710180471 in 3sec bestimmen. Die Programmierung erfolgte in der semi-interpretierten Sprache TUF-PL und einem Unterprogramm in C für die innerste Schleife, so dass das GNU-Programm *factor* mit 0.3sec immer noch um eine Größenordnung schneller ist. Wird die Schleife in TUF-PL programmiert, werden (ohne Parallelisierung) 760sec = 12min benötigt, was die Fermat'sche Methode, gleichfalls vollständig in TUF-PL, in 65sec erledigt, aber 74 Millionen mehrfach-genaue Rechenoperationen verwendet.

2. Hintergrund

Einen sehr umfassenden Übersicht liefert die Diplomarbeiten von Peter Hartmann [Hartmann2007] und Katja Schmidt-Samoa [SchmidtSamoa2002]. Lesenswert ist auch [SQUFOF].

Bis ins 17. Jahrhundert war lediglich die Probedivision mit einer Primzahltabelle (die mit dem *Sieb des Erathostenes* erzeugt werden kann) verfügbar. Da dieses alle Zahlen aufzählt und die Nicht-Primzahlen abstreicht, sind damit auch im Zeitalter des Computers bereits 32-Bit Zahlen nicht

effizient faktorisierbar.

Im folgenden liegt der Schwerpunkt auf dem Fall, dass die zu faktorisierende Zahl das Produkt von zwei Primfaktoren gleicher Größenordnung ist. Das muss aber keineswegs der Fall sein, weil die Faktoren lediglich mit einem probabilistischen Verfahren getestet werden.

3. Methode der Quadratdifferenzen

Erst Fermat hat mit seiner Methode der Quadratdifferenzen (https://de.wikipedia.org /wiki/Faktorisierungsmethode von Fermat]) einen Weg aufgezeigt, der nicht auf Probedivisionen bekannter Primzahlen beruht und besonders bei großen Faktoren schnell zu einer Lösung führt.

Im folgenden wird gezeigt, wie dieses Verfahren mit einfachen Mitteln effizient gestaltet werden kann, auch wenn damit keine Konkurrenz zu den hocheffizienten Methoden entsteht.

Bei Fermats Methode wird die zu faktorisierende Zahl als Differenz zweier Quadrate dargestellt:

$$n = x^2 - y^2 = (x + y) \cdot (x - y) = p \cdot q$$

$$p = x + y$$

$$q = x - y$$

$$x = \frac{p + q}{2}$$

$$y = \frac{p - q}{2}$$

Beginnend mit $x = \lceil \sqrt{n} \rceil$, wird $v = x^2 - n$ berechnet und geprüft, ob es eine Quadratzahl ist, ob es also ein y mit $y^2 = v$ gibt. Wenn das der Fall ist, dann sind $a = x + y$ und $b = x - y$ zwei Faktoren von n. Andernfalls wird x um eins erhöht und der Test wiederholt.

Da sich jede ungerade Zahl als Differenz von zwei Quadratzahlen darstellen lässt (man setze $q = n$ und $q = 1$, dann sind $x = \frac{n+1}{2}$ und $y = \frac{n-1}{2}$), bricht das Verfahren zwar theoretisch immer ab, würde aber bei Primzahlen (oder kleinen Faktoren) bis zu $n - \sqrt{n}$ Schritte erfordern, wenn nicht zuvor kleine Faktoren durch Probedivision der ersten Primzahlen ausgeschlossen wurden.

Ausserdem sind dann x^2 und y^2 nahe $\frac{n^2}{4}$, so dass bei einem 64-Bit Rechner nur Ausgangszahlen bis 32 Bit verwendet werden können. Insbesondere wenn man kleine Faktoren durch Probedivision berechnet, kann man das Verfahren abbrechen, wenn v>n ist.

Eine Verbesserung besteht darin, dass die Rechnung in einem Restklassenring erfolgt, um Kandidaten zu bestimmen.

Die Zahlen x und y sind nicht beide gleichzeitig gerade; dass x auch ungerade sein kann, zeigt 571*821 = (696-125)*(696+125).

3.1. Durchführung

Um Quadratzahlen aufzuzählen, sind keine Multiplikationen notwendig. Mit x' sei der Nachfolger von x bezeichnet:

$$x' = x + 1$$

$$x'^2 = x^2 + 2x + 1$$

Da wegen $v = x^2 - n$ die Differenz zwischen v und x^2 konstant n ist, wird v um genau dasselbe Inkrement weitergeschaltet:

$$v' = v + 2x + 1$$

Erste Methode

Sobald v eine Quadratzahl ist, ist die Lösung gefunden:

```
factor (n):
    x =: math int sqrt n
    if x^2 = n
        :> x, x
    x =+ 1
    ! x^2 > n
    v =: x^2 - n
    while v ~= ( math int sqrt v)^2
        v =+ 2*x + 1
        x =+ 1
    y =: math int sqrt v
    return x+y, x-y
```

Beispiel 463081 = 811 * 571:

x	v	y	v-y²
681	680	26	4
682	2043	45	18
683	3408	58	44
684	4775	69	14
685	6144	78	60
686	7515	86	119
687	8888	94	52
688	10263	101	62
689	11640	107	191
690	13019	114	23
691	14400	120	0

Sind die beiden Faktoren sehr nahe beieinander, ist schon im ersten Schritt v eine Quadratzahl.

Zeitbestimmend ist die (Ganzzahl-) Quadratwurzel zur Entscheidung, ob v eine Quadratzahl ist.

Die Werte von v werden recht schnell größer und können durchaus n übersteigen und — soferen n nahe dem nativen Rechenbereich ist — entweder einen Überlauf verursachen (natives C), oder benötigen die langsamere mehrfach genaue Bibliothek. Das führt dazu, dass bei Eingabe einer Primzahl sehr große Rechenzeiten entstehen. Aus diesem Grund wird die Berechnung abgebrochen, wenn $v \geq n$ ist; das ergibt als Abschätzung für den Bereich der berechneten Wurzeln:

$$v \approx n$$

$$x^2 \approx 2n$$

$$x \approx \sqrt{2}\sqrt{n}$$

$$p = x + v = \sqrt{2}\sqrt{n} + \sqrt{n} = \left(\sqrt{2} + 1\right)\sqrt{n} > \sqrt{2}\sqrt{n}$$

$$q = x - v = \sqrt{2}\sqrt{n} - \sqrt{n} = \left(\left(\sqrt{2}\right) - 1\right)\sqrt{n} \approx 0.4\sqrt{n}$$

Also ist der kleinere Faktor bei Abbruch kleiner als die halbe Wurzel aus n; ein Produkt aus zwei Primzahlen, die nicht allzuweit auseinander liegen, wird also faktorisiert.

Zweite Methode

Da die Entscheidung, ob n eine Quadratzahl ist, für die Laufzeit entscheidend ist, wird diese bei der zweiten Methode verbessert.

Man kann einige der fraglichen Zahlen von vornherein ausschließen, weil sie keine Quadratzahlen sein können.

So ist das Quadrat einer geraden Zahl ist gerade, und durch 4 dividierbar, da $(2n)^2 = 4n^2$. Das Quadrat einer ungeraden Zahl ist ungerade und 1 mod 8, da $n(n+1)$ immer gerade ist:

$$(2n+1)^2 = 4n^2 + 4n + 1 = 4n(n+1) + 1$$

Damit sieht der Test so aus:

```
is (n) square:
    ? is n odd
        ? n %% 8 = 1
            :> n = (math int sqrt n)^2
        :> ?-
    ? n %% 4 = 0
        :> n = (math int sqrt n)^2
    :> ?-
```

Eine bessere Möglichkeit ist es, den Rest modulo 16 auszuwerten; er kann nur 0, 1, 4 und 9 sein; erst dann wird die Quadratwurzel bestimmt. Dies beschleunigt die Rechnung etwa um den Faktor 2.

Damit kann die Anzahl der zeitaufwändigen Quadratwurzelberechnungen vermindert werden; eine Methode, ohne dies auszukommen, wurde bislang nicht gefunden.

Für die Quadratwurzel wird die Bibliotheksfunktion `math int sqrt ()` verwendet, die schneller ist als das Äquivalent in TUF.

Zweite Methode

Eine zweite Methode vermeidt die zeitaufwändige Berechnung der Quadratwurzel; die möglichen Wurzeln zu v werden aufgezählt:

```
x =: 1 + math int sqrt n
v =: x * x - n
y =: x - 1
w =: y * y
while v ~= w
    v =+ 2*x  + 1      \ v = v + x + x 1
    x =+ 1
    while w < v
```

```
        w =+ 2*y + 1
        y =+ 1
return x+y, x-y
```

Es sind nur Additionen notwendig, also einfach in Hardware aufzubauen. Auf modernen Rechnern, bei denen Multiplikation und Division sehr schnell sind, erfordert das Aufzählen allerdings mehr Rechenzeit, weil die zweite Schleife sehr häufig durchlaufen wird. Beispielsweise werden bei der 50-bit Zahl 769957617486611 14483 Quadratwurzeln berechnet, aber innere Schleife für die Quadratzahlen 889840 durchlaufen, also 100 mal mehr.

Dritte Methode

Eine dritte Methode berechnet die Anzahl d der Durchläufe vorab; dabei ist zu berücksichtigen, dass y in der Schleife inkremetiert wird.

Damit wird y um d erhöht wird:

$$y' = y + d$$

$$w' = (y')^2 = y^2 + 2yd + d^2 = w + 2yd + d^2$$

Gewünscht ist das d, für das $w' = v'$ ist, also

$$v' = w' = w + 2yd + d^2$$

$$v' - w = 2yd + d^2$$

$$v' - w + y^2 = v' = d^2 + 2dy + y^2$$

$$(d + y)^2 = v'$$

$$d = \sqrt{v'} - y$$

Damit sieht die dritte Methode so aus:

```
x =: 1 + math int sqrt n
v =: x * x - n
y =: math int sqrt v
w =: y * y
while v ~= w
    v =+ 2*x  + 1
    x =+ 1
    d =: (math int sqrt v) - y
    w =+ d * (2*y + d)
    y =+ d
return x+y, x-y
```

Da jedoch pro Durchlauf wieder eine Quadratwurzel berechnet werden muss und weitere Anweisungen erforderlich sind, ist diese Lösung zwar meist schneller als die zweite, aber langsamer als die erste Methode.

Zudem sind die erste und zweite Methode offensichtlich korrekt, während bei der dritten unklar ist, wieso das durch die abschneidende Quadratwuzel berechnete `d` korrekt ist, während die Herleitung dies nicht berücksichtigt.

Vierte Methode

Sie besteht aus einer Kombination der dritten und zweiten Methode, indem die Quadratwurzel durch eine Näherung ersetzt wird.

Mit der Näherung (Abbruch der Tayorreihe):

$$\sqrt{1+x} \geq 1 + \frac{x}{2}$$

$$\sqrt{p+q} = \sqrt{p}\sqrt{1+\frac{q}{p}} \geq \sqrt{p}\left(1+\frac{q}{2p}\right)$$

wird wegen $y \approx \sqrt{v}$:

$$\sqrt{v+2x+1} \geq \sqrt{v}\left(1+\frac{2x+1}{2v}\right) \geq \sqrt{v}\left(1+\frac{x+1}{2\cdot\sqrt{v}}\right) \geq \sqrt{v}+\frac{x}{2\sqrt{v}} \approx y+\frac{x}{2y}$$

$$d = \frac{x}{2y}$$

Da der Wert von d nur eine Näherung und entweder zu klein oder zu groß ist, muss anschließend noch durch Einzelschritte korrigiert werden:

```
?* v ~= w
    v =+ 2*x + 1
    x =+ 1

d =: x // y // 2
w =+ d * ( 2 * y + d)
y =+ d
?* w > v
    w =- 2*y -1
    y =- 1
?* w < v
    w =+ 2*y + 1
    y =+ 1
```

Weil aber im Schnitt d deutlich zu klein ist, und ein zu großes d durch die erste Unterschleife korrigiert wird, kann die Division durch 2 eingespart werden:

```
?* v ~= w
    v =+ 2*x + 1
    x =+ 1

d =: x // y
w =+ d * ( 2 * y + d)
y =+ d
?* w > v
    w =- 2*y -1
    y =- 1
?* w < v
    w =+ 2*y + 1
    y =+ 1
```

3.2. Parallelisierung

Eine Parallelisierung ist möglich, da die Werte für x mit der Schrittweite 1 aufsteigend sind, so dass Blöcke von x-Werten den Prozessen zugewiesen werden können.

Alternativ kann mit größerer Schrittweite e gerechnet werden, insbesondere der Anzahl der zu startenden Prozesse:

$$x' = x + e$$

$$x'^2 = x^2 + 2xe + e^2$$

$$v' = x'^2 - n = v + 2xe + e^2 = v + e(2x + e)$$

4. Methode der absteigenden Basis

Eine alternative zur Fermats Methode benötigt nach der Bestimmung der Anfangswerte keine Divisionen mehr, nur noch Additionen und Subtraktionen von Zahlen in der Größenordnung der Quadratwurzel, also der halben Stellenzahl. Das Verfahren konnte in der Literatur nicht identifiziert werden.

Ausgangspunkt ist die — zunächst nicht konstruktive — Aussage, dass der Rest einer Division einfach dadurch gefunden werden kann, dass die Zahl zur Basis des Divisors dargestellt wird; die letzte «Ziffer» ist dann der Rest.

Das Problem liegt darin, dass die Darstellung zu einer bestimmten Basis üblicherweise durch Divisionen bzw. Restbildung erfolgt, also ebenso aufwändig ist wie Probedivisionen.

Gleiches gilt für die Umrechnung von einer Basis in eine andere. So erfolgt die Umwandlung einer Zahl aus beispielsweise dem Oktalsystem ins Dezimalsystem unter Verwendung von Dezimalzahlen durch wiederholte Multiplikationen der Oktalziffern mit den Potenzen von 8; eine Umwandlung einer im Dezimalsystem gegebenen Zahl ins Oktalsystem durch wiederholte Division durch 8 mit Rest.

In einem Computer werden die Zahlen intern binär gespeichert; man kann sie als einziffrige Darstellung zur Basis des Maximalwerts auffassen. Damit sind Zahlen zu jeder Basis — solange die Basis unterhalb des Maximalwerts liegt — einfach dadurch darstellbar, dass für jede »Ziffer« eine Variable verwendet wird.

4.1. Die Methode

Da immer einer der Faktoren kleiner oder gleich der Quadratwurzel der zu faktorisierenden Zahl ist, beginnt man mit dem nächstniedrigeren ganzahligen ungeraden Wert der Quadratwurzel, und stellt sie zu dieser Basis da; das ergibt immer drei Ziffern zu dieser Basis. Mit $n = 1147$ und $\left\lfloor \sqrt{1147} \right\rfloor = 33$ ist der erste zu verprobende Faktor $p = 33$, und die Ziffern zur Basis 33 sind 1, 1 und 25:

$$1147 = 1 \cdot 33^2 + 1 \cdot 33 + 25$$

Die Darstellung zur Basis 31 ergibt:

$$1147 = 1 \cdot 31^2 + 6 \cdot 31 + 0$$

Damit hat man in diesem Fall bereits einen der Faktoren gefunden.

Anstatt also die Primzahlen unterhalb der Quadratwurzel zu bestimmen und mit jeder eine

Probedivision durchzuführen, kann man auch aus der Darstellung zur Basis x die zur Basis x-2, x-4, usw. bestimmen und dies solange fortführen, bis die letzte Ziffer Null ist. Eine Probedivision ist dabei nicht erforderlich.

Insbesondere wenn die zu faktorisierende Zahl aus zwei ungefähr gleich großen Faktoren besteht, wie es in der Kryptographie häufig der Fall ist, wird dieser Faktor schneller gefunden, als wenn aufsteigenden Primzahlen verwendet werden.

Da die eigentliche Suche mit einem halbierten Rechenbereich möglich ist, können Zahlen bis $2^{120} \approx 10^{40}$ auf Rechnern mit 64-Bit Arithmetik faktorisiert werden. Zudem ist das Verfahren sehr einfach parallelisierbar (s.u.).

4.2. Absteigende Basis

Sei n die zu faktorisierende Zahl, die o.B.d.A. ungerade ist. Da einer der Faktoren immer kleiner als die (oder gleich der) Quadratwurzel ist, kann die Suche für Faktoren von dort aus abwärts erfolgen. Dazu wird die Zahl mit drei Ziffern zur Basis b dargestellt, wobei b anfänglich die auf die nächste ungerade Zahl verminderte Quadratwurzel von n ist:

$$b = \lfloor \sqrt{n} \rfloor \text{ bzw. } \lfloor \sqrt{n} \rfloor - 1 \text{ (ungerade)}$$

$$n = xb^2 + yb + z$$

wobei dann anfänglich $x = 1$, $y < b$ und $z < b$ sind.

Nun wird b schrittweise um 2 reduziert; sei b' das reduzierte b (und nicht eine Ableitung nach der Zeit):

$$b = b' + 2$$

$$b^2 = b'^2 + 4b' + 4$$

Das ergibt für n:

$$n = xb'^2 + 4xb' + 4x + yb' + 2y + z$$

$$n = xb'^2 + (4x + y)b' + (4x + 2y + z)$$

$$n = x'b'^2 + y'b' + \dot{z}$$

Damit ist

$$b' = b - 2$$

$$y' = y + 4x$$

$$\dot{z} = z + 2y + 4x = z + y + y'$$

Um die neuen Stellen für die um zwei verminderte Basis zu finden, ist also lediglich die zweite Stelle um das vierfache der ersten zu erhöhen, und die dritte um die zweite und den neuen Werte der zweiten. Da nur der Wert $4x$ benötigt wird, ist keine Multiplikation notwendig. Hier kann entweder gleich der Wert $4x$ mitgeführt werden, oder es wird die Multiplikation durch viermal Addition oder einen Shift bewirkt; moderne Compiler optimieren dies ohnehin. Es sind also eine Subtraktion, drei Additionen und ein Shift notwendig.

Nach der Berechnung von $\dot{y}$ und $\dot{z}$ werden beide normalisiert:

```
y += z / b
z = z % b
x += y / b
y = y % b
```

Da (für $x < b$) immer $\dot{y}' < 5b$ und $\dot{z}' < 7b$ sind, sind Subtraktionen schneller als Divisionen:

```
while z >= b
    y += 1
    z -= b
while y >= b
    x += 1
    y -= b
```

Da die Additionen nicht länger dauern, wenn die Zahlen größer sind, braucht erst normalisiert zu werden, wenn der normale Rechenbereich — in TUF-PL 64 bit — überschritten wird; dann ist die Abfrage `z > 0` zu ersetzen durch `z %% b > 0`. Meist ist diese Modulo-Berechnung weniger aufwendig als die Normalisierungen.

Bei der Normalisierung handelt es sich nicht um eine Modulo-Rechnung, da die Werte für x und y ggf. erhöht werden müssen. Die Normalisierung für z ist notwendig, wenn die Bedingung $z = 0$ abgefragt wird; andernfalls ist `z %% b = 0` zu verwenden. Die Normalisierung von y vermeidet Überläufe und ist zudem geringfügig schneller.

Dies sei an einem ersten Beispiel $n = 589, b = 23$ gezeigt:

$$589 = 1 \cdot 23^2 + 2 \cdot 23 + 14$$

```
p   x   y   z   y'  z'
23  1   2   14  6   22
21  1   7   1   11  19
19  1   11  0
```

Die Normalisierung erfolgt bei der Übernahme in die nächste Zeile.

Sei als zweites Beispiel `589597 = 727 * 811 = 1 * 767² + 1*767 + 541`:

```
  p   x   y   z   y'  z'
767   1   1 541   5 547
765   1   5 547   9 561
763   1   9 561  13 583
761   1  13 583  17 613
759   1  17 613  21 651
757   1  21 651  25 697
755   1  25 697  29 751
753   1  29 751  33 813
751   1  34  62  38 134
749   1  38 134  42 214
747   1  42 214  46 302
745   1  46 302  50 398
743   1  50 398  54 502
741   1  54 502  58 614
739   1  58 614  62 734
737   1  62 734  66 862
735   1  67 127  71 265
733   1  71 265  75 411
731   1  75 411  79 565
729   1  79 565  83 727
```

```
 727    1   84    0
```

Hier das Beispiel `279 = 3² * 31 = 1*15² + 3*15 + 9`:

```
 p    x    y    z    y'   z'
15    1    3    9    7    19
13    1    8    6    12   26
11    2    3    4    11   18
 9    3    4    0
```

An diesem Beispiel wird auch deutlich, dass der gefundene Faktor kein Primfaktor sein muss; es ist lediglich der größte Faktor, der kleiner als die Wurzel ist. Der größte Primfaktor, hier `31`, kann durchaus oberhalb der Wurzel liegen.

4.3. Die Bedingung x < p

Sind die Faktoren relativ groß, wie es für RSA empfohlen wird, so wird der Faktor gefunden, während noch die Bedingung $x < b$ bzw. $b > \sqrt[3]{n}$ erfüllt ist.

Da es aber nur darauf ankommt, den Nulldurchgang von z zu erkennen, kann das Verfahren auch über $x > b$ hinaus fortgeführt werden, wenn der größer Faktor nicht nur unterhalb der Quadrat-, sondern auch unterhalb der Kubikwurzel der Ausgangszahl liegt, solange $7 \cdot b$ im Rechenbereich liegt. Dies ist nur bis $b = \sqrt[4]{n}$ sichergestellt, da dann

$$x = \frac{n}{b^2} = \frac{n}{\sqrt{n}} = \sqrt{n}$$

und somit auf den Anfangswert von b angewachsen ist. Wenn also beispielsweise die Ausgangszahl das Produkt von vier Primzahlen gleicher Größenordnung ist, kann das Verfahren auf Grund von Bereichsüberschreitungen misslingen. Zudem nimmt die Rechenzeit stark zu, wenn die Normalisierung über Schleifen und nicht über Hardware-Divisionen erfolgt.

Vielfach wird auch bei anderen Verfahren vorgeschlagen, auf andere Verfahren auszuweichen, wenn kein Faktor $b > \sqrt[3]{n}$ gefunden wird. Allerdings ist die Verwendung von Probedivisionen wohl nur bis $2^{21} \approx 10^7$ sinnvoll, also nur bei 63- bzw. 84-Bit Ausgangswerten (letzteres für die 4. Wurzel).

4.4. Mehr Ziffern

Der Vorteil der dargestellten Methode besteht sicherlich darin, dass sie sehr wenige Operationen benötigt, wenn der Zahlenbereich das Quadrat der Rechenbreite ist, z.B. 120 Bit für 64-Bit Arithmetik. Für Hardware-Lösungen ist das immer die bessere Lösung, insbesondere wenn serielle Arithmetik verwendet wird (s.u.).

Um mit einer 64-Bit Basis-Arithmetik dann 180 oder 240 Bit Zahlen behandeln zu können, ist es auch denkbar, mehr als zwei Ziffern zu verwenden:

$$n = x_3 b^3 + x_2 b^2 + x_1 b^1 + x_0$$

$$n = x_4 b^4 + x_3 b^3 + x_2 b^2 + x_1 b^1 + x_0$$

Im ersteren Fall ergibt sich

$$b = \dot{b} + 2$$

$$n = x_3\dot{b}^3 + 6x_3\dot{b}^2 + 12x_3\dot{b} + 8x_3 + x_2\dot{b}^2 + 4x_2\dot{b} + 4x_2 + x_1\dot{b} + 2x_1 + x_0$$

$$n = x_3\dot{b}^3 + (6x_3 + x_2)\dot{b}^2 + (12x_3 + 4x_2 + x_1)\dot{b} + (8x_3 + 4x_2 + 2x_1 + x_0)$$

$$\dot{x}_2 = x_2 + 6x_3 = x_2 + 4x_3 + 2x_3$$

$$\dot{x}_1 = 12x_3 + 4x_2 + x_1 = x_1 + 2\dot{x}_3 + 2x_2$$

$$\dot{x}_0 = x_0 + 2x_1 + 4x_2 + 8x_3$$

Damit werden 7 Additionen und 7 Shifts sowie die Subtraktion von 2 benötigt, also — ohne die Normalisierung — 15 Operationen (anstelle von 5). Dies könnte immer noch günstiger sein, als eine 128-Bit Arithmetik auf Basis von 64-Bit Elementar-Operationen zu verwenden.

Insofern wäre es — für Software-Lösungen — zu untersuchen, wie die Anzahl der Operationen mit der Anzahl der Ziffern wächst. Um 1024-Bit-Zahlen, d.h. 512-Bit Operationen mit 64-Bit Basis-Operationen zu bewältigen, wären dann 8 Stellen notwendig.

Da jedoch die Zwischenergebnisse erheblich über der Basis liegen, sind wohl eher 10 bis 12 Stellen realistisch.

4.5. Größere Abstände

Anstatt in Zweierschritten die Basis zu reduzieren, können auch andere Abstände verwendet werden. Natürlich wäre die Anzahl der Schritte minimal, wenn nur Primzahlen als Basis verwendet werden würden; jedoch ist für die Bestimmung der nächstkleineren Primzahl genausowenig ein schneller Algorithmus bekannt wie für die Bestimmung der nächstgrößeren.

Nehmen wir zunächst einmal an, dass man mehrere Prozesse parallel rechnen lassen kann und dass jeder Prozess in größeren Abständen die Faktoren ermittelt. Wenn der erste dieser Prozesse erfolgreich ist, können die anderen abgebrochen werden. Man kann also die Suchzeit entsprechend verkleinern.

Für einen ungeraden Anfangswert sind nur gerade Intervalle $2k$ sinnvoll. Dann müssen, um alle möglichen Teiler zu verwenden, zusätzlich die Basen $b-2$, $b-4$ usw bis $b-2(k-1)$ als Ausgangspunkte verwendet werden. Für jede dieser Sequenzen gilt dann:

$$b = \dot{b} + 2k$$

$$b^2 = \dot{b}^2 + 4k\dot{b} + 4k^2$$

$$n = x\dot{b}^2 + 4kx\dot{b} + 4k^2x + y\dot{b} + 2ky + z$$

$$n = x\dot{b}^2 + (4kx + y)\dot{b} + 4k^2x + 2ky + z$$

$$\dot{y} = y + 4kx$$

$$\dot{z} = z + 4k^2x + 2ky = z + k(\dot{y} + y)$$

Wenn keine Multiplikationen verwendet werden, müssen die beiden Summanden $4x$ und $y' + y$ k-mal addiert werden; die Anzahl der Additionen (ohne Normalisierung) erhöht sich von 3 auf 3+2k, jeder einzelne Prozess ist also um $\frac{3+2k}{3}$ langsamer. Durch die Verteilung auf k Prozesse ergibt sich ein Faktor $\frac{3+2k}{3k} = \frac{1}{k} + \frac{2}{3}$. Selbst bei sehr vielen Prozessen ist der Gewinn also bei Verzicht auf Multiplikationen gering.

Bei Schrittweiten $2k$, die eine Potenz von zwei sind, kann ein Shift verwendet werden, der wie einen Addition zählt; damit sind es 3+2 Operationen, und der Gewinn ist $\frac{5}{3k} \approx \frac{2}{k}$ und kann damit durch Einsatz von vielen Prozessen effizient gesteigert werden.

Man kann aber auch als Schrittweite das Produkt der ersten n Primzahlen verwenden, wie es für das Sieb des Erathostenes bekannt ist, also beispielsweise $k = 3$. Dann werden nicht drei, sondern nur zwei Prozesse benötigt, da die durch 3 teilbaren Fälle weggelassen werden können. Aufsteigend wäre das:

```
 1  3  5  7  9  11  13  15  17  19  21  23  25  27  29  31  33  35  37
    x     x     x     x     x     x     x     x
 1     7     13    19    25    31    37
    5     11    17    23    29    35
```

Man prüft also die ersten drei b auf Teilbarkeit durch 3 und verwendet nur die nicht teilbaren beiden. Damit wird die Geschwindigkeit um 50% vergrößert, und eine Aufteilung in 2 (oder ein vielfaches davon) von Prozessen ist sinnvoll.

Nimmt man noch die durch 5 teilbaren heraus, dann ist $k = 3 \cdot 5 = 15$. Ist beispielsweise anfänglich $b = 53$, dann werden mit 53 beginnend die 15 Werte 53, 51, 49, 47, 45, 43, 41, 39, 37, 35, 33, 31, 29, 27 und 25 auf Teilbarkeit durch 3 oder 5 untersucht und somit die 7 Werte 51, 45, 39, 35, 33, 27 und 25 nicht benutzt; es verbleiben die 8 Werte 53, 49, 47, 43, 41, 37, 31 und 29. Es wird ein vielfaches von 8 Prozessen anstelle von 15 verwendet; es werden also nur etwas mehr als die Hälfte der Basen verprobt.

Für die Berechnung ergibt sich:

$$\dot{y} = y + 60x$$

$$\dot{z} = z + 4k^2x + 2ky = z + 15(\dot{y} + y) = z + 16(\dot{y} + y) - (\dot{y} + y)$$

Allerdings sind — mit der angegebenen Optimierung der Multiplikation mit 15, die ein optimierender Compiler erkennen könnte — 4 Additionen, ein Shift und eine Multiplikation mit 60 sowie die Subtraktion von 2 notwendig. Hier kann es sich lohnen, anstelle von x den Wert 60x mitzuführen und bei der Reduktion von y durch eine Subtraktionsschleife 60 zu addieren.

Praktisch gesehen, wird man immer 8 Prozesse starten und es dem Betriebssystem überlassen, die Rechenzeit der Kerne auf die Prozesse zu verteilen.

Allerdings steigt die Zahl der Prozesse schnell an, und der Gewinn nur asymptotisch mit dem Logarithmus, da die Anzahl der Primzahlen bis n $\pi(n) \approx \frac{n}{\ln n}$ ist:

n	k	π(n)	p	k/p
6	3	3	2	1.5
30	15	10	8	1.9
210	105	46	43	2.4
2310	1155	343	339	3.4
30030	15015	3284	3219	4.6
450450	225225	37747	37741	5.9

Hierbei ist p die (minimale) Anzahl der Prozesse.

5. Implementationsaspekte

5.1. Software

Unter Unix ist mit dem Programm «factor» ein leistungsfähiges Programm unter Verwendung des Verfahrenes [SQUFOF] verfügbar, dass die Faktorisierung von bis zu 128-Bit Zahlen in Sekunden erledigt. Die folgenden Überlegungen sind daher mehr prinzipieller Natur denn als echte Alternative gedacht.

Bei Verwendung des Verfahrens der absteigenden Basis sind zu Beginn der Suche sind die Ziffern x, y und z (nach der Normalisierung) kleiner als p. Bei Verwendung von vorzeichenlosen (nicht-negativen) 32-Bit Zahlen sind also Produkte in der Größenordnung von 64 Bit faktorisierbar.

Betrachten wir zunächst den Fall, dass auch $x < p$ ist. Hierzu muss $p > \sqrt[3]{n}$ sein. Dann wird bei der Addition $y + 4x$ ein Zwischenergebnis $< 5p$ entstehen, und bei der Addition $z + y + y'$ bei nicht zuvor normalisiertem y' ein Zwischenergebnis $< 7p$. Somit sind bei 32-bit Zahlen 3 bit hierfür zu reservieren; also sind es max. 29 Bit pro Variable. Die Eingangszahl darf also maximal 58 Bit groß sein. Entsprechend ist bei 64-Bit Variablen nur 61 Bit verwendbar, und der Eingang ist auf 122 Bit, etwa 36 Dezimalziffern, zu beschränken. Dann allerdings bewältigt ein moderner Prozessor etwa 10^9 oder 2^{29} Schritte pro Sekunde. Da nur jede zweite Zahl verprobt wird, wird eine 58-Bit Zahl in weniger als 0.5sec faktorisiert; sind die Faktoren groß, deutlich schneller. Eine 122-Bit Zahl benötigt hingegen schon 2^{30} oder 10^{10} Sekunden, entsprechend 317 Jahren auf einem einzigen Prozessor. Da — siehe unten — das Verfahren gut parallelisierbar ist und die Faktoren bei den typischen Verschlüsslungsverfahren wie RSA groß sind, ist die Zeit wesentlich kleiner.

Zu Anfang wird x nur langsam, d.h. nicht mit jedem Durchlauf, größer. Dennoch wird, wenn vorher kein Faktor gefunden wird, irgendwann der Fall $x \geq p$ (s.o.) auftreten. Dann wird man mit einer vierziffrigen Darstellung (s.o) weiterrechnen, um keine mehrfach genaue Arithmetik einsetzen zu müssen, bis die Bereichsgrenzen erreicht werden.

5.1.1. Weitere Optimierungen

Zum einen kann man anstelle von x auch $4x$ mitführen und so die Multiplikation sparen; muss dies aber bei den Zahlenbereichen berücksichtigen.

Weniger aussichtsreich ist es, die Subtraktion von 2 von b durch eine Subtraktion von 1, d.h. das Mitführen von $\frac{b-1}{2}$, zu ersetzen, weil dann die restlichen Berechnungen wesentlich komplexer werden.

5.1.2. Modulare Arithmetik

Das die Darstellung zu Basis b auch zutrifft, wenn alle Rechnungen modulo einer anderen Basis, z.B. 2^{64} erfolgen, wäre es ein Option, die Sequenz modulo dieser Basis zu rechnen; dann entfällt die Normalisierung.

Allerdings ist noch herauszufinden, wie dann erkannt werden kann, dass z ein Vielfaches von b ist (Chinesischer Restesatz?). Außerdem hält, wie bei den quadratischen Sieben, die Sequenz auch vorzeitig an; es ist also immer das Ergebnis zu verifizieren und ggf. die Sequenz fortzuführen.

5.2. Hardware

Die benötige Chipfläche für die Additionen und Normalisierungen ist sehr gering.

Dabei ist es zweckmäßig, die Zahlen seriell zu verarbeiten; dann fallen die Summierer ohnehin nicht ins Gewicht, und Schieberegister mit 512 Bit Länge sind auch kein Problem. Zudem entfallen viele Probleme der Bitsynchronisation bei paralleler Verarbeitung, die den Takt begrenzen. Es kann also durchaus mit Taktfrequenzen um 10 GHz gearbeitet werden.

Dies erscheint zunächst wenig sinnvoll, da bei 512 Bit die einzelne Addition dann einem Takt von 20MHz entspricht.

Jedoch können viele Stufen als Pipeline hintereinander geschaltet werden, bis die Chipfläche aufgebraucht ist. Ein Wert von 1 Million Stufen ist wohl gut erreichbar, so dass ein einzelner Chip dann 10^{15} oder 2^{45} Schritte pro Sekunde erreichen könnte. Pro Jahr sind das dann 2^{67} Schritte, so dass Zahlen von 2^{128} Bit faktorisierbar sind. Da nur große Faktoren gesucht sind, erhöht sich der Wert auf 2^{136} Bit. Da $10^5 = 2^{15}$ Chips schnell gefertigt sind, ergeben sich 2^{151} Bit als Jahresschranke.

Somit sind RSA-Lösungen mit 512-Bit und mehr weiterhin als sicher gegenüber der dargestellten Lösung in ihrer jetzigen Form anzusehen.

5.3. Effizienzüberlegungen

Die Laufzeit ist etwa $\frac{1}{2}\sqrt{n}$, da alle (ungeraden) Zahlen unterhalb durchprobiert werden, sofern die Ausgangszahl eine Primzahl ist.

Hinzu kommt die Rechenzeit für die Normalisierung. Wenn hierzu nur Subtraktionen verwendet werden, steigt der Aufwand quadratisch, insb. wenn x>p wird und auf jeden Fall normalisiert werden muss.

Dies ist der Fall, wenn der kleinere Faktor kleiner als die dritte Wurzel von n ist. In diesem Fall kann auf eine dreiziffrige Darstellung umgeschaltet werden, bei der die Fortschaltung etwas aufwändiger ist.

Für RSA werden die Primfaktoren in gleicher Größenordnung gewählt; das sind in der Praxis weniger als 1%. D.h., der kleinere Faktor liegt nicht weiter als 1% unterhalb der Quadratwurzel. Damit sind bei der absteigenden Suche nur 1% der Zahlen zu verproben, bis der Faktor gefunden wird. Damit sind bei der absteigenden Suche bei 128-Bit RSA von den 2^{64} Werten zunächst nur 2^{63} zu verproben, da nur ungerade Faktoren gesucht werden, und davon nur $10^{-3} \approx 2^{-6}$ Exemplare. Es sind also maximal 2^{57} Durchläufe notwendig; und bei dem zusätzlichen Weglassen der Potenzen von 3 und 5 sind es 2^{56} Durchläufe. Ein moderner 64-Bit Prozessor bewältigt ca, $10^9 \approx 2^{27}$ Verprobungen pro Sekunde. Somit ist die maximale Laufzeit 2^{29} Sekunden. Das Jahr hat

ca. $10^{7.5} \approx 2^{24}$ Sekunden; es werden also auf einem einzigen Rechner $2^5 = 16$ Jahre benötigt. Verwendet man eine CPU mit 8 Kernen, reduziert sich die Zeit auf 2 Jahre. Man benötigt also 24 8-Kern-CPUs, um den Faktor eines 128-Bit RSA-Schlüssels in einem Monat zu finden.

Jedoch sind die Hardware-Anforderungen so gering, dass man 1 Million anstelle von 8 Hardware-Prozessoren auf einem Chip integrieren könnte. Damit reduziert sich die Zeit um den Faktor 10^5; damit kann ein 128-Bit RSA Schlüssel in 2.5 Sekunden gefunden werden, da ein Monat $2.5 \cdot 10^5$ Sekunden hat.

Selbstverständlich ist das in dieser Form keine ernsthafte Bedrohung selbst eines 256-Bit Schlüssels, da die benötigte Zeit um den Faktor $2^{128} \approx 10^{42}$ größer ist.

5.4. Parallelisierbarkeit

Das Verfahren der absteigenden Basis ist besonders einfach parallelisierbar.

5.4.7. Berechnung in Blöcken

Anstatt mit der nächstniedrigen ungeraden Quadratwurzel zu starten, kann mit jeder darunter liegenden Zahl angefangen werden, wenn Faktoren darüber nicht gefunden werden müssen, weil dies beispielsweise ein anderer Prozess bewerkstelligt.

Auf einem Rechner mit vier CPU-Kernen werden dann Bereiche von ca. 10^{10} (10G) oder 2^{32} Zahlen parallel berechnet; findet einer der Prozesse einen Faktor, können die anderen abgebrochen werden.

Die folgenden Beispiele sind mit `openssl` generierte 80-Bit RSA-Exponenten.

Ein ungewöhnlich günstiges Beispiel ist das folgende:

```
838386875135137090196257 = 883345709633 * 949103919329
915634684321 .. 905634684321
905634684321 .. 895634684321
895634684321 .. 885634684321
885634684321 .. 875634684321
```

Der letzte Block findet den Faktor bereits in 1 sec, während ein einzelner Prozess dazu 22 sec benötigt hätte.

Allerdings kann auch schon der erste Block (nach 5sec) fündig werden, und die anderen Prozesse sind vergebens:

```
759660371191114859072413 = 864456301817 * 878772437189
871584976459 .. 861584976459
861584976459 .. 851584976459
851584976459 .. 841584976459
841584976459 .. 831584976459
```

Typisch könnte folgendes Beispiel sein, bei dem der 3. Block den Schlüssel nach 5 sec (statt nach 18 sec) findet:

```
809687365220930168483101 = 873311734553 * 927145866917
899826297249 .. 889826297249
889826297249 .. 879826297249
879826297249 .. 869826297249
869826297249 .. 859826297249
```

Bei einem Versuch mit 16 Schlüsseln mit je 80 Bit und einer Blockgröße von 30G wurden maximal 20sec benötigt, die beiden Faktoren zu finden. In einem Fall war dies der letzte Block; hier wurden 8sec (elapsed) anstelle von 74 sec für einen einzigen Prozessor benötigt.

Der Vorteil dieses Verfahrens besteht auch darin, dass die Blöcke unabhängig von einander verteilt werden können, also eine freie CPU sich einen Block holt und bearbeitet.

5.4.8. Größere Schrittweite

Wie oben ausgeführt, erfordert eine Schrittweite von 15 (ungeraden Werten) ein Vielfaches von 8 anstelle von 15 Prozessen. Da sie mit der o.g. Blockbildung kombiniert werden kann und die Ausnutzung von mehreren Kernen automatisch vom Betriebssystem erfolgt, sollte sie immer eingesetzt werden.

Eine Blockbildung ist jedoch nur notwendig, wenn verteilt gerechnet wird oder mit ein Prozessor mit mehr als 8 Kernen verwendet werden kann.

6. Anhang

6.1. Quadrattests mit der Endziffer

Da eine Quadratzahl in der Dezimaldarstellung nie die Endziffern 2, 3, 7 oder 8 haben kann, müssen bei manueller Auswertung die entsprechenden Zeilen gar nicht verprobt zu werden.

Dies ist auch mit anderen Basen möglich. Die folgende Tabelle gibt Zahlenbasen `b` und die Anzahl `q` der unterschiedlichen *Endziffern* zu dieser Basis an, wobei Zeilen nur dann aufgeführt werden, wenn das Verhältnis `q/b` kleiner als das bisher niedrigste ist:

```
   b    q  q/b
   3    2  0.667
   4    2  0.500
   8    3  0.375
  12    4  0.333
  16    4  0.250
  32    7  0.219
  48    8  0.167
  80   12  0.150
  96   14  0.146
 112   16  0.143
 144   16  0.111
 240   24  0.100
 288   28  0.097
 336   32  0.095
 480   42  0.087
 560   48  0.086
 576   48  0.083
 720   48  0.067
1008   64  0.063
1440   84  0.058
1680   96  0.057
2016  112  0.056
2640  144  0.055
2880  144  0.050
3600  176  0.049
4032  192  0.048
```

Die letzte Zweierpotenz mit guten Wirkungsgrad ist 32, bei die zu testende Zahl modulo 32 nur 5 verschiedene Werte haben kann. Bei größeren Zweierpotenzen wird der Wirkungsgrad größer, steigt aber ab 256 nur noch wenig:

```
    b     q  q/b
    4     2  0.500
    8     3  0.375
   16     4  0.250
   32     7  0.219
   64    12  0.188
  128    23  0.180
  256    44  0.172
  512    87  0.170
 1024   172  0.168
 2048   343  0.167
 4096   684  0.167
 8192  1367  0.167
16384  2732  0.167
32768  5463  0.167
```

Da bei b=16 nur die vier Werte 0, 1, 4 und 9 vorkommen können, erscheint dies als die beste Wahl.

6.2. Aufgaben

Zu Fermats Quadratmethode:

1. Welches sind die Bedingungen, damit ein vorgegebener Rechenbereich, z.B. 64 Bit, nicht überschritten wird?
2. Wie sieht der Rechengang bei Benutzung der Differenz $\dot{v} - v$ aus? Gibt es eine obere Schranke für diese Differenz?
3. Kann man statt mit der Quadratwurzel für x auch mit y = 3 starten? Vor- und Nachteile?
4. Kann man anstelle von x das Doppelte mitführen?
5. Schätze und erprobe die Rechengeschwindigkeit mit und ohne Extrapolation.
6. Wie kann durch Ausschluss kleiner Primfaktoren mittels Probedivision ein Abbruchkriterium aussehen und das Verfahren mit der Aussage *Primzahl* anhalten, bevor $x - y = 1$ ist?
7. Kann man durch Rechnung modulo des Rechenbereichs m eine mehrfachgenaue Arithmetik vermeiden? Bleibt das Endekriterium $v - y^2 = 0$ auch $\mod m$ gültig? Gibt es mehrdeutige Ende-Kriterien?
8. Zeige, dass $x - y = 1$ ein stabiles Abbruchkriterium ist.

Zur Methode der absteigenden Basis:

6. Schätze die Zunahme der Rechenzeit bei $x > p$ und Verwendung von Additionen zur Normalisierung ab.
7. Quantifiziere den Vorteil von k Ziffern gegenüber einer durch Software realisierten mehrfach genauen Arithmetik.
8. Werden bei größeren Abständen $2k$ Multiplikationen eingesetzt, so steigt der Gewinn mit der Anzahl der Prozesse. Abschätzung?
9. Untersuche den Einsatz modularer Arithmetik
10. Skizziere die Methode der absteigenden Basis (ohne Normalisierung) als Kette von Rechenblöcken.
11. Ergänze die vorige Aufgabe um eine Normalisierung
12. Entwickle die Methode der aufsteigenden Basis für den Test auf kleine Primzahlen.

Sonstiges:

12. Lohnt es sich, zum Test kleiner Primzahlen zunächst den Rest der Division durch das Produkt der ersten n Primzahlen zu bestimmen?
13. Beweise die Aussagen zu den Endziffer-Tests von Quadratzahlen.

6.3. Literatur

SQUFOF:
Jason E. Gower, Samuel S. Wagstaff jr.: *SQFF Square Form Factorisation*. http://homes.cerias.purdue.edu/~ssw/squfof.pdf

Hartmann2007:
Peter Hartmann: *Faktorisierungsalgorithmen natürlicher Zahlen.* \n Diplomarbeit, Hamburg 2007. http://www.zahlen.mathematic.de/FnZ.pdf

SchmidtSamoa2002:
Katja Schmidt-Samoa: Das *Number Field Sieve*. Diplomarbeit, Kaiserslautern 2002. http://www.cdc.informatik.tu-darmstadt.de/~samoa/NFS.pdf

Inhalt