Technische Universität Wien
Institut für Diskrete Mathematik und Geometrie
Wiedner Hauptstraße 8
1040 Wien

Diplomarbeit

Big-Integer-Arithmetik in C#

und ihre Anwendung auf digitale Signaturen

Axel Heer

2011

Betreut durch
Ao. Univ. Prof. Dr. Johann Wiesenbauer

Für Petra.

Inhaltsverzeichnis

Vorwort

Integrität und Ursprung von elektronischen Dokumenten wollen genauso sichergestellt werden können, wie die Authentizität physischer Schriftstücke. Bei letzteren ist die händische Unterschrift das übliche Mittel zum Zweck, während immaterielle Bits & Bytes aufgrund der Leistungsfähigkeit zeitgemäßer Computersysteme mit einem deutlich höheren Aufwand geschützt werden müssen. Für eine sogenannte *digitale Signatur* hinreichende mathematische Methoden sind Gegenstand der vorliegenden Arbeit, wobei sowohl auf eine ausführliche Behandlung der dafür notwendigen theoretischen Grundlagen als auch auf eine nahezu praxistaugliche Implementierung Wert gelegt wurde.

Die in den vorgestellten Algorithmen implizit oder explizit verwendeten mathematischen Sätze werden inklusive (hoffentlich leicht) verständlichem Beweis angeführt, sollte es sich nicht gerade um Grundlagen aus Vorlesungen der ersten Semester für einen Studenten der technischen Mathematik handeln. Als Programmiersprache wurde C# .NET in der aktuellen Version 4.0 gewählt, da es sich dabei um eine moderne, gut lesbare, aber auch relativ leistungsstarke Technologie handelt – maschinennahe Optimierungen wie bei C oder sogar Assembler sind hier natürlich nicht möglich, aber im Rahmen dieser Arbeit mit mathematischem Fokus auch nicht angebracht.

Der Hauptteil der nun folgenden Seiten gliedert sich in sechs Kapitel: zu Beginn werden Techniken erarbeitet, die es ermöglichen, mit für aktuelle Computer unüblich großen Zahlen zu rechnen – an dieser Stelle seien ganz unbescheiden Abschnitt 1.9 sowie Abschnitt 5.8 hervorgehoben: die vorgestellte *Big-Integer-Arithmetik* ist trotz ihrer kurzen Entwicklungszeit relativ schnell! Danach werden mit Hilfe dieser Techniken Primzahlen zufälliger Natur erzeugt, um dann im dritten Teil der vorliegenden Arbeit als Grundlage für prominente Kryptosysteme verwendet zu werden. Bevor endlich anhand von praktischen Beispielen die Verwendung von *digitalen Signaturen* demonstriert werden kann, werden noch in einer eigenen Passage drei ausgewählte Hashfunktionen behandelt. Zum Abschluss werden mit Hilfe von weiteren mathematischen Methoden die Algorithmen dieser Arbeit noch einmal beschleunigt, um mittels größerer Schlüsselpaare entsprechend sicherere „Unterschriften" erzeugen zu können.

Axel Heer im Jänner 2011

1. „Große“ Zahlen

1.1. Einleitung

Ein gewöhnlicher Computer kann heutzutage in der Regel mit 64-Bit Binärzahlen umgehen. In der Kryptologie – und somit bei digitalen Signaturen – werden jedoch oft deutlich größere Zahlen benötigt, wenn mathematische Probleme als Grundlage verwendet werden, welche so schwer zu lösen sind, dass sich der entsprechende Rechenaufwand entweder nicht lohnt oder einfach nicht innerhalb eines vernünftigen Zeitraumes durchführen lässt. Es muss also für die vorliegende Arbeit auf einer 64-Bit Rechenmaschine möglich sein, mit Zahlen zu operieren, für deren Darstellung mehrere hundert oder sogar tausende Bits notwendig sind.

Definition 1.1.1 (Basisdarstellung)**.** Die Darstellung einer nicht negativen ganzen Zahl a als Summe der Form $a = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_0 b^0$, kurz $a = (a_n a_{n-1} \ldots a_0)_b$, mit $0 \leqslant a_i < b$ für $0 \leqslant i \leqslant n$, nennt man Basisdarstellung von a zur Basis b.

Hat man die Darstellung einer Zahl a zu einer Basis b, so kann diese leicht in eine alternative Darstellung zu einer Basis b' konvertiert werden. Dazu muss einfach a durch b' geteilt werden: der Divisionsrest ergibt $(a_0)_{b'}$ und mit dem Quotienten $\lfloor \frac{a}{b'} \rfloor$ wird dann dieses Verfahren so lange fortgesetzt, bis alle relevanten a_i gefunden worden sind. So kann beispielsweise 1138 von der gewohnten Dezimalschreibweise zu $(472)_{16}$ oder $(100.0111.0010)_2$ umgerechnet werden.

Zahlen größer als $2^{64} - 1$, also Zahlen, welche sich *nicht* mit maximal 64 Bits darstellen lassen, können mit Hilfe dieser Erkenntnis als Array gespeichert werden, wobei die einzelnen Elemente dieses Arrays als Koeffizienten a_i zur Basis 2^{64} aufgefasst werden. Auf zwei solchen Arrays $(a_n a_{n-1} \ldots a_0)_{2^{64}}$ und $(b_m b_{m-1} \ldots b_0)_{2^{64}}$ können nun Operationen definiert werden, welche den mathematischen Grundrechnungsarten entsprechen. Es wird also im Prinzip ein und dasselbe durchgeführt, wie beim klassischen Rechnen mit Papier und Bleistift, nur eben zu einer etwas größeren Basis.

Einziges Problem an diesem Ansatz ist jedoch, dass bei den einzelnen Schritten dieser Operationen immer wieder „Overflows“ auftreten können. Bei der klassischen dezimalen Addition von 17 und 4 bleibt im ersten Schritt ein Rest, welcher im zweiten Schritt natürlich berücksichtigt werden muss, damit auch 21 herauskommt. Bei der Implementierung der Klasse *IntBig*, welche all diese Operationen auf den oben beschriebenen

Arrays abstrahiert, hat sich herausgestellt, dass es sinnvoller ist, nur 2^{32} als Basis zu wählen, um die einzelnen Schritte der jeweiligen Operationen einfach und vor allem so zu entwickeln, dass die Performance der einzelnen Algorithmen verhältnismäßig (siehe Abschnitt 1.9) gut ist.

1.2. Aufbau der Datenstruktur

Als Grundlage dient eine Klasse *IntBig*, deren Operatoren überladen wurden, um einen syntaktisch möglichst ähnlichen Umgang wie mit den gewöhnlichen Strukturen *int* und *long* zu ermöglichen – zumindest innerhalb von C# bzw. einer alternativen .NET Programmiersprache, welche mit überladenen Operatoren umgehen kann.

Damit diese Datenstruktur auch „vollständig“ mit der .NET Runtime kompatibel ist, also nicht nur gut innerhalb eines C#-Programmes verwendet werden kann, wurde regelmäßig mittels *FxCop*, welcher unter [17] bezogen werden kann, überprüft, ob auch alle Regeln für professionelle .NET-Bibliotheken erfüllt werden. Einzig die Regelgruppe *Globalization Rules* wurde der Einfachheit halber ignoriert, während die Regel [18] (Berücksichtigung von Programmiersprachen, welche das Überladen von Operatoren nicht unterstützen bzw. erlauben) eindeutig am meisten Aufwand verursacht hat.

Des Weiteren wurde darauf geachtet, dass die Klasse in jeder Situation *immutable* ist. Das bedeutet: wenn einmal eine Instanz der Klasse *IntBig* fertig initialisiert wurde, kann ihr Wert auf gar keinen Fall mehr verändert werden. Das ist nicht nur konsistent mit *int* sowie *long*, sondern hat noch weitere Vorteile, welche beispielsweise unter [27] zusammengefasst werden. Vor allem ist der Umgang mit solchen Klassen auf alle Fälle einfacher, wenn man sich unter anderem darauf verlassen kann, dass bei einer Operation $c := a \circ b$ die Werte von a sowie b nicht verändert werden.

Innerhalb der Klasse wird nur zweierlei gespeichert: der Array, welcher den Wert der nicht negativen Zahl in einer Art Darstellung zur Basis 2^{32} enthält, sowie ein Bit, welches festlegt, ob dieser Wert noch mit -1 multipliziert werden soll. Es können also alle Elemente aus $\mathbb{Z}$ dargestellt werden – unendlich viel Hauptspeicher sowie beliebig große Arrays vorausgesetzt. Für die vorliegende Arbeit sind es auf jeden Fall betragsmäßig hinreichend große ganze Zahlen.

Neben den Operatoren und Funktionen, welche dann zur weiteren Verwendung der Klasse *IntBig* zur Verfügung stehen, also als *public* markiert sind, wurden die eigentlichen Algorithmen in Funktionen implementiert, welche ausschließlich mit Arrays sowie Zahlen operieren und eben nicht auf diverse Objekte zugreifen. Auch wenn das auf den ersten Blick ein wenig seltsam klingen mag, es wird somit dem Compiler mehr Spielraum für Optimierungen gegeben, was vor allem bei der Performance des Multiplikationsalgorithmus aufgefallen ist.

Beispiel 1.2.1. Die öffentlich aufrufbare Funktion *Subtract* führt nur die zuerst notwendigen Validierungen und Fallunterscheidungen durch, bevor der eigentliche Algorithmus *subtract* ausgeführt wird.

```
public static IntBig Subtract(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (left.isNegative != right.isNegative) {
        return Add(left, Negate(right));
    }
    var diff = compare(left.innerBits, right.innerBits);
    if (diff < 0) {
        var bits = subtract(right.innerBits, left.innerBits);
        return new IntBig(bits, !left.isNegative);
    }
    else {
        var bits = subtract(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative);
    }
}
```

Der Algorithmus *subtract* kann dann davon ausgehen, dass der Wert von *left* betragsmäßig größer oder gleich dem Wert von *right* ist. Um das Vorzeichen muss er sich nicht kümmern, ihm ist dieses Bit gar nicht erst bekannt.

```
static uint[] subtract(uint[] left, uint[] right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");
    Debug.Assert(compare(left, right) >= 0,
                 "compare(left, right) >= 0");

    // subtracts right from left
}
```

Bei einem Debug-Build werden jedoch die Bedingungen, welche für das korrekte Ausführen eines Algorithmus erfüllt sein müssen, sicherheitshalber überprüft, um Programmierfehler möglichst früh zu bemerken. Alle Aufrufe der im .NET Framework enthaltenen Klasse *System.Diagnostics.Debug* werden praktischerweise im Release-Build einfach nicht übersetzt – die Verwendung von Preprocessor-Makros o. ä. entfällt.

Bemerkung 1.2.2. Im weiteren Verlauf dieses Kapitels werden derartige Sicherheitsmechanismen der Übersicht halber weggelassen. Die jeweils ungekürzte Fassung des Quelltextes kann im Anhang A nachgeschlagen werden.

1.3. Ordnung

Wie bereits in Beispiel 1.2.1 angedeutet, ist es für manche Algorithmen der Klasse *IntBig* notwendig zu wissen, welcher der beiden Operanden der betragsmäßig größere ist. C# bietet hier die Möglichkeit, die Operatoren $=$, $\neq$, $<$, $\leqslant$, $>$ sowie $\geqslant$ zu überladen – das wäre jedoch immer fast der gleiche Code.

Deswegen wurde hier ein Konzept übernommen, welches in vielen anderen Klassen des .NET Frameworks gefunden werden kann, zum Beispiel bei *System.String*. Neben den Vergleichsoperatoren existiert dort eine Methode *Compare*, welche zwei Instanzen der zugehörigen Klasse vergleicht und folgendes Resultat liefert: -1 bzw. einen Integer < 0, sollte der erste Operand der kleinere sein, $+1$ bzw. einen Integer > 0 im umgekehrten Fall sowie 0 bei Gleichheit.

Beispiel 1.3.1. Für einen der aufgelisteten Relationsoperatoren R gilt ganz einfach a R b genau dann, wenn *Compare*(a, b) R 0. Somit sind die Implementierungen dieser Operatoren nur mehr triviale „Einzeiler“.

```
public static bool operator <(IntBig left, IntBig right) {
    return Compare(left, right) < 0;
}
```

Innerhalb von *Compare* müssen zuerst die Vorzeichen der beiden Operanden behandelt werden: bei Ungleichheit ist ganz einfach der negative Wert der kleinere; bei Gleichheit muss das Ergebnis von *compare* unter Umständen noch negiert werden.

Algorithmus 1.3.2. Die vorzeichenunabhängige Funktion *compare*.

```
static int compare(uint[] left, uint[] right) {
    if (left.Length < right.Length) {
        return -1;
    }
```

```
5         if (left.Length > right.Length) {
6             return 1;
7         }
8         for (int i = left.Length - 1; i >= 0; i--) {
9             if (left[i] < right[i]) {
10                return -1;
11            }
12            if (left[i] > right[i]) {
13                return 1;
14            }
15        }
16        return 0;
17    }
```

In den Zeilen 2 bis 7 werden zuerst einmal die Längen der beiden Arrays verglichen, wobei davon ausgegangen wird, dass keine führenden Nullen vorhanden sind. Anschließend, also bei gleicher Länge, werden in einer *for*-Schleife die einzelnen Elemente verglichen, wobei so früh wie nur möglich abgebrochen wird. Sollte kein Schleifendurchgang den Algorithmus abgeschlossen haben, liegt eine Gleichheit vor und es wir 0 retourniert.

1.4. Shifts

Bei einem binären Shift werden die einzelnen Bits in der Binärdarstellung einer Zahl verschoben, wobei die dadurch leer werdenden Stellen in der Regel mit Nullen gefüllt werden. Bei *int* und *long* bekommen hingegen negative Zahlen, welche nach rechts verschoben werden, Einser angehängt, was mit der Darstellung von negativen Zahlen als Zweierkomplement (siehe [6]) zusammenhängt.

Neben diversen dadurch möglichen Tricks hat ein Shift auf jeden Fall den Vorteil, dass auf diese Art und Weise sehr schnell mit 2^n multipliziert bzw. dividiert werden kann. Bei „normalen" Rechenoperation werden solche Situationen in der Regel vom Compiler erkannt und automatisch optimiert. Bei der Klasse *IntBig* hingegen müssen derartige Verbesserungen, also das Ersetzen von einer Multiplikation bzw. Division mit 2^n durch einen passenden Shift um n Stellen, händisch gemacht werden.

Algorithmus 1.4.1. Der Shift nach rechts um n Stellen.

```
static uint[] rightShift(uint[] value, int shift) {
    var fastShift = shift / 32;
    shift = shift % 32;

```

```
5      if (value.Length <= fastShift) {
6          return new uint[1];
7      }
8
9      var bits = new uint[value.Length - fastShift];
10
11     if (shift == 0) {
12         for (int i = 0; i < bits.Length; i++) {
13             bits[i] = value[i + fastShift];
14         }
15     }
16     else {
17         for (int i = 0; i < bits.Length - 1; i++) {
18             bits[i] = (value[i+fastShift] >> shift)
19                     | (value[i+fastShift+1] << (32-shift));
20         }
21         bits[bits.Length-1] = value[value.Length-1] >> shift;
22     }
23
24     return bits;
25 }
```

Zu Beginn wird die Anzahl der zu verschiebenden Stellen in den Anteil, welcher komplette Sprünge innerhalb des Arrays verursacht, und in den Rest, welcher mit separaten 32-Bit Shifts behandelt werden muss, unterteilt. Wird der komplette Array „weggeschoben“, so sind wir bereits fertig.

In den Zeilen 11 bis 22 wird je nachdem, ob mit zusätzlichen 32-Bit Shifts gearbeitet werden muss, die Verschiebung entsprechend durchgeführt, wobei im zweiten Fall immer jene Bits, welche durch den 32-Bit Shift nach rechts verloren gehen, mit Hilfe eines inversen Shifts auf das jeweils nächste Element übertragen werden müssen.

Bemerkung 1.4.2. Bei einem Shift nach rechts kann eine unerwünschte führende Null entstehen. Jedoch wird diese dann beim Initialisieren einer neuen Instanz der Klasse *IntBig* vom Konstruktor automatisch wieder entfernt. Genauso wird im Konstruktor sichergestellt, dass keine „negative Null“ als Wert möglich ist.

Bemerkung 1.4.3. Der Shift nach links verläuft zum Großteil völlig analog und wird deswegen nicht extra behandelt. Einzig zu beachten ist, dass bei einer Verschiebung nach links keine Bits verloren gehen können.

1.5. Addition

Werden zwei nicht negative ganze Zahlen $(a_n a_{n-1} \dots a_0)_{2^{32}}$ und $(b_n b_{n-1} \dots b_0)_{2^{32}}$ gleicher Länge dargestellt zur Basis 2^{32} addiert, so geht man analog zur klassischen Methode mit Papier und Bleistift folgendermaßen vor: im ersten Schritt wird die Summe der ersten beiden Koeffizienten modulo der Basis 2^{32} als erster Koeffizient des Ergebnisses eingetragen. Sollte der Quotient $\lfloor \frac{a_0+b_0}{2^{32}} \rfloor$ ungleich Null sein, so wird er für den nächsten Schritt übertragen.

Im i-ten Schritt wird dann der Übertrag u aus dem vorherigen Schritt zur Summe dazu genommen. Da ein Koeffizient maximal den Wert $2^{32}-1$ haben kann, gilt für die Summe $a_i + b_i + u \leqslant 3(2^{32}-1) < 2^{34}$, wir können das Ergebnis also in einer Variable vom Type *long* speichern. Da im letzten Schritt der Übertrag u natürlich nicht gleich Null sein muss, werden für das Ergebnis $n+2$ Koeffizienten vorgesehen.

Modulo 2^{32} wird am schnellsten gerechnet, indem einfach nach den ersten 32 Bits abgeschnitten, also auf *uint* konvertiert, wird. Der Quotient $\lfloor \frac{a_i+b_i+u}{2^{32}} \rfloor$ lässt sich in einem Computerprogramm am einfachsten mit einem Shift des Zählers um 32 Bits nach rechts auswerten. Es wird also die Summe $a_i + b_i + u$ von drei 32-Bit Zahlen in einer 64-Bit Zahl gespeichert, wobei die ersten 32 Bits den gewünschten Koeffizienten und die restlichen Bits den Übertrag u für den nächsten Schritt ergeben.

Algorithmus 1.5.1. Die Addition zweier nicht negativer ganzer Zahlen.

```
static uint[] add(uint[] left, uint[] right) {
    var bits = new uint[left.Length + 1];
    var carry = 0L;

    for (int i = 0; i < right.Length; i++) {
        var digit = (left[i] + carry) + right[i];
        bits[i] = (uint)digit;
        carry = digit >> 32;
    }
    for (int i = right.Length; i < left.Length; i++) {
        var digit = left[i] + carry;
        bits[i] = (uint)digit;
        carry = digit >> 32;
    }
    bits[bits.Length - 1] = (uint)carry;

    return bits;
}
```

Hier wird vorausgesetzt, dass der linke Array mehr oder gleich viele Werte wie der rechte Operand umfasst, was bei einem Aufruf von *add* entsprechend berücksichtigt werden muss. Der rechte Teil wird indirekt mit führenden Nullen auf die gleiche Länge gebracht, indem bei Indizes größer oder gleich dessen Länge der rechte Koeffizient in der Summe einfach weggelassen wird. Der Übertrag wird in der Variable *carry* gespeichert, welche durch die Initialisierung mit Suffix *L* implizit als *long* deklariert ist.

In der ersten der beiden *for*-Schleifen werden die zuvor beschriebenen Operationen Schritt für Schritt durchgeführt, wobei die Variable *digit* aufgrund der Klammerung 64 Bits umfasst. Die zweite Schleife ist nur dazu da, um den rechten Array nicht explizit auf die Länge des linken Arrays bringen zu müssen – das spart wertvolle Millisekunden!

Bemerkung 1.5.2. Der eigentliche Additionsoperator muss, bevor er die Arrays an die Funktion *add* übergeben kann, die Vorzeichen der beiden Operanden berücksichtigen: sind die beiden Vorzeichen gleich, so werden die Werte mittels *add* addiert, das Ergebnis bekommt dann das gleiche Vorzeichen. Sind die Vorzeichen jedoch unterschiedlich, so wird, nachdem der zweite Operand negiert wurde, der Subtraktionsoperator aufgerufen. Schließlich gilt $a + (-b) = a - b$ bzw. $(-a) + b = (-a) - (-b)$.

Bemerkung 1.5.3. Die Variable *carry* kann genaugenommen immer nur den Wert 1 oder 0 annehmen, wie man sich leicht überlegen kann.

1.6. Subtraktion

Das Subtrahieren von zwei nicht negativen ganzen Zahlen $a = (a_n a_{n-1} \dots a_0)_{2^{32}}$ und $b = (b_n b_{n-1} \dots b_0)_{2^{32}}$ gleicher Länge mit $b \leqslant a$ verläuft nach einem relativ ähnlichen Schema wie die Addition. Statt der Summe $a_i + b_i + u$ wird einfach $a_i - b_i + u$ berechnet, wobei der Übertrag u diesmal anders zustande kommt: ist das Ergebnis von $a_i - b_i + u$ kleiner als Null, so muss im nächsten Schritt der Koeffizient a_i des Minuenden um Eins vermindert werden, ansonsten wird u gleich Null gesetzt.

Diese Regel lässt sich in Binärarithmetik wieder sehr effizient implementieren. Dank der Darstellung von negativen Zahlen als Zweierkomplement ist das letzte Bit gesetzt, sollte der Wert kleiner Null sein; umgekehrt ist es das nicht für nicht negative Zahlen. Somit lässt sich diese Fallunterscheidung auflösen, indem das Ergebnis von $a_i - b_i + u$ um 63 Bits nach rechts geschoben wird.

Algorithmus 1.6.1. Die Subtraktion zweier nicht negativer Zahlen verläuft analog zur Addition, nur wird $a_i - b_i + u$ gerechnet und für den Wert von *carry* um 63 statt um nur 32 Bits verschoben. Der Algorithmus wird deswegen nicht weiter behandelt.

Bemerkung 1.6.2. Wie immer kann der vollständige Quelltext, in diesem Fall von *subtrakt*, im Anhang A nachgeschlagen werden.

1.7. Multiplikation

Beim Multiplizieren von zwei nicht negativen ganzen Zahlen $(a_n a_{n-1} \ldots a_0)_{2^{32}}$ und $(b_m b_{m-1} \ldots b_0)_{2^{32}}$ – wobei diesmal keine gleiche Länge vorausgesetzt werden muss – multipliziert sich auch der Rechenaufwand: bis jetzt war für die Laufzeit nur einer der beiden Arrays ausschlaggebend (nämlich der größere), was hier nicht mehr der Fall ist, wie dann am Algorithmus sofort zu sehen ist.

Wieder wird genauso vorgegangen, wie beim klassischen Rechnen mit Papier und Bleistift: ein Koeffizient a_i des ersten Faktors wird mit einem b_j des zweiten Operanden multipliziert und zum Koeffizienten c_{i+j} des Ergebnisses $(c_{n+m+1} c_{n+m} \ldots c_0)_{2^{32}}$ addiert. Diese Rechenschritte müssen für alle i mit $0 \leqslant i \leqslant n$ und für alle j mit $0 \leqslant j \leqslant m$ durchgeführt werden.

Da unter anderem das Produkt von zwei 32-Bit Zahlen etwas mehr als nur 32 Bits umfassen kann, ist wieder die Berücksichtigung eines Übertrags u notwendig, welcher dann im nächsten Schritt entsprechend dazu addiert werden muss. Schließlich darf der Übertrag u, welcher nicht mehr in c_{i+j} Platz findet, bei der Berechnung von c_{i+j+1} nicht vergessen werden.

Zusammenfassend wird also in jedem Schritt $c_{i+j} + a_i b_j + u$ berechnet, was sich nach oben mit $2(2^{32} - 1) + (2^{32} - 1)^2 = 2^{64} - 1$ abschätzen lässt. Und das passt perfekt (!) in eine Variable vom Typ *ulong*.

Algorithmus 1.7.1. Die Multiplikation zweier nicht negativer ganzer Zahlen.

```
static uint[] multiply(uint[] left, uint[] right) {
    var bits = new uint[left.Length + right.Length];

    for (int i = 0; i < right.Length; i++) {
        var carry = 0UL;
        for (int j = 0; j < left.Length; j++) {
            var digits = bits[i + j] + carry
                + (ulong)left[j] * (ulong)right[i];
            bits[i + j] = (uint)digits;
            carry = digits >> 32;
        }
        bits[i + left.Length] = (uint)carry;
    }

    return bits;
}
```

Das Array für das Ergebnis der Multiplikation wird mit einer Länge von $n + m + 2$ reserviert. Anschließend wird mittels zwei ineinander verschachtelten *for*-Schleifen das beschriebene Verfahren durchgeführt: jeder Koeffizient des rechten Faktors wird der Reihe nach mit den Koeffizienten des linken Operanden multipliziert.

Die Laufzeit von Algorithmus 1.7.1 lässt sich noch mit einem kleinen Trick für gleichwertige Parameter nahezu halbieren. Da in Kapitel 2 Zahlen sehr häufig quadriert werden, um unter anderem Potenzen innerhalb eines Restklassenkörpers zu berechnen, wurde in den Multiplikationsoperator der Klasse *IntBig* eine Weiche eingebaut, welche mittels *object.ReferenceEquals* erkennt, ob es sich bei der angeforderten Multiplikation eigentlich um das Quadrieren von nur einer Zahl handelt.

Grundlage der Optimierung ist die Tatsache, dass für $a = (a_n a_{n-1} \dots a_0)_{2^{32}}$ und $(b_n b_{n-1} \dots b_0)_{2^{32}}$, mit $a_i = b_i$ für alle i, $a_j b_i = a_i b_j$ gelten muss. Die innere *for*-Schleife kann also entsprechend gekürzt werden, wobei diesmal auf „Overflows“ geachtet werden muss. Der Wert von $c_{i+j} + 2a_i b_j + u$ hat naturgemäß nicht mehr innerhalb einer 64-Bit Variable Platz, was mit diversen Shifts kompensiert wird, welche wiederum ein wenig „Overhead“ verursachen, der allerdings in Relation zur Halbierung der inneren Schleifendurchläufe seine Rechenzeit wert ist.

Algorithmus 1.7.2. Das Quadrieren einer nicht negativen ganzen Zahl.

```
static uint[] square(uint[] value) {
    var bits = new uint[value.Length * 2];

    for (int i = 0; i < value.Length; i++) {
        var carry = 0UL;
        for (int j = 0; j < i; j++) {
            var digits1 = bits[i + j] + carry;
            var digits2 = (ulong)value[j] * (ulong)value[i];
            bits[i + j] = (uint)(digits1 + (digits2 << 1));
            carry = (digits2 + (digits1 >> 1)) >> 31;
        }
        var digits = (ulong)value[i] * (ulong)value[i] + carry;
        bits[i * 2] = (uint)digits;
        bits[i * 2 + 1] = (uint)(digits >> 32);
    }

    return bits;
}
```

Die Berechnung von $c_{i+j} + 2a_i b_j + u$ wird auf zwei Schritte aufgeteilt, um im Rahmen von 64-Bit Variablen zu bleiben. Da in C# das „Overflow"-Bit einer Addition nicht abgefangen werden kann, ist eine zusätzliche Operation notwendig, wobei zuerst die Bits von $a_i b_j$ einfach nach links verschoben werden und danach das erste Bit von $c_{i+j} + u$ weggelassen wird, welches aufgrund des Shifts sowieso keinen Übertrag verursachen kann.

Bemerkung 1.7.3. Es gibt effizientere Ansätze für derartige Multiplikationsalgorithmen (zum Beispiel [7]), jedoch zahlt sich so eine Implementierung für Zahlen mit „nur" ein paar tausend Bits nicht aus. Der hier vorgestellte Algorithmus ist von der Laufzeit her mehr als ausreichend (siehe auch Abschnitt 1.9).

1.8. Division und Divisionsrest

Der mit Abstand komplizierteste und auch (mit weniger Abstand) rechenaufwendigste Algorithmus der Klasse *IntBig* ist die Division bzw. der Divisionsrest. Für dessen Implementierung hat grundsätzlich auch das gewöhnliche Rechnen mit der Hand eine ganz gute Grundlage geboten, allerdings mussten noch ein paar „Tricks" eingebaut werden, um eine akzeptable Laufzeit zu erreichen.

Im ersten Schritt wird herausgefunden, wie viele führende Nullen der letzte Koeffizient des Divisors im Rahmen seiner 32-Bit Binärdarstellung hat. Um möglichst wenige Rechenschritte für die Division durchführen zu müssen, werden sowohl der Divisor als auch der Dividend um eben diese Anzahl an Bits nach links geschoben, wodurch der Wert des gesuchten Quotienten natürlich nicht verändert wird.

Dann wird der in seiner Darstellung zur Basis 2^{32} im Verhältnis zum Dividenden ganz nach links geschobene Divisor vom Dividenden abgezogen, so lange er in dieser Form kleiner oder gleich dem Dividenden ist. Der entsprechende Koeffizient des Ergebnisses muss dann nach der Subtraktion um Eins erhöht werden. Aufgrund des Shifts im ersten Schritt kann pro Position maximal eine Subtraktion durchgeführt werden.

Anschließend, wenn also der „ganz nach links" geschobene Divisor größer als der Dividend ist, müssen Schritt für Schritt die weiteren Koeffizienten des Quotienten abgeschätzt werden. Als Grundlage dient dabei $\lfloor \frac{a_n 2^{32} + a_{n-1}}{b_m} \rfloor$, wofür einmal die „Probe" gerechnet wird. Ist die Schätzung zu hoch (zu niedrig kann sie ja nicht sein), so wird sie um Eins vermindert. Dieser Vorgang wird so lange fortgesetzt, bis die Schätzung passt. Damit ist der gesuchte Koeffizient des Ergebnisses gefunden und der Dividend kann wieder entsprechend reduziert werden.

Zum Schluss muss noch der Shift aus dem ersten Schritt repariert werden, schließlich passt zwar der gesuchte Quotient, der Divisionsrest wird jedoch um die verschobenen Bits zu groß sein. Dieser „Fix" ist natürlich nur notwendig, wenn auch der Divisionsrest und nicht nur der Quotient gesucht ist.

Algorithmus 1.8.1. Die Division zweier nicht negativer ganzer Zahlen.

```
var bits = new uint[dividend.Length - divisor.Length + 1];

var shifted = leadingZeroCount(divisor[divisor.Length - 1]);
dividend = leftShift(dividend, shifted);
divisor = leftShift(divisor, shifted);

var divisorLength = actualLength(divisor);
var dividendLength = actualLength(dividend);
var divHigh = divisor[divisorLength - 1];
```

Der Algorithmus geht davon aus, dass der Dividend betragsmäßig größer als der Divisor ist. Die Hilfsfunktion *leadingZeroCount* führt eine Art binäre Suche durch, um die Anzahl der führenden Nullen herauszufinden. Da hier ausnahmsweise die Arrays direkt weiterverarbeitet werden (um Zeit zu sparen), wird jeweils in *divisorLength* als auch in *dividendLength* die „richtige“ Länge (ohne führende Nullen) ermittelt und gespeichert.

```
var diff = compareWithFastShift(dividend, dividendLength,
    divisor, divisorLength, dividendLength - divisorLength);
while (diff >= 0) {
    ++bits[dividendLength - divisorLength];
    subtractInplaceWithFastShift(dividend, dividendLength,
      divisor, divisorLength, dividendLength - divisorLength);
    dividendLength = actualLength(dividend, dividendLength);
    diff = compareWithFastShift(dividend, dividendLength,
      divisor, divisorLength, dividendLength - divisorLength);
}
```

Mittels *compareWithFastShift* wird festgestellt, ob der entsprechend positionierte Divisor kleiner oder gleich dem Dividenden ist. Innerhalb der *while*-Schleife wird die zuvor beschriebene Subtraktion durchgeführt sowie der passende Koeffizient des Quotienten erhöht. Die Werte *dividendLength* und *diff* müssen dabei immer aktualisiert werden.

```
var i = dividendLength - 1;
while (i >= divisorLength) {
    var guess = 0xffffffff;
    if (dividend[i] != divHigh) {
        guess = (uint)((dividend[i - 1]
            + ((ulong)dividend[i] << 32)) / divHigh);
    }
```

Die beschriebene Schätzung des aktuellen Koeffizienten des Ergebnisses. Sollte a_n gleich b_m sein, so kommt es zu einem numerischen „Overflow“ – in dem Fall wird gleich mit dem maximal möglichen Wert gearbeitet. Dass a_n größer als b_m ist, kann dabei aufgrund der vorherigen Schritte nicht vorkommen.

```
    var guessedDivisor = multiply(divisor, guess);
    diff = compareWithFastShift(dividend, dividendLength,
        guessedDivisor, actualLength(guessedDivisor),
        i - divisorLength);
    while (diff < 0) {
        --guess;
        guessedDivisor = multiply(divisor, guess);
        diff = compareWithFastShift(dividend, dividendLength,
            guessedDivisor, actualLength(guessedDivisor),
            i - divisorLength);
    }
```

Die Schätzung wird so lange nach unten korrigiert, bis sie passt. Dafür wurde eine eigene *multiply* Funktion implementiert, welche für die Multiplikation eines Arrays mit nur einem Skalar ausgelegt ist, womit wieder wertvolle Rechenzeit gespart werden kann, da das Anlegen eines einelementigen Arrays unnötige Ressourcen verbraucht.

```
    subtractInplaceWithFastShift(dividend, dividendLength,
        guessedDivisor, actualLength(guessedDivisor),
        i - divisorLength);
    dividendLength = actualLength(dividend, dividendLength);
    bits[i - divisorLength] = guess;
    i = dividendLength - 1;
}
```

Am Ende der Schleife wird dann der Dividend um den mit der korrekten Schätzung multiplizierten Divisor, welcher zusätzlich um die passende Anzahl an Stellen nach links geschoben wird, reduziert. Dieser Teil entspricht also wieder weitgehendst der händischen Division im Dezimalsystem.

```
modulus = rightShift(dividend, shifted);
return bits;
```

Für einen korrekten Divisionsrest muss noch der Shift aus dem ersten Schritt rückgängig gemacht werden. Die Variable *bits* enthält das Ergebnis.

1.9. Performance

Im .NET Framework ist seit der Version 4.0 eine *Big-Integer-Arithmetik* enthalten, welche es genauso wie die hier vorgestellte Klasse *IntBig* ermöglicht, mit relativ großen Zahlen zu rechnen. Sie ist in einem eigenen Assembly *System.Numerics* enthalten und hört auf den Namen *BigInteger*.

Der Gedanke, eine Arithmetik zu entwickeln, welche schneller rechnen kann als die des Softwaregiganten Microsoft, war naturgemäß äußerst reizvoll, wodurch auf die Performance der Klasse *IntBig* sehr großen Wert gelegt wurde. Die Datenstruktur von Microsoft wurde bereits Anfang 2007 vorgestellt (siehe [28]), dann aber aufgrund von Performanceproblemen wieder zurückgezogen (siehe [29]), um schließlich Anfang diesen Jahres (also 2010) in der finalen Fassung gemeinsam mit der Version 4.0 des .NET Frameworks endgültig freigegeben zu werden.

Um es gleich vorweg zu nehmen: die Klasse *IntBig* rechnet in der Tat schneller als *BigInteger* von Microsoft!

Getestet wurde folgendermaßen: mittels Maple wurden 100.000 Paare (a_i, b_i) von Zufallszahlen generiert, mit denen dann die einzelnen Operationen durchgeführt wurden, wobei $2^{512} \leqslant a_i < 2^{1024}$ und $2^{256} \leqslant b_i < 2^{512}$ für alle i vorgegeben wurde. Dadurch sind nicht nur die Zahlen selbst sondern auch deren Länge relativ abwechslungsreich. Und das Behandeln von Vorzeichen wurde absichtlich vermieden – es sollten nur die reinen vorzeichenunabhängigen Rechenvorgänge miteinander verglichen werden.

Der erste Anlauf, also die erste funktionstüchtige Version von *IntBig* gegen *BigInteger* antreten zu lassen, war noch nicht erfolgreich – die Ergebnisse jedoch motivierend: *IntBig* war, abgesehen von der Division, nur im zweistelligen Prozentbereich langsamer als *BigInteger*. Da die Division von *IntBig* sehr stark von den anderen Operationen abhängt, wurde dieser Bereich zuerst einmal ignoriert.

Die *Addition* war sehr leicht zu beschleunigen: in der ursprünglichen Version wurden die beiden Arrays auf dieselbe Länge gebracht, was in der aktuellen Version durch das Aufteilen in zwei *for*-Schleifen vermieden wird. Bei der *Subtraktion* war zusätzlich der in Abschnitt 1.6 beschriebene „Shift-Trick“ notwendig, um eine explizite Fallunterscheidung nach dem Vorzeichen der Variable *digit* zu vermeiden.

Bei der *Multiplikation* ist am meisten Zeit in die Optimierung geflossen, und das, obwohl diese überraschend einfach ausgefallen ist: die beiden *for*-Schleifen haben in der Urversion direkt auf die Objekte vom Typ *IntBig* zugegriffen. Eine Separation, wodurch ausschließlich mit Zahlen und Arrays gearbeitet wird, gibt offensichtlich der .NET Runtime bessere Möglichkeiten, den Code auszuführen. Den Sonderfall des *Quadrierens* zu behandeln (siehe Algorithmus 1.7.2), hat im Benchmark den Abstand zu *BigInteger* um weitere Millisekunden vergrößern können.

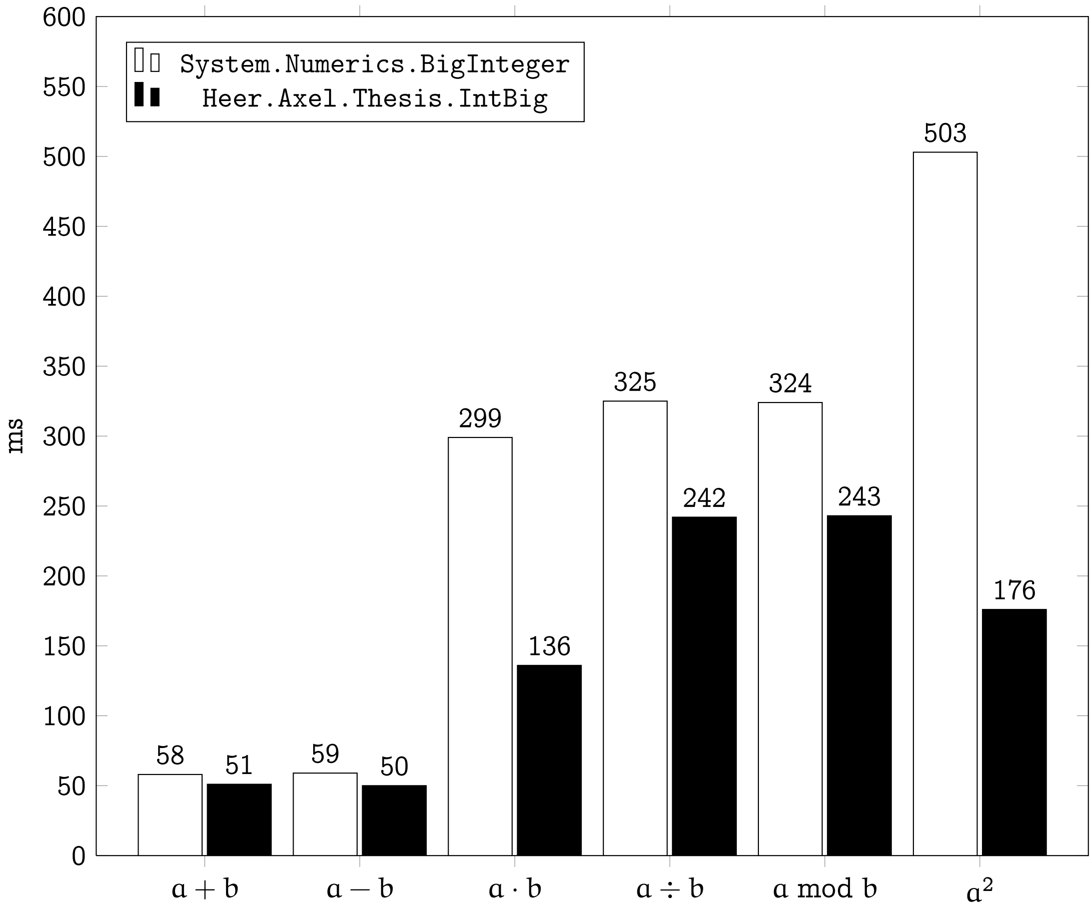

Abbildung 1.1.: Laufzeit von 100.000 Operationen (gemischt)

Last but not least wurde der Algorithmus zur *Division* öfters umstrukturiert, um möglichst wenig neue Arrays zu instantiieren und vor allem den Array des Dividenden immer weiter zu verwenden. Dadurch war es notwendig, einige Methoden in einer angepassten Version zu entwickeln, welche den Divisor immer um die angegebene Anzahl an Stellen implizit verschieben. Auch die Implementierung einer eigenen Multiplikationsroutine, welche auf die Verarbeitung eines Arrays mit einem Skalar ausgelegt ist, hat die Laufzeit um einige Millisekunden reduziert.

Das Ergebnis des Benchmarks *IntBig* vs. *BigInteger*, welcher auf einer 64-Bit CPU mit 3 GHz unter Windows 7 durchgeführt wurde, ist in Abbildung 1.1 dargestellt. Die Datenstruktur des .NET Frameworks in der Version 4.0 arbeitet also teilweise nicht einmal halb so schnell wie die für diese Arbeit in C# entwickelte Arithmetik.

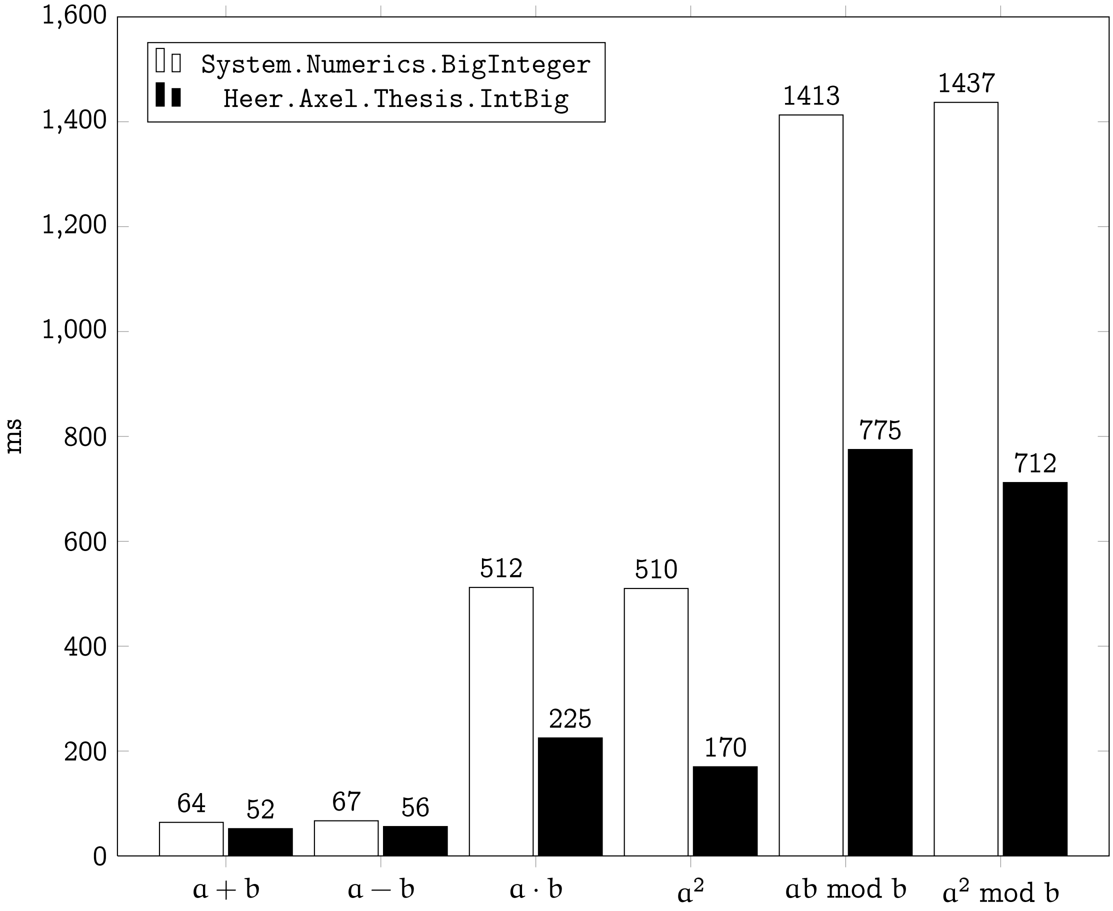

Abbildung 1.2.: Laufzeit von 100.000 Operationen (1024-Bit)

Um den üblichen Verhältnissen in der Kryptographie eher zu entsprechen und somit ein aussagekräftigeres Ergebnis für die Anwendung bei *digitalen Signaturen* präsentieren zu können, wurde weiters ein Benchmark durchgeführt, für den $2^{1023} \leqslant a_i < 2^{1024}$ und $2^{1023} \leqslant b_i < 2^{1024}$ für alle i vorgegeben wurde. Zusätzlich wurde anstatt reiner Division- bzw. Moduloperationen näher an der Praxis gemessen, wodurch man sieht, dass im Verhältnis zur Rechenzeit des Moduloperators der Geschwindigkeitsvorteil beim Quadrieren nahezu untergeht. Wie wir in Kapitel 5 sehen werden, ist diese Optimierung dennoch entscheidend.

In Abschnitt 5.6 wird die Multiplikationsroutine für Zahlen mit mehr als ca. 2048 Bits weiter beschleunigt werden. Als kleinen Ausblick auf die dann mögliche Rechenzeit sei in Abbildung 1.3 der gleiche Benchmark aber mit 4096-Bit Werten angegeben.

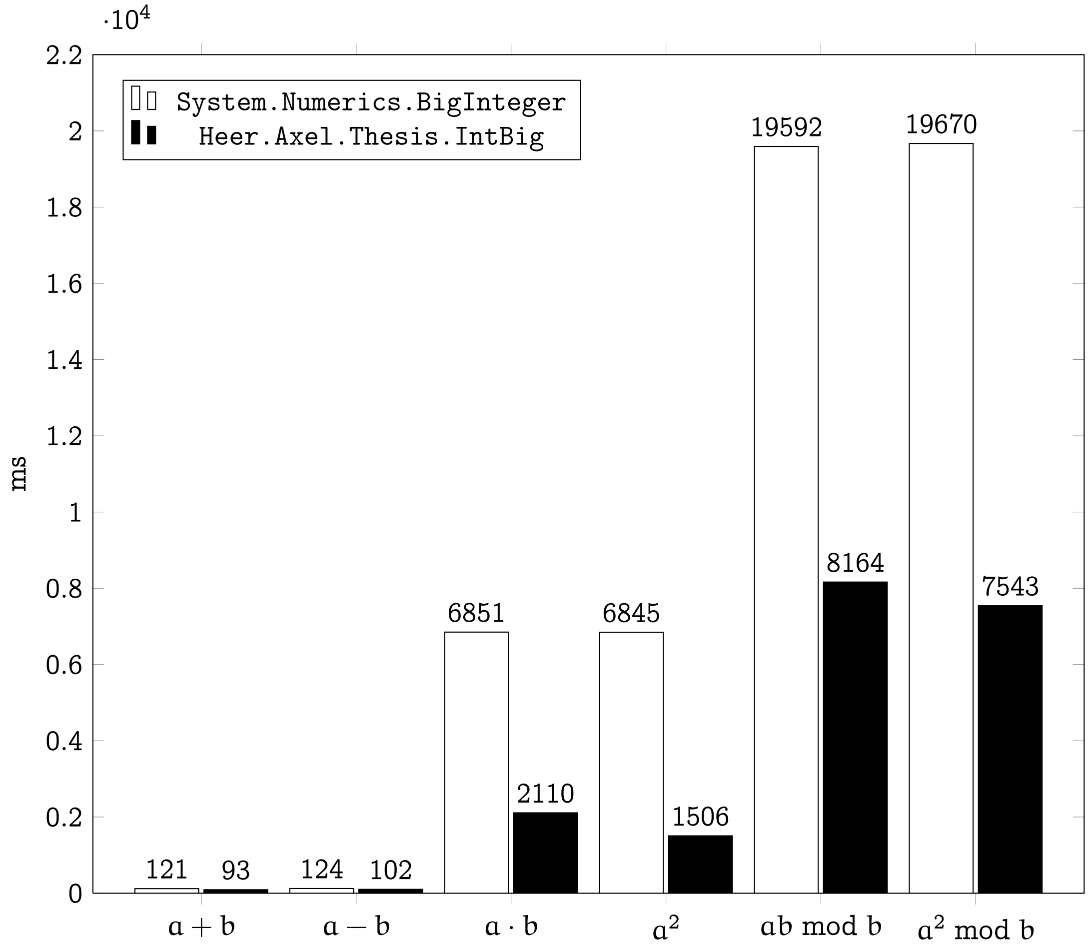

Abbildung 1.3.: Laufzeit von 100.000 Operationen (4096-Bit)

2. Primzahlen

2.1. Einleitung

Positive Zahlen größer oder gleich 2, die nur 1 sowie sich selbst als Teiler besitzen, spielen in asymmetrischen Kryptosystemen, welche unter anderem für digitale Signaturen verwendet werden können, oft eine wichtige Rolle, da gewisse Probleme in Zusammenhang mit solchen Primzahlen derartig komplex sind, dass sich deren Berechnung quasi nicht durchführen lässt. Natürlich muss hier auch leicht zu Berechnendes mit im Spiel sein: konkret sei als Beispiel die Gleichung

$$h = g^x \pmod p$$

gegeben. Ist h zu berechnen, so ist die Lösung – unter der Annahme, dass g, x sowie p bekannt sind – verhältnismäßig schnell gefunden, siehe Algorithmus 2.1.3. Soll hingegen x gefunden werden, während h, g und p angegeben sind, so sind keine effizienten Methoden bekannt, welche dieses Problem lösen.

Bemerkung 2.1.1. Das angeführte Beispiel bildet die Grundlage für den Schlüsselaustausch nach Diffie und Hellman, welche diese kryptographisch interessante Gleichung 1976 entdeckt haben. Mehr dazu ist unter [2] oder unter [8] zu finden.

Eine weitere gute Grundlage für derartige Kryptosysteme bildet die Komplexität der Primfaktorzerlegung, also die Darstellung einer natürlichen Zahl n als Produkt der Form

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

wobei p_i für alle i eine Primzahl ist und zwecks Eindeutigkeit $1 < p_1 < p_2 < \cdots < p_k$ vorausgesetzt wird. Dass diese Darstellung einer Zahl relativ komplex zu berechnen ist, wurde von Rivest, Shamir und Adleman benutzt, um das nach wie vor sehr populäre RSA-Kryptosystem zu entwickeln (siehe [9]).

Bemerkung 2.1.2. Eine kryptographische „Endlösung“ stellen solche Systeme leider nicht dar. Zahlen, von denen noch 1977 angenommen wurde, dass deren Zerlegung in Primfaktoren so gut wie unmöglich ist, wurden 17 Jahre danach mit fortschrittlicheren Methoden sowie modernerer Hardware erfolgreich zerlegt (siehe [10]).

Da für kryptographische Zwecke Primzahlen benötigt werden, welche aufgrund ihrer Größe nicht in einer 64-Bit Variable gespeichert werden können, wurde in Kapitel 1 eine Datenstruktur vorbereitet, welche nun genutzt werden soll, um bei einer „großen“ Zahl testen zu können, ob es sich um eine Primzahl handelt. In weiterer Folge sollen dann Primzahlen gefunden werden, für deren ungefähre Größe eine Anzahl an Bits (exklusive führende Nullen) vorgegeben ist.

Doch bevor ein konkreter Primzahltest behandelt werden kann, wird noch eine grundlegende mathematische Operation benötigt, welche von der Klasse *IntBig* nicht geboten wird – das Potenzieren. Da diese Datenstruktur analog zu *int* und *long* verwendet werden soll und eben in C# kein Potenzieroperator vorhanden ist, wurde dafür ein passendes Gegenstück zur Klasse *System.Math* entwickelt: *MathBig*.

Für einen effizienten Potenzieralgorithmus kann die Binärdarstellung des Exponenten ausgenutzt werden. Angenommen der Algorithmus soll a^b berechnen, so kann b in der Form $k_0 2^0 + k_1 2^1 + \cdots + k_n 2^n$ dargestellt werden, womit

$$a^b = a^{k_0 2^0 + k_1 2^1 + \cdots + k_n 2^n} = (a^{2^0})^{k_0} (a^{2^1})^{k_1} \cdots (a^{2^n})^{k_n}$$

gilt. Diese per definitionem $b - 1$ Multiplikationen können also drastisch reduziert werden, indem schrittweise quadriert und bei $k_i = 1$ zusätzlich multipliziert wird.

Algorithmus 2.1.3. Das Potenzieren von *value* mit Exponenten *power*.

```
static IntBig Pow(IntBig value, IntBig power) {
    IntBig result = 1;

    while (!power.IsZero) {
        if (power.IsOdd) {
            result = value * result;
        }
        value = value * value;
        power = power >> 1;
    }

    return result;
}
```

Das Iterieren über alle Bits des Exponenten in Binärdarstellung wird gelöst, indem abgefragt wird, ob der aktuelle Wert von *power* ungerade ist, also $k_0 = 1$ gilt. Vor dem jeweils nächsten Schleifendurchlauf wird *power* um ein Bit nach rechts geschoben, also unter anderem $k_0 := k_1$ gesetzt.

Sollte nun das aktuelle Bit des Exponenten gesetzt sein, so wird mit dem aktuellen Wert von *value* multipliziert, dessen Wert in jedem Durchlauf quadriert wird.

Bemerkung 2.1.4. Zeile 8 in Algorithmus 2.1.3 profitiert von der in Algorithmus 1.7.2 durchgeführten Optimierung des Multiplikationsalgorithmus für zwei gleiche Zahlen. Der hier vorgestellte Potenzieralgorithmus war auch der Anlass für diese Erweiterung des Multiplikationsoperators.

2.2. Rabin-Miller-Test

Um eine Zahl $n > 1$ zu testen, könnte man einfach für alle $1 < t \leqslant \sqrt{n}$ probieren, ob t ein Teiler von n ist. Das dauert allerdings für „große“ Zahlen viel zu lange – genauso wie alle anderen bekannten Verfahren, welche in deterministischer Art und Weise einen Primzahltest durchführen.

Deswegen werden in der Kryptographie sogenannte probabilistische Tests verwendet, also Primzahltests, welche mit einer gewissen Fehlerwahrscheinlichkeit feststellen, ob n eine Primzahl ist. Sowohl diese Fehlerwahrscheinlichkeit als auch die Laufzeit eines derartigen Tests gilt es, so gering wie nur möglich zu halten, was naturgemäß nicht kompromisslos durchführbar ist.

Solche Tests nutzen in der Regel Eigenschaften aus, welche zwar notwendig, jedoch nicht hinreichend sind dafür, dass es sich bei n um eine Primzahl handelt.

Hilfssatz 2.2.1 (Kleiner Fermat)**.** *Sei* p *prim,* $a \in \mathbb{Z}$ *mit* $1 \leqslant a < p$*. Dann gilt*

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

Beweis. Induktion nach a für $1 \leqslant a < p - 1$.

$a = 1$:

$$1^{p-1} \equiv 1 \pmod{p}.$$

$a \to a + 1$:

$$\begin{aligned} (a+1)^p &= \sum_{k=0}^{p} \binom{p}{k} a^{p-k} \equiv a^p + 1 \pmod{p}, \\ \overset{\text{Ind.}}{\Rightarrow} \quad (a+1)^p &\equiv a + 1 \pmod{p}, \\ \Rightarrow \quad (a+1)^{p-1} &\equiv 1 \pmod{p}. \end{aligned}$$

□

Bemerkung 2.2.2. Schreibt man $a^p \equiv a$ statt $a^{p-1} \equiv 1$ so gilt der Satz für beliebige $a \in \mathbb{Z}$. Jedoch ist diese allgemeinere Version für den weiteren Verlauf dieses Abschnitts nicht interessant.

Satz 2.2.3. *Sei* p *eine ungerade Primzahl und* $2^s r = p - 1$ *mit* r *ungerade. Weiters sei* $a \in \mathbb{Z}$ *mit* $1 \leqslant a < p$. *Dann gilt entweder*

$$a^r \equiv 1 \pmod p$$

oder

$$a^{2^j r} \equiv -1 \pmod p$$

für ein j *mit* $0 \leqslant j < s$.

Beweis. Ist p eine Primzahl, dann gilt nach Hilfssatz 2.2.1

$$a^{2^s r} \equiv 1 \pmod p.$$

Sei q eine Zahl mit der Eigenschaft

$$q^2 \equiv 1 \pmod p,$$

dann gilt

$$\begin{aligned} q^2 \equiv 1 \pmod p \quad &\Leftrightarrow \quad p \mid (q^2 - 1) \\ &\Leftrightarrow \quad p \mid (q-1) \ \vee \ p \mid (q+1) \\ &\Leftrightarrow \quad q \equiv 1 \pmod p \ \vee \ q \equiv -1 \pmod p. \end{aligned}$$

Die Folge

$$(a^r, a^{2r}, \dots, a^{2^s r})$$

muss also entweder gleich

$$(1, 1, \dots, 1)$$

oder von der Form

$$(a^r, a^{2r}, \dots, -1, 1, \dots, 1)$$

sein. □

Diese Eigenschaft einer Primzahl kann nun genutzt werden, um auf probabilistische Art und Weise zu überprüfen, ob es sich bei n um eine Primzahl handelt. Dazu wird a zufällig gewählt, und das t-mal, wobei in [1], Tabelle 4.4, eine Übersicht über empfohlene t in Abhängigkeit von n zu finden ist.

Algorithmus 2.2.4. Der Rabin-Miller-Primzahltest.

```
bool isProbablePrime(IntBig value, int tryCount) {
    if (value.IsNegative || value.IsZero || value.IsOne) {
        return false;
    }
    if (!value.IsOdd) {
        return false;
    }
```

Zu Beginn werden einmal die trivialen Fälle abgefangen. Für Zahlen kleiner oder gleich 1 und gerade Zahlen ist der Aufwand von Satz 2.2.3 einfach nicht notwendig. Die Variable *tryCount* entspricht dem Parameter t, welcher zuvor von der Methode *IsProbablePrime* der Klasse *PrimeBig* in Abhängigkeit von *value* gewählt wird.

```
    var s = 0;
    IntBig r = value - 1;

    while (!r.IsOdd) {
        ++s;
        r = r >> 1;
    }
```

Die Berechnung von $2^s r = p - 1$ mit r ungerade wird mit Hilfe von Shifts nach rechts erledigt. Da in weiterer Folge Zufallszahlen getestet werden, sollte die Anzahl der Nullen nach dem ersten Bit gering sein – schließlich werden die Werte der Bits mit jeweils gleicher Wahrscheinlichkeit gewählt.

```
    for (int i = 0; i < tryCount; i++) {
        IntBig a = generator.Next(value - 3) + 2;
        IntBig y = MathBig.Pow(a, r, value);
```

Die Variable *generator* ist eine Instanz der Klasse *RandomBig*, welche in Abschnitt 2.3 genauer behandelt wird – die Methode *Next* generiert eine nicht negative Zufallszahl, die kleiner als der übergebene Wert sein soll. Somit enthält a eine Zufallszahl mit $2 \leqslant a \leqslant n - 2$, für die gleich anschließend a^r mittels einer alternativen Version von Algorithmus 2.1.3 berechnet wird, welche zusätzlich in jedem Schritt modulo n arbeitet.

Ist der Wert von y gleich 1 oder gleich $n - 1$, so sind wir fertig – bei n handelt es sich dann wahrscheinlich um eine Primzahl. In Integer-Arithmetik entspricht ein $\equiv -1 \pmod n$ schließlich einem $\equiv n-1 \pmod n$, was in Abschnitt 1.8 nachvollzogen werden kann. Anderenfalls muss y im Rahmen der Folge aus dem Beweis von Satz 2.2.3 so lange quadriert werden, bis es $n - 1$ entspricht.

```
        if (!y.IsOne && !(value - y).IsOne) {
            var j = 1;
            while (j < s && !(value - y).IsOne) {
                y = (y * y) % value;
                if (y.IsOne) {
                    return false;
                }
                ++j;
            }
            if (!(value - y).IsOne) {
                return false;
            }
        }
```

Da in diesem letzten Abschnitt des Rabin-Miller-Tests nur mehr ein Wert von -1 interessant ist, wird bei 1 gleich abgebrochen, schließlich kann dann dieser Wert in einem Restklassenkörper durch quadrieren nicht mehr zu -1 werden.

```
    }

    return true;
}
```

Sollte der Parameter *value* die beschriebenen Schritte t-mal überstanden haben, so kann mit einer gewissen Fehlerwahrscheinlichkeit *true* retourniert werden. Implementiert ist dieser Algorithmus in der Klasse *PrimeBig*, welche auch eine Methode enthält, die den Parameter *tryCount* automatisch, wenn auch etwas vorsichtiger als in [1], Tabelle 4.4, angegeben, wählt.

2.3. Zufallsprimzahlen

Wie schon im letzten Abschnitt angedeutet, wird ein Zufallszahlengenerator für Instanzen der Klasse *IntBig* benötigt. Sprachen bzw. Frameworks wie C# und die .NET Runtime bieten in der Regel einen einfachen Mechanismus, um Zufallszahlen zu erzeugen – in unserem Fall ist es die Klasse *System.Random*.

Diese Klasse generiert initial einen sogenannten *seed*, also einen zufälligen 32-Bit Startwert (laut [19] auf Basis der aktuellen Uhrzeit), mit dem dann alle folgenden 32-Bit „Zufallszahlen“ auf eben nicht zufällige Art und Weise berechnet werden. Für die meisten Anwendungen ist so eine Methode vollkommen ausreichend, für kryptographische

Zwecke jedoch mangelhaft. Was hilft schon eine mühsam gesuchte Primzahl im 1.000-Bit-Bereich, wenn nur die ersten 32 Bits zufällig erzeugt wurden?

Um sicher zu gehen, dass auch wirklich alle Bits von möglichst zufälliger Natur sind, müssen in der Regel Messdaten der Hardware (Temperatur, Auslastung, Lüfterdrehzahl, ...) oder (und) Eingaben des Benutzers (Tastaturanschläge, Mausbewegungen, Mausklicks, ...) für *jeden* 32-Bit Block einer Zufallszahl berücksichtigt werden. Somit könnte dann für jeden dieser Blöcke eine neue Instanz von *System.Random* mit jeweils unterschiedlichem *seed* initialisiert werden, was natürlich um einiges zeitaufwendiger, allerdings unbedingt notwendig ist.

Gerade im Zeitalter des Internets gibt es eine noch viel elegantere Variante, um an „echte" Zufallszahlen zu kommen: das Konsumieren eines professionellen Webdienstes, der auf Basis diverser physikalischer Messdaten gute Zufallszahlen generiert. Unter *random.org* ist zum Beispiel ein derartiges Projekt zu finden. Diese Seite würde sogar eine gute Grundlage bieten, um Zufallszahlen für diese Arbeit zu erstellen – der entsprechende Teil des Dienstes wird kostenlos zur Verfügung gestellt. Allerdings wäre dann eine permanente Internetverbindung notwendig.

Da unter Windows bereits eine Schnittstelle existiert, um Zufallszahlen für kryptographische Anwendungen zu erzeugen, wurde dieser API der Vorzug gegeben. Der für diese Zwecke interessante Teil des sogenannten „Cryptographic service providers" wird von der im .NET Framework enthaltenen Klasse *RNGCryptoServiceProvider* für .NET Sprachen wie C# zur Verfügung gestellt und ist somit relativ einfach für die Erzeugung zufälliger Werte vom Typ *IntBig* zu verwenden.

Die Klasse *RandomBig* abstrahiert diesen Vorgang, womit in anderen Teilen des Projekts direkt „gute" Zufallszahlen benutzt werden können.

Algorithmus 2.3.1. Der Zufallszahlengenerator für *IntBig*.

```
public IntBig Next(int bitCount, bool makeOdd) {
    var bytes = new byte[(bitCount + 7) / 8];
    generator.GetBytes(bytes);
    if (bitCount % 8 != 0) {
        var value = (byte)
            (bytes[bytes.Length - 1] << (8 - bitCount % 8));
        bytes[bytes.Length - 1] = (byte)
            (value >> (8 - bitCount % 8));
    }
    if (makeOdd) {
        bytes[0] |= 1;
    }
```

```
13      return IntBig.FromByteArray(bytes);
14  }
```

Zuerst wird ein für die in *bitCount* geforderte Anzahl an Bits hinreichend dimensionierter neuer Byte-Array initialisiert, welcher anschließend mittels *generator*, einer Instanz der Klasse *RNGCryptoServiceProvider*, befüllt wird. Sollten die letzten Bits zu viel sein – schließlich werden immer 8-Bit-Blöcke mit zufälligen Bits erzeugt – so werden diese in den Zeilen 4 bis 9 wieder abgeschnitten.

Für das Erzeugen neuer Primzahlen sind sowieso nur ungerade Zahlen als Kandidaten interessant, weswegen noch ein zusätzliches Flag *makeOdd* eingebaut wurde, welches für das Setzen des entsprechenden Bits zuständig ist. Die Funktion *FromByteArray* von *IntBig* wandelt schlussendlich das generierte Array um.

Bemerkung 2.3.2. Die in Algorithmus 2.2.4 benutzte Version von *Next* zählt zuerst die Anzahl der Bits des übergebenen Parameters und generiert dann ähnlich wie der soeben vorgestellte Algorithmus eine neue Zahl, von der ggf. noch ein Bit wieder abgeschnitten wird. Aufgrund der hohen Ähnlichkeit mit Algorithmus 2.3.1 wird dieser Algorithmus jedoch nicht extra behandelt.

Das Finden von neuen Zufallszahlen, bei denen es sich um eine Primzahl handelt, ist mit den bis jetzt vorgestellten Mitteln ganz einfach: erzeuge so lange eine neue ungerade Zufallszahl mit der gewünschten Anzahl an Bits, bis sie laut *PrimeBig.IsProbablePrime* wahrscheinlich eine Primzahl ist!

Algorithmus 2.3.3. Das Erzeugen einer neuen Zufallsprimzahl.

```
public IntBig NextProbablePrime(int bitCount) {
    var result = generator.Next(bitCount, true);
    while (!IsProbablePrime(result)) {
        result = generator.Next(bitCount, true);
    }
    return result;
}
```

2.4. Stabile Primzahlen

Bei Kryptosystemen, welche auf der Komplexität der Primfaktorzerlegung aufbauen (wie zum Beispiel RSA), ist es laut [1] sinnvoll, noch weitere Bedingungen an eine Zahl p zu stellen – abgesehen davon, dass es eine Primzahl sein muss. Diese Bedingungen führen dazu, dass diverse Attacken möglichst schwierig sind. Am wichtigsten ist es jedoch nach wie vor, p möglichst groß und zufällig zu wählen.

Definition 2.4.1. Eine Primzahl p nennt man *stabil*, falls ganze Zahlen r, s und t existieren, sodass die folgenden Bedingungen erfüllt sind.

(i) $p - 1$ hat einen großen Primfaktor r.

(ii) $p + 1$ hat einen großen Primfaktor s.

(iii) $r - 1$ hat einen großen Primfaktor t.

Bemerkung 2.4.2. Die Attacken, welche den Anlass für diese Definition darstellen, werden in [1], Abschnitt 8.2.2, beschrieben.

Um also *stabile* Primzahlen zu erhalten, darf man nicht alles dem Zufall überlassen. (Und das „Kontrollieren“ einer Zufallszahl wäre auch nicht sonderlich sinnvoll – Stichwort Komplexität der Primfaktorzerlegung.) Im nun folgenden Algorithmus werden s und t zufällig gewählt, aus denen dann die eigentliche Primzahl p abgeleitet wird.

Algorithmus 2.4.3. Das Erzeugen einer *stabilen* Zufallsprimzahl.

```
public IntBig NextProbableStrongPrime(int bitCount) {
    var s = NextProbablePrime(bitCount / 2 - 32);
    var t = NextProbablePrime(bitCount / 2 - 32);
```

Die Primzahlen s und t müssen nur halb so groß wie die eigentlich geforderte Primzahl p sein. Zusätzlich muss noch ein wenig Spielraum vorhanden sein, um dem Wert der Variable *bitCount* auch möglichst genau entsprechen zu können.

```
    var t2 = t << 1;
    var r = (t << 32) + 1;

    while (!IsProbablePrime(r)) {
        r = r + t2;
    }
```

Hier wird das erste Element der Folge $(2^{32}t + 1, (2^{32} + 2)t + 1, (2^{32} + 4)t + 1, (2^{32} + 6)t + 1, \ldots)$ gesucht, bei dem es sich um eine Primzahl handelt. Dieses entspricht der Zahl r aus Definition 2.4.1, wobei r höchstwahrscheinlich $\frac{bitCount}{2}$ Bits haben wird.

```
    var rs2 = (r * s) << 1;
    var p = ((MathBig.Pow(s, r - 2, r) * s) << 1) - 1;

    p = p + (rs2 << (bitCount - rs2.CountBits()));
    while (!IsProbablePrime(p)) {
        p = p + rs2;
    }
```

Die gesuchte Primzahl ist das erste Element der Folge $(p + 2^\beta rs, p + (2^\beta + 2)rs, p + (2^\beta + 4)rs, p + (2^\beta + 6)rs, \ldots)$, bei dem es sich um eine Primzahl handelt. Dabei muss der Exponent β ganz einfach so gewählt werden, dass der Wert von $2^\beta rs$ genau die gewünschte Anzahl an Bits liefert.

```
    return p;
}
```

Die von *NextProbableStrongPrime* gefundene Zahl erfüllt alle Bedingungen von Definition 2.4.1, was an dieser Stelle natürlich noch begründet werden soll.

Beweis (Korrektheit von Algorithmus 2.4.3). Nach Hilfssatz 2.2.1 gilt

$$s^{r-1} \equiv 1 \pmod r.$$

Daraus folgt

$$\begin{aligned} p_0 := 2(s^{r-2} \bmod r)s - 1 &\equiv 1 \pmod r \text{ und} \\ p_0 &\equiv -1 \pmod s. \end{aligned}$$

Die Punkte von Definition 2.4.1 können folgendermaßen erfüllt werden.

(i) $p - 1 = p_0 + 2jrs - 1 \equiv 0 \pmod r$. Also gilt $r \mid (p-1)$.

(ii) $p + 1 = p_0 + 2jrs + 1 \equiv 0 \pmod s$. Also gilt $s \mid (p+1)$.

(iii) $r - 1 = 2it \equiv 0 \pmod t$. Also gilt $t \mid (r-1)$.

□

2.5. Performance

Wie schnell kann nun eine *stabile* Zufallsprimzahl erzeugt werden? Mit den in dieser Arbeit vorgestellten Algorithmen in der Programmiersprache C# können auf einem aktuellen Computer in durchaus annehmbarer Zeit entsprechende Zahlen im 1000-Bit-Bereich generiert werden.

Die Performance-Tests wurden auf einer 64-Bit CPU mit 3 GHz unter Windows 7 durchgeführt, wobei jeweils mehrere Durchläufe mit je 100 Iterationen gemacht wurden – aufgrund der Zufallskomponente waren die Ergebnisse teilweise sehr unterschiedlich. Das gemittelte Ergebnis eines dieser durchgeführten Benchmarks ist in Abbildung 2.1 dargestellt.

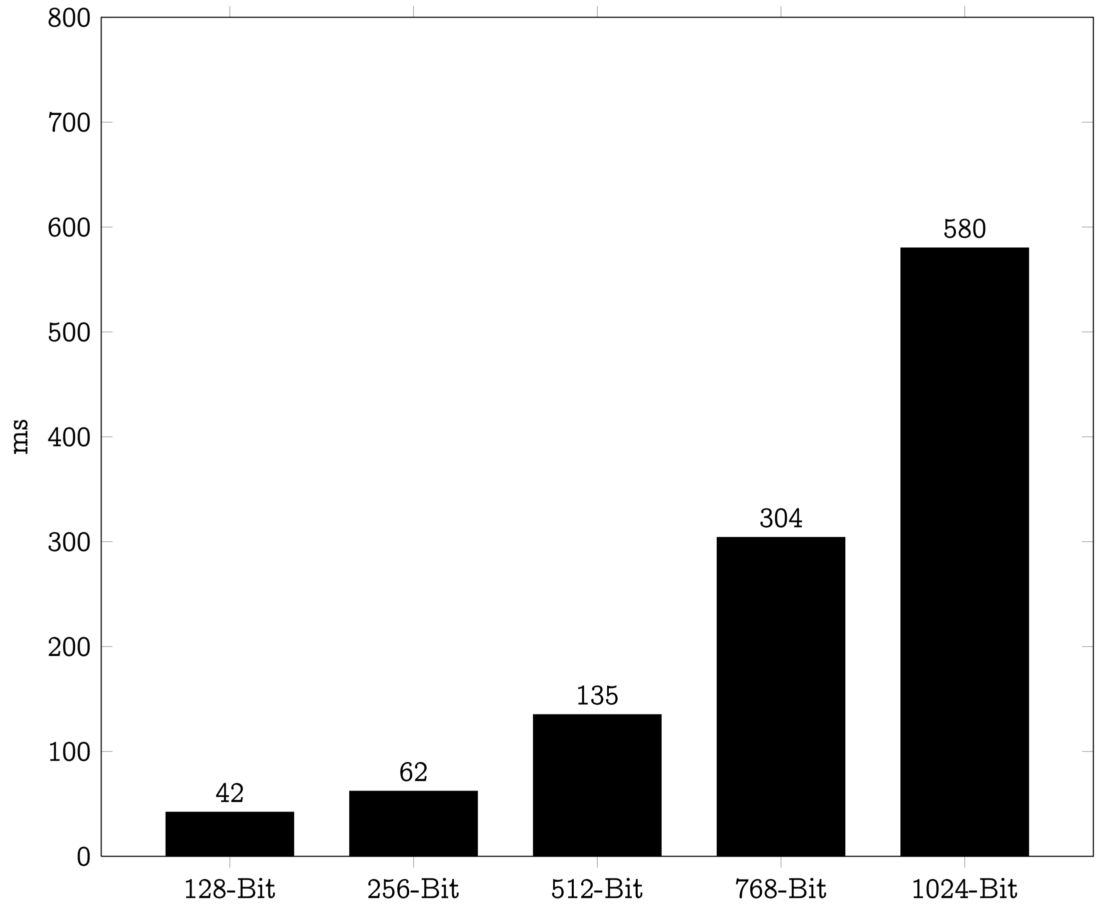

Abbildung 2.1.: Erzeugung einer *stabilen* Zufallsprimzahl

3. Asymmetrische Kryptosysteme

3.1. Einleitung

Wie schon in der Einleitung zu Kapitel 2 angedeutet, können asymmetrische Kryptosysteme für die Erstellung von digitalen Signaturen verwendet werden. In so einem System generiert jeder Anwender ein Schlüsselpaar, wobei jeweils ein Schlüssel für das „Aufsperren" – das Entschlüsseln – sowie der andere Schlüssel für das „Zusperren" – das Verschlüsseln – zuständig ist. Dabei muss der erste Schlüssel unbedingt geheim gehalten werden, während der zweite veröffentlicht werden kann.

Formaler: ein derartiges System stellt zwei Funktionsscharen E_x und D_x, mit $x \in S$, zur Verfügung, sowie zusätzlich eine Methode bzw. einen Algorithmus zur Erzeugung von gültigen Schlüsselpaaren (e, d), mit $(e, d) \in S \times S$, wobei die Menge S möglichst „groß" sein soll, was anhand von „klassischen" Schlüsseln leicht nachvollzogen werden kann. Jedem solchen Schlüsselpaar (e, d) sind nun in eindeutiger Art und Weise die zwei Funktionen E_e und D_d zugeordnet – es wird also strikt zwischen einerseits der „Nummer" des Schlüssels und andererseits dessen „Funktion" unterschieden.

Für jede Nachricht m aus einer vom System definierten Nachrichtenmenge M soll nun $m = D_d(E_e(m))$ bzw. ganz allgemein $id = D_d \circ E_e$ für ein gültiges – und nur für ein gültiges! – Schlüsselpaar (e, d) gelten. Die Abbildungen E_x und D_y müssen also zueinander invers sein, wenn es sich bei (x, y) um ein ordnungsgemäß nach der definierten Methode generiertes Schlüsselpaar handelt; im Optimalfall nur dann. Bei der Menge M wird in der Regel eine geeignete Teilmenge der ganzen Zahlen gewählt, welche gut mit Byte-Arrays einer gewissen Größe identifiziert werden kann.

Damit so ein Kryptosystem auch seinen Zweck erfüllt, darf aus dem öffentlichen Schlüssel e nicht auf den privaten Schlüssel d geschlossen werden können – zumindest muss der dafür notwendige Rechenaufwand verhältnismäßig gewaltig sein. Schlüsselpaare sind dann in der Regel von der Form $(f(d), d)$, mit $d \in S$ und $f(d) \in S$, wobei $e := f(d)$ relativ leicht zu berechnen sein soll und der Aufwand für das Finden von $d := f^{-1}(e)$ hingegen mit guter Hardware „Jahrtausende" benötigen sollte.

Ein Anwender so eines Systems kann also von seinem Schlüsselpaar den Wert e veröffentlichen, um mittels E_e verschlüsselte Nachrichten zu erhalten, welche sich unter D_d wieder auf die ursprüngliche Form abbilden lassen. Bleibt dabei d geheim und ist es weiters in Bezug auf den Rechenaufwand nicht wirklich möglich, aus e den sogenannten privaten Schlüssel d zu konstruieren, so kann auf eben asymmetrische Art und Weise ein sicherer Kommunikationskanal aufgebaut werden.

In der Praxis werden solche Verfahren allerdings nur verwendet, um „kleine“ Nachrichten zu verarbeiten, da ein deutlich höherer Rechenaufwand als bei symmetrischen Systemen zu bewältigen ist. Für einen sicheren Kommunikationskanal kann zum Beispiel über einen asymmetrischen Prozess der Schlüssel für ein schnelleres symmetrisches Protokoll ausgetauscht werden, um dann mit weniger „Overhead“ kommunizieren zu können. Umgekehrt kann der Ursprung von Daten, welche mittels D_d „verschlüsselt“ wurden, mit Hilfe von E_e kontrolliert werden – dazu später mehr.

Im Folgenden werden die beiden populärsten Kryptosysteme, die auf dem soeben beschriebenen Prinzip basieren, vorgestellt, um diese dann im darauffolgenden Kapitel für die Erstellung digitaler Signaturen verwenden zu können.

3.2. Allgemeine Algorithmen

In diesem Abschnitt werden Algorithmen behandelt, welche von den hier vorgestellten Kryptosystemen benötigt werden. Da diese von mathematisch allgemeiner Natur sind, also nicht speziell für ein derartiges Kryptosystem entwickelt wurden, werden die nun folgenden Algorithmen an dieser Stelle separat zusammengefasst.

Satz 3.2.1. *Sei* $a \in \mathbb{Z}^+$ *und* $b \in \mathbb{Z}^+$. *Dann gilt*

$$\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a \bmod b).$$

Beweis. Sei $d := \mathrm{ggT}(a, b)$, $d' := \mathrm{ggT}(b, a \bmod b)$ und $q := \lfloor \frac{a}{b} \rfloor$. Dann gilt

$$\begin{aligned} & d \mid a \wedge d \mid b \\ \Rightarrow\quad & d \mid a \wedge d \mid qb \\ \Rightarrow\quad & d \mid a - qb \\ \Rightarrow\quad & d \mid a \bmod b, \end{aligned}$$

also $d \mid d'$. Umgekehrt gilt

$$\begin{aligned} & d' \mid b \wedge d' \mid a \bmod b \\ \Rightarrow\quad & d' \mid qb \wedge d' \mid a - qb \\ \Rightarrow\quad & d' \mid a, \end{aligned}$$

also $d' \mid d$. Woraus die Behauptung folgt. □

Algorithmus 3.2.2. Die Berechnung des größten gemeinsamen Teilers.

```
public static IntBig Gcd(IntBig left, IntBig right) {
    var a = Abs(left);
    var b = Abs(right);

    while (!b.IsZero) {
        var c = a % b;
        a = b;
        b = c;
    }

    return a;
}
```

Die Implementierung eines effizienten Algorithmus zur Berechnung des größten gemeinsamen Teilers auf Basis der zuvor angeführten theoretischen Grundlage. Dieser verläuft ähnlich zur klassischen Version des euklidischen Algorithmus, nur wird mittels Divisionen schneller das Ziel erreicht.

Bemerkung 3.2.3. Für noch viel „größere“ Zahlen existieren deutlich kompliziertere Algorithmen, um den größten gemeinsamen Teiler noch schneller berechnen zu können (siehe zum Beispiel [2], Kapitel 9).

Bemerkung 3.2.4. Für zwei ganze Zahlen a und b gilt $\mathrm{kgV}(a, b) = \frac{|ab|}{\mathrm{ggT}(a,b)}$, wie man sich leicht überlegen kann, weswegen der entsprechende Algorithmus *Lcm* der Klasse *MathBig* an dieser Stelle nicht extra besprochen wird.

Eine erweiterte Version des euklidischen Algorithmus kann benutzt werden, um zu einem $a \in \mathbb{Z}_n$ das multiplikativ inverse Element b, mit $ab \equiv 1 \pmod n$, zu finden – sofern es existiert. Dazu wird $\mathrm{ggT}(a, b)$ in der Form $xa + yb$ als ganzzahlige Linearkombination von a und b dargestellt.

Um die Existenz dieser Darstellung einzusehen, sei $x_1 = 0$, $x_2 = 1$, $y_1 = 1$ sowie $y_2 = 0$. Für a und b gilt nun

$$\begin{aligned} a &= x_2 a + y_2 b \text{ und} \\ b &= x_1 a + y_1 b. \end{aligned}$$

Multipliziert man die zweite Zeile mit $q := \lfloor \frac{a}{b} \rfloor$, um diese anschließend von der ersten Zeile abzuziehen, so folgt

$$a - qb = (x_2 - qx_1)a + (y_2 - qy_1)b.$$

Jetzt entspricht $a - qb$ der Variable c in Algorithmus 3.2.2, es muss also nur noch die Berechnung der x_i und y_i in die Iterationen entsprechend integriert werden. Nach dem letzten Schleifendurchlauf gilt dann

$$\mathrm{ggT}(a, b) = x_2 a + y_2 b.$$

Wählt man $a := n$, so folgt sofort

$$\mathrm{ggT}(n, b) \equiv y_2 b \pmod{n}.$$

Sollten also n und b teilerfremd sein, so enthält die Variable *y2* des dann folgenden Algorithmus am Ende – nach der Schleife – den gesuchten Wert des multiplikativ inversen Elements zu b. Es gilt dann

$$y_2 b \equiv 1 \pmod{n}$$

für $\mathrm{ggT}(n, b) = 1$. Anderenfalls kann dieses Inverse gar nicht existieren: der gemeinsame Teiler von n und b wäre dann auch ein Teiler von 1. Widerspruch.

Algorithmus 3.2.5. Die Berechnung des multiplikativ inversen Elements.

```
public static IntBig Inv(IntBig value, IntBig mod) {
    var a = Abs(mod);
    var b = Abs(value);

 // IntBig x2 = 1, x1 = 0;
    IntBig y2 = 0, y1 = 1;

    while (!b.IsZero) {
        var q = a / b;
        var r = a - q * b;
     // var x = x2 - q * x1;
        var y = y2 - q * y1;

        a = b; b = r;
     // x2 = x1; x1 = x;
        y2 = y1; y1 = y;
    }

    if (!a.IsOne) {
        throw new ArgumentOutOfRangeException("value");
    }
```

```
    if (y2.IsNegative) {
        y2 = y2 + Abs(mod);
    }

    return y2;
}
```

Die Variablen *x*, *x1* sowie *x2* werden nicht benötigt, wenn nur das inverse Element und nicht die komplette Linearkombination gesucht ist. Sie wurden deshalb auskommentiert, jedoch der Vollständigkeit halber nicht entfernt. Da in dieser Variante explizit das Inverse gesucht wird, führt ein $\mathrm{ggT}(a, n) \neq 1$ zu einer *ArgumentOutOfRangeException*. Zum Schluss wird noch sichergestellt, dass ein positiver Wert retourniert wird.

Alternativ könnte per *out*-Parameter bzw. via Definition einer eigenen Datenstruktur das Tripel (a, x_2, y_2) retourniert werden, was dann genau dem *Erweiterten euklidischen Algorithmus* entsprechen würde (siehe auch [11]).

3.3. RSA

Das wohl bekannteste asymmetrische Kryptosystem ist nach dessen Erfindern R. Rivest, A. Shamir und L. Adleman benannt, die *RSA* in den 70er Jahren entwickelt und erstmals vorgestellt haben. Trotz des hohen Alters gilt es heute noch als sicher – auch wenn immer größere Schlüssel verwendet werden müssen – und ist neben *ElGamal* eines der populärsten Systeme im Bereich der Verschlüsselung und der digitalen Signatur. Von den drei Mathematikern wurde auch die Firma *RSA Security Inc.* gegründet, die oft mit dem eigentlichen, mathematischen Verfahren verwechselt wird.

An dieser Stelle soll nun *RSA* in einer Variante behandelt werden, in der mittels D_d signiert sowie mit Hilfe von E_e validiert werden soll. Für eine digitale Signatur werden also die Schlüsselfunktionen in umgekehrter Reihenfolge verwendet: mit dem privaten Schlüssel wird nicht entschlüsselt sondern „verschlüsselt“ bzw. signiert, damit der öffentliche Schlüssel verwendet werden kann, um den Ursprung sowie die Integrität einer Nachricht $m \in M$ überprüfen zu können. Natürlich muss dabei auch sichergestellt sein, dass es sich bei e um den richtigen Schlüssel handelt...

Ein Vorteil von *RSA* ist, dass dieser „Rollentausch“ keine spezielle Implementierung erfordert, da die beiden Funktionsscharen E_x und D_x ident sind. Auch wenn die Namen der Algorithmen dann *Sign* bzw. *Verify* lauten werden; sie könnten genauso gut zum klassischen Verschlüsseln sowie Entschlüsseln verwendet werden. Das ist allerdings nicht bei jedem asymmetrischen Kryptosystem der Fall!

Grundsätzlich baut das System darauf auf, dass zwei „große“ Primzahlen sehr schnell miteinander multipliziert werden können, das Produkt jedoch nicht mehr so leicht in seine Primfaktorzerlegung gebracht werden kann, wenn kein einziger Faktor bekannt ist. Zusätzlich erschweren die in Definition 2.4.1 angeführten Eigenschaften eine erfolgreiche Faktorisierung dieses Produkts, wenn diese bei der Erzeugung der beiden Primzahlen entsprechend berücksichtigt werden (siehe auch [1]).

Bei *RSA* sind der öffentliche sowie der private Schlüssel jeweils Zahlenpaare (n, e) bzw. (n, d) mit der Eigenschaft, dass $m^{de} \equiv m \pmod{n}$ gilt, für $0 \leqslant m < n$. Eine kleine Nachricht m kann also mittels $m^d \bmod n$ signiert und danach mit Hilfe von e validiert werden. Bei beiden Vorgängen handelt es sich um einen einfachen Potenzieralgorithmus, welcher in der Klasse *MathBig* bereits implementiert worden ist.

Bemerkung 3.3.1. Die doch sehr starke Einschränkung für m wird dann im nächsten Kapitel gelockert – an dieser Stelle wird das Signieren von größeren Nachrichten der Übersicht halber nicht weiter berücksichtigt.

Algorithmus 3.3.2. Die Erzeugung eines *RSA*-Schlüsselpaares.

```
void createRsaKeyPair(out dynamic publicKey,
                      out dynamic privateKey) {
    var p = primes.NextProbableStrongPrime(KeyStrength / 2);
    var q = primes.NextProbableStrongPrime(KeyStrength / 2);

    var n = p * q;
    var l = MathBig.Lcm(p - 1, q - 1);
```

In der ursprünglichen Fassung von *RSA*, wird nicht $\lambda = \mathrm{kgV}(p-1, q-1)$ berechnet, sondern $\phi = (p-1)(q-1)$ genommen. Die Verwendung des kleinsten gemeinsamen Vielfachen führt jedoch zu kleinerem e sowie d, was die Rechenzeit verkürzt.

```
    var e = random.Next(l - 2) + 2;
    while (!MathBig.Gcd(e, l).IsOne) {
        e = random.Next(l - 2) + 2;
    }
    var d = MathBig.Inv(e, l);
```

Der öffentliche Schlüssel e wird zufällig gewählt, muss jedoch relativ prim zu λ sein, schließlich wird e gleich im Anschluss modulo λ invertiert, was dann den privaten Schlüssel ergibt. Werden also p und q „entsorgt“, so kann mittels n nicht mehr so leicht von e auf d geschlossen werden.

```
    publicKey = new ExpandoObject();
    publicKey.n = n;
    publicKey.e = e;
```

```
    privateKey = new ExpandoObject();
    privateKey.n = n;
    privateKey.d = d;
}
```

Zum Schluss werden noch die Werte für den öffentlichen sowie den privaten Schlüssel per *out*-Parameter zurückgegeben.

Bemerkung 3.3.3. Zur Speicherung von Schlüsseln sowie Signaturen wurde der Einfachheit halber auf die neue *Dynamic Language Runtime* des .NET Frameworks zurückgegriffen, um die unterschiedlichen „Formate“ der hier behandelten Kryptosysteme möglichst unkompliziert behandeln zu können.

Bleibt noch zu zeigen, dass die auf diese Art und Weise generierten Schlüsselpaare auch die gewünschten Eigenschaften mitbringen:

Beweis. Nach Konstruktion gilt

$$ed \equiv 1 \pmod{\lambda},$$

mit $\lambda = \mathrm{kgV}(p-1, q-1)$. Also lässt sich ed in der Form

$$\begin{aligned} ed &= 1 + i(p-1) \text{ bzw.} \\ ed &= 1 + j(q-1) \end{aligned}$$

schreiben. Weiters gilt

$$m^{p-1} \equiv 1 \pmod{p},$$

falls m und p teilerfremd sind (Hilfssatz 2.2.1). Werden nun beide Seiten mit i potenziert und einmal noch mit m multipliziert, so erhält man

$$m^{1+i(p-1)} \equiv m \pmod{p}.$$

Sollten m und p nicht teilerfremd sein, so ist diese Gleichung sofort erfüllt, schließlich wird dann m von p geteilt – womit auch dieser Sonderfall behandelt wäre. Analog folgt

$$m^{1+j(q-1)} \equiv m \pmod{q}.$$

Aus

$$\begin{aligned} m^{ed} &\equiv m \pmod{p} \text{ und} \\ m^{ed} &\equiv m \pmod{q} \end{aligned}$$

folgt nun die Behauptung, da

$$p \mid (m^{ed} - m) \text{ und}$$
$$q \mid (m^{ed} - m)$$

sowie die Tatsache, dass es sich bei p und q um zwei Primzahlen handelt, zu

$$pq \mid (m^{ed} - m)$$

also

$$m^{ed} \equiv m \pmod{n}$$

führen. □

Fehlt nur noch die Verwendung des generierten Schlüsselpaares, um eine Nachricht m, für die zuerst einmal $0 < m < n$ vorausgesetzt werden muss, zu signieren bzw. später auch validieren zu können:

Algorithmus 3.3.4. Das Erstellen einer Signatur mittels *RSA*.

```
dynamic signWithRsa(IntBig message, dynamic privateKey) {
    IntBig n = privateKey.n;
    IntBig d = privateKey.d;

    if (message.IsNegative || message.IsZero || message >= n) {
        throw new ArgumentOutOfRangeException("message");
    }

    var s = MathBig.Pow(message, d, n);

    dynamic signature = new ExpandoObject();
    signature.s = s;

    return signature;
}
```

Nach einer Überprüfung der Nachricht m wird mit Hilfe der Klasse *MathBig* das zuvor beschriebene Potenzieren durchgeführt. Das Resultat wird wieder in einem dynamischen Objekt gespeichert, schließlich kann die Signatur eines alternativen Systems ganz anders aussehen (siehe Anhang A).

Bemerkung 3.3.5. Das Validieren funktioniert völlig analog, anstatt der Nachricht m wird zuerst die Signatur s überprüft, schließlich sollte für diese genauso $0 < s < n$ gelten. Der Algorithmus *verifyWithRsa* wird deswegen nicht extra behandelt.

3.4. DSA

Das Erstellen einer digitalen Signatur mittels des *Digital Signature Algorithm* – kurz *DSA* – basiert auf dem *ElGamal* Kryptosystem. Das Verfahren *DSA* wurde 1991 vorgeschlagen und ist laut [1] das erste von jeder Regierung akzeptierte System für digitale Signaturen, auch wenn nach wie vor bei den meisten Amtshandlungen eine händische Unterschrift erforderlich ist.

Die Grundidee von *ElGamal* ist die Ausnutzung des Aufwandes, einen diskreten Logarithmus zu berechnen: sind Elemente α, β aus einem Restklassenkörper $\mathbb{Z}_p$ gegeben, so sind bekannte Algorithmen (siehe [1], Abschnitt 3.6) zur Berechnung von $\log_\alpha \beta$, also das Finden eines Elements $x \in \mathbb{Z}_p$ mit der Eigenschaft

$$\beta = \alpha^x \pmod{p},$$

derartig zeitaufwendig, dass ein Kryptosystem gut darauf aufgebaut werden kann. Analog zu *RSA* ist die Umkehrung des diskreten Logarithmus – das Potenzieren – nämlich wieder sehr schnell erledigt.

Der Vorteil von *ElGamal* im Vergleich zu anderen Systemen wie *RSA* ist, dass die Algorithmen für allgemeine zyklische Gruppen verwendbar sind, es müssen also nicht unbedingt ganze Zahlen modulo einer Primzahl p verwendet werden.

Eine Schwierigkeit hingegen stellt die Forderung von *ElGamal* dar, dass es sich bei α um ein erzeugendes Element handeln soll: das Finden eines solchen Elements in $\mathbb{Z}_p$ erfordert das Faktorisieren von $p - 1$ (siehe [1], Algorithmus 4.80). In der Spezifikation des *Digital Signature Algorithm* wurde diese Voraussetzung allerdings gelockert, indem anstatt eines erzeugenden Elements nur mehr ein Element von „hoher" Ordnung verlangt wird. Für das Suchen eines Elements mit dieser Eigenschaft wird nur mehr ein einziger Primfaktor von $p - 1$ benötigt, welcher bei der Verwendung von Algorithmus 2.4.3 zur Erzeugung eines neuen p praktischerweise nicht extra gesucht werden muss.

Wäre α ein Element niedriger Ordnung, so gäbe es nur wenige β, für die eine Berechnung des diskreten Logarithmus überhaupt möglich ist – das würde Angriffe auf den öffentlichen Schlüssel massiv vereinfachen, wie man dann anhand der in diesem Abschnitt vorgestellten Algorithmen sehen kann. Will man also ein Element α der Ordnung q, mit $q \mid (p - 1)$, so muss nur die Eigenschaft

$$\alpha = g^{(p-1)/q} \bmod p \neq 1$$

sichergestellt werden. Grund: betrachtet man die Primfaktorzerlegung $q q_1^{e_1} q_2^{e_2} \cdots q_k^{e_k}$ von $p - 1$, so führt eine Erfüllung obiger Eigenschaft zu

$$g^d \bmod p \neq 1,$$

für beliebige d mit $d \mid q_1^{e_1} q_2^{e_2} \cdots q_k^{e_k}$, also einer Ordnung von mindestens q für g.

Bemerkung 3.4.1. Bei anderen *ElGamal* Varianten wird mit Primzahlen p gearbeitet, wobei $\mathrm{p} - 1 = 2\mathrm{q}$ verlangt wird, sodass auch q eine Primzahl ist. Es werden also Primzahlen q gesucht, für die zusätzlich geprüft wird, ob $2\mathrm{q} + 1$ prim ist (Vergleiche unter anderem [1], Algorithmus 4.86). Auf diese Art und Weise ist dann die Primfaktorzerlegung von $\mathrm{p} - 1$ auch sofort bekannt.

Algorithmus 3.4.2. Die Erzeugung eines *DSA*-Schlüsselpaares.

```
void createDsaKeyPair(out dynamic publicKey,
                      out dynamic privateKey) {
    IntBig q;
    var p = primes.NextProbableStrongPrime(KeyStrength, out q);
```

Hier wird eine Variante von Algorithmus 2.4.3 aufgerufen, welche den Wert r (siehe auch Definition 2.4.1) als *out*-Parameter zur Verfügung stellt. Für die Variablen q und p gilt nun automatisch $\mathrm{q} \mid (\mathrm{p} - 1)$ laut Konstruktion.

```
    IntBig alpha = 1;
    while (alpha.IsOne) {
        var g = random.Next(p - 3) + 2;
        alpha = MathBig.Pow(g, (p - 1) / q, p);
    }
```

Die Variable *alpha* enthält nun ein Element aus $\mathbb{Z}_\mathrm{p}$ der Ordnung q. Dazu wird die zuvor beschriebene Eigenschaft von Elementen der Ordnung q ausgenutzt.

```
    var a = random.Next(q - 1) + 1;
    var y = MathBig.Pow(alpha, a, p);
```

Ist nur der Wert von der Variable y bekannt, wird also die Variable a geheim gehalten, so handelt es sich bei a um genau die Unbekannte des diskreten Logarithmus $\log_\alpha y$.

```
    publicKey = new ExpandoObject();
    publicKey.p = p;
    publicKey.q = q;
    publicKey.alpha = alpha;
    publicKey.y = y;

    privateKey = new ExpandoObject();
    privateKey.p = p;
    privateKey.q = q;
    privateKey.alpha = alpha;
    privateKey.a = a;
}
```

Jeder Schlüssel besteht somit aus den Werten p, q sowie α, wobei p und q Primzahlen mit der Eigenschaft $q \mid (p-1)$ sind. Bei α handelt es sich per definitionem um ein Element aus $\mathbb{Z}_p$ mit der Ordnung q. Für den privaten Schlüssel wird ein zufälliger Wert a gewählt, während für den öffentlichen Schlüssel $\alpha^a \bmod p$ bekanntgegeben wird.

Algorithmus 3.4.3. Das Erstellen einer Signatur mittels *DSA*.

```
dynamic signWithDsa(IntBig message, dynamic privateKey) {
    IntBig p = privateKey.p;
    IntBig q = privateKey.q;
    IntBig alpha = privateKey.alpha;
    IntBig a = privateKey.a;

    if (message.IsNegative || message.IsZero || message >= q) {
        throw new ArgumentOutOfRangeException("message");
    }

    var k = random.Next(q - 1) + 1;

    var r = MathBig.Pow(alpha, k, p) % q;
    var s = (MathBig.Inv(k, q) * (message + a * r)) % q;

    dynamic signature = new ExpandoObject();
    signature.r = r;
    signature.s = s;

    return signature;
}
```

Die Signatur bei *DSA* besteht sogar aus zwei Werten, wobei die Zufallskomponente k zu beachten ist: die Signatur für eine Nachricht m ist nicht eindeutig!

Algorithmus 3.4.4. Das Verifizieren einer Signatur mittels *DSA*.

```
bool verifyWithDsa(IntBig message, dynamic signature,
                                   dynamic publicKey) {
    IntBig p = publicKey.p;
    IntBig q = publicKey.q;
    IntBig alpha = publicKey.alpha;
    IntBig y = publicKey.y;

```

```
    IntBig r = signature.r;
    IntBig s = signature.s;

    if (r.IsNegative || r.IsZero || r >= q) {
        throw new ArgumentOutOfRangeException("signature");
    }
    if (s.IsNegative || s.IsZero || s >= q) {
        throw new ArgumentOutOfRangeException("signature");
    }

    var w = MathBig.Inv(s, q);

    var u1 = (w * message) % q;
    var u2 = (w * r) % q;

    var v1 = MathBig.Pow(alpha, u1, p);
    var v2 = MathBig.Pow(y, u2, p);

    var v = ((v1 * v2) % p) % q;

    return v == r;
}
```

Nachdem erst einmal sichergestellt wurde, dass r sowie s Elemente von $\mathbb{Z}_q$ sind, was ja laut *signWithDsa* der Fall sein muss, wird die Gültigkeit der Signatur überprüft.

Dass die beiden Algorithmen auch wie gewünscht funktionieren, ist leider nicht mehr ganz so offensichtlich wie bei *RSA* und wird daher wie folgt nachgerechnet:

Beweis. Für eine Nachricht m gilt laut Konstruktion

$$s \equiv k^{-1}(m + ar) \pmod{q},$$

also

$$m \equiv sk - ar \pmod{q}.$$

Eine Multiplikation mit s^{-1} ergibt

$$s^{-1}m \equiv k - ars^{-1} \pmod{q},$$

und somit

$$k \equiv s^{-1}m + ars^{-1} \pmod{q}.$$

Wird $\alpha \in \mathbb{Z}_p$ mit beiden Seiten potenziert, so erhält man

$$\alpha^{u_1+au_2} \bmod p \equiv \alpha^k \bmod p \pmod{q}.$$

Es muss daher

$$\alpha^{u_1}y^{u_2} \bmod p \equiv r \pmod{q}$$

überprüft werden.

□

Bemerkung 3.4.5. In der Spezifikation von *DSA* wird sehr genau vorgeschrieben, wie viele Bits die Werte von p und q haben müssen – mangels mathematischer Attraktivität wurde auf eine derartige Implementierung verzichtet. Weiters gibt es einen vorgeschlagenen Algorithmus zur Erzeugung der Zahlen p und q, welcher diese strikten Vorgaben erfüllen kann: siehe [1], Algorithmus 4.56.

3.5. Performance

Bis auf den „kgV-Trick" bei *RSA* ist bei so streng spezifizierten Verfahren naturgemäß kein wirkliches Potential vorhanden, die Laufzeit weiter zu optimieren. Die Performance hängt hier also primär von den in den letzten beiden Kapiteln vorgestellten Algorithmen ab, wobei aufgrund der zahlreichen Zufallskomponenten ein genaues Messen so gut wie unmöglich ist.

Um dem Zufall nicht allzu viel Spielraum in den nun folgenden Benchmarks zu gewähren, wurden die Algorithmen der hier vorgestellten Kryptosysteme jeweils 100-mal ausgeführt, um anschließend das Mittel der gemessenen Laufzeiten zu berechnen. Dessen ungeachtet sind die Ergebnisse natürlich mit Vorsicht zu genießen: ein erneuter Benchmark könnte eine Spur anders aussehen, auch wenn im Großen und Ganzen die Verhältnisse erhalten bleiben sollten.

Die Performance-Tests wurden wie immer auf einer 64-Bit CPU mit 3 GHz unter Windows 7 durchgeführt, wobei diesmal der komplette Vorgang – das Erstellen des Schlüsselpaares, die Berechnung der digitalen Signatur sowie die entsprechende Kontrolle – aufgeteilt wurde: ein Schlüsselpaar wird schließlich nicht so oft generiert wie angewendet, zumindest nicht bei diesen Verfahren.

Im ersten Schritt gewinnt eindeutig *RSA*, da hier für einen 1024-Bit Schlüssel zwei 512-Bit Primzahlen benötigt werden – für *DSA* ist hingegen ein entsprechender 1024-Bit Wert notwendig, vergleiche auch Abbildung 2.1 aus dem vorherigen Kapitel. Bei dem eigentlich wichtigeren Teil ist jedoch dann *DSA* leicht im Vorteil, da trotz großem Schlüssel mit verhältnismäßig kleineren Werten gerechnet wird.

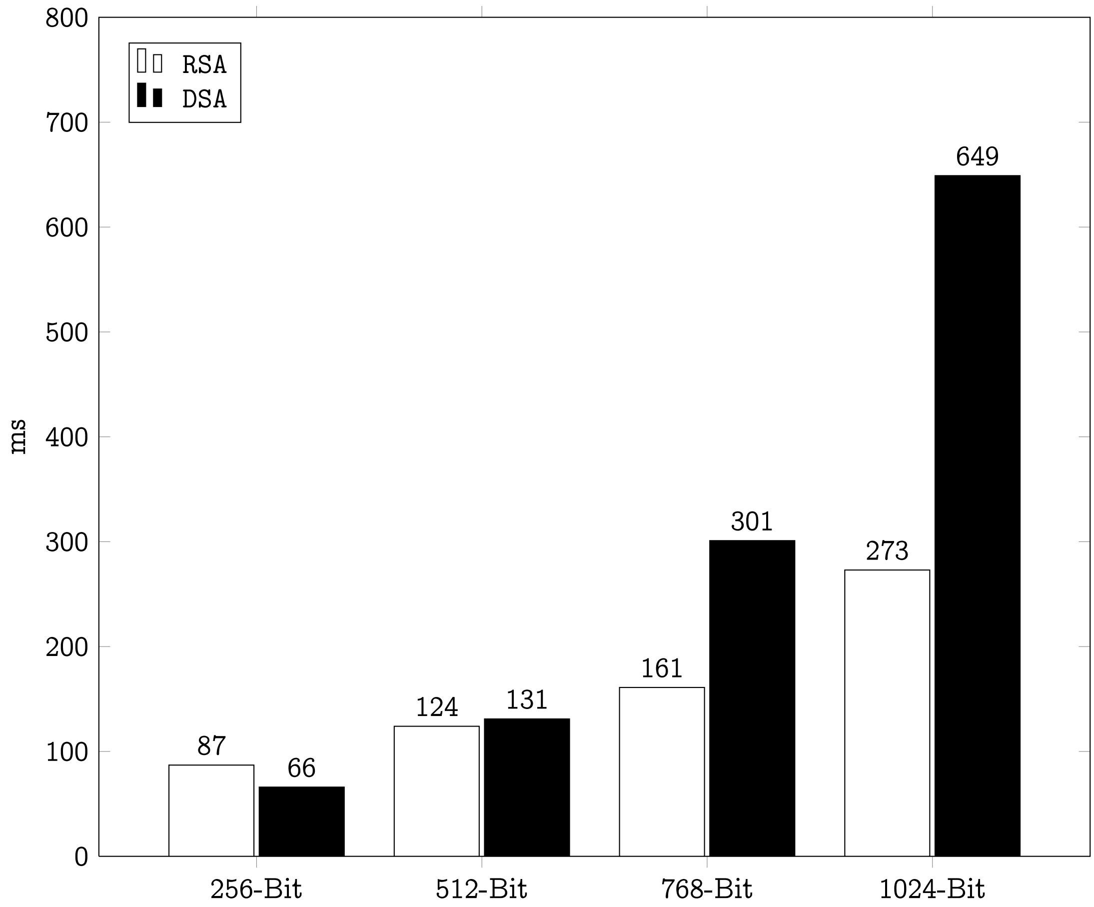

Abbildung 3.1.: Erzeugung eines neuen Schlüsselpaares

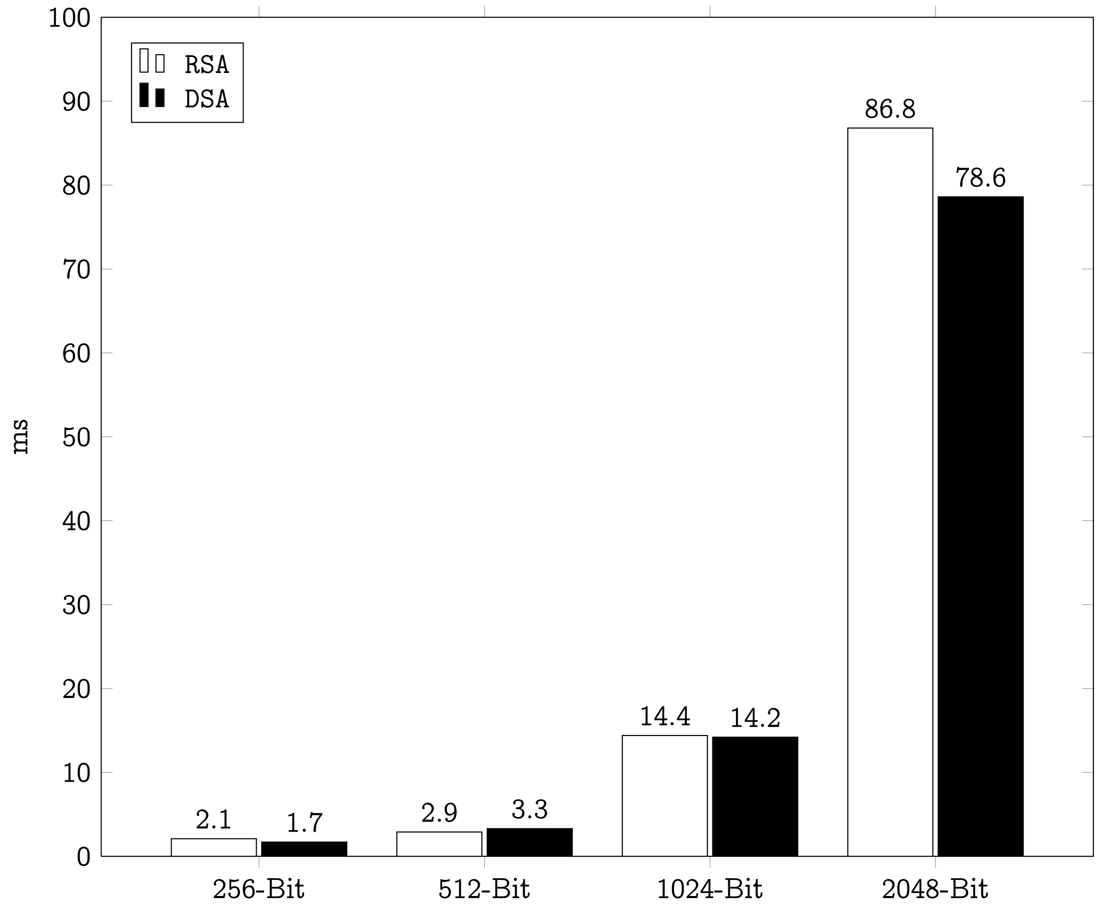

Abbildung 3.2.: Erstellung und Validierung einer digitalen Signatur

4. Hashfunktionen

4.1. Einleitung

Mit den Algorithmen aus dem vorherigen Kapitel lassen sich nur sehr kleine Nachrichten signieren – die Größe der Nachricht m wird in Abhängigkeit des zu verwendeten Kryptosystems durch das jeweilige Schlüsselpaar beschränkt. Da das Generieren von „riesigen“ Schlüsseln nicht gerade zielführend ist, genausowenig wie ein aufwändiges Signieren von zahlreichen Blöcken – Stichwort Performance und Speicherverbrauch –, welche zusammengesetzt wieder die zu übermittelnde Nachricht ergeben, werden in diesem Abschnitt Methoden vorgestellt, um eine Art Kurzfassung h der Nachricht m zu berechnen, deren Signatur ausreichen soll, um m zu validieren.

Werden alle Nachrichten $m \in \mathbb{Z}^+$ auf eine deutlich kleinere Menge der Kurzfassungen $h \in [0, 2^n)$ abgebildet, so können diese Kurzfassungen naturgemäß nicht eindeutig sein, die entsprechende Abbildung – auch Hashfunktion genannt – ist alles andere als bijektiv. „Triviale“ Hashfunktionen, wie sie oft in der Informatik für diverse Datenstrukturen verwendet werden, sind also für kryptologische Zwecke denkbar ungeeignet, da es technisch nicht möglich sein darf, zu einer Signatur beliebig gültige Nachrichten zu erstellen. Die Problemstellung ist also der eines asymmetrischen Kryptosystems relativ ähnlich: in die eine Richtung soll der *Hash* einer Nachricht schnell berechnet werden können, umgekehrt dürfen zu einer (signierten) Kurzfassung nicht beliebig Nachrichten „gefälscht“ werden.

Beispiel 4.1.1. Ein bei Datenstrukturen beliebtes Schema ist das einfache Unterteilen in k-Bit große Blöcke, welche mittels XOR verknüpft werden. Zu so einem Hashwert h kann jedoch ganz leicht eine Nachricht „gefälscht“ werden: man nehme eine beliebige Nachricht m' und berechne deren Hashwert h'. Die Erstellung von $m' \mid (h \,\mathrm{XOR}\, h')$, also das Anhängen der unter XOR abgebildeten Hashwerte, führt zu einer fälschlicherweise gültigen Nachricht.

Ganz unmöglich kann ein derartiger Fälschungsvorgang naturgemäß nicht sein – es besteht immer die Möglichkeit einer *Brute-Force-Attacke*, also einem nacheinander Ausprobieren aller in Frage kommenden Werte. Es gilt wieder die entsprechende Wahrscheinlichkeit eines „Erfolges“ möglichst gering bzw. den damit verbundenen Aufwand möglichst gewaltig zu halten: bei einem Hashverfahren, welches theoretisch keine andere Angriffsmöglichkeit bietet und darüber hinaus zum Beispiel 128-Bit große Hashwerte liefert, gäbe es 2^{128} bzw. ca. $3{,}40 \cdot 10^{38}$ verschiedene Zusammenfassungen, was ein einfaches Durchprobieren mit heutiger Hardware zum Scheitern verurteilt.

Das Konzept ist nun das Folgende: der Urheber einer Nachricht m berechnet mit Hilfe eines standardisierten Verfahrens eine Kurzfassung $h(m)$, welche anschließend mit seinem privaten Schlüssel eines asymmetrischen Kryptosystems signiert wird – die digitale Signatur ist nun nichts anderes als $D_d(h(m))$. Ein Empfänger dieser Nachricht inkl. der digitalen Signatur kann mit ein und derselben Hashfunktion erneut die dazugehörige Kurzfassung berechnen, um diese mittels der beigefügten Unterschrift zu kontrollieren.

Konkret werden jetzt „die“ Standardverfahren zu diesem Thema behandelt, welche in jeweils mehreren Runden Blöcke der zu hashenden Nachricht bearbeiten. In jeder dieser Runden werden dabei Teile dieser Blöcke vertauscht, nachdem sie untereinander mit verschiedenen binären Operationen verknüpft wurden. Mathematische Eleganz wie bei asymmetrischen Kryptosystemen ist hier eher nicht anzutreffen – es werden auch immer wieder Möglichkeiten gefunden, diese Verfahren zumindest teilweise zu brechen. Auch hier ist keine Endlösung in Sicht, sondern ein andauernder Wettlauf zwischen Kryptographen sowie Kryptoanalytikern im Gange...

4.2. MD5

Der kryptographische Hashalgorithmus *MD5* wurde Anfang der 90er Jahre von Ronald L. Rivest entwickelt und galt lange Zeit als absolut sicher, weswegen er neben der *SHA* Familie zu den am weitesten verbreiteten Hashmethoden gezählt werden kann. Nach den neuesten Erkenntnissen (siehe [13]) ist es jedoch relativ schnell möglich, sogenannte Kollisionen, also unterschiedliche Nachrichten mit ein und demselben *MD5*-Hashwert, zu erzeugen. Am sichersten gelten daher die *SHA*-Algorithmen der 2. Generation (*SHA-2*), welche dann im nächsten Abschnitt behandelt werden.

Wie bereits in der Einleitung angedeutet, handelt es sich hier nicht um einen kurzen und eleganten Algorithmus. Das Verfahren ist leider relativ aufwendig, weswegen der komplette Vorgang für die vorliegende Arbeit in drei Abschnitte unterteilt wurde: zuerst müssen diverse für den Hashvorgang konstante Werte initialisiert werden, damit sie den darauf folgenden Hashdurchgängen zur Verfügung stehen. Im 2. Schritt wird die Nachricht $m \in \mathbb{Z}^+$, welche mit einem endlichen Byte-Array identifiziert wird, in 512-Bit große Blöcke unterteilt, wobei der letzte Block noch extra behandelt werden muss – mehr dazu dann in der Detailbeschreibung des Algorithmus selbst. Der letzte Abschnitt, welcher immer wieder durch den 2. Schritt für jeden 512-Bit Block ausgeführt wird, erledigt die eigentlichen zahlreichen Hashoperationen.

Algorithmus 4.2.1. Initialisierung von *MD5*.

```
static void initMd5() {
    y = new uint[64];
    for (var i = 0; i < 64; i++) {
        y[i] = (uint)(Math.Abs(Math.Sin(i+1))*Math.Pow(2,32));
    }

    z = new[] {
        0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
        1,6,11,0,5,10,15,4,9,14,3,8,13,2,7,12,
        5,8,11,14,1,4,7,10,13,0,3,6,9,12,15,2,
        0,7,14,5,12,3,10,1,8,15,6,13,4,11,2,9
    };

    s = new[] {
        7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22,
        5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20,
        4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23,
        6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21
    };
}
```

Bei dem Array y handelt es sich um konstante Werte, welche in jeder Runde des Hashvorganges eines 512-Bit Blocks bei einer Operation dazu addiert werden – diese sind also für jede Nachricht gleich! Die Werte von z werden verwendet, um auf die einzelnen 32-Bit Segmente solch eines Blocks zuzugreifen: in der ersten Runde (jede Runde greift 16 Mal auf den aktuellen Block zu) werden diese also der Reihe nach eingelesen, während für die anderen drei Durchgänge eine alternative Zugriffsreihenfolge gilt. Zusätzlich finden jedes Mal bitweise Rotationen statt, deren Größe durch s vorgegeben ist.

Bemerkung 4.2.2. Bevor der nächste Schritt behandelt werden kann, muss noch auf die Identifikation von vier Bytes $\mathtt{b}_i$ mit einem 32-Bit Integer $\mathtt{x}$ eingegangen werden: in manchen Systemen wird das erste Byte für die „niedrigsten“ Bits der Zahl verwendet, andere sehen das genau umgekehrt. In der sogenannten *little-endian* Kodierung gilt nun $\mathtt{x} = 2^{24}\mathtt{b}_3 + 2^{16}\mathtt{b}_2 + 2^8\mathtt{b}_1 + \mathtt{b}_0$, während in der entsprechend genannten *big-endian* Version $\mathtt{x} = 2^{24}\mathtt{b}_0 + 2^{16}\mathtt{b}_1 + 2^8\mathtt{b}_2 + \mathtt{b}_3$ genommen wird.

Die Übersetzung zwischen Bytes sowie Integers wird von den Funktionen *toIntArrayLittleEndian*, *toByteArrayLittleEndian*, *toIntArrayBigEndian* und *toByteArrayBigEndian* übernommen (siehe auch Anhang A).

Algorithmus 4.2.3. Hauptfunktion von *MD5*.

```
static IntBig computeMd5Hash(Stream input) {
    var h = new uint[] {
        0x67452301,
        0xEFCDAB89,
        0x98BADCFE,
        0x10325476
    };
    var chunk = new uint[16];

    var buffer = new byte[64];
    var read = input.Read(buffer, 0, 64);

    while (read == 64) {
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
        read = input.Read(buffer, 0, 64);
    }
```

In *h* wird pro 512-Bit Block ein Hashwert dazu addiert, wobei die Startwerte wieder per Spezifikation vorgegeben sind. Da das Ergebnis von *MD5* eine 128-Bit Zahl sein soll, wird hier mit vier 32-Bit Werten vom Typ *UInt32* gearbeitet. Der aktuelle Block wird nach einer Konvertierung mittels *toIntArrayLittleEndian* in *chunk* gespeichert – um möglichst sparsam mit den zur Verfügung stehenden Ressourcen zu arbeiten, wird das dafür notwendige Array global für den zu hashenden Datenstrom gespeichert.

Aufgrund der Verwendung des .NET Datentyps *Stream*, welcher im Namespace *System.IO* gefunden werden kann, ist ein Verarbeiten von sehr großen Datenmengen mit der hier vorgestellten *MD5*-Implementierung überhaupt kein Problem! Mittels *Stream.Read* werden so lange 512-Bit große Blöcke nach *buffer* geladen, um anschließend mit *computeMd5HashChunk* verarbeitet zu werden, bis die restlichen Bits wie folgt noch „abgeschlossen“ werden:

```
    if (read > 55) {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 64; i++) {
            buffer[i] = 0;
        }
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
```

```
        for (var i = 0; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                >> ((i - 56) * 8));
        }
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
    }
    else {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                >> ((i - 56) * 8));
        }
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
    }
```

Der Datenstrom wird per definitionem mit einem Byte beendet, in dem das höchste Bit gesetzt ist – das also dem Wert 2^7 bzw. 128 entspricht. Last but not least muss noch die eigentliche Anzahl an gehashten Bits mitverarbeitet werden, um Attacken, welche bestimmte Datenströme miteinander verbinden, möglichst einzuschränken. Zu beachten ist hier, dass *Stream.Length* die Anzahl an Bytes liefert, in der Spezifikation von *MD5* jedoch von Bits die Rede ist. Diese Zahl wird ebenfalls nach der *little-endian* Definition in das Byte Array eingepflegt.

Da die Bitanzahl auch noch etwas Platz benötigt, ist die Fallunterscheidung nach der Anzahl der zuletzt eingelesenen Bytes notwendig: für das abschließende Byte sowie die Bitanzahl sind 9 Bytes vorgesehen. Zusätzlich wird der Platz zwischen dem Byte mit dem Wert 2^7 sowie der kodierten Anzahl an Bits mit Nullen aufgefüllt, um nicht am Ende einen „unvollständigen“ Block zu hashen.

```

    return IntBig.FromByteArray(toByteArrayLittleEndian(h));
}
```

Algorithmus 4.2.4. Blockfunktion von *MD5*.

```
static void computeMd5HashChunk(uint[] chunk, uint[] h) {
    var a = h[0];
    var b = h[1];
    var c = h[2];
    var d = h[3];
```

Die vier Teile des aktuellen Hashwertes werden der Übersicht halber in „gewöhnliche“ Variablen kopiert – diese Zahlen gilt es nun in vier Runden mit jeweils 16 Durchgängen zu verarbeiten: um Codewiederholungen zu vermeiden, sind diese Runden in nur einer *for*-Schleife implementiert. Der einzige Unterschied zwischen den vier Durchgängen ist die Berechnung der temporären Variable t, wie man an der *if-else*-Konstruktion erkennen kann. Direkt nach dem letzten *else*-Zweig ist die Verwendung der konstanten Arrays z, y sowie s zu sehen. Die Funktion *rotateLeft* führt eine entsprechende Rotation der Bits durch und ist aufgrund ihrer Einfachheit an dieser Stelle nicht extra angeführt.

```
    for (var i = 0; i < 64; i++) {
        var t = 0u;

        if (i < 16) {
            t = (b & c) | ((~b) & d);
        }
        else if (i < 32) {
            t = (b & d) | (c & (~d));
        }
        else if (i < 48) {
            t = b ^ c ^ d;
        }
        else {
            t = c ^ (b | (~d));
        }
        t = rotateLeft(a + t + chunk[z[i]] + y[i], s[i]);

        a = d;
        d = c;
        c = b;
        b = b + t;
    }

```

```
    h[0] = h[0] + a;
    h[1] = h[1] + b;
    h[2] = h[2] + c;
    h[3] = h[3] + d;
}
```

Nachdem ein 512-Bit Block verarbeitet worden ist, wird das Zwischenergebnis der Variablen a, b, c und d zu dem Hashwert stückweise modulo 2^{32} dazu addiert.

Beispiel 4.2.5. Konvertiert man den String *"Digitale Signaturen und ihre Implementierung in C#"* mittels UTF-8 in ein Byte Array, so ergibt dessen *MD5*-Hash den Wert

$$0x7E030286CD5102C29D81D9BBFAAA624B.$$

Ändert man diesen Text nur in einer Stelle auf *"Digitale Signaturen und ihre Implementierung in F#"*, so schaut das Ergebnis ganz anders aus:

$$0xCC511F182B20FF91B531E3F7AEEB5386.$$

Bemerkung 4.2.6. Die beiden Ergebnisse in dem vorherigen Beispiel sind als hexadezimale Zahlen angegeben, da im Rahmen dieser Arbeit eine Auffassung als Zahl am naheliegendsten ist – oft ist es üblich, das Resultat einer Hashfunktion als hexadezimale Folge von Bytes anzugeben, was dann in die andere Richtung zu lesen wäre.

4.3. SHA

Der erste offiziell vom *NIST (National Institute of Standards and Technology)* abgesegnete Hashalgorithmus hört auf den Namen *SHA-n*, wobei n für die jeweilige Version steht. Die Hashfunktion *SHA-1* ist auch das ursprünglich für das *DSA*-Kryptosystem vorgesehene Verfahren, um eine beliebige Nachricht m zu signieren.

Während der zuvor vorgestellte Algorithmus *MD5* Zahlen im 128-Bit Bereich liefert, wurden bei *SHA-1* sogar 160 Bits vorgesehen. Die jüngeren Versionen der *SHA*-Familie generieren noch einmal größere Werte, deren Anzahl an Bits ganz einfach in die jeweilige Bezeichnung aufgenommen wurde *(SHA-224, SHA-256, SHA-384, SHA-512)*. Im weiteren Verlauf werden *SHA-1* sowie *SHA-256* ausführlich behandelt – die anderen Varianten sind *SHA-256* sehr ähnlich und wurden deswegen nicht auch noch implementiert. Oft werden die vier neueren *SHA*-Algorithmen einfach *SHA-2* genannt, wenn die genaue Anzahl an Bits des Ergebnisses nicht gerade im Vordergrund steht.

Darüber hinaus wird vom *NIST* eine Art Wettbewerb veranstaltet (siehe [14]), aus dem voraussichtlich im Jahr 2012 der Algorithmus *SHA-3* hervorgehen soll – ein Nachfolger der *SHA-2*-Algorithmen ist also bereits in Sicht!

Algorithmus 4.3.1. Hauptfunktion von *SHA-1*.

```
static IntBig computeSha1Hash(Stream input) {
    var h = new uint[] {
        0x67452301,
        0xEFCDAB89,
        0x98BADCFE,
        0x10325476,
        0xC3D2E1F0
    };
    var chunk = new uint[80];

    var buffer = new byte[64];
    var read = input.Read(buffer, 0, 64);

    while (read == 64) {
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
        read = input.Read(buffer, 0, 64);
    }
```

Anhand von h ist die nahe „Verwandtschaft“ zwischen *MD5* und *SHA-1* zu sehen: die ersten vier Werte des 32-Bit Integer Arrays sind gleich. Das hat den ganz einfachen Grund, dass beide Algorithmen auf dem noch älteren Verfahren *MD4* basieren, wobei *MD4* heutzutage nicht mehr verwendet wird – es ist viel zu unsicher. Da *SHA-1* am Ende einen 160-Bit Wert liefern soll, besteht h aus fünf – in der Regel natürlich unterschiedlichen – 32-Bit Werten.

Ein weiterer signifikanter Unterschied in der Hauptfunktion von *SHA-1* im Vergleich zu der von *MD5* ist die Verwendung der *big-endian* Kodierung. Implementierungen näher an der Hardware, zum Beispiel in *C* oder sogar *Assembler* geschriebene, sind hier ein wenig im Nachteil, sofern sie für x86 CPUs gemacht werden, da bei der Übersetzung zwischen Byte sowie Integer Arrays nicht direkt „gecastet“ werden kann.

```
    if (read > 55) {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 64; i++) {
            buffer[i] = 0;
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
```

```
        for (var i = 0; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                  >> ((7 - (i - 56)) * 8));
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
    }
    else {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                  >> ((7 - (i - 56)) * 8));
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
    }
```

Auch bei der Integration der Anzahl an Bits der ursprünglichen Nachricht ist hier eine *big-endian* Kodierung durchzuführen, was auf den ersten Blick ein wenig seltsam aussehen mag: es werden zuerst die *höchsten* Bits an das Byte Array angehängt, der entsprechende Wert muss also zuerst um 56 Bits, dann um 48, usw. verschoben werden. Die Funktionen *toIntArrayBigEndian* sowie *toByteArrayBigEndian* können im Anhang dieser Arbeit nachgeschlagen werden.

```

    return IntBig.FromByteArray(toByteArrayBigEndian(h));
}
```

Bemerkung 4.3.2. Die Hauptfunktion zum neueren *SHA-256*-Algorithmus, welche in der Implementierung zu dieser Arbeit *computeSha2Hash* genannt wurde, ist mit Algorithmus 4.3.1 nahezu ident – nur die konstanten Werte von h sind andere, schließlich soll ja ein 256-Bit Hashwert generiert werden. Die Funktion *computeSha2Hash* wird daher an dieser Stelle nicht extra behandelt.

Algorithmus 4.3.3. Blockfunktion von *SHA-1*.

```
static void computeSha1HashChunk(uint[] chunk, uint[] h) {
    for (var i = 16; i < 80; i++) {
        chunk[i] = rotateLeft(chunk[i-3] ^ chunk[i-8] ^
                              chunk[i-14] ^ chunk[i-16], 1);
    }

    var a = h[0];
    var b = h[1];
    var c = h[2];
    var d = h[3];
    var e = h[4];
```

Anstatt wie bei *MD5* auf die 32-Bit Segmente des aktuellen 512-Bit Blocks in unterschiedlicher Reihenfolge zuzugreifen, wird bei *SHA-1* auf einen 2560-Bit Block hochgerechnet, indem immer vier vorhergehende Segmente miteinander verknüpft werden. Anschließend werden vier Runden durchgeführt, wobei diesmal jede Runde aus 20 Schritten besteht: jede Runde berechnet wieder auf unterschiedliche Art und Weise die Variable *t*, wobei es pro Runde eine additive Konstante *y* gibt – bei *MD5* gab es für jeden Schritt ein eigenes *y[i]*. Auch auf das Verwenden von unterschiedlichen Rotationen wurde im Vergleich zu *MD5* verzichtet, durch das Expandieren auf einen 2560-Bit Block werden die Bits offenbar ausreichend „gemischt“.

```
    for (var i = 0; i < 80; i++) {
        var t = 0u;
        var y = 0u;

        if (i < 20) {
            t = (b & c) | ((~b) & d);
            y = 0x5A827999;
        }
        else if (i < 40) {
            t = b ^ c ^ d;
            y = 0x6ED9EBA1;
        }
        else if (i < 60) {
            t = (b & c) | (b & d) | (c & d);
            y = 0x8F1BBCDC;
        }
```

```
        else {
            t = b ^ c ^ d;
            y = 0xCA62C1D6;
        }
        t = rotateLeft(a, 5) + t + e + chunk[i] + y;

        e = d;
        d = c;
        c = rotateLeft(b, 30);
        b = a;
        a = t;
    }

    h[0] = h[0] + a;
    h[1] = h[1] + b;
    h[2] = h[2] + c;
    h[3] = h[3] + d;
    h[4] = h[4] + e;
}
```

Am Ende der Verarbeitung eines 512-Bit Blocks wird wieder stückweise modulo 2^{32} addiert – auch hier bleibt die Grundidee von *MD4* im Wesentlichen erhalten.

Beispiel 4.3.4. Konvertiert man den String *"Digitale Signaturen und ihre Implementierung in C#"* mittels UTF-8 in ein Byte Array, so ergibt dessen *SHA-1*-Hash den Wert

$$0xF08368AA59441AB882627828BB54D600C576F928.$$

Ändert man diesen Text nur in einer Stelle auf *"Digitale Signaturen und ihre Implementierung in F#"*, so schaut das Ergebnis wieder ganz anders aus:

$$0x78EACAF6148DB8E53EC3003A38E0793386891842.$$

Um diesen Abschnitt abzurunden, wird an dieser Stelle noch die Blockfunktion von *SHA-256* vorgestellt: wie der Name bereits andeutet, liefert dieser Algorithmus einen 256-Bit Wert als Ergebnis. Die anderen Varianten der *SHA-2*-Familie unterscheiden sich im Wesentlichen in den Startwerten und in der Art und Weise wie der 512-Bit Block expandiert wird. Im Unterschied zu *SHA-1* sind nicht mehr verschiedene Runden vorgesehen, es wird einfach der expandierte Block Segment für Segment abgearbeitet. Das Array k, dessen Initialisierung an dieser Stelle nicht extra angeführt ist, enthält die notwendigen Konstanten.

Algorithmus 4.3.5. Blockfunktion von *SHA-256*.

```
static void computeSha2HashChunk(uint[] chunk, uint[] hi) {
    for (var i = 16; i < 64; i++) {
        var x0 = rotateRight(chunk[i-15], 7) ^
            rotateRight(chunk[i-15], 18) ^ (chunk[i-15] >> 3);
        var x1 = rotateRight(chunk[i-2], 17) ^
            rotateRight(chunk[i-2], 19) ^ (chunk[i-2] >> 10);
        chunk[i] = chunk[i-16] + x0 + chunk[i-7] + x1;
    }

    var a = hi[0];
    var b = hi[1];
    var c = hi[2];
    var d = hi[3];
    var e = hi[4];
    var f = hi[5];
    var g = hi[6];
    var h = hi[7];
```

Wieder wird der aktuelle Block vergrößert, indem aus vorhergehenden Segmenten neue Werte berechnet werden – diesmal werden diese Integer zusätzlich noch mehrmals rotiert und verknüpft, um das Verfahren noch sicherer zu machen. In den nachfolgenden Schritten, wobei diesmal nicht zwischen Runden unterschieden wird, gibt es vier „Hauptoperationen“: *Ch*, $\Sigma 1$, *Ma* und $\Sigma 0$. Die Operation $\Sigma 1$ (im Code *s1*) verarbeitet den Wert e, *Ch* vermischt die Werte e, f und g, wobei das Ergebnis von $\Sigma 1$ und *Ch* gleich zweimal weiterverwendet wird. Weiters werden von $\Sigma 0$ (im Code *s0*) die Bits von a durcheinander gebracht, um anschließend zum Resultat von *Ma*, einer Verknüpfung von a, b und c, addiert zu werden.

```
    for (var i = 0; i < 64; i++) {
        var s0 = rotateRight(a, 2) ^
            rotateRight(a, 13) ^ rotateRight(a, 22);
        var maj = (a & b) ^ (a & c) ^ (b & c);
        var t2 = s0 + maj;
        var s1 = rotateRight(e, 6) ^
            rotateRight(e, 11) ^ rotateRight(e, 25);
        var ch = (e & f) ^ ((~e) & g);
        var t1 = h + s1 + ch + k[i] + chunk[i];

```

```
        h = g;
        g = f;
        f = e;
        e = d + t1;
        d = c;
        c = b;
        b = a;
        a = t1 + t2;
    }

    hi[0] = hi[0] + a;
    hi[1] = hi[1] + b;
    hi[2] = hi[2] + c;
    hi[3] = hi[3] + d;
    hi[4] = hi[4] + e;
    hi[5] = hi[5] + f;
    hi[6] = hi[6] + g;
    hi[7] = hi[7] + h;
}
```

Beispiel 4.3.6. Konvertiert man den String *"Digitale Signaturen und ihre Implementierung in C#"* mittels UTF-8 in ein Byte Array, so ergibt dessen *SHA-256*-Hash den Wert

0xF97E989C67F7235C42B5A6D5AA5836E217C0CBC31CDEFDFB7681EA42F6548F9C.

Ändert man diesen Text nur in einer Stelle auf *"Digitale Signaturen und ihre Implementierung in F#"*, so schaut das Ergebnis wie gewohnt ganz anders aus:

0x3E1E907CBB6DBFE7D84E953D16B602BE388A0530AE8718A8AFCBF0E138337583.

4.4. Performance

Wichtig war bei der Implementierung der hier vorgestellten Hashalgorithmen, dass nicht pro Blockoperation neue Arrays initialisiert werden, weswegen die jeweilige Variable *chunk* immer wiederverwendet wird. Um auch Daten hashen zu können, welche nicht in den Hauptspeicher passen, wurde die Klasse *Stream* anstatt eines einfachen Byte Arrays als Grundlage verwendet. Wieder wurde ein entsprechender Benchmark durchgeführt, dessen Ergebnis in Abbildung 4.1 dargestellt ist.

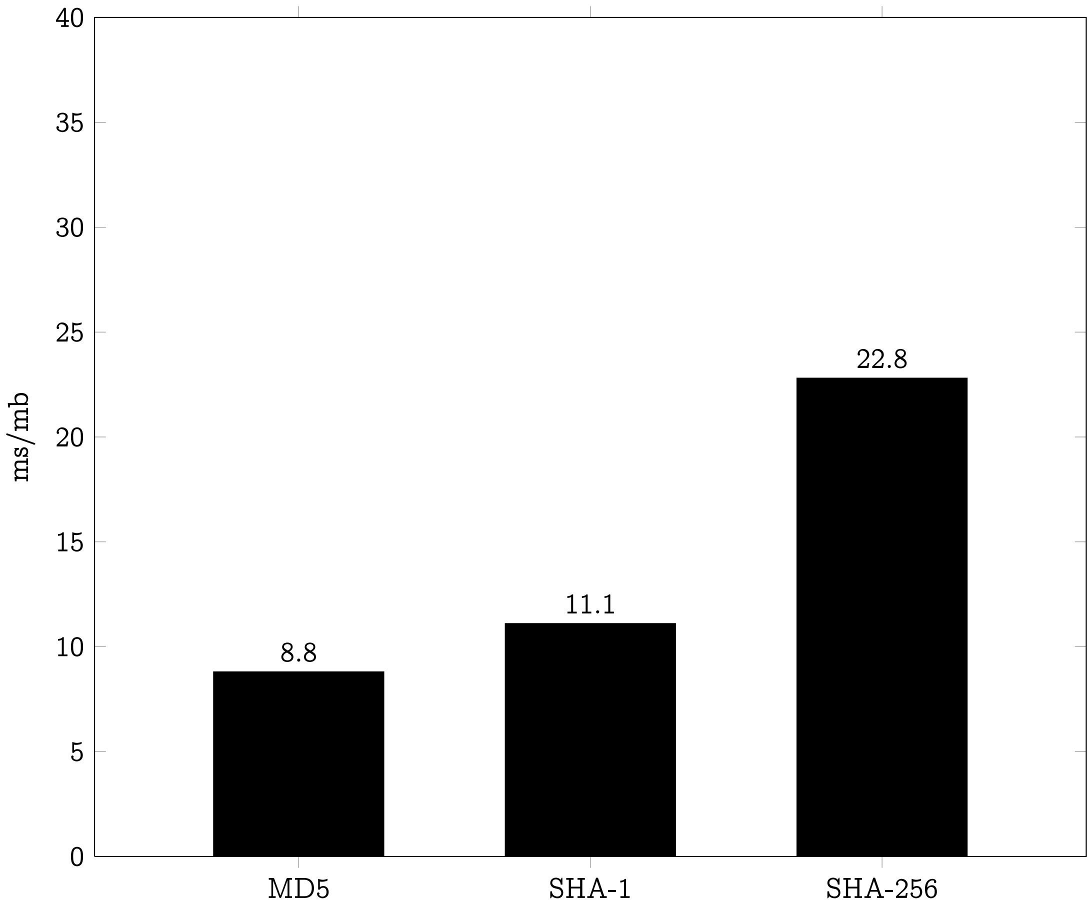

Abbildung 4.1.: Berechnung eines Hashwertes

5. „Noch größere“ Zahlen

5.1. Einleitung

In den bisherigen Abschnitten wurde vor allem in Bezug auf die Performance mit max. 1024 Bits gerechnet – genau genommen mit höchstens doppelt so vielen Stellen, da ja vor einer entsprechenden Reduktion meistens eine verhältnismäßig große Multiplikation o. ä. durchgeführt wird (vergleiche unter anderem Algorithmus 2.1.3). Auf jeden Fall ist es natürlich noch interessant, wie sich die bis jetzt vorgestellten Algorithmen verhalten, wenn man in Bezug auf die Stelligkeit noch weiter geht, was wiederum zu deutlich sichereren *digitalen Signaturen* führen würde.

Den „Flaschenhals“ der hier vorgestellten Implementierung stellt eindeutig das Erzeugen von Primzahlen dar, wie man sich leicht mit ein paar Tests der Beispielapplikation überzeugen kann. Genauere Untersuchungen der Laufzeit unterschiedlicher Operationen haben sogar ergeben, dass bei der Generierung einer 4096-Bit Primzahl über 90% der Rechenzeit durch den Moduloalgorithmus verbraucht werden! Nahezu die gesamte Rechenzeit wird auf höherer Ebene gesehen von der Potenzieroperation innerhalb des Rabin-Miller-Tests (siehe Algorithmus 2.2.4) in Anspruch genommen.

Es gilt also, zumindest zwei Optimierungen vorzunehmen: erstens soll beim Primzahltest möglichst früh „ausgesiebt“ werden, indem für kleine Primzahlen auf möglichst schnelle Art und Weise der Divisionsrest berechnet wird – also auf entsprechende Teilbarkeit geprüft wird, um das Potenzieren so weit wie nur möglich überhaupt zu vermeiden. Zweitens muss der Aufwand der Modulooperation unbedingt reduziert werden, wobei wir sogar sehen werden, dass mit Hilfe von ein wenig „Preprocessing“ auf die meisten Divisionsschritte sogar komplett verzichtet werden kann – allerdings nicht ersatzlos, dazu später mehr. Weiters wären schnellere Potenziermethoden interessant, welche mehr als nur ein Bit pro Rechenschritt verarbeiten können – tatsächlich wurden auch hier in der Literatur Möglichkeiten gefunden und selbstverständlich angewandt.

Dass die Methoden aus diesem Abschnitt zwar Verbesserungen, jedoch voraussichtlich keine Wunder bewirken, lässt sich mit Hilfe von folgendem Code abschätzen, welcher von Maple – einer hochoptimierten Software – ausgeführt werden kann.

```
roll := rand(2^4096 - 1):
p := nextprime(roll());
```

5.2. Divisionen mit „kleinem“ Divisor

Um wie bereits angedeutet möglichst selten bei der Suche nach Primzahlen potenzieren zu müssen, kann zu Beginn des Rabin-Miller-Primzahltests nicht nur überprüft werden, ob es sich bei dem Kandidaten um eine gerade Zahl handelt – also der Kandidat durch 2 teilbar ist –, sondern noch auf eine Teilbarkeit durch weitere kleine Primzahlen getestet werden. Ist die zu testende Zahl beispielsweise durch 7 teilbar, so kann man sich den theoretisch folgenden Primzahltest sparen – fehlt nur noch eine effiziente Methode, um dies zu überprüfen, sodass sich entsprechende Tests gegenüber dem Potenzieren auch eindeutig auszahlen.

Für eine Division durch die ersten n Primzahlen p_i, mit $p_i > 2$ und $p_i < 2^{32}$, ist der Divisionsalgorithmus der Klasse *IntBig* jedoch viel zu aufwändig, wie man anhand der händischen dezimalen Division bei einstelligem Divisor leicht feststellen kann. Das „teure“ Abschätzen sowie Korrigieren des Ergebnisses ist einfach nicht notwendig, wenn der Divisor nur eine einzige „Ziffer“ ungleich Null in seiner Darstellung zur Basis 2^{32} besitzt. Da sich Operatoren in C# genauso überladen lassen wie gewöhnliche Methoden, wurden sowohl der Divisionsoperator als auch der Moduloperator in einer jeweils zusätzlichen Version entwickelt: während der ursprüngliche Divisionsoperator mit der Signatur

public static IntBig operator /(IntBig dividend, IntBig divisor)

definiert ist, kann nun auf die alternative Version

public static IntBig operator /(IntBig dividend, int divisor)

zugegriffen werden. Analog wird eine für Divisoren mit nur 32 Bits angepasste Version des Moduloperators zur Verfügung gestellt.

Algorithmus 5.2.1. Der vereinfachte Divisionsalgorithmus mit „kleinem“ Divisor.

```
static uint[] divide(uint[] dividend, uint divisor) {
    var bits = new uint[dividend.Length];

    var carry = 0UL;
    for (var i = dividend.Length - 1; i >= 0; i--) {
        var value = (carry << 32) | dividend[i];
        bits[i] = (uint)(value / divisor);
        carry = value % divisor;
    }

    return bits;
}
```

Die Abschätzung aus Algorithmus 1.8.1 ist nicht mehr notwendig, für den Divisionsrest kann auf eine einfache Maschineninstruktion zurückgegriffen werden. Am Ende des Algorithmus enthält die Variable *carry* den Divisionsrest – ist man nur an diesem interessiert, so können noch einmal Rechenoperationen und somit wertvolle Rechenzeit eingespart werden.

Algorithmus 5.2.2. Der weiter optimierte Divisionsalgorithmus für den Divisionsrest.

```
static uint mod(uint[] dividend, uint divisor) {
    var carry = 0UL;
    for (var i = dividend.Length - 1; i >= 0; i--) {
        var value = (carry << 32) | dividend[i];
        carry = value % divisor;
    }
    return (uint)carry;
}
```

Für einen Kandidaten mit n Koeffizienten in seiner Darstellung zur Basis 2^{32} kann also ein Teilbarkeitstest für kleine Primzahlen mit nur n 32-Bit Moduloperationen durchgeführt werden!

Bemerkung 5.2.3. Da sich auf diese Art und Weise natürlich auch alle anderen Operatoren optimieren lassen und, Dank der Möglichkeit des Überladens, der restliche Code für eine derartige Verbesserung gar nicht angepasst werden muss, wurden gleich alle Algorithmen der Klasse *IntBig* für entsprechende 32-Bit Operanden modifiziert. Diese Modifikationen werden jedoch aufgrund ihrer geringen Auswirkung für diese Arbeit nicht extra besprochen (siehe wie immer Anhang A).

Fehlt nur noch die entsprechende Erweiterung von Algorithmus 2.2.4: bei Programmstart soll ein Array mit allen Primzahlen kleiner einer Schranke T erstellt werden. Wird jetzt eine Zahl x getestet, so muss zuerst überprüft werden, ob $x < T$ gilt – schließlich würde eine Teilbarkeit durch eine der zuvor berechneten Primzahlen nicht ausschließen, dass es sich bei x um eine Primzahl handelt. Wie auch immer dieser Fall behandelt wird, für diese Arbeit sind x interessant, für die $x \geqslant T$ gilt. Für jene x kann nun mittels Algorithmus 5.2.2 schnell überprüft werden, ob der eigentliche verhältnismäßig teure Rabin-Miller-Test überhaupt durchgeführt werden muss.

Dieses Array, welches alle Primzahlen kleiner T enthält, kann beispielsweise mit dem klassischen *Sieb des Eratosthenes* berechnet werden: hier wird mit der kleinsten Primzahl gestartet, alle Vielfachen kleiner T „gestrichen“ und mit der um 1 erhöhten Zahl fortgesetzt. Dieses Verfahren wird so lange durchgeführt, bis alle zusammengesetzten Zahlen kleiner T „ausgesiebt“ wurden.

Algorithmus 5.2.4. Das *Sieb des Eratosthenes.*

```
void initSmallPrimes() {
    var sieve = new BitArray(Threshold);

    for (var i = 2; i * i < Threshold; i++) {
        if (!sieve[i]) {
            for (var j = i * i; j < Threshold; j += i) {
                sieve[j] = true;
            }
        }
    }

    var result = new List<int>();
    for (var i = 3; i < Threshold; i++) {
        if (!sieve[i]) {
            result.Add(i);
        }
    }

    smallPrimes = result.ToArray();
}
```

Bei der Klasse *BitArray* handelt es sich um eine genau für solche Situationen optimierte Datenstruktur. Da es keinen Typ *bit* gibt, müsste ansonsten mit einem für diese Anforderung eher unhandlichen *byte*-Array gearbeitet werden. Die Klasse *BitArray* wird vom .NET Framework zur Verfügung gestellt und kann im Namespace *System.Collections* gefunden werden.

Die erste *for*-Schleife führt das Iterieren über alle natürlichen Zahlen größer gleich 2 und kleiner T durch, wobei, wie man sich leicht überlegen kann, nur bis $\sqrt{T}$ gearbeitet werden muss, da alle zusammengesetzten Zahlen zwischen $\sqrt{T}$ und T bereits einen Primfaktor kleiner $\sqrt{T}$ enthalten müssen.

Sollte die aktuelle Zahl eine Primzahl sein, so werden von der inneren *for*-Schleife alle Vielfachen davon „ausgesiebt“, das heißt auf *true* gesetzt. Diese Schleife kann wiederum für eine Primzahl p erst bei p^2 starten, da alle kleineren Vielfachen von p bereits bei einem früheren Durchgang auf *true* gesetzt wurden.

Zum Schluss wird noch von einer Darstellung als Bit-Array auf die ursprüngliche Variante eines Integer-Arrays konvertiert, welcher dann in der entsprechend erweiterten Version des Rabin-Miller-Tests verwendet wird.

5.3. Der Ring der Montgomery Zahlen

Da, wie bereits angesprochen, die Moduloperation verhältnismäßig extrem teuer ist, wäre ein zu $(\mathbb{Z}_m, +, \cdot)$ isomorpher algebraischer Ring interessant, welcher mit „billigen“ Operationen multiplizieren kann – wo also nicht nach jeder gewöhnlichen Multiplikation eine Moduloperation durchgeführt werden muss. Um somit schneller potenzieren zu können, müsste man zu Beginn von Algorithmus 2.1.3 die gegebene Zahl mit Hilfe eines entsprechenden Isomorphismus konvertieren, innerhalb dieser alternativen algebraischen Struktur den eigentlichen Algorithmus durchführen und anschließend das Ergebnis wieder in seine ursprünglich verlangte Darstellung bringen.

Dazu geht man wie folgt vor (siehe auch [1] bzw. [2]): sei R eine zu m teilerfremde Zahl mit $R > m$, sodass Divisionen durch R sowie Reduktionen modulo R leicht durchführbar sind. Die *Montgomery-Darstellung* $\tilde{x}$ einer Zahl $x \in \mathbb{Z}_m$ lautet nun

$$\tilde{x} := xR \pmod{m}.$$

Liegen die Elemente von $\mathbb{Z}_m$ so wie gehabt in ihrer Darstellung zur Basis 2^{32} vor, also in der Form

$$x := \sum_{k=0}^{n-1} a_k 2^{32k},$$

mit $a_{n-1} \neq 0$, so stellt 2^{32n} eine gute Wahl für $m = x$ dar. Divisionen durch R bzw. Reduktionen modulo R wären somit nur mehr ganz einfache *Array*-Operationen bei einer entsprechenden Implementierung.

Die *Addition* in $(\mathbb{Z}_m, +, \cdot)$ ist mit dieser Darstellung praktischerweise bereits verträglich, schließlich gilt das Distributivgesetz und somit $(x + y)R = xR + yR$. Genauso werden alle anderen Gruppeneigenschaften von $(\mathbb{Z}_m, +)$ geerbt. Bei der *Multiplikation* muss jedoch nachgebessert werden, hier ist ein eigener Operator notwendig.

Definition 5.3.1 (Montgomery-Produkt)**.** Seien $(\mathbb{Z}_m, +, \cdot)$ sowie R wie beschrieben gegeben, dann nennt man die Operation

$$x \star y := xyR^{-1} \pmod{m}$$

das *Montgomery-Produkt* von x und y.

Der algebraische Ring $(\mathbb{Z}_m, +, \cdot)$ kann also mittels des Isomorphismus $x \to \tilde{x}$ mit $(\mathbb{Z}_m, +, \star)$ identifiziert werden: es gilt

$$\widetilde{xy} = xyR = xRyRR^{-1} = \tilde{x} \star \tilde{y}$$

und bei $\tilde{1}$ handelt es sich um das Einselement von $(\mathbb{Z}_m, \star)$.

Aufgrund dieser Isomorphie ist es nun möglich, Algorithmus 2.1.3 in $(\mathbb{Z}_m, +, \star)$ durchzuführen. Schließlich beruht der „Trick“ dieses Verfahrens ganz einfach auf der Assoziativität von $(\mathbb{Z}_m, \cdot)$. Somit muss zu Beginn für die Berechnung von x^a ein passendes R gewählt werden, sowie $\tilde{x}$ für den Parameter x oder das neutrale Element $\tilde{1}$ berechnet werden. Für $\tilde{x}$ bzw. $\tilde{1}$ sind noch entsprechende Moduloperationen notwendig (!) – die dann folgenden *Montgomery-Produkte* werden jedoch anders aufgelöst.

Satz 5.3.2 (Montgomery)**.** *Seien* m, R *teilerfremde natürliche Zahlen mit* $R > m$ *und* $m' := -m^{-1} \bmod R$ *gegeben. Dann gilt für* $0 \leqslant x < Rm$, *dass die Zahl*

$$x + m(xm' \bmod R)$$

durch R *teilbar ist, sowie*

$$\frac{x + m(xm' \bmod R)}{R} \equiv xR^{-1} \pmod{m}.$$

Weiters gilt

$$0 \leqslant \frac{x + m(xm' \bmod R)}{R} < 2m.$$

Beweis. Die Teilbarkeit durch R folgt sofort, da

$$x + m(xm' \bmod R) \equiv x + xmm' \equiv x - x \equiv 0 \pmod{R}.$$

Nach einer Multiplikation mit R^{-1} von

$$x + m(xm' \bmod R) \equiv x \pmod{m}$$

lässt sich die verlangte Äquivalenz einsehen. Schlussendlich erhält man noch die gewünschte Abschätzung indem man

$$0 \leqslant x < Rm$$

und

$$0 \leqslant xm' \bmod R < R$$

zu

$$0 \leqslant x + m(xm' \bmod R) < Rm + Rm = 2Rm$$

zusammenfasst und anschließend durch R dividiert. □

Da m' nicht von x abhängt und somit für den gesamten Algorithmus nur einmal berechnet werden muss, bestehen die einzelnen *Montgomery-Produkte* im Wesentlichen aus drei Multiplikationen, einer Addition sowie ggf. einer Subtraktion – die aufwändige Division durch m entfällt. Der für den Algorithmus interessante Teil wird in der Literatur häufig separat behandelt und einfach *Montgomery-Reduktion* genannt.

Algorithmus 5.3.3. Das „Preprocessing“ für die *Montgomery-Reduktion*.

```
uint[] beginMontgomery(uint[] mod) {
    var bits = new uint[mod.Length + 1];
    bits[bits.Length - 1] = 1;
    return bits;
}
```

Für die Wahl von R wird die *Array*-Darstellung direkt ausgenutzt: es wird ein um ein Element größeres *Array* gewählt, dessen letzte Stelle auf Eins gesetzt wird.

Algorithmus 5.3.4. Die *Montgomery-Reduktion*.

```
uint[] montgomery(uint[] value, uint[] mod, uint[] inv) {
    var r = fastMod(multiply(value, inv), mod.Length);
    r = fastDiv(add(multiply(mod, r), value), mod.Length);
    if (fastCompare(r, mod) >= 0) {
        r = subtract(r, mod);
    }
    return new IntBig(r, false);
}
```

Um möglichst schnell die entsprechenden Divisionen durch R bzw. Reduktionen modulo R durchführen zu können, wurden spezielle Funktionen entwickelt, welche R gar nicht explizit benötigen. Es werden einfach die *Arrays* entsprechend gekürzt. Der Algorithmus selbst setzt direkt die Erkenntnisse aus Satz 5.3.2 um.

Beispiel 5.3.5. Die Verwendung von Montgomery Zahlen.

```
static IntBig Pow(IntBig value, IntBig power, IntBig mod)
    var R = IntBig.BeginMontgomery(mod);
    var inv = R - Inv(mod, R);

    // IntBig.Montgomery(x * y, mod, inv)

    return IntBig.Montgomery(result, mod, inv);
}
```

Will man jetzt zahlreiche Operationen modulo m durch entsprechende *Montgomery-Produkte* ersetzen, so müssen zuerst R sowie m' aufgestellt werden. Nach einer Konvertierung der Parameter können nun *Montgomery-Reduktionen* genutzt werden, wobei auf eine zusätzliche Reduktion ganz am Schluss nicht vergessen werden darf: dadurch wird xR wieder zu x.

5.4. Die Barrett Methode

Eine weitere Möglichkeit, um auf effiziente Art und Weise in $\mathbb{Z}_m$ zu multiplizieren, wird in der *Barrett-Reduktion* durchgeführt: das Ergebnis von $\lfloor \frac{x}{m} \rfloor$ wird möglichst genau mittels gewöhnlicher Multiplikationen sowie eines vorberechneten und nur von m abhängigen Wertes unter der Ausnutzung der Darstellung zur Basis 2^{32} abgeschätzt, was dann implizit dazu verwendet wird, um den Divisionsrest r mit ggf. weniger Subtraktionen zu erhalten (siehe auch [1] bzw. [2]).

Satz 5.4.1 (Barrett). *Seien x, m natürliche Zahlen dargestellt zur Basis $b \geqslant 4$, mit $x < b^{2k}$ und $b^{k-1} < m < b^k$, sowie $\mu := \lfloor \frac{b^{2k}}{m} \rfloor$ gegeben. Dann existieren natürliche Zahlen q und r, mit $x = qm + r$ und $r < m$, sowie eine Abschätzung*

$$\tilde{q} := \left\lfloor \frac{\lfloor \frac{x}{b^{k-1}} \rfloor \lfloor \frac{b^{2k}}{m} \rfloor}{b^{k+1}} \right\rfloor,$$

mit $q - 3 \leqslant \tilde{q} \leqslant q$. Weiters gilt

$$0 \leqslant x - \tilde{q}m < 4m < b^{k+1}.$$

Beweis. Zuerst wird $q := \lfloor \frac{x}{m} \rfloor$ in drei Produkte aufgeteilt, nämlich

$$q := \left\lfloor \left(\frac{x}{b^{k-1}} \right) \left(\frac{b^{2k}}{m} \right) \left(\frac{1}{b^{k+1}} \right) \right\rfloor.$$

Diesen Ausdruck gilt es nun folgendermaßen abzuschätzen:

$$\begin{aligned}
q &= \left\lfloor \frac{\left(\frac{x}{b^{k-1}} \right) \left(\frac{b^{2k}}{m} \right)}{b^{k+1}} \right\rfloor \\
&= \left\lfloor \frac{\left(\lfloor \frac{x}{b^{k-1}} \rfloor + \varepsilon_1 \right) \left(\lfloor \frac{b^{2k}}{m} \rfloor + \varepsilon_2 \right)}{b^{k+1}} \right\rfloor \\
&\leqslant \left\lfloor \frac{\lfloor \frac{x}{b^{k-1}} \rfloor \lfloor \frac{b^{2k}}{m} \rfloor + \left(\lfloor \frac{x}{b^{k-1}} \rfloor + \lfloor \frac{b^{2k}}{m} \rfloor \right) \max(\varepsilon_1, \varepsilon_2) + \varepsilon_1 \varepsilon_2}{b^{k+1}} \right\rfloor \\
&\leqslant \left\lfloor \frac{\lfloor \frac{x}{b^{k-1}} \rfloor \lfloor \frac{b^{2k}}{m} \rfloor}{b^{k+1}} + (\varepsilon_3 + \varepsilon_4) \max(\varepsilon_1, \varepsilon_2) + \varepsilon_1 \varepsilon_2 \right\rfloor \\
&\leqslant \tilde{q} + 3,
\end{aligned}$$

mit $0 \leqslant \varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4 < 1$, da

$$x < b^{2k} \Rightarrow \lfloor \frac{x}{b^{k-1}} \rfloor < b^{k+1} \Rightarrow \frac{\lfloor \frac{x}{b^{k-1}} \rfloor}{b^{k+1}} < 1$$

sowie

$$b^{k-1} < m \Rightarrow b^{2k} < mb^{k+1} \Rightarrow \lfloor \frac{b^{2k}}{m} \rfloor < b^{k+1} \Rightarrow \frac{\lfloor \frac{b^{2k}}{m} \rfloor}{b^{k+1}} < 1$$

gilt. Umgekehrt sieht man leicht $\tilde{q} \leqslant q$ ein. Betrachtet man nun

$$x - \tilde{q}m = (q - \tilde{q})m + r \leqslant 3m + r,$$

so folgt sofort die letzte Ungleichung. □

Bemerkung 5.4.2. Der zweite Teil von Satz 5.4.1 sagt ganz einfach aus, dass $x - \tilde{q}m$ modulo b^{k+1} berechnet werden kann, wobei anschließend nur wenige Subtraktionen notwendig sind, um implizit auf das „richtige“ q schließen zu können. Analog zur *Montgomery-Reduktion* sind alle Rechenschritte in der Form von Divisionen durch b^k sowie Reduktionen modulo b^k durch sehr schnelle *Array*-Operationen realisierbar.

Bemerkung 5.4.3. Laut [1] ist es eigentlich möglich zu zeigen, dass in Satz 5.4.1 sogar $q-2 \leqslant \tilde{q} \leqslant q$ gilt. Allerdings wurden in der Literatur nur Hinweise auf die Existenz eines derartigen Beweises gefunden und leider nicht der Beweis selbst. Wie auch immer – auf die eine Subtraktion mehr oder weniger kommt es bei der Laufzeit des dann folgenden Algorithmus 5.4.5 nicht wirklich an.

Algorithmus 5.4.4. Das „Preprocessing“ für die *Barrett-Reduktion.*

```
uint[] beginBarrett(uint[] mod) {
    var bits = new uint[mod.Length * 2 + 1];
    bits[bits.Length - 1] = 1;
    return new IntBig(bits, false);
}
```

Ähnlich wie bei der *Montgomery-Reduktion* muss zuerst ein $R := b^{2k}$ gewählt werden, welches anschließend durch m dividiert werden soll.

Algorithmus 5.4.5. Die *Barrett-Reduktion.*

```
uint[] barrett(uint[] value, uint[] mod, uint[] mu) {
    var q = fastDiv(value, mod.Length - 1);
    q = fastDiv(multiply(q, mu), mod.Length + 1);

    var r1 = fastMod(value, mod.Length + 1);
```

```
6       var r2 = fastMod(multiply(q, mod), mod.Length + 1);
7       var r = subtract(r1, r2);
8   
9       while (fastCompare(r, mod) >= 0) {
10          r = subtract(r, mod);
11      }
12  
13      return new IntBig(r, false);
14  }
```

Wieder werden die Funktionen *fastDiv* sowie *fastMod* eingesetzt, um den Reduktionsprozess zu beschleunigen. Im Unterschied zur *Montgomery-Reduktion* muss diesmal innerhalb einer *while*-Schleife subtrahiert werden, da dies natürlich öfters notwendig sein kann. Die in Zeile 7 aufgerufene Funktion *subtract* arbeitet praktischerweise „unabsichtlich“ modulo 2^{k+1}, da sie zwei Operanden x und y, mit $x \geqslant y$, erwartet und somit den letzten Übertrag nicht mehr berücksichtigt.

Bemerkung 5.4.6. Verwendet wird die *Barrett-Reduktion*, indem ähnlich wie bei der *Montgomery-Reduktion* zuerst mittels *beginBarrett* der Wert für R festgelegt und gleich danach mittels einer gewöhnlichen Division μ berechnet wird. Anschließend können zahlreiche Reduktionen durchgeführt werden, ohne ein weiteres Mal dividieren zu müssen – im Wesentlichen werden wieder zwei Multiplikationen durchgeführt. Aufgrund der Ähnlichkeit zur *Montgomery-Reduktion* wird an dieser Stelle auf ein ausführliches Beispiel verzichtet.

5.5. Alternatives Potenzieren

In Kapitel 2 wurde die Binärdarstellung einer Zahl α ausgenutzt, um x^α möglichst effizient zu berechnen, also das explizite Durchführen von $\alpha - 1$ Multiplikationen zu vermeiden. Dazu wurde α in die Form $\alpha_0 2^0 + \alpha_1 2^1 + \cdots + \alpha_n 2^n$ gebracht sowie

$$x^\alpha = x^{\alpha_0 2^0 + \alpha_1 2^1 + \cdots + \alpha_n 2^n} = (x^{2^0})^{\alpha_0} (x^{2^1})^{\alpha_1} \cdots (x^{2^n})^{\alpha_n}$$

betrachtet. Da für alle Koeffizienten $0 \leqslant \alpha_k < 2$ gilt, entscheiden die α_k ganz einfach, ob der Ausdruck $(x^{2^k})^{\alpha_k}$ zum Ergebnis multipliziert werden soll oder eben nicht. So oder so wird $(x^{2^k})^2$ für den nächsten Schritt berechnet, damit immer nur eine Quadrieroperation sowie ggf. eine Multiplikation durchgeführt werden muss. Auch wenn mit diesem „Trick“ die Anzahl der notwendigen Multiplikationen drastisch gesenkt wird und weiters mittels Algorithmus 1.7.2 die Ausdrücke x^{2^k} noch effizienter berechnet werden können, wäre es interessant, zu einer etwas größeren Basis ein ähnliches Verfahren durchführen zu können, um in „größeren Schritten“ ans Ziel zu kommen.

Bringt man nun α in die allgemeinere Form $\alpha_0 b^0 + \alpha_1 b^1 + \cdots + \alpha_n b^n$, mit $0 \leqslant \alpha_k < b$, so lässt sich genauso

$$x^\alpha = x^{\alpha_0 b^0 + \alpha_1 b^1 + \cdots + \alpha_n b^n} = (x^{b^0})^{\alpha_0} (x^{b^1})^{\alpha_1} \cdots (x^{b^n})^{\alpha_n}$$

untersuchen: die Ausdrücke $(x^{b^k})^{\alpha_k}$ verlangen jedoch nun $\alpha_k - 1$ Multiplikationen, womit wir wieder bei dem ursprünglichen Problem angelangt sind. Abgesehen davon müsste für die Auswertung von x^{b^k} wieder umständlich aufgeteilt werden, weswegen nur b in Frage kommen, bei denen es sich um eine Zweierpotenz, also $b = 2^i$, handelt.

Unabhängig von der Gestalt von b kann man mit Hilfe eines weiteren „Tricks“ schon noch weiterarbeiten:

$$\alpha_0 b^0 + \alpha_1 b^1 + \alpha_2 b^2 + \cdots + \alpha_{n-1} b^{n-1} + \alpha_n b^n$$

lässt sich ähnlich wie im *Horner-Schema* zu

$$\alpha_0 + b\Bigg(\alpha_1 + b\bigg(\alpha_2 + \cdots + b\Big(\alpha_{n-1} + b\big(\alpha_n\big)\Big)\bigg)\Bigg)$$

umformen! Beginnt man also nicht beim niedrigsten, sondern beim höchsten Koeffizienten α_n von α, und arbeitet sozusagen in die andere Richtung, so erhält man

$$x^\alpha = x^{\alpha_0}\Bigg(x^{\alpha_1}\bigg(x^{\alpha_2} \cdots \Big(x^{\alpha_{n-1}}\big(x^{\alpha_n}\big)^b\Big)^b\bigg)^b\Bigg)^b.$$

Weiters können die $a_k := x^k$, mit $0 \leqslant k < b$ vorberechnet werden, sollte α hinreichend viele Stellen haben, sodass sich ein entsprechendes „Preprocessing“ rentiert – die Wahl von $b := 2^i$ muss also wohl überlegt sein.

Für diese Arbeit hat sich $i = 8$ als eine gute Lösung herausgestellt: für ein α mit 1024 Bits sind dann neben den Quadrieroperationen 254 Multiplikationen als Vorarbeit, sowie 127 Multiplikationen im Laufe des eigentlichen Algorithmus notwendig. Das sind ca. 131 Produkte weniger, wenn man von einem zufällig erzeugten α ausgeht, bei dem jedes Bit mit gleicher Wahrscheinlichkeit gesetzt ist oder eben nicht.

Darüber hinaus können noch weitere Multiplikationen durch Quadrieroperationen ersetzt werden, indem beim „Preprocessing“ alle a_k mit geradem k mittels $a_k := (a_{\frac{k}{2}})^2$ ausgewertet werden. Um bei dem Beispiel mit dem 1024-Bit α zu bleiben: es sind dann insgesamt nur mehr 254 Multiplikationen sowie 1143 Quadrieroperationen notwendig. Je größer α gewählt wird, desto mehr zahlt sich diese Optimierung dann auch aus.

Algorithmus 5.5.1. Ein schnelleres Potenzieren von *value* mit Exponenten *power*.

```
static IntBig Pow(IntBig value, IntBig power) {
    var v = new IntBig[256];
    v[0] = 1;
    v[1] = value;
    v[2] = (v[1] * v[1]);

    for (var j = 4; j < v.Length; j *= 2) {
        v[j] = (v[j / 2] * v[j / 2]);
    }
    for (var i = 3; i < v.Length; i += 2) {
        v[i] = (v[i - 1] * value);
        for (var j = i * 2; j < v.Length; j *= 2) {
            v[j] = (v[j / 2] * v[j / 2]);
        }
    }
```

Ab Index 4 werden zuerst einmal alle a_k, im Code *v[i]* bzw. *v[j]*, mittels der schnelleren Quadrieroperation berechnet, für die $k = 2^i$ gilt. Anschließend wird für alle ungeraden k eine halb so schnelle Multiplikation durchgeführt, wobei dann für gerade k eine eigene innere Schleife vorgesehen ist.

```
    var p = power.ToByteArray();

    IntBig result = v[p[p.Length - 1]];

    for (var i = p.Length - 2; i >= 0; i--) {
        for (var j = 0; j < 8; j++) {
            result = (result * result);
        }
        result = (result * v[p[i]]);
    }

    return result;
}
```

Mit der Funktion *ToByteArray* wird *power* von seiner Darstellung zur Basis 2^{32} in die hier benötigte Form konvertiert, welche dann als *Array* abgearbeitet werden kann. Das Potenzieren hoch 2^8 wird durch achtmaliges Quadrieren gelöst, für die darauffolgende Multiplikation wird auf ein vorberechnetes a_k zurückgegriffen.

5.6. Rekursives Multiplizieren

Dank der Modifikationen aus Abschnitt 5.3 bzw. Abschnitt 5.4 stellt jetzt die Multiplikation die mit Abstand wichtigste Rechenoperation der Big-Integer-Arithmetik *IntBig* dar, weswegen nun eine entsprechende Beschleunigung von Algorithmus 1.7.1 durchgeführt werden soll. Die Leistung der Multiplikation ist zwar grundsätzlich schon relativ in Ordnung (siehe Abschnitt 1.9), jedoch werden für zwei Faktoren $u = (u_0 u_1 \cdots u_{n-1})_b$ und $v = (v_0 v_1 \cdots v_{n-1})_b$ insgesamt n^2 native Integer-Multiplikationen durchgeführt – für jeweils doppelt so große Faktoren vervierfacht sich also der Rechenaufwand, wenn man die zusätzlich notwendigen Additionen sowie Shifts außer Acht lässt.

Um den Anstieg der benötigten Rechenzeit ein wenig zu entschleunigen, kann man die beiden Faktoren u und v der Multiplikation mit den Polynomen

$$U(x) = u_{n-1}x^{n-1} + \cdots + u_1 x + u_0 \text{ und } V(x) = v_{n-1}x^{n-1} + \cdots + v_1 x + v_0$$

identifizieren – klarerweise gilt $u = U(b)$ bzw. $v = V(b)$. Es wird also die Darstellung der beiden Zahlen verallgemeinert, um in dieser generalisierten Form einen Vorteil zu entdecken, der für einen schnelleren Multiplikationsalgorithmus ausgenutzt werden kann. In diesem Fall lautet der Nutzen für $w = uv$ ganz einfach $W(x) = U(x)V(x)$. Das Produkt muss also auf hinreichend vielen Stützstellen ausgewertet werden, um W vollständig interpolieren und somit $w = W(b)$ erhalten zu können.

Leider verursacht sowohl das Berechnen der Stützstellen als auch das Interpolieren des Produktpolynoms W einen nicht zu verachtenden Aufwand – laut [2] müsste die Maschine, auf der ein entsprechender Algorithmus ausgeführt werden soll, im „Schneckentempo“ multiplizieren, sodass sich ein derartiges Vorgehen auch praktisch auszahlt. Theoretisch müssten ja nur $2n - 1$ Stützstellen berechnet, also $2n - 1$ Multiplikationen der Form $W(k) = U(k)V(k)$ durchgeführt werden, was grundsätzlich sehr vielversprechend klingt. In der Praxis wurde jedoch noch kein allgemeiner Algorithmus (siehe [15]) entwickelt, der dies auch wirklich verlustarm umsetzen kann.

Bleibt man jedoch bei dieser Idee für $u = (u_0 u_1 \cdots u_{2n-1})_b$ bzw. $v = (v_0 v_1 \cdots v_{2n-1})_b$ und vereinfacht diese auf zwei „kleine Polynome“ der Form

$$u = u_h b^n + u_l \text{ und } v = v_h b^n + v_l,$$

mit $u_h = (u_n u_{n+1} \cdots u_{2n-1})_b$ sowie $u_l = (u_0 u_1 \cdots u_{n-1})_b$ (analog für v), so lässt sich ein einfacher Weg finden, den Multiplikationsalgorithmus effizienter zu gestalten. In diesem Fall rechnet man direkt, also ohne Verwendung von Stützstellen sowie Interpolationen, das Produkt

$$uv = u_h v_h b^{2n} + (u_h v_l + u_l v_h) b^n + u_l v_l$$

aus, um festzustellen, dass auf diese Art und Weise ein Produkt von zwei Faktoren mit jeweils $2n$ Stellen durch vier Produkte mit n-stelligen Faktoren ersetzt wird, was den Rechenaufwand noch nicht wirklich senkt. Ein Umformen auf

$$uv = u_h v_h b^{2n} + ((u_h + u_l)(v_h + v_l) - u_h v_h - u_l v_l)b^n + u_l v_l$$

bringt dann das gewünschte Resultat. Zusammengefasst wird mit den drei halb so großen und somit jeweils vier Mal so schnell zu berechnenden Produkten

$$\begin{aligned} p_1 &:= u_h v_h \\ p_2 &:= u_l v_l \\ p_3 &:= (u_h + u_l)(v_h + v_l) \end{aligned}$$

der Ausdruck

$$uv = p_1 b^{2n} + (p_3 - p_1 - p_2)b^n + p_2$$

ausgewertet. Ignoriert man fürs Erste den Aufwand, welcher von den dafür notwendigen Additionen sowie Subtraktionen verursacht wird, so verdreifacht sich nun die Rechenzeit für eine Multiplikation, wenn man die Stelligkeit der beiden Faktoren verdoppelt, was sich ab einer gewissen Bitanzahl trotz „Overhead“ auf jeden Fall auszahlen soll.

Algorithmus 5.6.1 (Karatsuba)**.** Die Multiplikation mit integrierter Optimierung für besonders große Zahlen, welche ursprünglich von *Karatsuba* entwickelt wurde.

```
unsafe uint[] multiply(uint *left, int leftLength,
                       uint *right, int rightLength) {
    var bits = new uint[leftLength + rightLength];
```

Um wie beschrieben die Faktoren ohne Kopiervorgänge oder umständliche Indexberechnungen aufteilen zu können, wurde diese Routine auf *Pointer-Arithmetik* umgestellt – es werden also keine *Arrays* mehr verwendet. Weiters wurde eine Schranke T definiert, ab der das besprochene Aufteilen stattfinden soll – darunter ist der „Overhead“ einfach zu groß. Für T wurde nach ausführlichen Tests 64 gewählt, Zahlen mit weniger als 2048 Bits werden also wie gewohnt behandelt.

```
    if (leftLength < Threshold || rightLength < Threshold) {
        // multiply the bits...
    }
    else {
        var n = (((leftLength > rightLength) ?
                  leftLength : rightLength) + 1) / 2;

```

```
        var leftLow = left; var leftHigh = left + n;
        var rightLow = right; var rightHigh = right + n;

        var leftLowLength = (n < leftLength) ?
                              n : leftLength;
        var rightLowLength = (n < rightLength) ?
                              n : rightLength;
        var leftHighLength = leftLength - leftLowLength;
        var rightHighLength = rightLength - rightLowLength;
```

Nach einer Wahl für ein passendes n – bei Zahlen mit ungerader Länge wird implizit eine führende Null angehängt – werden sowohl die Speicheradressen als auch die vier ev. ungleichen Längen von u_h, u_l, v_h sowie v_l berechnet, um auch mit deutlich unterschiedlich großen Zahlen effizient umgehen zu können.

```
        var p1 = multiply(leftHigh, leftHighLength,
                          rightHigh, rightHighLength);
        var p2 = multiply(leftLow, leftLowLength,
                          rightLow, rightLowLength);
        var p3 = multiply(add(leftLow, leftLowLength,
                              leftHigh, leftHighLength),
                          add(rightLow, rightLowLength,
                              rightHigh, rightHighLength));

        subtractInplace(p3, p1);
        subtractInplace(p3, p2);
```

Für die Berechnung von p_1, p_2 und p_3 wird die Multiplikationsroutine rekursiv aufgerufen und somit unter Umständen wiederholt die *Karatsuba*-Methode angewandt. Um ein wenig Rechenzeit sowie Speicher zu sparen, wurde die Funktion *subtractInplace* entwickelt, welche den Minuenden direkt überschreibt.

```
        addInplaceWithFastShift(bits, p2, 0);
        addInplaceWithFastShift(bits, p3, n);
        addInplaceWithFastShift(bits, p1, n * 2);
    }
    return bits;
}
```

Ähnlich wie in Algorithmus 1.8.1 müssen zum Schluss die Werte um eine gewisse Anzahl an Stellen verschoben verarbeitet – in diesem Fall addiert – werden, wofür auch eine spezielle Additionsmethode entwickelt wurde.

Betrachtet man nun den neuen Multiplikationsalgorithmus, so liegt dessen Parallelisierbarkeit auf der Hand. Die Werte von p_1, p_2 und p_3 könnten – sofern vorhanden – auf unterschiedlichen Prozessoren berechnet werden, weil der eine Rechenvorgang mit dem anderen überhaupt nichts zu tun hat.

Da ein moderner PC der „Mittelklasse“ bereits zumindest über vier CPUs verfügt – genauer um eine CPU mit mindestens vier „Cores“ – und das .NET Framework seit der aktuellen Version 4.0 einige praktische Hilfsmittel zur Parallelisierung von Software enthält, wurden weitere Schranken definiert, ab denen ein Aufteilen der Kalkulationen auf mehrere CPUs eine Verbesserung darstellt.

Bemerkung 5.6.2. Mehr zum Thema Parallelisierung auf Basis des .NET Frameworks kann unter [23] nachgelesen werden. Für Algorithmus 5.6.1 wurde im Wesentlichen auf die Klasse *Task*, welche im Namespace *System.Threading.Tasks* gefunden werden kann, zurückgegriffen. Die entsprechende Implementierung wird an dieser Stelle nicht extra behandelt, wobei der Sourcecode im Anhang A nachgeschlagen werden kann.

Bemerkung 5.6.3. Analog lässt sich naturgemäß Algorithmus 1.7.2 beschleunigen – hier können genauso Quadrieroperationen durch drei jeweils halb so große Rechenschritte ersetzt werden (Anhang A...).

5.7. „Lineares“ Multiplizieren

Die in Abschnitt 5.6 vorgestellte Erweiterung von Algorithmus 1.7.1 ist eine stark spezialisierte sowie optimierte Variante einer deutlich allgemeineren Idee, welche bereits angedeutet wurde – diese gilt es nun ausführlicher zu behandeln. Wie bereits besprochen, werden die beiden Faktoren u und v mit

$$U(x) = u_{n-1}x^{n-1} + \cdots + u_1 x + u_0 \text{ und } V(x) = v_{n-1}x^{n-1} + \cdots + v_1 x + v_0$$

identifiziert. Das Produkt $w = uv$ hat in dieser Darstellung die Gestalt

$$W(x) = w_{2n-2}x^{2n-2} + w_{2n-3}x^{2n-3} + \cdots + w_2 x^2 + w_1 x + w_0.$$

Kennt man nun den Wert von $W(x) = U(x)V(x)$ an $2n-1$ Stützstellen, welche punktweise ausgewertet werden können, so lässt sich mit Hilfe dieser Punkte $W(x)$ und daher w vollständig rekonstruieren.

Da somit im Wesentlichen $2n-1$ Produkte ausgerechnet werden müssen, wird in der Literatur von annähernd linearem Aufwand gesprochen, was natürlich mehr als nur vielversprechend klingt. Jedoch sind drei Punkte nicht zu unterschätzen: die beiden Faktoren müssen erst einmal in ihrer Polynomdarstellung ausgewertet werden. Nach der

	$b = 2^8$	$b = 2^{16}$	$b = 2^{32}$	$b = 2^{64}$
$u = 2^{1024} - 1$	1023	456	217	138
$u = 2^{2048} - 1$	2302	1031	472	249
$u = 2^{4096} - 1$	5117	2310	1047	504
$u = 2^{8192} - 1$	11260	5125	2326	1079

Tabelle 5.1.: Bitanzahl für $U(2n-2)$ in Abhängigkeit von b und u

Berechnung der $2n-1$ Multiplikationen muss $W(x)$ noch interpoliert werden. Schlussendlich können die Stützstellen bei schlechter Wahl der Basis b für u und v unter Umständen sogar größere Werte hervorbringen, als es die beiden ursprünglichen Faktoren sind: wählt man $\{0, 1, \dots, 2n-2\}$ um die Polynome $U(x)$ und $V(x)$ auszuwerten, so kann für $u = (u_0 u_1 \cdots u_{n-1})_b$ der höchste Wert mit

$$U(2n-2) \leqslant \sum_{k=0}^{n-1} (b-1)(2n-2)^k$$

abgeschätzt werden (analog für v). Man befindet sich also in der Zwickmühle, dass ein im Verhältnis zu u und v großes b dazu führt, dass diese Idee nur wenig „greift“, sowie ein entsprechend kleines b die Stützstellen „explodieren“ lässt.

In Tabelle 5.1 sind explizite Werte angegeben, welche zeigen, dass b nicht allzu klein gewählt werden darf, also bereits eine Big-Integer-Arithmetik vorhanden sein muss, um dieses Verfahren überhaupt umsetzen zu können. Grundsätzlich ist es nun möglich, mit Hilfe dieser Idee einen Algorithmus zu entwickeln, welcher eine bereits existierende Arithmetik beschleunigen soll.

Bevor dieser Algorithmus, welcher ursprünglich von *Toom* und *Cook* entwickelt wurde (siehe [15]), konkret umgesetzt werden kann, muss noch die Interpolation von $W(x)$ genauer betrachtet werden. Nach der Berechnung der bereits ausführlich besprochenen Produkte $W(0), W(1), \dots, W(2n-2)$ kann nämlich relativ leicht eine alternative Darstellung von $W(x)$ gewonnen werden.

Definition 5.7.1. Die *fallenden Faktorielle* sind definiert als

$$x^{\underline{k}} := x(x-1)\cdots(x-k+1).$$

Hilfssatz 5.7.2. *Für die* fallenden Faktorielle *gilt*

$$\Delta x^{\underline{k}} := (x+1)^{\underline{k}} - x^{\underline{k}} = k x^{\underline{k-1}}.$$

Beweis. Nachrechnen. □

Betrachtet man nun $W(x)$ in der Form

$$W(x) = \tilde{w}_{2n-2}x^{2n-2} + \tilde{w}_{2n-3}x^{2n-3} + \cdots + \tilde{w}_2x^2 + \tilde{w}_1x + \tilde{w}_0$$

dargestellt, so kann Hilfssatz 5.7.2 benutzt werden, um die Koeffizienten $\tilde{w}_k$ mit Hilfe der zuvor an den entsprechenden Stützstellen berechneten Werte zu rekonstruieren. Wendet man den Operator Δ wiederholt auf $W(x)$ an, so erhält man folgendes Schema.

$$\begin{array}{rclcl} \Delta W(x) & = & W(x+1) - W(x) & = & \cdots + \tilde{w}_1 \\ \Delta^2 W(x) & = & \Delta W(x+1) - \Delta W(x) & = & \cdots + 2\tilde{w}_2 \\ \vdots & & & \vdots & \\ \Delta^{2n-2} W(x) & = & \Delta^{2n-3} W(x+1) - \Delta^{2n-3} W(x) & = & (2n-2)!\tilde{w}_{2n-2} \end{array}$$

Wird also dieses Differenzenschema angewandt, so gilt

$$\tilde{w}_k = \frac{\Delta^k W(0)}{k!}.$$

Bevor man endlich $w = W(b)$ erhält, muss also nur noch von dieser alternativen Darstellung umgerechnet, also die w_k berechnet werden, was sich mit einer Art Erweiterung des *Horner-Schemas* bewerkstelligen lässt – dazu später mehr. Zum besseren Verständnis wird in [5] anhand von zwei Beispielwerten dieses Verfahren vorgerechnet.

Algorithmus 5.7.3 (Toom-Cook)**.** Die wesentlichen Schritte des ausführlich motivierten *Toom-Cook-Algorithmus* zur Multiplikation der Zahlen u und v zur Basis 2^β. Als Parameter werden u und v in Binärdarstellung sowie ein β übergeben, wobei β die zu verwendende Basis festlegt.

1. Spalte die Bits von u und v zur Basis 2^β auf, sodass $u = (u_{n-1}u_{n-2}...u_2u_1u_0)_{2^\beta}$ und $v = (v_{m-1}v_{m-2}...v_2v_1v_0)_{2^\beta}$ gilt.

2. Berechne die Stützstellen $U_x := u_{n-1}x^{n-1} + u_{n-2}x^{n-2} + \cdots + u_2x^2 + u_1x + u_0$ und $V_x := v_{m-1}x^{m-1} + v_{m-2}x^{m-2} + \cdots + v_2x^2 + v_1x + v_0$ für $x \in \{0, 1, \ldots, n+m-2\}$.

3. Berechne die Produkte $W_x := U_xV_x$ für $x \in \{0, 1, \ldots, n+m-2\}$.

4. Berechne die Koeffizienten $\tilde{w}_x$ mit Hilfe des zuvor beschriebenen Differenzenschemas, angewandt auf die W_x für $x \in \{0, 1, \ldots, n+m-2\}$.

5. Berechne die Koeffizienten w_x mit Hilfe der zuvor berechneten Koeffizienten $\tilde{w}_x$ für $x \in \{0, 1, \ldots, n+m-2\}$.

6. Berechne $w = w_{n+m-2}2^{\beta^{n+m-2}} + w_{n+m-3}2^{\beta^{n+m-3}} + \cdots + w_2 2^{\beta^2} + w_1 2^\beta + w_0$.

Algorithmus 5.7.4 (Toom-Cook, Schritt 1)**.** Das Aufspalten der Bits von u und v.

```
static IntBig[] ToomCookSplit(IntBig value, int baseBits) {
    var n = (value.CountBits() + baseBits - 1) / baseBits];
    var v = new IntBig[n];
    for (var i = 0; i < v.Length; i++) {
        var shifted = value >> baseBits;
        v[i] = value - (shifted << baseBits);
        value = shifted;
    }
    return v;
}
```

Nachdem ein entsprechend dimensioniertes *Array* angelegt wurde, werden schrittweise mit Hilfe von Shifts und einer Subtraktion Bitblöcke der geforderten Größe aus der Binärdarstellung von *value* extrahiert.

Algorithmus 5.7.5 (Toom-Cook, Schritt 2)**.** Das Auswerten der Polynome $U(x)$ und $V(x)$ an den entsprechenden Stützstellen $\{0, 1, \ldots, n + m - 2\}$.

```
static IntBig[] ToomCookEval(IntBig[] value, int pointCount) {
    var v = new IntBig[pointCount];
    for (var i = 0; i < v.Length; i++) {
        v[i] = 0;
        for (var j = value.Length - 1; j >= 0; j--) {
            v[i] = (v[i] * i + value[j]);
        }
    }
    return v;
}
```

Hier wird direkt das bekannte *Horner-Schema* angewandt, um die Polynome an den Stützstellen auszuwerten – der entsprechende Vorgang wird dabei von der inneren *for*-Schleife durchgeführt. Die äußere Schleife ist nur für das Iterieren über die einzelnen Stützstellen zuständig.

Algorithmus 5.7.6 (Toom-Cook, Schritt 3)**.** Dieser Schritt beinhaltet nur das Auswerten der Produkte $W_x := U_x V_x$ und wird deswegen nicht extra behandelt.

Algorithmus 5.7.7 (Toom-Cook, Schritt 4)**.** Das Interpolieren von $W(x)$ mittels des zuvor besprochenen Differenzenschemas.

```
static IntBig[] ToomCookInterpolate(IntBig[] value) {
    var q = new IntBig[value.Length];
    q[0] = value[0];
    for (var i = 1; i < q.Length; i++) {
        var h = new IntBig[value.Length - 1];
        for (var j = 0; j < h.Length; j++) {
            h[j] = (value[j + 1] - value[j]) / i;
        }
        value = h;
        q[i] = value[0];
    }
    return q;
}
```

Hier ist für den Übergang von $\Delta^k W(x)$ zu $\Delta^{k+1} W(x)$ ein Hilfsarray h im Einsatz, welches die Ergebnisse des aktuellen Durchgangs speichert. Nachdem alle Differenzen berechnet wurden, wird das aktuelle Array mit dem Hilfsarray überschrieben – genauer gesagt die Referenz auf das Array – und der niedrigste Koeffizient übernommen.

Algorithmus 5.7.8 (Toom-Cook, Schritt 5)**.** Das Transformieren der Koeffizienten $\tilde{w}_k$ in die eigentlich gewünschte Darstellung w_k.

```
1  static IntBig[] ToomCookTransform(IntBig[] value) {
2      var s = new IntBig[value.Length];
3      for (var i = 0; i < s.Length; i++) {
4          s[i] = value[i];
5      }
6      for (var i = s.Length - 2; i > 0; i--) {
7          for (var j = i; j < s.Length - 1; j++) {
8              s[j] = (s[j] - s[j + 1] * i);
9          }
10     }
11     return s;
12 }
```

An dieser Stelle muss zuerst eine Kopie des ursprünglichen Arrays angefertigt werden, welche anschließend schrittweise aktualisiert wird. Im Wesentlichen entspricht dieser Vorgang einer Erweiterung des *Horner-Schemas*, schließlich sind die $x^{\underline{k}}$ etwas schwieriger zu behandeln als die x^k, da jedes „Ausmultiplizieren“ mit einer zusätzlichen Subtraktion (Zeile 8) verbunden ist.

	$\log_2 u, v \approx 2^{24}$	$\log_2 u, v \approx 2^{22}$	$\log_2 u, v \approx 2^{20}$
$\beta = 2^{24}$	6690 ms		
$\beta = 2^{23}$	6515 ms		
$\beta = 2^{22}$	5169 ms	660 ms	
$\beta = 2^{21}$	4374 ms	657 ms	
$\beta = 2^{20}$	4316 ms	565 ms	72 ms
$\beta = 2^{19}$	5154 ms	534 ms	71 ms
$\beta = 2^{18}$	8247 ms	626 ms	70 ms
$\beta = 2^{17}$	15293 ms	964 ms	78 ms
$\beta = 2^{16}$	30296 ms	1741 ms	112 ms

Tabelle 5.2.: Laufzeit einer Operation *ToomCook*(u, v, β)

Algorithmus 5.7.9 (Toom-Cook, Schritt 6)**.** Das Auswerten von $W(2^\beta)$ besteht naturgemäß aus trivialen Shifts sowie Addieroperationen und wird aufgrund seiner Einfachheit nicht extra besprochen.

Bemerkung 5.7.10. Wie an den einzelnen Algorithmen zu sehen ist, handelt es sich bei dem Verfahren, so wie es hier vorgestellt wurde, nicht um einen speziellen Algorithmus der Datenstruktur *IntBig* – mit einem leicht angepassten Code könnte jede beliebige Big-Integer-Arithmetik verwendet werden. Der Nutzen des *Toom-Cook-Algorithmus* nimmt zu, je langsamer jene Bit-Integer-Arithmetik multipliziert (siehe [2]).

Um einen Vorteil von Algorithmus 5.7.3 auf Basis der Bit-Integer-Arithmetik *IntBig* messen zu können, mussten – im Verhältnis zu für *digitale Signaturen* üblich dimensionierten Bitfolgen – extrem große Zufallszahlen generiert werden. Für jene Testwerte wurde schrittweise der Parameter β halbiert, um anschließend die Ergebnisse der Benchmarks vergleichen zu können. In Tabelle 5.2 ist zu sehen, wie das Verfahren eine Zeit lang die Multiplikation beschleunigt, jedoch dann relativ bald „einbricht".

5.8. Performance

Um den praktischen Mehrwert der mathematischen Methoden aus diesem Abschnitt zu veranschaulichen, wurde ähnlich wie in Abschnitt 1.9 ein Vergleich zwischen der Datenstruktur *BigInteger* von Microsoft sowie der Datenstruktur *IntBig* dieser Arbeit durchgeführt. Der Benchmark wurde unter denselben Bedingungen wie die bisherigen Performancemessungen ausgeführt, wobei diesmal die genaue Laufzeit eines Potenziervorganges gemessen wurde. Die entsprechenden Ergebnisse sind in Abbildung 5.1 dargestellt – diesmal musste auf eine logarithmische y-Achse zurückgegriffen werden.

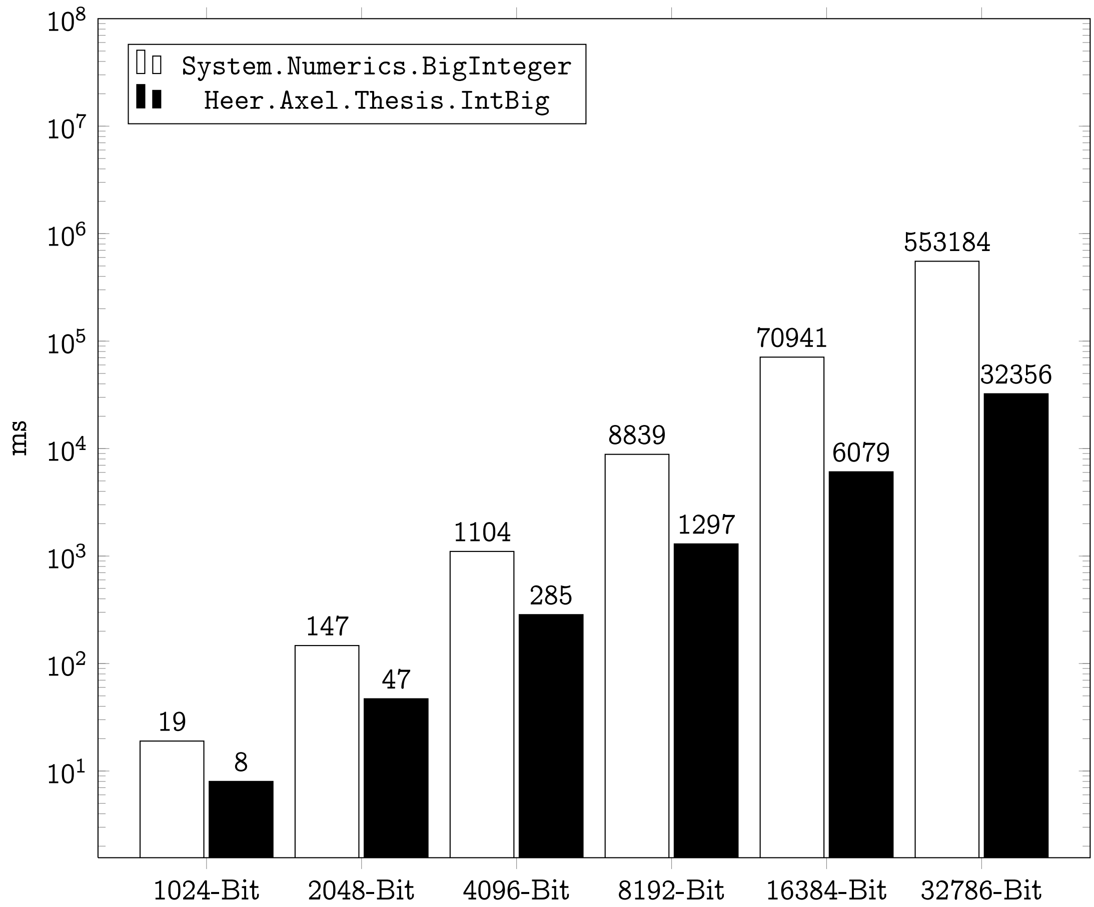

Abbildung 5.1.: Laufzeit einer Operation $a^b \bmod n$

6. Praktische Anwendungen

6.1. Einleitung

Nach dem theoretischen Teil dieser Arbeit soll nun der „Nachricht m“ ein Gesicht verpasst werden: mit einer digitalen Signatur können elektronische Daten von unterschiedlichster Bedeutung „unterschrieben“ werden, wobei sogar auch eine Art Transitivität möglich und durchaus üblich ist.

Um gleich bei dem Begriff der Transitivität zu bleiben: ein heutzutage alltägliches Szenario ist der Verbindungsaufbau eines *HTTPS*-Tunnels. Wird beispielsweise mit Hilfe eines Webbrowsers die Webseite *tiss.tuwien.ac.at* aufgerufen, so wird von einer Art Server der TU Wien unter anderem ein öffentlicher Schlüssel übertragen, welcher wiederum von einer weiteren Stelle signiert wurde. Diese „Signatur-Kette“ endet relativ bald bei einer *vertrauenswürdigen* Unterschrift – also einer digitalen Signatur, welche von einer sogenannten *Stammzertifizierungsstelle* ausgestellt wurde, die offiziell anerkannt und somit dem Webbrowser bzw. dem Betriebssystem bekannt ist. So eine Kette von Signaturen kann üblicherweise in aktuellen Webbrowsern vom Benutzer eingesehen und auch genauer untersucht werden.

In diesem Szenario dient das verwendete asymmetrische Kryptosystem also nicht nur dem Schlüsselaustausch für eine Absicherung der zu übertragenden Daten, sondern auch einer Sicherstellung der Identität des Kommunikationspartners am anderen Ende der Leitung. Wäre diese Absicherung nicht gegeben, so wäre es technisch durchaus möglich, die Adresse *tiss.tuwien.ac.at* auf einen bösartigen Server weiterzuleiten, um gleich anschließend den Benutzer nach seinen Zugangsdaten zu fragen – ohne digitale Signaturen könnte die browsende Person nicht gewarnt werden, dass an der Identität des aktuellen Kommunikationspartners etwas „faul“ ist. Die Aufgabe der *Stammzertifizierungsstellen* ist dabei naturgemäß eine äußerst verantwortungsvolle und heikle Angelegenheit – vergleichbar mit der des zuständigen Amtes eines Staates, um Pässe auszustellen.

Eine weitere Anwendungsmöglichkeit für die digitale Unterschrift stellt das Schützen von beispielsweise Software dar. Erhält bzw. erwirbt ein Anwender die Lizenz, um eine kommerzielle Anwendung zu nutzen, so muss diese Anwendung sicherstellen, dass es sich bei der angegebenen Lizenz nicht um eine Fälschung handelt. Das lässt sich natürlich nicht mit einfachen Buchstaben- und Zahlenfolgen bewerkstelligen, die bei der

Installation eingegeben werden – diese „Keys“ sind schnell abgeschrieben oder sogar auf Knopfdruck via E-Mail weitergeleitet. Soll also ein entsprechender Schutzmechanismus bei der Verwendung einer ungültigen Lizenz mehr als nur Gewissensbisse verursachen, so sind digitale Signaturen eine gute Grundlage für die Entwicklung so eines Kopierschutzes. Dieses Szenario wird in Abschnitt 6.3 ausführlich behandelt.

Natürlich darf man bei der digitalen Unterschrift nicht auf die wahrscheinlich naheliegendste Anwendung vergessen: die *Unterschrift*. Wird ein digitales Dokument erstellt, so kann dieses in offensichtlicher Weise mit den hier vorgestellten Mitteln auch unterschrieben werden. Bei diesem Dokument kann es sich um ein E-Mail, ein Bild, ein Video, ein Musikstück, eine Präsentation – kurz: um eine beliebige elektronische Datei handeln. Dieser Anwendungsfall wird Gegenstand des nun folgenden Abschnittes sein.

Bemerkung 6.1.1. Im Gegensatz zu den vorhergehenden Kapiteln ist der Sourcecode der hier vorgestellten Programme nicht im Anhang enthalten. Die Programme aus diesem Kapitel dienen nur der Veranschaulichung von den in dieser Arbeit behandelten mathematischen Methoden und können auch leicht mit .NET Mitteln nachgebaut werden – dazu sei vor allem auf den Namespace *System.Security.Cryptography* verwiesen.

Bemerkung 6.1.2. Die hier vorgestellten Beispielprogramme wurden auf Basis der im .NET Framework enthaltenen *Windows Presentation Foundation* (WPF) entwickelt – durch eine damit leicht mögliche Separation von Benutzeroberfläche sowie Programmablauf sind die interessanten Programmteile, welche in diesem Abschnitt behandelt werden, frei von den Komponenten einer Benutzeroberfläche.

Bemerkung 6.1.3. Die Konzepte und Patterns von WPF werden in [24] zum Teil erläutert. Für die Beispiele im Rahmen dieser Arbeit wurden diese Richtlinien nur stark vereinfacht übernommen, schließlich handelt es sich um relativ kleine Anwendungen, für die auch keine Erweiterungen geplant sind. Es wurde nur zwischen *View* sowie *ViewModel* unterschieden, wobei das *ViewModel* der Einfachheit halber auch gleich für die Abwicklung der notwendigen „Kommandos“ zuständig ist. Für eine Beschreibung der dafür notwendigen Schnittstellen siehe [20] und [21].

Bemerkung 6.1.4. Rechen- und damit auch zeitaufwändige Vorgänge sollten in einer Windowsapplikation in einem eigenen Thread ausgeführt werden, um nicht die Benutzeroberfläche zu blockieren. Der für den Anwender sichtbare Teil wird in einem speziellen Thread ausgeführt, welcher auch immer die entsprechenden „Kommandos“ in seinem eigenen Kontext aufruft, was zu Blockaden von eben diesem Thread führen kann. Prozesse, wie zum Beispiel das Generieren eines neuen Schlüsselpaares, wurden deswegen mit Hilfe der Komponenten *BackgroundWorker* (siehe [22]) ausgelagert – mit diesem Modul ist es relativ einfach möglich, das Ergebnis so eines ausgelagerten Prozesses wieder mit der Benutzeroberfläche zu synchronisieren.

6.2. „Klassische“ Unterschrift

In diesem Beispiel handelt es sich bei der „Nachricht m“ um eine beliebige Datei, welche eine digitale Unterschrift erhalten soll. Sowohl das Schlüsselpaar als auch die Signatur werden von dem Beispielprogramm in einem XML-Format gespeichert, wobei diese XML-Dateien auch später wieder geladen werden können, um beliebige Dateien zu signieren bzw. Signaturen zu überprüfen.

Die Funktionen, welche im Namen mit dem Kürzel *Async* enden, werden, so wie in Bemerkung 6.1.4 beschrieben, nicht direkt von der Benutzeroberfläche aufgerufen. Der Unterschied ist vor allem für die Fehlerbehandlung relevant, da dann *Exceptions* unterschiedlich zu behandeln sind – der Übersicht halber wurden jedoch die jeweils angeführten Programmcodes auf den wesentlichen Teil gekürzt, wodurch derartige Unterschiede nicht mehr ersichtlich sind.

Die Anwendung selbst ist in Abbildung 6.1 dargestellt: im obersten Bereich ist die Konfiguration globaler Optionen möglich. So können das zu verwendende asymmetrische Kryptosystem sowie der anzuwendende Hashalgorithmus festgelegt werden. Mittels Schlüsselstärke ist die gewünschte Anzahl an Bits des Schlüsselpaares festzulegen, wobei natürlich darauf zu achten ist, dass diese mit der gewählten Hashfunktion kompatibel ist – der Schlüssel darf also nicht zu klein gewählt werden, wobei Schlüssel mit weniger als 100 Bits erst gar nicht unterstützt werden.

Programmcode 6.2.1. Mit Klick auf den Button „Generieren...“ wird ein neues den gewählten Optionen entsprechendes Schlüsselpaar generiert.

```
void executeGenerateKeyAsync() {
    using (var crypto = new CryptoBig()) {
        crypto.CurrentStrategy = CurrentCryptoStrategy;
        crypto.KeyStrength = CurrentKeyStrength;

        dynamic privateKey;
        dynamic publicKey;
        crypto.CreateKeyPair(out publicKey, out privateKey);

        CurrentPrivateKey =
            (IDictionary<string, object>)privateKey;
        CurrentPublicKey =
            (IDictionary<string, object>)publicKey;
    }
    IsCurrentSignatureValid = null;
}
```

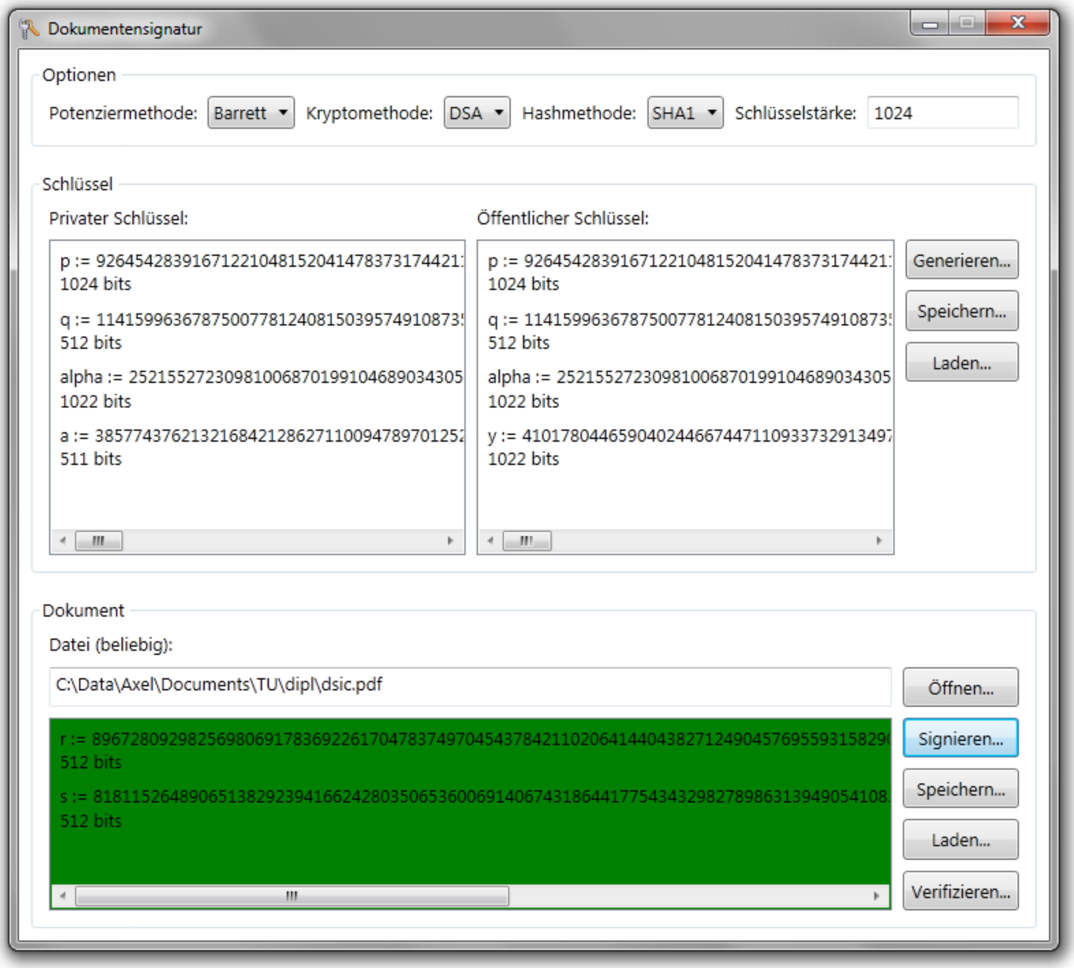

Abbildung 6.1.: Erstellung einer „klassischen" Unterschrift

Die von der Klasse *CryptoBig* verwendeten Objekte vom Type *ExpandoObject* sind mit der Schnittstelle *IDictionary<TKey, TValue>* kompatibel, welche in WPF leicht eingebunden werden kann.

Die beiden Schaltflächen „Laden...“ sowie „Speichern...“ im mittleren Bereich der Anwendung werden ihrem Namen gerecht, indem sie das eingangs erwähnte Laden und Speichern in ein XML-Format auslösen. Dabei wird das Entfernen des privaten Schlüssels aus der XML-Datei unterstützt, um das Konzept der digitalen Signatur auch richtig nachstellen zu können.

Programmcode 6.2.2. Der Button „Speichern...“ führt zu:

```
void executeSaveKey() {
    var dialog = new SaveFileDialog();
    dialog.FileName = "keyPair";
    dialog.DefaultExt = ".xml";

    if (dialog.ShowDialog() == true) {
        var xml = new XElement("keyPair",
            new XElement("privateKey",
                from x in CurrentPrivateKey
                select new XElement(x.Key, x.Value)),
            new XElement("publicKey",
                from x in CurrentPublicKey
                select new XElement(x.Key, x.Value)));
        using (var stream = dialog.OpenFile()) {
            xml.Save(stream);
        }
    }
}
```

Das „Root“-Element wird *KeyPair* genannt und enthält unter sich zwei Knoten – einen für den privaten und einen für den öffentlichen Schlüssel. Auch bei der Generierung des XMLs ist wieder die Schnittstelle *IDictionary* sehr hilfreich. Unabhängig von dem verwendeten Kryptosystem wird einfach pro Wert, der dem Objekt vom Typ *ExpandoObject* zugewiesen wurde, ein eigenes XML-Element erzeugt.

Mit Hilfe eines einfachen Texteditors kann aus der erstellten XML-Datei der Knoten *privateKey* entfernt werden. Dabei ist darauf zu achten, dass der komplette Knoten gelöscht wird und nicht nur dessen Inhalt. Die Funktion *executeLoadKey* kann damit entsprechend umgehen – in der Anwendung ist dann einfach nur der öffentliche Schlüssel geladen. Das Erstellen von Signaturen wird in diesem Fall natürlich deaktiviert.

Die Laden-Funktion funktioniert analog und wird deswegen nicht extra angeführt. Genauso arbeiten die beiden entsprechenden Schaltflächen im unteren Bereich, nur wird eben die Signatur in eine XML-Datei geschrieben bzw. aus einer solchen geladen.

Der Button „Öffnen...“ im Signaturbereich bietet einen unter Windows bekannten *OpenFileDialog*, welcher im Namespace *Microsoft.Win32* zu finden ist, damit der Benutzer die zu signierende Datei auswählen kann. Von dieser Datei wird im ersten Schritt auch nur der genaue Pfad benötigt – interessant sind also erst wieder die Schaltflächen „Signieren...“ und „Verifizieren...“.

Programmcode 6.2.3. Endlich kann eine digitale Signatur erstellt...

```
void executeGenerateSignatureAsync() {
    IntBig h = null;
    using (var stream = File.OpenRead(CurrentFile)) {
        using (var hash = new HashBig()) {
            hash.CurrentStrategy = CurrentHashStrategy;
            h = hash.ComputeHash(stream);
        }
    }
    using (var crypto = new CryptoBig()) {
        crypto.CurrentStrategy = CurrentCryptoStrategy;
        var signature = crypto.Sign(h, CurrentPrivateKey);
        CurrentSignature =
            (IDictionary<string, object>)signature;
        IsCurrentSignatureValid = true;
    }
}
```

Programmcode 6.2.4. ...und auch verifizert werden.

```
void executeVerifySignatureAsync() {
    IntBig h = /* ... */
    using (var crypto = new CryptoBig()) {
        crypto.CurrentStrategy = CurrentCryptoStrategy;
        IsCurrentSignatureValid = crypto.Verify(h,
            CurrentSignature, CurrentPublicKey);
    }
}
```

6.3. Kopierschutzmechanismen

Ein weiteres leicht zu demonstrierendes Beispiel ist die Verwendung von digitalen Signaturen für die Entwicklung eines Kopierschutzmechanismus. Dank der in dieser Arbeit behandelten kryptologischen Methoden können „Aktivierungsschlüssel“ erstellt werden, welche nicht zu „knacken“ sein sollten – ganz im Gegensatz zu oft üblichen Methoden, um kommerzielle Software zu schützen.

Die Anforderungen an ein System, welches eine Anwendung vor einer unerlaubten Weitergabe schützen soll, passen ausgezeichnet zu der Funktionalität eines asymmetrischen Kryptosystems für digitale Signaturen: so eine schützenswerte Anwendung „will“ bei ihrem Start feststellen können, ob sie lizenziert ist und ob ihre Lizenz auch ordnungsgemäß ausgestellt wurde. Wird also eine derartige Lizenz in eine Form gebracht, die der Softwarehersteller signieren kann, so muss einfach diese Signatur überprüft werden, nachdem der Inhalt der Lizenz verarbeitet wurde.

Um die Weitergabe einer Lizenz zu verhindern, ist die Verknüpfung eben dieser an eine Person bzw. an einen Computer notwendig. Aktivierungsschlüssel, die einfach nur aufgrund ihrer Beschaffenheit oder Struktur gültig sind, können leicht wiederverwendet werden. Eine Überprüfung, ob jeder Schlüssel nur genau ein einziges Mal in Verwendung ist, wäre naturgemäß relativ aufwendig – sowohl für den Anwender als auch den Herausgeber, wie die Praxis zeigt.

Der Anwender muss also im ersten Schritt eine Art Schlüssel erhalten, welcher rein der Identifikation dient. Dieser Key könnte dann dazu berechtigen, eine Software freizuschalten – und hier kommt die digitale Signatur ins Spiel. Die Lizenz müsste Kennzeichen des Computers bzw. des Benutzers enthalten, welche sich nicht fälschen lassen – viele Hardwarekomponenten verfügen über entsprechende Merkmale. Darüber hinaus wäre es kein Problem, in diese Lizenz weitere Merkmale wie freigeschaltete Funktionalität oder ein Ablaufdatum einzupflegen.

Sobald diese – in der Regel kommerzielle – Software aktiviert wurde, verliert der ursprüngliche Schlüssel seine Gültigkeit, was dessen Weitergabe ad absurdum führt. Dieses Vorgehen kann natürlich gelockert werden, schließlich kommt es durchaus vor, dass eine Person den Computer wechselt bzw. erneuert. Alternativ wäre noch eine regelmäßige Erneuerung der Lizenz möglich. Aufgrund der Kurzlebigkeit von Software ist eine Einschränkung auf ca. 3 bis 5 Freischaltungen wahrscheinlich die einfachste Variante.

Das hier vorgestellte Beispiel verwendet als Grundlage die ID der primären CPU sowie die ID des Windows Accounts eines Benutzers. Weiters wird nur das Generieren und Verifizieren der Lizenz selbst demonstriert – der komplette Vorgang inklusive dem beschriebenen Schlüssel zur Identifikation ist zumindest im Rahmen der digitalen Signaturen weniger interessant. Darüber hinaus soll die Lizenz eine Aktivierung von Zusatzfunktionalitäten erlauben – im Beispiel „Funktionalität 1“ sowie „Funktionalität 2“ genannt –, um auch diesen Anwendungsfall demonstrieren zu können. Mit Hilfe eines Ablaufdatums wäre dann noch die Erstellung von temporären Lizenzen möglich – das Verstellen der Systemuhr kann dabei natürlich nicht verhindert werden, genausowenig wie ein Formatieren der Festplatte.

Die beiden Beispielanwendungen sind in Abbildung 6.2 dargestellt. Es wurde eine Applikation zur Erstellung neuer Lizenzen entwickelt, sowie eine Minianwendung, welche einfach bei ihrem Start die installierte Lizenz überprüft und Informationen über eben diese anzeigt – im Bild läuft die Lizenz Ende des Jahres ab. Für die Signatur wurde ein fixes *DSA*-Schlüsselpaar sowie *SHA-256* als Hashalgorithmus gewählt

Zuerst benötigen wir die besprochenen IDs von Hardware und User:

Programmcode 6.3.1. Die ID der primären CPU kann per WMI ausgelesen werden. Um auf die dafür notwendigen APIs zugreifen zu können, muss das Assembly *System.Management* eingebunden werden (siehe auch [25]).

```
string getCurrentCpuId() {
    var cpu = new ManagementObject(
        "Win32_Processor.DeviceId=\"CPU0\"");
    return (string)cpu.Properties["ProcessorId"].Value;
}
```

Programmcode 6.3.2. Für den aktuellen Windows Benutzer ist auch bereits eine passende Klasse im .NET Framework enthalten.

```
string getCurrentUserId() {
    var user = WindowsIdentity.GetCurrent();
    return user.User.Value;
}
```

Es gibt zwar Möglichkeiten, die ID des aktuellen Benutzers zu ändern, jedoch können nicht zwei Benutzer auf einem Computer dieselbe ID erhalten. Und unterschiedliche Computer haben in der Regel auch unterschiedliche CPUs...

Die Lizenz wird von dem ersten Beispielprogramm in der Windows Registry eingetragen, was mit dem Windows Tool *regedit.exe* überprüft werden kann. Signiert wird ein String, welcher aus den Werten der Lizenz sowie den beiden IDs zusammengesetzt wird – dieser String wird bei jedem Start der zweiten Anwendung neu generiert, um gleich anschließend die digitale Signatur überprüfen zu können. Für die Konvertierung des Strings in ein Byte Array wird die Klasse *UTF8* verwendet, welche im Namespace *System.Text* zu finden ist.

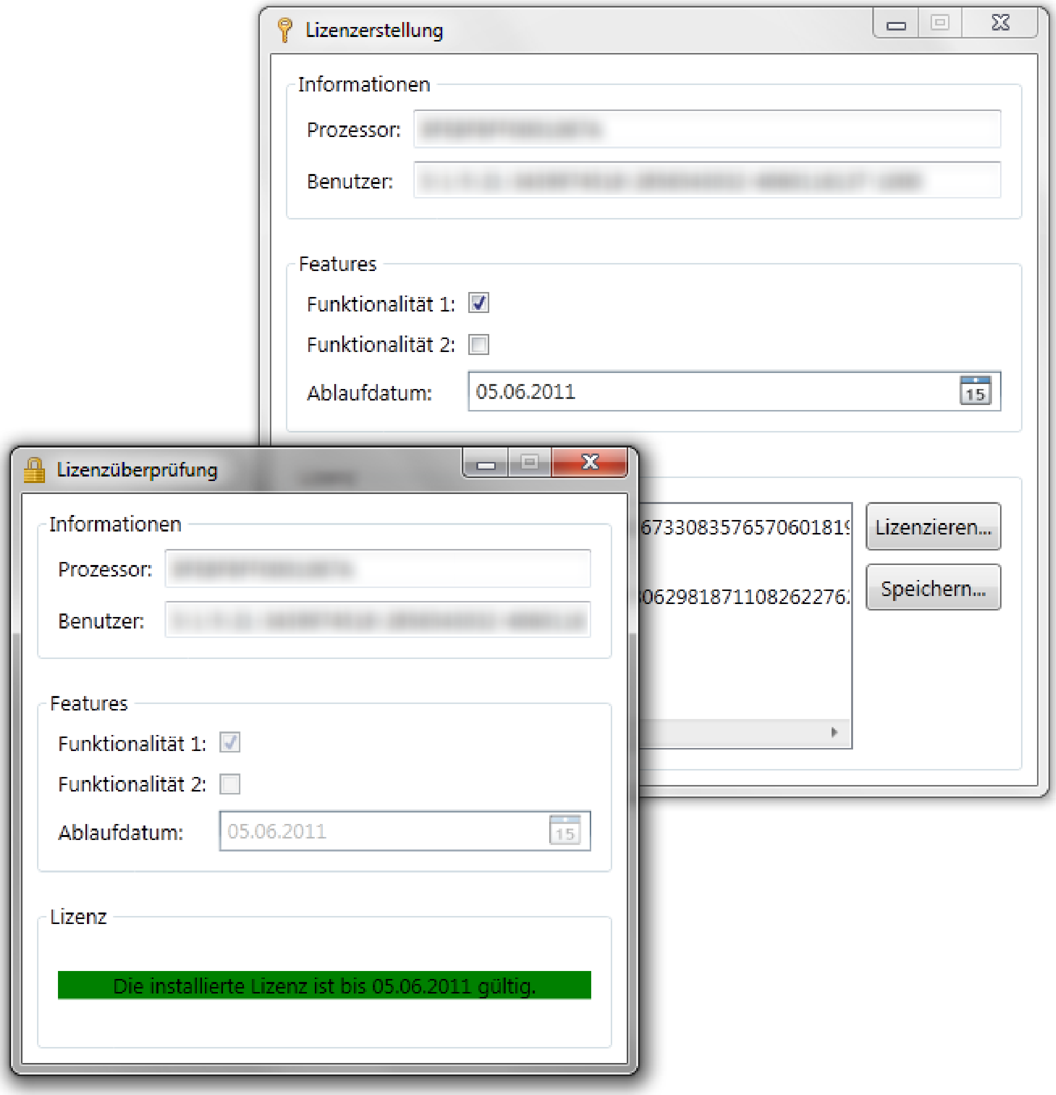

Abbildung 6.2.: Erstellung und Überprüfung einer Lizenz

Programmcode 6.3.3. Die Erstellung einer neuen Lizenz.

```
void executeGenerateLicense() {
    var license = new StringBuilder();
    license.Append(CurrentCpuId);
    license.Append(CurrentUserId);
    license.Append(HasFeature1);
    license.Append(HasFeature2);

    if (ValidUntil.HasValue) {
        license.Append(ValidUntil.Value.ToString("yyyyMMdd"));
    }

    var rawLicense = Encoding.UTF8.GetBytes(
        license.ToString());

    IntBig h;
    using (var hash = new HashBig()) {
        hash.CurrentStrategy = HashBigStrategy.Sha2;
        using (var stream = new MemoryStream(rawLicense)) {
            h = hash.ComputeHash(stream);
        }
    }

    dynamic privateKey = new ExpandoObject();
    privateKey.p = IntBig.Parse("...");
    privateKey.q = IntBig.Parse("...");
    privateKey.alpha = IntBig.Parse("...");
    privateKey.a = IntBig.Parse("...");

    dynamic l;
    using (var crypto = new CryptoBig()) {
        crypto.CurrentStrategy = CryptoBigStrategy.Dsa;
        l = crypto.Sign(h, privateKey);
    }

    CurrentLicense = (IDictionary<string, object>)l;
}
```

Im Anschluss werden von *executeSaveLicense* die Werte von *HasFeature1*, *HasFeature2* sowie *ValidUntil1* zusammen mit der Lizenz *CurrentLicense* in der Windows Registry abgelegt:

Programmcode 6.3.4. Das Speichern der Lizenz für die Minianwendung.

```
void executeSaveLicense() {
    var baseKey = Registry.CurrentUser
        .CreateSubKey(@"SOFTWARE\Heer.Axel.Thesis");

    baseKey.SetValue("HasFeature1", HasFeature1);
    baseKey.SetValue("HasFeature2", HasFeature2);
    if (ValidUntil.HasValue) {
        baseKey.SetValue("ValidUntil", ValidUntil.Value
            .ToString("yyyyMMdd"));
    }
    else {
        baseKey.DeleteValue("ValidUntil", false);
    }

    baseKey = baseKey.CreateSubKey("License");

    foreach (var i in CurrentLicense) {
        baseKey.SetValue(i.Key, i.Value);
    }
}
```

Auch an dieser Stelle ist die *IDictionary* Implementierung der Klasse *ExpandoObject* von Nutzen. Die API für den Zugriff auf die Windows Registry ist im Namespace *Microsoft.Win32* untergebracht.

In der Praxis würden diese beiden Schritte natürlich nicht von ein und derselben Applikation durchgeführt werden. Die Erstellung der Lizenz muss beim Hersteller liegen, da der private Schlüssel naturgemäß nicht herausgegeben werden darf. Und die „Installation“ der Lizenz liegt natürlich in der Hand des Anwenders – um eine Weitergabe der Lizenz zu simulieren, können die Daten mittels *regedit.exe* exportiert werden. Die von *regedit.exe* generierte *reg*-Datei kann dann unter einem anderen Account bzw. auf einem anderen Computer registriert werden.

Die Minianwendung, welche die in der Registry hinterlegte Lizenz überprüft, führt im Wesentlichen analoge Schritte durch, nur eben in die andere Richtung. Der grobe Ablauf davon ist im letzten Programmcode dargestellt:

Programmcode 6.3.5. Startroutine der Minianwendung.

```
try {
    validateLicense(extractFromRegistry());

    if (ValidUntil.HasValue && ValidUntil < DateTime.Today) {
        throw new Exception("Die Testlizenz ist abgelaufen.");
    }

    IsLicenseValid = true;

    if (ValidUntil.HasValue) {
        LicenseMessage = "Die installierte Lizenz ist bis "
            + ValidUntil.Value.ToShortDateString()
            + " gueltig.";
    }
    else {
        LicenseMessage =
            "Die installierte Lizenz ist gueltig.";
    }
}
catch (Exception ex) {
    IsLicenseValid = false;

    LicenseMessage =
        "Es ist keine gueltige Lizenz installiert. Fehler: "
        + ex.Message;
}
```

Die nicht extra angeführte Funktion *extractFromRegistry* liest die von *executeSaveLicense* gespeicherten Werte aus der Windows Registry ein, welche im Anschluss von *validateLicense* überprüft werden – sollte es sich dabei um eine Fälschung handeln, so wird von *validateLicense* eine Exception geworfen.

A. Sourcecode

Im Folgenden ist der komplette Quelltext der Klassenbibliothek, welche in den einzelnen Anwendungen verwendet wird, aufgelistet. Sie enthält den mathematisch interessanten Teil der C#-Implementierung.

A.1. IntBig

```
const int ThresholdMul1 = 2048 / 32;
const int ThresholdMul2 = 8192 / 32;
const int ThresholdSqu1 = 4096 / 32;
const int ThresholdSqu2 = 16384 / 32;

private uint[] innerBits;
private bool isNegative;

public bool IsNegative {
    get { return isNegative; }
}

public bool IsZero {
    get { return innerBits.Length == 1 && innerBits[0] == 0; }
}

public bool IsOne {
    get { return !isNegative
        && innerBits.Length == 1 && innerBits[0] == 1; }
}

public bool IsMinusOne {
    get { return isNegative
        && innerBits.Length == 1 && innerBits[0] == 1; }
}

public bool IsOdd {
    get { return (innerBits[0] & 1) != 0; }
}

private IntBig(uint[] innerBits, bool isNegative) {
```

```
    Debug.Assert(innerBits != null, "innerBits != null");
    Debug.Assert(innerBits.Length > 0, "innerBits.Length > 0");

    // check for leading zeros
    var length = innerBits.Length;
    while (length > 1 && innerBits[length - 1] == 0) {
        --length;
    }

    // don't share bits between IntBig instances
    var bits = new uint[length];
    Buffer.BlockCopy(innerBits, 0, bits, 0, 4 * length);

    // zero is not negative
    if (bits.Length == 1 && bits[0] == 0) {
        isNegative = false;
    }

    this.innerBits = bits;
    this.isNegative = isNegative;
}

#region conversions

private IntBig(int value) {
    isNegative = value < 0;
    innerBits = new[] { (uint)Math.Abs(value) };
}

private IntBig(long value) {
    isNegative = value < 0;
    if ((Math.Abs(value) >> 32) != 0) {
        innerBits = new[] { (uint)Math.Abs(value),
                            (uint)(Math.Abs(value) >> 32) };
    }
    else {
        innerBits = new[] { (uint)Math.Abs(value) };
    }
}

public static implicit operator IntBig(int value) {
    return new IntBig(value);
}

public static implicit operator IntBig(long value) {
    return new IntBig(value);
}

```

```
public static explicit operator int(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    if (value.innerBits.Length > 1) {
        throw new OverflowException();
    }
    if (value.innerBits[0] > (uint)int.MaxValue) {
        throw new OverflowException();
    }
    if (value.isNegative) {
        return -1 * (int)value.innerBits[0];
    }
    return (int)value.innerBits[0];
}

public static explicit operator long(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    if (value.innerBits.Length > 2) {
        throw new OverflowException();
    }
    if (value.innerBits.Length == 2) {
        if (value.innerBits[1] > (uint)int.MaxValue) {
            throw new OverflowException();
        }
        if (value.isNegative) {
            return -1L * (long)((ulong)value.innerBits[0]
                                + ((ulong)value.innerBits[1] << 32));
        }
        return (long)((ulong)value.innerBits[0]
                    + ((ulong)value.innerBits[1] << 32));
    }
    if (value.isNegative) {
        return -1L * (long)value.innerBits[0];
    }
    return (long)value.innerBits[0];
}

#endregion

#region operators

public static IntBig operator >>(IntBig value, int shift) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
```

```
    if (shift < 0) {
        return LeftShift(value, -shift);
    }
    var bits = rightShift(value.innerBits, shift);
    return new IntBig(bits, value.isNegative);
}

public static IntBig operator <<(IntBig value, int shift) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    if (shift < 0) {
        return RightShift(value, -shift);
    }
    var bits = leftShift(value.innerBits, shift);
    return new IntBig(bits, value.isNegative);
}

public static bool operator ==(IntBig left, IntBig right) {
    return Compare(left, right) == 0;
}

public static bool operator !=(IntBig left, IntBig right) {
    return Compare(left, right) != 0;
}

public static bool operator <(IntBig left, IntBig right) {
    return Compare(left, right) < 0;
}

public static bool operator <=(IntBig left, IntBig right) {
    return Compare(left, right) <= 0;
}

public static bool operator >(IntBig left, IntBig right) {
    return Compare(left, right) > 0;
}

public static bool operator >=(IntBig left, IntBig right) {
    return Compare(left, right) >= 0;
}

public static IntBig operator +(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return new IntBig(value.innerBits, value.isNegative);
}
```

```

public static IntBig operator +(IntBig left, int right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (left.isNegative != (right < 0)) {
        return Negate(Subtract(Negate(left), right));
    }
    var bits = add(left.innerBits, (uint)Math.Abs(right));
    return new IntBig(bits, left.isNegative);
}

public static IntBig operator +(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (left.isNegative != right.isNegative) {
        return Subtract(left, Negate(right));
    }
    if (left.innerBits.Length < right.innerBits.Length) {
        var bits = add(right.innerBits, left.innerBits);
        return new IntBig(bits, left.isNegative);
    }
    else {
        var bits = add(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative);
    }
}

public static IntBig operator ++(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return Add(value, 1);
}

public static IntBig operator -(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return new IntBig(value.innerBits, !value.isNegative);
}

public static IntBig operator -(IntBig left, int right) {
    if (object.ReferenceEquals(left, null)) {
```

```
        throw new ArgumentNullException("left");
    }
    if (left.isNegative != (right < 0)) {
        return Negate(Add(Negate(left), right));
    }
    if (left.innerBits.Length == 1 &&
        left.innerBits[0] < (uint)Math.Abs(right)) {
        var bits = new[] { (uint)Math.Abs(right) - left.innerBits[0] };
        return new IntBig(bits, !left.isNegative);
    }
    else {
        var bits = subtract(left.innerBits, (uint)Math.Abs(right));
        return new IntBig(bits, left.isNegative);
    }
}

public static IntBig operator -(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (left.isNegative != right.isNegative) {
        return Add(left, Negate(right));
    }
    var diff = compare(left.innerBits, right.innerBits);
    if (diff < 0) {
        var bits = subtract(right.innerBits, left.innerBits);
        return new IntBig(bits, !left.isNegative);
    }
    else {
        var bits = subtract(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative);
    }
}

public static IntBig operator --(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return Subtract(value, 1);
}

public static IntBig operator *(IntBig left, int right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
```

```
    var bits = multiply(left.innerBits, (uint)Math.Abs(right));
    return new IntBig(bits, left.isNegative ^ (right < 0));
}

public static IntBig operator *(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (object.ReferenceEquals(left, right)) {
        var bits = square(left.innerBits);
        return new IntBig(bits, false);
    }
    else {
        var bits = multiply(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative ^ right.isNegative);
    }
}

public static IntBig operator /(IntBig dividend, int divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (divisor == 0) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    uint[] bits = null;
    var modulus = 0U;

    // the dividend should be bigger
    if (dividendBits.Length > 1 ||
        dividendBits[0] > (uint)Math.Abs(divisor)) {
        bits = divide(dividendBits, (uint)Math.Abs(divisor),
            out modulus);
    }
    else if (dividendBits[0] == (uint)Math.Abs(divisor)) {
        bits = new uint[] { 1 };
    }
    else {
        bits = new uint[] { 0 };
    }

    return new IntBig(bits, dividend.isNegative ^ (divisor < 0));
}
```

```
public static IntBig operator /(IntBig dividend, IntBig divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (object.ReferenceEquals(divisor, null)) {
        throw new ArgumentNullException("divisor");
    }
    if (divisor.IsZero) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    uint[] bits = null;

    // the dividend should be bigger
    var diff = compare(dividendBits, divisor.innerBits);
    if (diff > 0) {
        bits = divide(dividendBits, divisor.innerBits, out dividendBits);
    }
    else if (diff == 0) {
        bits = new uint[] { 1 };
    }
    else {
        bits = new uint[] { 0 };
    }

    return new IntBig(bits, dividend.isNegative ^ divisor.isNegative);
}

public static int operator %(IntBig dividend, int divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (divisor == 0) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    var modulus = 0U;

    // the dividend should be bigger
    if (dividendBits.Length > 1 ||
        dividendBits[0] > (uint)Math.Abs(divisor)) {
        modulus = divideModOnly(dividendBits, (uint)Math.Abs(divisor));
    }
    else if (dividendBits[0] == (uint)Math.Abs(divisor)) {
        modulus = 0;
```

```
    }
    else {
        modulus = dividendBits[0];
    }

    return (int)modulus;
}

public static IntBig operator %(IntBig dividend, IntBig divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (object.ReferenceEquals(divisor, null)) {
        throw new ArgumentNullException("divisor");
    }
    if (divisor.IsZero) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;

    // the dividend should be bigger
    var diff = compare(dividendBits, divisor.innerBits);
    if (diff > 0) {
        divide(dividendBits, divisor.innerBits, out dividendBits);
    }
    else if (diff == 0) {
        dividendBits = new uint[] { 0 };
    }

    return new IntBig(dividendBits, dividend.isNegative);
}

#endregion

#region operator functions

public static IntBig RightShift(IntBig value, int shift) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    if (shift < 0) {
        return LeftShift(value, -shift);
    }
    var bits = rightShift(value.innerBits, shift);
    return new IntBig(bits, value.isNegative);
}

```

```
public static IntBig LeftShift(IntBig value, int shift) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    if (shift < 0) {
        return RightShift(value, -shift);
    }
    var bits = leftShift(value.innerBits, shift);
    return new IntBig(bits, value.isNegative);
}

public static int Compare(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null) &&
        object.ReferenceEquals(right, null)) {
        return 0;
    }
    if (object.ReferenceEquals(left, null)) {
        return -1;
    }
    if (object.ReferenceEquals(right, null)) {
        return 1;
    }

    var sign = 1;

    // sign handling
    if (left.isNegative && right.isNegative) {
        sign = -1;
    }
    else {
        if (left.isNegative) {
            return -1;
        }
        if (right.isNegative) {
            return 1;
        }
    }

    return sign * compare(left.innerBits, right.innerBits);
}

public static IntBig Plus(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return new IntBig(value.innerBits, value.isNegative);
}

```

```
public static IntBig Add(IntBig left, int right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (left.isNegative != (right < 0)) {
        return Negate(Subtract(Negate(left), right));
    }
    var bits = add(left.innerBits, (uint)Math.Abs(right));
    return new IntBig(bits, left.isNegative);
}

public static IntBig Add(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (left.isNegative != right.isNegative) {
        return Subtract(left, Negate(right));
    }
    if (left.innerBits.Length < right.innerBits.Length) {
        var bits = add(right.innerBits, left.innerBits);
        return new IntBig(bits, left.isNegative);
    }
    else {
        var bits = add(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative);
    }
}

public static IntBig Increment(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return Add(value, 1);
}

public static IntBig Negate(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return new IntBig(value.innerBits, !value.isNegative);
}

public static IntBig Subtract(IntBig left, int right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
```

```
    }
    if (left.isNegative != (right < 0)) {
        return Negate(Add(Negate(left), right));
    }
    if (left.innerBits.Length == 1 &&
        left.innerBits[0] < (uint)Math.Abs(right)) {
        var bits = new[] { (uint)Math.Abs(right) - left.innerBits[0] };
        return new IntBig(bits, !left.isNegative);
    }
    else {
        var bits = subtract(left.innerBits, (uint)Math.Abs(right));
        return new IntBig(bits, left.isNegative);
    }
}

public static IntBig Subtract(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (left.isNegative != right.isNegative) {
        return Add(left, Negate(right));
    }
    var diff = compare(left.innerBits, right.innerBits);
    if (diff < 0) {
        var bits = subtract(right.innerBits, left.innerBits);
        return new IntBig(bits, !left.isNegative);
    }
    else {
        var bits = subtract(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative);
    }
}

public static IntBig Decrement(IntBig value) {
    if (object.ReferenceEquals(value, null)) {
        throw new ArgumentNullException("value");
    }
    return Subtract(value, 1);
}

public static IntBig Multiply(IntBig left, int right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    var bits = multiply(left.innerBits, (uint)Math.Abs(right));
```

```
    return new IntBig(bits, left.isNegative ^ (right < 0));
}

public static IntBig Multiply(IntBig left, IntBig right) {
    if (object.ReferenceEquals(left, null)) {
        throw new ArgumentNullException("left");
    }
    if (object.ReferenceEquals(right, null)) {
        throw new ArgumentNullException("right");
    }
    if (object.ReferenceEquals(left, right)) {
        var bits = square(left.innerBits);
        return new IntBig(bits, false);
    }
    else {
        var bits = multiply(left.innerBits, right.innerBits);
        return new IntBig(bits, left.isNegative ^ right.isNegative);
    }
}

[SuppressMessage("Microsoft.Design", "CA1021:AvoidOutParameters")]
public static IntBig Divide(IntBig dividend, int divisor,
                                        out int modulus) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (divisor == 0) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    uint[] bits = null;
    var mod = 0U;

    // the dividend should be bigger
    if (dividendBits.Length > 1 ||
        dividendBits[0] > (uint)Math.Abs(divisor)) {
        bits = divide(dividendBits, (uint)Math.Abs(divisor), out mod);
    }
    else if (dividendBits[0] == (uint)Math.Abs(divisor)) {
        bits = new uint[] { 1 };
        mod = 0;
    }
    else {
        bits = new uint[] { 0 };
        mod = dividendBits[0];
    }

```

```
    modulus = (int)mod;

    return new IntBig(bits, dividend.isNegative);
}

[SuppressMessage("Microsoft.Design", "CA1021:AvoidOutParameters")]
public static IntBig Divide(IntBig dividend, IntBig divisor,
                                          out IntBig modulus) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (object.ReferenceEquals(divisor, null)) {
        throw new ArgumentNullException("divisor");
    }
    if (divisor.IsZero) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    uint[] bits = null;

    // the dividend should be bigger
    var diff = compare(dividendBits, divisor.innerBits);
    if (diff > 0) {
        bits = divide(dividendBits, divisor.innerBits, out dividendBits);
    }
    else if (diff == 0) {
        dividendBits = new uint[] { 0 };
        bits = new uint[] { 1 };
    }
    else {
        bits = new uint[] { 0 };
    }

    modulus = new IntBig(dividendBits, dividend.isNegative);
    return new IntBig(bits, dividend.isNegative ^ divisor.isNegative);
}

public static IntBig Divide(IntBig dividend, int divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (divisor == 0) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    uint[] bits = null;
```

```
    var modulus = 0U;

    // the dividend should be bigger
    if (dividendBits.Length > 1 ||
        dividendBits[0] > (uint)Math.Abs(divisor)) {
        bits = divide(dividendBits, (uint)Math.Abs(divisor),
            out modulus);
    }
    else if (dividendBits[0] == (uint)Math.Abs(divisor)) {
        bits = new uint[] { 1 };
    }
    else {
        bits = new uint[] { 0 };
    }

    return new IntBig(bits, dividend.isNegative ^ (divisor < 0));
}

public static IntBig Divide(IntBig dividend, IntBig divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (object.ReferenceEquals(divisor, null)) {
        throw new ArgumentNullException("divisor");
    }
    if (divisor.IsZero) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    uint[] bits = null;

    // the dividend should be bigger
    var diff = compare(dividendBits, divisor.innerBits);
    if (diff > 0) {
        bits = divide(dividendBits, divisor.innerBits, out dividendBits);
    }
    else if (diff == 0) {
        bits = new uint[] { 1 };
    }
    else {
        bits = new uint[] { 0 };
    }

    return new IntBig(bits, dividend.isNegative ^ divisor.isNegative);
}

public static int Mod(IntBig dividend, int divisor) {
```

```
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (divisor == 0) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;
    var modulus = 0U;

    // the dividend should be bigger
    if (dividendBits.Length > 1 ||
        dividendBits[0] > (uint)Math.Abs(divisor)) {
        modulus = divideModOnly(dividendBits, (uint)Math.Abs(divisor));
    }
    else if (dividendBits[0] == (uint)Math.Abs(divisor)) {
        modulus = 0;
    }
    else {
        modulus = dividendBits[0];
    }

    return (int)modulus;
}

public static IntBig Mod(IntBig dividend, IntBig divisor) {
    if (object.ReferenceEquals(dividend, null)) {
        throw new ArgumentNullException("dividend");
    }
    if (object.ReferenceEquals(divisor, null)) {
        throw new ArgumentNullException("divisor");
    }
    if (divisor.IsZero) {
        throw new DivideByZeroException();
    }

    var dividendBits = dividend.innerBits;

    // the dividend should be bigger
    var diff = compare(dividendBits, divisor.innerBits);
    if (diff > 0) {
        divide(dividendBits, divisor.innerBits, out dividendBits);
    }
    else if (diff == 0) {
        dividendBits = new uint[] { 0 };
    }

    return new IntBig(dividendBits, dividend.isNegative);
```

```
}

#endregion

#region interfaces

public override bool Equals(object obj) {
    return Compare(this, obj as IntBig) == 0;
}

public bool Equals(IntBig other) {
    return Compare(this, other) == 0;
}

public int CompareTo(object obj) {
    return Compare(this, obj as IntBig);
}

public int CompareTo(IntBig other) {
    return Compare(this, other);
}

public override int GetHashCode() {
    var hash = 0U;
    for (int i = 0; i < innerBits.Length; i++) {
        hash ^= innerBits[i];
    }
    return (int)hash;
}

public static IntBig Parse(String value) {
    if (string.IsNullOrEmpty(value)) {
        throw new ArgumentNullException("value");
    }
    var isNegative = value[0] == '-';
    if (isNegative) {
        value = value.Remove(0, 1);
    }
    IntBig result = 0;
    while (value.Length >= 9) {
        var chunk = value.Substring(0, 9);
        result = (result * 1000000000) + int.Parse(chunk);
        value = value.Remove(0, 9);
    }
    for (int i = 0; i < value.Length; i++) {
        if (value[i] >= '0' && value[i] <= '9') {
            result = (result * 10) + (value[i] - '0');
        }
```

```
        else {
            throw new FormatException();
        }
    }
    if (isNegative) {
        result = Negate(result);
    }
    return result;
}

public override string ToString() {
    if (IsZero) {
        return "0";
    }
    var result = "";
    var quotient = isNegative ? Negate(this) : Plus(this);
    while (!quotient.IsZero) {
        var modulus = 0;
        quotient = Divide(quotient, 1000000000, out modulus);
        result = modulus.ToString("000000000") + result;
    }
    result = result.TrimStart('0');
    if (isNegative) {
        result = '-' + result;
    }
    return result;
}

public string ToHexString() {
    var result = "";
    for (int i = 0; i < innerBits.Length; i++) {
        result = innerBits[i].ToString("X8") + result;
    }
    return result;
}

public static IntBig FromByteArray(byte[] value) {
    if (value == null) {
        throw new ArgumentNullException("value");
    }
    var bits = new uint[(value.Length + 3) / 4];
    Buffer.BlockCopy(value, 0, bits, 0, value.Length);
    return new IntBig(bits, false);
}

public byte[] ToByteArray() {
    if (IsZero) {
        return new byte[] { 0 };
```

```
    }
    var length = innerBits.Length * 4
        - leadingZeroCount(innerBits[innerBits.Length - 1]) / 8;
    var bytes = new byte[length];
    Buffer.BlockCopy(innerBits, 0, bytes, 0, length);
    return bytes;
}

public int CountBits() {
    return innerBits.Length * 32
        - leadingZeroCount(innerBits[innerBits.Length - 1]);
}

#endregion

#region big integer algorithms

static uint[] rightShift(uint[] value, int shift) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");
    Debug.Assert(shift >= 0, "shift >= 0");

    // big shifts move entire blocks
    var fastShift = shift / 32;
    shift = shift % 32;

    // too big shifts...
    if (value.Length <= fastShift) {
        return new uint[1];
    }

    // temporary storage for shifted bits
    var bits = new uint[value.Length - fastShift];

    // shifts the bits to the right
    if (shift == 0) {
        for (int i = 0; i < bits.Length; i++) {
            bits[i] = value[i + fastShift];
        }
    }
    else {
        for (int i = 0; i < bits.Length - 1; i++) {
            bits[i] = (value[i + fastShift] >> shift)
                    | (value[i + fastShift + 1] << (32 - shift));
        }
        bits[bits.Length - 1] = value[value.Length - 1] >> shift;
    }

```

```
    return bits;
}

static uint[] leftShift(uint[] value, int shift) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");
    Debug.Assert(shift >= 0, "shift >= 0");

    // big shifts move entire blocks
    var fastShift = shift / 32;
    shift = shift % 32;

    // temporary storage for shifted bits
    var bits = new uint[value.Length + fastShift + 1];

    // shifts the bits to the left
    if (shift == 0) {
        for (int i = fastShift; i < bits.Length - 1; i++) {
            bits[i] = value[i - fastShift];
        }
    }
    else {
        for (int i = fastShift + 1; i < bits.Length - 1; i++) {
            bits[i] = (value[i - fastShift] << shift)
                    | (value[i - fastShift - 1] >> (32 - shift));
        }
        bits[fastShift] = value[0] << shift;
        bits[bits.Length - 1] = value[value.Length - 1] >> (32 - shift);
    }

    return bits;
}

static int compare(uint[] left, uint[] right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");

    // different length
    if (left.Length < right.Length) {
        return -1;
    }
    if (left.Length > right.Length) {
        return 1;
    }

    // check the bit blocks
```

```
    for (int i = left.Length - 1; i >= 0; i--) {
        if (left[i] < right[i]) {
            return -1;
        }
        if (left[i] > right[i]) {
            return 1;
        }
    }

    return 0;
}

static uint[] add(uint[] left, uint right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");

    // temporary storage for added bits
    var bits = new uint[left.Length + 1];

    // first operation
    var digit = (long)left[0] + right;
    bits[0] = (uint)digit;
    var carry = digit >> 32;

    // adds the bits
    for (int i = 1; i < left.Length; i++) {
        digit = left[i] + carry;
        bits[i] = (uint)digit;
        carry = digit >> 32;
    }
    bits[bits.Length - 1] = (uint)carry;

    return bits;
}

static uint[] add(uint[] left, uint[] right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");
    Debug.Assert(left.Length >= right.Length);

    // temporary storage for added bits
    var bits = new uint[left.Length + 1];
    var carry = 0L;

    // adds the bits
    for (int i = 0; i < right.Length; i++) {
```

```
            var digit = (left[i] + carry) + right[i];
            bits[i] = (uint)digit;
            carry = digit >> 32;
        }
        for (int i = right.Length; i < left.Length; i++) {
            var digit = left[i] + carry;
            bits[i] = (uint)digit;
            carry = digit >> 32;
        }
        bits[bits.Length - 1] = (uint)carry;

        return bits;
}

static uint[] subtract(uint[] left, uint right) {
        Debug.Assert(left != null, "left != null");
        Debug.Assert(left.Length > 0, "left.Length > 0");
        Debug.Assert(left.Length > 1 || left[0] >= right);

        // temporary storage for subtracted bits
        var bits = new uint[left.Length];

        // first operation
        var digit = (long)left[0] - right;
        bits[0] = (uint)digit;
        var carry = digit >> 63;

        // substract the bits
        for (int i = 1; i < left.Length; i++) {
            digit = left[i] + carry;
            bits[i] = (uint)digit;
            carry = digit >> 63;
        }

        return bits;
}

static uint[] subtract(uint[] left, uint[] right) {
        Debug.Assert(left != null, "left != null");
        Debug.Assert(left.Length > 0, "left.Length > 0");
        Debug.Assert(right != null, "right != null");
        Debug.Assert(right.Length > 0, "right.Length > 0");

        // temporary storage for subtracted bits
        var bits = new uint[left.Length];
        var carry = 0L;

        // substract the bits
```

```
    for (int i = 0; i < right.Length; i++) {
        var digit = (left[i] + carry) - right[i];
        bits[i] = (uint)digit; // equal to digit + 0x100000000 if digit < 0
        carry = digit >> 63; // equal to -1 if digit < 0, to 0 otherwise
    }
    for (int i = right.Length; i < left.Length; i++) {
        var digit = left[i] + carry;
        bits[i] = (uint)digit;
        carry = digit >> 63;
    }

    return bits;
}

static unsafe uint[] square(uint[] value) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");

    // switch to fast pointer arithmetic
    fixed (uint* v = value) {
        return square(v, value.Length);
    }
}

static unsafe uint[] square(uint* value, int valueLength) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(valueLength > 0, "valueLength > 0");

    // temporary storage for squared bits
    var bits = new uint[valueLength * 2];

    if (valueLength < ThresholdSqu1) {
        // squares the bits
        for (int i = 0; i < valueLength; i++) {
            var carry = 0UL;
            for (int j = 0; j < i; j++) {
                var digits1 = bits[i + j] + carry;
                var digits2 = (ulong)value[j] * (ulong)value[i];
                bits[i + j] = (uint)(digits1 + (digits2 << 1));
                carry = (digits2 + (digits1 >> 1)) >> 31;
            }
            var digits = (ulong)value[i] * (ulong)value[i] + carry;
            bits[i * 2] = (uint)digits;
            bits[i * 2 + 1] = (uint)(digits >> 32);
        }
    }
    else if (valueLength < ThresholdSqu2) {
        // divide & conquer (using Karatsuba's algorithm)
```

```
            var n = (valueLength + 1) / 2;

            var valueLow = value; var valueHigh = value + n;

            var valueLowLength = n; // n < valueLength (!)
            var valueHighLength = valueLength - valueLowLength;

            var p1 = square(valueHigh, valueHighLength);
            var p2 = square(valueLow, valueLowLength);
            var p3 = square(add(valueLow, valueLowLength,
                                valueHigh, valueHighLength));

            subtractInplace(p3, p1);
            subtractInplace(p3, p2);

            // merge the result
            addInplaceWithFastShift(bits, p2, 0);
            addInplaceWithFastShift(bits, p3, n);
            addInplaceWithFastShift(bits, p1, n * 2);
        }
        else {
            // divide & conquer (using more CPUs)
            var n = (valueLength + 1) / 2;

            var valueLow = value; var valueHigh = value + n;

            var valueLowLength = n; // n < valueLength (!)
            var valueHighLength = valueLength - valueLowLength;

            var t1 = Task.Factory.StartNew<uint[]>(
                () => square(valueHigh, valueHighLength));
            var t2 = Task.Factory.StartNew<uint[]>(
                () => square(valueLow, valueLowLength));
            var p3 = square(add(valueLow, valueLowLength,
                                valueHigh, valueHighLength));
            var p1 = t1.Result;
            var p2 = t2.Result;

            subtractInplace(p3, p1);
            subtractInplace(p3, p2);

            // merge the result
            addInplaceWithFastShift(bits, p2, 0);
            addInplaceWithFastShift(bits, p3, n);
            addInplaceWithFastShift(bits, p1, n * 2);
        }

        return bits;
```

```
}

static uint[] multiply(uint[] left, uint right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");

    // temporary storage for multiplied bits
    var bits = new uint[left.Length + 1];

    // multiplies the bits
    var carry = 0UL;
    for (int j = 0; j < left.Length; j++) {
        var digits = (ulong)left[j] * right + carry;
        bits[j] = (uint)digits;
        carry = digits >> 32;
    }
    bits[left.Length] = (uint)carry;

    return bits;
}

static unsafe uint[] multiply(uint[] left, uint[] right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");

    // switch to fast pointer arithmetic
    fixed (uint* l = left, r = right) {
        return multiply(l, left.Length, r, right.Length);
    }
}

static unsafe uint[] multiply(uint *left, int leftLength,
                              uint *right, int rightLength) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(leftLength >= 0, "leftLength >= 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(rightLength >= 0, "rightLength >= 0");

    // temporary storage for multiplied bits
    var bits = new uint[leftLength + rightLength];

    if (leftLength < ThresholdMul1 || rightLength < ThresholdMul1) {
        // multiplies the bits
        for (int i = 0; i < rightLength; i++) {
            var carry = 0UL;
            for (int j = 0; j < leftLength; j++) {
```

```
                var digits = bits[i + j] + carry
                    + (ulong)left[j] * (ulong)right[i];
                bits[i + j] = (uint)digits;
                carry = digits >> 32;
            }
            bits[i + leftLength] = (uint)carry;
        }
    }
    else if (leftLength < ThresholdMul2 || rightLength < ThresholdMul2) {
        // divide & conquer (using Karatsuba's algorithm)
        var n = (leftLength > rightLength) ? leftLength : rightLength;
        n = (n + 1) / 2;

        var leftLow = left; var leftHigh = left + n;
        var rightLow = right; var rightHigh = right + n;

        var leftLowLength = (n < leftLength) ? n : leftLength;
        var leftHighLength = leftLength - leftLowLength;
        var rightLowLength = (n < rightLength) ? n : rightLength;
        var rightHighLength = rightLength - rightLowLength;

        var p1 = multiply(leftHigh, leftHighLength,
                          rightHigh, rightHighLength);
        var p2 = multiply(leftLow, leftLowLength,
                          rightLow, rightLowLength);
        var p3 = multiply(add(leftLow, leftLowLength,
                              leftHigh, leftHighLength),
                          add(rightLow, rightLowLength,
                              rightHigh, rightHighLength));

        subtractInplace(p3, p1);
        subtractInplace(p3, p2);

        // merge the result
        addInplaceWithFastShift(bits, p2, 0);
        addInplaceWithFastShift(bits, p3, n);
        addInplaceWithFastShift(bits, p1, n * 2);
    }
    else {
        // divide & conquer (using more CPUs)
        var n = (leftLength > rightLength) ? leftLength : rightLength;
        n = (n + 1) / 2;

        var leftLow = left; var leftHigh = left + n;
        var rightLow = right; var rightHigh = right + n;

        var leftLowLength = (n < leftLength) ? n : leftLength;
        var leftHighLength = leftLength - leftLowLength;
```

```
        var rightLowLength = (n < rightLength) ? n : rightLength;
        var rightHighLength = rightLength - rightLowLength;

        var t1 = Task.Factory.StartNew<uint[]>(
            () => multiply(leftHigh, leftHighLength,
                           rightHigh, rightHighLength));
        var t2 = Task.Factory.StartNew<uint[]>(
            () => multiply(leftLow, leftLowLength,
                           rightLow, rightLowLength));
        var p3 = multiply(add(leftLow, leftLowLength,
                              leftHigh, leftHighLength),
                          add(rightLow, rightLowLength,
                              rightHigh, rightHighLength));
        var p1 = t1.Result;
        var p2 = t2.Result;

        subtractInplace(p3, p1);
        subtractInplace(p3, p2);

        // merge the result
        addInplaceWithFastShift(bits, p2, 0);
        addInplaceWithFastShift(bits, p3, n);
        addInplaceWithFastShift(bits, p1, n * 2);
    }

    return bits;
}

static unsafe uint[] add(uint* left, int leftLength,
                         uint* right, int rightLength) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(leftLength > 0, "leftLength > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(rightLength >= 0, "rightLength >= 0");

    // temporary storage for added bits
    var bits = new uint[leftLength + 1];
    var carry = 0L;

    // adds the bits
    for (int i = 0; i < rightLength; i++) {
        var digit = (left[i] + carry) + right[i];
        bits[i] = (uint)digit;
        carry = digit >> 32;
    }
    for (int i = rightLength; i < leftLength; i++) {
        var digit = left[i] + carry;
        bits[i] = (uint)digit;
```

```
        carry = digit >> 32;
    }
    bits[bits.Length - 1] = (uint)carry;

    return bits;
}

static void subtractInplace(uint[] left, uint[] right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");

    // we'll overwrite left...
    var carry = 0L;

    // substract the bits
    for (int i = 0; i < right.Length; i++) {
        var digit = (left[i] + carry) - right[i];
        left[i] = (uint)digit;
        carry = digit >> 63;
    }
    for (int i = right.Length; i < left.Length; i++) {
        var digit = left[i] + carry;
        left[i] = (uint)digit;
        carry = digit >> 63;
    }
}

static void addInplaceWithFastShift(uint[] left, uint[] right,
                                    int rightShift) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");
    Debug.Assert(rightShift >= 0, "rightShift >= 0");

    // assuming leading zeros on right...
    var shiftedRightLength = right.Length + rightShift;
    if (shiftedRightLength > left.Length) {
        shiftedRightLength = left.Length;
    }

    // we'll overwrite left...
    var carry = 0L;

    // adds the bits
    int i = rightShift;
```

```
    for (; i < shiftedRightLength; i++) {
        var digit = (left[i] + carry) + right[i - rightShift];
        left[i] = (uint)digit;
        carry = digit >> 32;
    }
    if (i < left.Length) {
        var digit = left[i] + carry;
        left[i] = (uint)digit;
        // assuming this is it...
    }
}

static uint[] divide(uint[] dividend, uint divisor, out uint modulus) {
    Debug.Assert(dividend != null, "dividend != null");
    Debug.Assert(dividend.Length > 0, "dividend.Length > 0");
    Debug.Assert(divisor != 0, "divisor != 0");
    Debug.Assert(dividend.Length > 1 || dividend[0] > divisor);

    // temporary storage for divided bits
    var bits = new uint[dividend.Length];

    // divides the bits
    var carry = 0UL;
    for (var i = dividend.Length - 1; i >= 0; i--) {
        var value = (carry << 32) | dividend[i];
        bits[i] = (uint)(value / divisor);
        carry = value % divisor;
    }

    modulus = (uint)carry;

    return bits;
}

static uint divideModOnly(uint[] dividend, uint divisor) {
    Debug.Assert(dividend != null, "dividend != null");
    Debug.Assert(dividend.Length > 0, "dividend.Length > 0");
    Debug.Assert(divisor != 0, "divisor != 0");
    Debug.Assert(dividend.Length > 1 || dividend[0] > divisor);

    // divides the bits
    var carry = 0UL;
    for (var i = dividend.Length - 1; i >= 0; i--) {
        var value = (carry << 32) | dividend[i];
        carry = value % divisor;
    }

    return (uint)carry;
```

```
}

static uint[] divide(uint[] dividend, uint[] divisor, out uint[] modulus) {
    Debug.Assert(dividend != null, "dividend != null");
    Debug.Assert(dividend.Length > 0, "dividend.Length > 0");
    Debug.Assert(divisor != null, "divisor != null");
    Debug.Assert(divisor.Length > 0, "divisor.Length > 0");
    Debug.Assert(compare(dividend, divisor) > 0, "");

    // temporary storage for divided bits
    var bits = new uint[dividend.Length - divisor.Length + 1];

    // get more bits into the highest bit block
    var shifted = leadingZeroCount(divisor[divisor.Length - 1]);
    dividend = leftShift(dividend, shifted);
    divisor = leftShift(divisor, shifted);

    // useful values
    var divisorLength = divisor.Length - 1; // i know that!
    var dividendLength = dividend[dividend.Length - 1] == 0 ?
        dividend.Length - 1 : dividend.Length;
    var divHigh = divisor[divisorLength - 1];

    // these values are useful too :-)
    var guessedDivisor = new uint[divisor.Length];
    var guessedDivisorLength = 0;

    // sub the divisor (shifted to the left, so they have equal length)
    var diff = compareWithFastShift(dividend, dividendLength,
        divisor, divisorLength, dividendLength - divisorLength);
    while (diff >= 0) {
        ++bits[dividendLength - divisorLength];
        subtractInplaceWithFastShift(dividend, dividendLength,
            divisor, divisorLength, dividendLength - divisorLength);
        dividendLength = actualLength(dividend, dividendLength);
        diff = compareWithFastShift(dividend, dividendLength,
            divisor, divisorLength, dividendLength - divisorLength);
    }

    // divides the rest of the bits
    var i = dividendLength - 1;
    while (i >= divisorLength) {
        // first guess for the current bit of the quotient
        var guess = 0xffffffff;
        if (dividend[i] != divHigh) {
            guess = (uint)((dividend[i - 1] + ((ulong)dividend[i] << 32))
                            / divHigh);
        }
```

```
            // the guess may be a little bit to big
            multiplyDivisor(divisor, guess, guessedDivisor);
            guessedDivisorLength =
                guessedDivisor[guessedDivisor.Length - 1] == 0 ?
                guessedDivisor.Length - 1 : guessedDivisor.Length;
            diff = compareWithFastShift(dividend, dividendLength,
                guessedDivisor, guessedDivisorLength,
                i - divisorLength);
            while (diff < 0) {
                --guess;
                multiplyDivisor(divisor, guess, guessedDivisor);
                guessedDivisorLength =
                    guessedDivisor[guessedDivisor.Length - 1] == 0 ?
                    guessedDivisor.Length - 1 : guessedDivisor.Length;
                diff = compareWithFastShift(dividend, dividendLength,
                    guessedDivisor, guessedDivisorLength,
                    i - divisorLength);
            }
            // we have the bit!
            subtractInplaceWithFastShift(dividend, dividendLength,
                guessedDivisor, guessedDivisorLength,
                i - divisorLength);
            dividendLength = actualLength(dividend, dividendLength);
            bits[i - divisorLength] = guess;
            i = dividendLength - 1;
        }

        // repair the cheated shift
        modulus = rightShift(dividend, shifted);

        return bits;
    }

    static uint[] multiplyDivisor(uint[] left, uint right, uint[] bits) {
        Debug.Assert(left != null, "left != null");
        Debug.Assert(left.Length > 0, "left.Length > 0");
        Debug.Assert(bits != null, "bits != null");
        Debug.Assert(bits.Length == left.Length);

        // multiplies the bits
        var carry = 0UL;
        for (int j = 0; j < left.Length - 1; j++) {
            var digits = (ulong)left[j] * right + carry;
            bits[j] = (uint)digits;
            carry = digits >> 32;
        }
        bits[bits.Length - 1] = (uint)carry;

```

```
    return bits;
}

static int compareWithFastShift(uint[] left, int leftLength,
                                uint[] right, int rightLength,
                                int shift) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");
    Debug.Assert(shift >= 0, "shift >= 0");
    Debug.Assert(actualLength(left) == leftLength,
                 "actualLength(left) == leftLength");
    Debug.Assert(actualLength(right) == rightLength,
                 "actualLength(right) == rightLength");

    if (leftLength < rightLength + shift) {
        return -1;
    }
    if (leftLength > rightLength + shift) {
        return 1;
    }
    for (int i = leftLength - 1; i >= shift; i--) {
        if (left[i] < right[i - shift]) {
            return -1;
        }
        if (left[i] > right[i - shift]) {
            return 1;
        }
    }
    for (int i = shift - 1; i >= 0; i--) {
        if (left[i] < 0) {
            return -1;
        }
        if (left[i] > 0) {
            return 1;
        }
    }
    return 0;
}

static void subtractInplaceWithFastShift(uint[] left, int leftLength,
                                         uint[] right, int rightLength,
                                         int shift) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");
```

```
    Debug.Assert(shift >= 0, "shift >= 0");
    Debug.Assert(actualLength(left) == leftLength,
                 "actualLength(left) == leftLength");
    Debug.Assert(actualLength(right) == rightLength,
                 "actualLength(right) == rightLength");

    var carry = 0L;
    for (int i = shift; i < rightLength + shift; i++) {
        var digit = (left[i] + carry) - right[i - shift];
        left[i] = (uint)digit;
        carry = digit >> 63;
    }
    for (int i = rightLength + shift; i < leftLength; i++) {
        var digit = left[i] + carry;
        left[i] = (uint)digit;
        carry = digit >> 63;
    }
}

static int actualLength(uint[] value) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");

    var length = value.Length;
    while (length > 1 && value[length - 1] == 0) {
        --length;
    }
    return length;
}

static int actualLength(uint[] value, int length) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");
    Debug.Assert(actualLength(value) <= length,
                 "actualLength(value) <= length");

    while (length > 1 && value[length - 1] == 0) {
        --length;
    }
    return length;
}

static int leadingZeroCount(uint value) {
    if (value == 0) {
        return 32;
    }
    int count = 0;
    if ((value & 0xffff0000) == 0) {
```

```
            count += 16;
            value = value << 16;
        }
        if ((value & 0xff000000) == 0) {
            count += 8;
            value = value << 8;
        }
        if ((value & 0xf0000000) == 0) {
            count += 4;
            value = value << 4;
        }
        if ((value & 0xc0000000) == 0) {
            count += 2;
            value = value << 2;
        }
        if ((value & 0x80000000) == 0) {
            count += 1;
        }
        return count;
    }

    #endregion

    #region montgomery / barret helper

    internal static IntBig BeginMontgomery(IntBig mod) {
        Debug.Assert(mod != null, "mod != null");

        var bits = new uint[mod.innerBits.Length + 1];
        bits[bits.Length - 1] = 1;
        return new IntBig(bits, false);
    }

    internal static IntBig Montgomery(IntBig value, IntBig mod, IntBig inv) {
        Debug.Assert(value != null, "value != null");
        Debug.Assert(mod != null, "mod != null");
        Debug.Assert(inv != null, "inv != null");

        var v = value.innerBits;
        var m = mod.innerBits;
        var i = inv.innerBits;

        var r = fastMod(multiply(fastMod(v, m.Length), i), m.Length);
        r = fastDiv(add(multiply(m, r), v), m.Length);

        while (fastCompare(r, m) >= 0) {
            r = subtract(r, m);
        }
```

```
    return new IntBig(r, false);
}

internal static IntBig BeginBarrett(IntBig mod) {
    Debug.Assert(mod != null, "mod != null");

    var bits = new uint[mod.innerBits.Length * 2 + 1];
    bits[bits.Length - 1] = 1;
    return new IntBig(bits, false);
}

internal static IntBig Barrett(IntBig value, IntBig mod, IntBig mu) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(mod != null, "mod != null");
    Debug.Assert(mu != null, "mu != null");

    var v = value.innerBits;
    var m = mod.innerBits;
    var u = mu.innerBits;

    var q = fastDiv(v, m.Length - 1);
    q = fastDiv(multiply(q, u), m.Length + 1);

    var r1 = fastMod(v, m.Length + 1);
    var r2 = fastMod(multiply(q, m), m.Length + 1);
    var r = subtract(r1, r2);

    while (fastCompare(r, m) >= 0) {
        r = subtract(r, m);
    }

    return new IntBig(r, false);
}

static int fastCompare(uint[] left, uint[] right) {
    Debug.Assert(left != null, "left != null");
    Debug.Assert(left.Length > 0, "left.Length > 0");
    Debug.Assert(right != null, "right != null");
    Debug.Assert(right.Length > 0, "right.Length > 0");

    // left is bigger than right
    for (int i = left.Length - 1; i >= right.Length; i--) {
        if (left[i] != 0) {
            return 1;
        }
    }

```

```
    // check the bit blocks
    for (int i = right.Length - 1; i >= 0; i--) {
        if (left[i] < right[i]) {
            return -1;
        }
        if (left[i] > right[i]) {
            return 1;
        }
    }

    return 0;
}

static uint[] fastDiv(uint[] value, int count) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");
    Debug.Assert(count >= 0, "count >= 0");

    if (count == 0) {
        return value;
    }
    if (value.Length <= count) {
        return new uint[] { 0 };
    }
    var bits = new uint[value.Length - count];
    Buffer.BlockCopy(value, count * 4, bits, 0, bits.Length * 4);
    return bits;
}

static uint[] fastMod(uint[] value, int count) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length > 0, "value.Length > 0");
    Debug.Assert(count >= 0, "count >= 0");

    var bits = new uint[count];
    if (value.Length <= count) {
        Buffer.BlockCopy(value, 0, bits, 0, value.Length * 4);
    }
    else {
        Buffer.BlockCopy(value, 0, bits, 0, bits.Length * 4);
    }
    return bits;
}

#endregion
```

A.2. MathBig

```
static MathBig() {
    CurrentPowStrategy = PowBigStrategy.Barrett;
}

public static IntBig Abs(IntBig value) {
    if (value == null) {
        throw new ArgumentNullException("value");
    }
    return value.IsNegative ? -value : +value;
}

public static IntBig Pow(IntBig value, IntBig power) {
    if (value == null) {
        throw new ArgumentNullException("value");
    }
    if (power == null) {
        throw new ArgumentNullException("power");
    }
    if (power.IsNegative) {
        throw new ArgumentOutOfRangeException("power");
    }

    IntBig result = 1;

    while (!power.IsZero) {
        if (power.IsOdd) {
            result = value * result;
        }
        value = value * value;
        power = power >> 1;
    }

    return result;
}

public static PowBigStrategy CurrentPowStrategy { get; set; }

static IntBig PowClassic(IntBig value, IntBig power, IntBig mod) {
    if (value == null) {
        throw new ArgumentNullException("value");
    }
    if (power == null) {
        throw new ArgumentNullException("power");
    }
    if (mod == null) {
```

```
            throw new ArgumentNullException("mod");
        }
        if (value.IsNegative) {
            throw new ArgumentOutOfRangeException("value");
        }
        if (power.IsNegative) {
            throw new ArgumentOutOfRangeException("power");
        }
        if (mod.IsNegative || mod.IsZero || mod.IsOne || !mod.IsOdd) {
            throw new ArgumentOutOfRangeException("mod");
        }

        // do some preprocessing...
        var v = new IntBig[256];
        v[0] = 1;
        v[1] = value % mod;
        v[2] = (v[1] * v[1]) % mod;

        for (var j = 4; j < v.Length; j *= 2) {
            v[j] = (v[j / 2] * v[j / 2]) % mod;
        }
        for (var i = 3; i < v.Length; i += 2) {
            v[i] = (v[i - 1] * value) % mod;
            for (var j = i * 2; j < v.Length; j *= 2) {
                v[j] = (v[j / 2] * v[j / 2]) % mod;
            }
        }

        var p = power.ToByteArray();

        IntBig result = v[p[p.Length - 1]];

        for (var i = p.Length - 2; i >= 0; i--) {
            for (var j = 0; j < 8; j++) {
                result = (result * result) % mod;
            }
            result = (result * v[p[i]]) % mod;
        }

        return result;
    }

    static IntBig PowMontgomery(IntBig value, IntBig power, IntBig mod) {
        if (value == null) {
            throw new ArgumentNullException("value");
        }
        if (power == null) {
            throw new ArgumentNullException("power");
```

```
    }
    if (mod == null) {
        throw new ArgumentNullException("mod");
    }
    if (value.IsNegative) {
        throw new ArgumentOutOfRangeException("value");
    }
    if (power.IsNegative) {
        throw new ArgumentOutOfRangeException("power");
    }
    if (mod.IsNegative || mod.IsZero || mod.IsOne || !mod.IsOdd) {
        throw new ArgumentOutOfRangeException("mod");
    }

    var R = IntBig.BeginMontgomery(mod);
    var inv = R - Inv(mod, R);

    // do some preprocessing...
    var v = new IntBig[256];
    v[0] = R % mod;
    v[1] = (value << (R.CountBits() - 1)) % mod;
    v[2] = IntBig.Montgomery(v[1] * v[1], mod, inv);

    for (var j = 4; j < v.Length; j *= 2) {
        v[j] = IntBig.Montgomery(v[j / 2] * v[j / 2], mod, inv);
    }
    for (var i = 3; i < v.Length; i += 2) {
        v[i] = IntBig.Montgomery(v[i - 1] * v[1], mod, inv);
        for (var j = i * 2; j < v.Length; j *= 2) {
            v[j] = IntBig.Montgomery(v[j / 2] * v[j / 2], mod, inv);
        }
    }

    var p = power.ToByteArray();

    IntBig result = v[p[p.Length - 1]];
    for (var i = p.Length - 2; i >= 0; i--) {
        for (var j = 0; j < 8; j++) {
            result = IntBig.Montgomery(result * result, mod, inv);
        }
        result = IntBig.Montgomery(result * v[p[i]], mod, inv);
    }
    result = IntBig.Montgomery(result, mod, inv);

    return result;
}

static IntBig PowBarrett(IntBig value, IntBig power, IntBig mod) {
```

```
    if (value == null) {
        throw new ArgumentNullException("value");
    }
    if (power == null) {
        throw new ArgumentNullException("power");
    }
    if (mod == null) {
        throw new ArgumentNullException("mod");
    }
    if (value.IsNegative) {
        throw new ArgumentOutOfRangeException("value");
    }
    if (power.IsNegative) {
        throw new ArgumentOutOfRangeException("power");
    }
    if (mod.IsNegative || mod.IsZero || mod.IsOne || !mod.IsOdd) {
        throw new ArgumentOutOfRangeException("mod");
    }

    var R = IntBig.BeginBarrett(mod);
    var mu = R / mod;

    // do some preprocessing...
    var v = new IntBig[256];
    v[0] = 1;
    v[1] = IntBig.Barrett(value, mod, mu);
    v[2] = IntBig.Barrett(v[1] * v[1], mod, mu);

    for (var j = 4; j < v.Length; j *= 2) {
        v[j] = IntBig.Barrett(v[j / 2] * v[j / 2], mod, mu);
    }
    for (var i = 3; i < v.Length; i += 2) {
        v[i] = IntBig.Barrett(v[i - 1] * v[1], mod, mu);
        for (var j = i * 2; j < v.Length; j *= 2) {
            v[j] = IntBig.Barrett(v[j / 2] * v[j / 2], mod, mu);
        }
    }

    var p = power.ToByteArray();

    IntBig result = v[p[p.Length - 1]];
    for (var i = p.Length - 2; i >= 0; i--) {
        for (var j = 0; j < 8; j++) {
            result = IntBig.Barrett(result * result, mod, mu);
        }
        result = IntBig.Barrett(result * v[p[i]], mod, mu);
    }

```

```
    return result;
}

public static IntBig Pow(IntBig value, IntBig power, IntBig mod) {
    switch (CurrentPowStrategy) {
        case PowBigStrategy.Classic:
            return PowClassic(value, power, mod);
        case PowBigStrategy.Montgomery:
            return PowMontgomery(value, power, mod);
        case PowBigStrategy.Barrett:
            return PowBarrett(value, power, mod);
        default:
            throw new NotImplementedException();
    }
}

public static IntBig Gcd(IntBig left, IntBig right) {
    if (left == null) {
        throw new ArgumentNullException("left");
    }
    if (right == null) {
        throw new ArgumentNullException("right");
    }
    if (left.IsZero && right.IsZero) {
        throw new InvalidOperationException();
    }

    var a = Abs(left);
    var b = Abs(right);

    while (!b.IsZero) {
        var c = a % b;
        a = b;
        b = c;
    }

    return a;
}

public static IntBig Lcm(IntBig left, IntBig right) {
    if (left == null) {
        throw new ArgumentNullException("left");
    }
    if (right == null) {
        throw new ArgumentNullException("right");
    }
    if (left.IsZero && right.IsZero) {
        throw new InvalidOperationException();
```

```
    }

    var ab = Abs(left * right);

    return ab / Gcd(left, right);
}

public static IntBig Inv(IntBig value, IntBig mod) {
    if (value == null) {
        throw new ArgumentNullException("value");
    }
    if (mod == null) {
        throw new ArgumentNullException("mod");
    }
    if (mod.IsNegative || mod.IsZero || mod.IsOne) {
        throw new ArgumentOutOfRangeException("mod");
    }
    if (value.IsNegative || value.IsZero || value >= mod) {
        throw new ArgumentOutOfRangeException("value");
    }

    var a = Abs(mod);
    var b = Abs(value);

 // IntBig x2 = 1, x1 = 0;
    IntBig y2 = 0, y1 = 1;

    while (!b.IsZero) {
        var q = a / b;
        var r = a - q * b;
     // var x = x2 - q * x1;
        var y = y2 - q * y1;

        a = b; b = r;
     // x2 = x1; x1 = x;
        y2 = y1; y1 = y;
    }

    if (!a.IsOne) {
        throw new ArgumentOutOfRangeException("value");
    }

    if (y2.IsNegative) {
        y2 = y2 + Abs(mod);
    }

    return y2;
}
```

```
public static IntBig ToomCook(IntBig left, IntBig right, int baseBits) {
    if (left == null) {
        throw new ArgumentNullException("left");
    }
    if (right == null) {
        throw new ArgumentNullException("right");
    }
    if (left.IsNegative || left.IsZero) {
        throw new ArgumentOutOfRangeException("left");
    }
    if (right.IsNegative || right.IsZero) {
        throw new ArgumentOutOfRangeException("right");
    }
    if (baseBits <= 0) {
        throw new ArgumentOutOfRangeException("bits");
    }

    // split left and right
    var l = ToomCookSplit(left, baseBits);
    var r = ToomCookSplit(right, baseBits);

    // evaluate sampling points
    var L = ToomCookEval(l, l.Length + r.Length - 1);
    var R = ToomCookEval(r, l.Length + r.Length - 1);

    // multiply sampling points
    var P = ToomCookProduct(L, R);

    // interpolate sampling points
    var Q = ToomCookInterpolate(P);

    // transform to "ordinary" representation
    var S = ToomCookTransform(Q);

    // join coefficients
    return ToomCookJoin(S, baseBits);
}

private static IntBig ToomCookJoin(IntBig[] value, int baseBits) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(baseBits > 0, "baseBits > 0");

    var r = value[0];

    for (var i = 1; i < value.Length; i++) {
        r += (value[i] << (baseBits * i));
    }
```

```

    return r;
}

private static IntBig[] ToomCookTransform(IntBig[] value) {
    Debug.Assert(value != null, "value != null");

    var s = new IntBig[value.Length];
    for (var i = 0; i < s.Length; i++) {
        s[i] = value[i];
    }

    for (var i = s.Length - 2; i > 0; i--) {
        for (var j = i; j < s.Length - 1; j++) {
            s[j] = (s[j] - s[j + 1] * i);
        }
    }

    return s;
}

private static IntBig[] ToomCookInterpolate(IntBig[] value) {
    Debug.Assert(value != null, "value != null");

    var q = new IntBig[value.Length];
    q[0] = value[0];

    for (var i = 1; i < q.Length; i++) {
        var h = new IntBig[value.Length - 1];
        for (var j = 0; j < h.Length; j++) {
            h[j] = (value[j + 1] - value[j]) / i;
        }
        value = h;
        q[i] = value[0];
    }

    return q;
}

private static IntBig[] ToomCookProduct(IntBig[] left, IntBig[] right) {
    Debug.Assert(left != null, "right != null");
    Debug.Assert(left != null, "right != null");
    Debug.Assert(left.Length == right.Length);

    var p = new IntBig[left.Length];

    for (var i = 0; i < p.Length; i++) {
        p[i] = left[i] * right[i];
```

```
    }

    return p;
}

private static IntBig[] ToomCookEval(IntBig[] value, int pointCount) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(pointCount > 0, "pointCount > 0");

    var v = new IntBig[pointCount];

    for (var i = 0; i < v.Length; i++) {
        v[i] = 0;
        for (var j = value.Length - 1; j >= 0; j--) {
            v[i] = (v[i] * i + value[j]);
        }
    }

    return v;
}

private static IntBig[] ToomCookSplit(IntBig value, int baseBits) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value > 0, "value > 0");
    Debug.Assert(baseBits > 0, "baseBits > 0");

    var v = new IntBig[(value.CountBits() + baseBits - 1) / baseBits];

    for (var i = 0; i < v.Length; i++) {
        var shifted = value >> baseBits;
        v[i] = value - (shifted << baseBits);
        value = shifted;
    }

    return v;
}
```

A.3. PrimeBig

```
RandomBig generator = new RandomBig();

const int SmallPrimesThreshold = 100000;

static PrimeBig() {
    initSmallPrimes();
}

public IntBig NextProbablePrime(int bitCount) {
    if (bitCount < 2) {
        throw new ArgumentOutOfRangeException("bitCount");
    }
    var result = generator.NextBits(bitCount, true, true);
    while (!IsProbablePrime(result)) {
        result = generator.NextBits(bitCount, true, true);
    }
    return result;
}

public IntBig NextProbableStrongPrime(int bitCount) {
    IntBig r;
    return NextProbableStrongPrime(bitCount, out r);
}

[SuppressMessage("Microsoft.Design", "CA1021:AvoidOutParameters")]
public IntBig NextProbableStrongPrime(int bitCount, out IntBig factor) {
    if (bitCount < 128) {
        throw new ArgumentOutOfRangeException("bitCount");
    }

    var s = NextProbablePrime(bitCount / 2 - 32);
    var t = NextProbablePrime(bitCount / 2 - 32);

    var t2 = t << 1;
    var r = (t << 32) + 1;

    while (!IsProbablePrime(r)) {
        r = r + t2;
    }

    var rs2 = (r * s) << 1;
    var p = ((MathBig.Pow(s, r - 2, r) * s) << 1) - 1;

    p = p + (rs2 << (bitCount - rs2.CountBits()));

```

```
    while (!IsProbablePrime(p)) {
        p = p + rs2;
    }

    factor = r;
    return p;
}

public bool IsProbablePrime(IntBig value) {
    if (value == null) {
        throw new ArgumentNullException("value");
    }

    var bits = value.CountBits();

    if (bits < 128) {
        return isProbablePrime(value, 100);
    }
    if (bits < 256) {
        return isProbablePrime(value, 27);
    }
    if (bits < 512) {
        return isProbablePrime(value, 12);
    }
    if (bits < 768) {
        return isProbablePrime(value, 6);
    }
    if (bits < 1024) {
        return isProbablePrime(value, 4);
    }
    if (bits < 2048) {
        return isProbablePrime(value, 3);
    }

    return isProbablePrime(value, 2);
}

#region interfaces

public void Dispose() {
    generator.Dispose();
}

#endregion

#region algorithms for primes

static int[] smallPrimes;
```

```
static void initSmallPrimes() {
    var sieve = new BitArray(SmallPrimesThreshold);

    for (var i = 2; i * i < SmallPrimesThreshold; i++) {
        if (!sieve[i]) {
            for (var j = i * i; j < SmallPrimesThreshold; j += i) {
                sieve[j] = true;
            }
        }
    }

    var result = new List<int>();

    for (var i = 3; i < SmallPrimesThreshold; i++) {
        if (!sieve[i]) {
            result.Add(i);
        }
    }

    smallPrimes = result.ToArray();
}

bool isProbablePrime(IntBig value, int tryCount) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(tryCount > 0, "tryCount > 0");

    if (value.IsNegative || value.IsZero || value.IsOne) {
        return false;
    }
    if (!value.IsOdd) {
        return false;
    }

    // just search for small values...
    if (value < SmallPrimesThreshold) {
        return (Array.BinarySearch<int>(smallPrimes, (int)value) >= 0);
    }

    // do fast trial divisions for small factors...
    for (int i = 0; i < smallPrimes.Length; i++) {
        if ((value % smallPrimes[i]) == 0) {
            return false;
        }
    }

    // preprocessing for rabin-miller
    var s = 0;
```

```
    IntBig r = value - 1;

    while (!r.IsOdd) {
        ++s;
        r = r >> 1;
    }

    // t times with a big base
    for (int i = 0; i < tryCount; i++) {
        IntBig a = generator.Next(value - 3) + 2;
        IntBig y = MathBig.Pow(a, r, value);
        if (!y.IsOne && !(value - y).IsOne) {
            var j = 1;
            while (j < s && !(value - y).IsOne) {
                y = (y * y) % value;
                if (y.IsOne) {
                    return false;
                }
                ++j;
            }
            if (!(value - y).IsOne) {
                return false;
            }
        }
    }

    return true;
}

#endregion
```

A.4. RandomBig

```
RNGCryptoServiceProvider generator = new RNGCryptoServiceProvider();

public IntBig NextBits(int bitCount, bool makeOdd, bool forceCount) {
    if (bitCount < 0) {
        throw new ArgumentOutOfRangeException("bitCount");
    }
    if (bitCount == 0) {
        return 0;
    }
    var bytes = new byte[(bitCount + 7) / 8];
    generator.GetBytes(bytes);
    if (bitCount % 8 != 0) {
        var value = (byte)(bytes[bytes.Length - 1] << (8 - bitCount % 8));
        bytes[bytes.Length - 1] = (byte)(value >> (8 - bitCount % 8));
    }
    if (makeOdd) {
        bytes[0] |= 1;
    }
    if (forceCount) {
        bytes[bytes.Length - 1] |= (byte)(1 << ((bitCount - 1) % 8));
    }
    return IntBig.FromByteArray(bytes);
}

public IntBig Next(IntBig maxValue) {
    if (maxValue == null) {
        throw new ArgumentNullException("maxValue");
    }
    if (maxValue.IsNegative) {
        throw new ArgumentOutOfRangeException("maxValue");
    }
    var roll = NextBits(maxValue.CountBits(), false, false);
    if (roll >= maxValue) {
        roll = roll >> 1;
    }
    return roll;
}

#region interfaces

public void Dispose() {
    generator.Dispose();
}

#endregion
```

A.5. CryptoBig

```
RandomBig random = new RandomBig();
PrimeBig primes = new PrimeBig();

public int KeyStrength { get; set; }

public CryptoBigStrategy CurrentStrategy { get; set; }

[SuppressMessage("Microsoft.Design", "CA1021:AvoidOutParameters")]
[SuppressMessage("Microsoft.Design", "CA1007:UseGenericsWhereAppropriate")]
public void CreateKeyPair(out dynamic publicKey, out dynamic privateKey) {
    if (KeyStrength < 256) {
        throw new InvalidOperationException();
    }

    switch (CurrentStrategy) {
        case CryptoBigStrategy.RSA:
            createRsaKeyPair(out publicKey, out privateKey);
            break;
        case CryptoBigStrategy.DSA:
            createDsaKeyPair(out publicKey, out privateKey);
            break;
        default:
            throw new NotImplementedException();
    }
}

public dynamic Sign(IntBig message, dynamic privateKey) {
    if (message == null) {
        throw new ArgumentNullException("message");
    }
    if (privateKey == null) {
        throw new ArgumentNullException("privateKey");
    }

    switch (CurrentStrategy) {
        case CryptoBigStrategy.RSA:
            return signWithRsa(message, privateKey);
        case CryptoBigStrategy.DSA:
            return signWithDsa(message, privateKey);
        default:
            throw new NotImplementedException();
    }
}

public bool Verify(IntBig message, dynamic signature, dynamic publicKey) {
```

```
    if (message == null) {
        throw new ArgumentNullException("message");
    }
    if (signature == null) {
        throw new ArgumentNullException("signature");
    }
    if (publicKey == null) {
        throw new ArgumentNullException("publicKey");
    }

    switch (CurrentStrategy) {
        case CryptoBigStrategy.RSA:
            return verifyWithRsa(message, signature, publicKey);
        case CryptoBigStrategy.DSA:
            return verifyWithDsa(message, signature, publicKey);
        default:
            throw new NotImplementedException();
    }
}

#region interfaces

public void Dispose() {
    primes.Dispose();
    random.Dispose();
}

#endregion

#region algorithms for RSA

void createRsaKeyPair(out dynamic publicKey, out dynamic privateKey) {
    Debug.Assert(KeyStrength >= 256, "KeyStrength >= 256");

    var p = primes.NextProbableStrongPrime(KeyStrength / 2);
    var q = primes.NextProbableStrongPrime(KeyStrength / 2);

    var n = p * q;
    var l = MathBig.Lcm(p - 1, q - 1);

    var e = random.Next(l - 2) + 2;
    while (!MathBig.Gcd(e, l).IsOne) {
        e = random.Next(l - 2) + 2;
    }
    var d = MathBig.Inv(e, l);

    publicKey = new ExpandoObject();
    publicKey.n = n;
```

```
    publicKey.e = e;

    privateKey = new ExpandoObject();
    privateKey.n = n;
    privateKey.d = d;
}

static dynamic signWithRsa(IntBig message, dynamic privateKey) {
    Debug.Assert(message != null, "message != null");

    IntBig n = privateKey.n;
    IntBig d = privateKey.d;

    if (message.IsNegative || message.IsZero || message >= n) {
        throw new ArgumentOutOfRangeException("message");
    }

    var s = MathBig.Pow(message, d, n);

    dynamic signature = new ExpandoObject();
    signature.s = s;

    return signature;
}

static bool verifyWithRsa(IntBig message, dynamic signature,
                                          dynamic publicKey) {
    Debug.Assert(message != null, "message != null");

    IntBig n = publicKey.n;
    IntBig e = publicKey.e;

    IntBig s = signature.s;

    if (s.IsNegative || s.IsZero || s >= n) {
        throw new ArgumentOutOfRangeException("signature");
    }

    return MathBig.Pow(s, e, n) == message;
}

#endregion

#region algorithms for DSA

void createDsaKeyPair(out dynamic publicKey, out dynamic privateKey) {
    Debug.Assert(KeyStrength >= 256, "KeyStrength >= 256");

```

```
    IntBig q; // will divide p − 1
    var p = primes.NextProbableStrongPrime(KeyStrength, out q);

    IntBig alpha = 1;
    while (alpha.IsOne) {
        var g = random.Next(p − 3) + 2;
        alpha = MathBig.Pow(g, (p − 1) / q, p);
    }

    var a = random.Next(q − 1) + 1;
    var y = MathBig.Pow(alpha, a, p);

    publicKey = new ExpandoObject();
    publicKey.p = p;
    publicKey.q = q;
    publicKey.alpha = alpha;
    publicKey.y = y;

    privateKey = new ExpandoObject();
    privateKey.p = p;
    privateKey.q = q;
    privateKey.alpha = alpha;
    privateKey.a = a;
}

dynamic signWithDsa(IntBig message, dynamic privateKey) {
    Debug.Assert(message != null, "message != null");

    IntBig p = privateKey.p;
    IntBig q = privateKey.q;
    IntBig alpha = privateKey.alpha;
    IntBig a = privateKey.a;

    if (message.IsNegative || message.IsZero || message >= q) {
        throw new ArgumentOutOfRangeException("message");
    }

    var k = random.Next(q − 1) + 1;

    var r = MathBig.Pow(alpha, k, p) % q;
    var s = (MathBig.Inv(k, q) * (message + a * r)) % q;

    dynamic signature = new ExpandoObject();
    signature.r = r;
    signature.s = s;

    return signature;
}
```

```
static bool verifyWithDsa(IntBig message, dynamic signature,
                                          dynamic publicKey) {
    Debug.Assert(message != null, "message != null");

    IntBig p = publicKey.p;
    IntBig q = publicKey.q;
    IntBig alpha = publicKey.alpha;
    IntBig y = publicKey.y;

    IntBig r = signature.r;
    IntBig s = signature.s;

    if (r.IsNegative || r.IsZero || r >= q) {
        throw new ArgumentOutOfRangeException("signature");
    }
    if (s.IsNegative || s.IsZero || s >= q) {
        throw new ArgumentOutOfRangeException("signature");
    }

    var w = MathBig.Inv(s, q);

    var u1 = (w * message) % q;
    var u2 = (w * r) % q;

    var v1 = MathBig.Pow(alpha, u1, p);
    var v2 = MathBig.Pow(y, u2, p);

    var v = ((v1 * v2) % p) % q;

    return v == r;
}

#endregion
```

A.6. HashBig

```
static HashBig() {
    initMd5();
    initSha2();
}

public HashBigStrategy CurrentStrategy { get; set; }

public IntBig ComputeHash(Stream input) {
    if (input == null) {
        throw new ArgumentNullException("input");
    }

    switch (CurrentStrategy) {
        case HashBigStrategy.MD5:
            return computeMd5Hash(input);
        case HashBigStrategy.SHA1:
            return computeSha1Hash(input);
        case HashBigStrategy.SHA2:
            return computeSha2Hash(input);
        default:
            throw new NotImplementedException();
    }
}

#region interfaces

public void Dispose() {
}

#endregion

#region algorithms for MD5

static uint[] y;
static int[] z, s;

static void initMd5() {
    y = new uint[64];
    for (var i = 0; i < 64; i++) {
        y[i] = (uint)(Math.Abs(Math.Sin(i + 1)) * Math.Pow(2, 32));
    }

    z = new[] {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12,
```

```
        5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2,
        0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9
    };

    s = new[] {
        7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
        5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
        4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
        6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
    };
}

static IntBig computeMd5Hash(Stream input) {
    Debug.Assert(input != null, "input != null");

    var h = new uint[] {
        0x67452301,
        0xEFCDAB89,
        0x98BADCFE,
        0x10325476
    };
    var chunk = new uint[16];

    var buffer = new byte[64];
    var read = input.Read(buffer, 0, 64);

    while (read == 64) {
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
        read = input.Read(buffer, 0, 64);
    }
    if (read > 55) {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 64; i++) {
            buffer[i] = 0;
        }
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
        for (var i = 0; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                >> ((i - 56) * 8));
        }
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
    }
```

```
    else {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                >> ((i - 56) * 8));
        }
        toIntArrayLittleEndian(buffer, chunk);
        computeMd5HashChunk(chunk, h);
    }

    return IntBig.FromByteArray(toByteArrayLittleEndian(h));
}

static void computeMd5HashChunk(uint[] chunk, uint[] h) {
    Debug.Assert(chunk != null, "chunk != null");
    Debug.Assert(chunk.Length == 16, "chunk.Length == 16");
    Debug.Assert(h != null, "h != null");
    Debug.Assert(h.Length == 4, "h.Length == 4");

    var a = h[0];
    var b = h[1];
    var c = h[2];
    var d = h[3];

    for (var i = 0; i < 64; i++) {
        var t = 0u;

        if (i < 16) {
            t = (b & c) | ((~b) & d);
        }
        else if (i < 32) {
            t = (b & d) | (c & (~d));
        }
        else if (i < 48) {
            t = b ^ c ^ d;
        }
        else {
            t = c ^ (b | (~d));
        }
        t = rotateLeft(a + t + chunk[z[i]] + y[i], s[i]);

        a = d;
        d = c;
        c = b;
        b = b + t;
```

```
    }

    h[0] = h[0] + a;
    h[1] = h[1] + b;
    h[2] = h[2] + c;
    h[3] = h[3] + d;
}

#endregion

#region algorithms for SHA1

static IntBig computeSha1Hash(Stream input) {
    Debug.Assert(input != null, "input != null");

    var h = new uint[] {
        0x67452301,
        0xEFCDAB89,
        0x98BADCFE,
        0x10325476,
        0xC3D2E1F0
    };
    var chunk = new uint[80];

    var buffer = new byte[64];
    var read = input.Read(buffer, 0, 64);

    while (read == 64) {
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
        read = input.Read(buffer, 0, 64);
    }
    if (read > 55) {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 64; i++) {
            buffer[i] = 0;
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
        for (var i = 0; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                >> ((7 - (i - 56)) * 8));
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
```

```
    }
    else {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                                  >> ((7 - (i - 56)) * 8));
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha1HashChunk(chunk, h);
    }

    return IntBig.FromByteArray(toByteArrayBigEndian(h));
}

static void computeSha1HashChunk(uint[] chunk, uint[] h) {
    Debug.Assert(chunk != null, "chunk != null");
    Debug.Assert(chunk.Length == 80, "chunk.Length == 80");
    Debug.Assert(h != null, "h != null");
    Debug.Assert(h.Length == 5, "h.Length == 5");

    for (var i = 16; i < 80; i++) {
        chunk[i] = rotateLeft(chunk[i - 3] ^ chunk[i - 8] ^
                              chunk[i - 14] ^ chunk[i - 16], 1);
    }

    var a = h[0];
    var b = h[1];
    var c = h[2];
    var d = h[3];
    var e = h[4];

    for (var i = 0; i < 80; i++) {
        var t = 0u;
        var y = 0u;

        if (i < 20) {
            t = (b & c) | ((~b) & d);
            y = 0x5A827999;
        }
        else if (i < 40) {
            t = b ^ c ^ d;
            y = 0x6ED9EBA1;
        }
        else if (i < 60) {
            t = (b & c) | (b & d) | (c & d);
```

```
            y = 0x8F1BBCDC;
        }
        else {
            t = b ^ c ^ d;
            y = 0xCA62C1D6;
        }
        t = rotateLeft(a, 5) + t + e + chunk[i] + y;

        e = d;
        d = c;
        c = rotateLeft(b, 30);
        b = a;
        a = t;
    }

    h[0] = h[0] + a;
    h[1] = h[1] + b;
    h[2] = h[2] + c;
    h[3] = h[3] + d;
    h[4] = h[4] + e;
}

#endregion

#region algorithms for SHA2

static uint[] k;

static void initSha2() {
    k = new uint[] {
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
        0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
        0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
        0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
        0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
        0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
        0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
    };
}
```

```
static IntBig computeSha2Hash(Stream input) {
    Debug.Assert(input != null, "input != null");

    var h = new uint[] {
        0x6a09e667,
        0xbb67ae85,
        0x3c6ef372,
        0xa54ff53a,
        0x510e527f,
        0x9b05688c,
        0x1f83d9ab,
        0x5be0cd19
    };
    var chunk = new uint[64];

    var buffer = new byte[64];
    var read = input.Read(buffer, 0, 64);

    while (read == 64) {
        toIntArrayBigEndian(buffer, chunk);
        computeSha2HashChunk(chunk, h);
        read = input.Read(buffer, 0, 64);
    }
    if (read > 55) {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 64; i++) {
            buffer[i] = 0;
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha2HashChunk(chunk, h);
        for (var i = 0; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
                               >> ((7 - (i - 56)) * 8));
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha2HashChunk(chunk, h);
    }
    else {
        buffer[read] = 0x80;
        for (var i = read + 1; i < 56; i++) {
            buffer[i] = 0;
        }
        for (var i = 56; i < 64; i++) {
            buffer[i] = (byte)((input.Length * 8)
```

```
                                >> ((7 - (i - 56)) * 8));
        }
        toIntArrayBigEndian(buffer, chunk);
        computeSha2HashChunk(chunk, h);
    }

    return IntBig.FromByteArray(toByteArrayBigEndian(h));
}

static void computeSha2HashChunk(uint[] chunk, uint[] hi) {
    Debug.Assert(chunk != null, "chunk != null");
    Debug.Assert(chunk.Length == 64, "chunk.Length == 64");
    Debug.Assert(hi != null, "hi != null");
    Debug.Assert(hi.Length == 8, "hi.Length == 8");

    for (var i = 16; i < 64; i++) {
        var x0 = rotateRight(chunk[i - 15], 7) ^
            rotateRight(chunk[i - 15], 18) ^ (chunk[i - 15] >> 3);
        var x1 = rotateRight(chunk[i - 2], 17) ^
            rotateRight(chunk[i - 2], 19) ^ (chunk[i - 2] >> 10);
        chunk[i] = chunk[i - 16] + x0 + chunk[i - 7] + x1;
    }

    var a = hi[0];
    var b = hi[1];
    var c = hi[2];
    var d = hi[3];
    var e = hi[4];
    var f = hi[5];
    var g = hi[6];
    var h = hi[7];

    for (var i = 0; i < 64; i++) {
        var s0 = rotateRight(a, 2) ^
            rotateRight(a, 13) ^ rotateRight(a, 22);
        var maj = (a & b) ^ (a & c) ^ (b & c);
        var t2 = s0 + maj;
        var s1 = rotateRight(e, 6) ^
            rotateRight(e, 11) ^ rotateRight(e, 25);
        var ch = (e & f) ^ ((~e) & g);
        var t1 = h + s1 + ch + k[i] + chunk[i];

        h = g;
        g = f;
        f = e;
        e = d + t1;
        d = c;
        c = b;
```

```
        b = a;
        a = t1 + t2;
    }

    hi[0] = hi[0] + a;
    hi[1] = hi[1] + b;
    hi[2] = hi[2] + c;
    hi[3] = hi[3] + d;
    hi[4] = hi[4] + e;
    hi[5] = hi[5] + f;
    hi[6] = hi[6] + g;
    hi[7] = hi[7] + h;
}

#endregion

#region tools

static uint rotateLeft(uint value, int count) {
    return (value << count) | (value >> (32 - count));
}

static uint rotateRight(uint value, int count) {
    return (value >> count) | (value << (32 - count));
}

static void toIntArrayLittleEndian(byte[] value, uint[] target) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length % 4 == 0, "value.Length % 4 == 0");
    Debug.Assert(target != null, "target != null");
    Debug.Assert(target.Length >= value.Length / 4);

    for (int i = 0; i < value.Length / 4; i++) {
        target[i] = (uint)value[4 * i];
        target[i] |= (uint)value[4 * i + 1] << 8;
        target[i] |= (uint)value[4 * i + 2] << 16;
        target[i] |= (uint)value[4 * i + 3] << 24;
    }
}

static byte[] toByteArrayLittleEndian(uint[] value) {
    Debug.Assert(value != null, "value != null");

    var result = new byte[value.Length * 4];
    for (int i = 0; i < result.Length; i++) {
        result[i] = (byte)(value[i / 4] >> ((i % 4) * 8));
    }
    return result;
```

```
}

static void toIntArrayBigEndian(byte[] value, uint[] target) {
    Debug.Assert(value != null, "value != null");
    Debug.Assert(value.Length % 4 == 0, "value.Length % 4 == 0");
    Debug.Assert(target != null, "target != null");
    Debug.Assert(target.Length >= value.Length / 4);

    for (int i = 0; i < value.Length / 4; i++) {
        target[i] = (uint)value[4 * i] << 24;
        target[i] |= (uint)value[4 * i + 1] << 16;
        target[i] |= (uint)value[4 * i + 2] << 8;
        target[i] |= (uint)value[4 * i + 3];
    }
}

static byte[] toByteArrayBigEndian(uint[] value) {
    Debug.Assert(value != null, "value != null");

    var result = new byte[value.Length * 4];
    for (int i = 0; i < result.Length; i++) {
        result[i] = (byte)(value[i / 4] >> ((3 - (i % 4)) * 8));
    }
    return result;
}

#endregion
```

Literaturverzeichnis

[1] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone:
Handbook of Applied Cryptography,
CRC Press Inc. (1997)

[2] Richard Crandall, Carl Pomerance:
Prime Numbers. A Computational Perspective. Second Edition,
Springer (2005)

[3] Jeffrey Richter:
CLR via C#. Second Edition,
Microsoft Press (2006)

[4] G. Eigenthaler: *Algebra,*
Skriptum zur gleichnamigen Vorlesung (2004)

[5] J. Wiesenbauer: *Analyse von Algorithmen,*
Skriptum zur gleichnamigen Vorlesung (2009)

[6] Wikipedia: *Zweierkomplement,*
http://de.wikipedia.org/wiki/Zweierkomplement
(Abruf am 25. 07. 2010)

[7] Wikipedia: *Karatsuba-Algorithmus,*
http://de.wikipedia.org/wiki/Karatsuba-Algorithmus
(Abruf am 25. 07. 2010)

[8] Wikipedia: *Diffie-Hellman-Schlüsselaustausch,*
http://de.wikipedia.org/wiki/Diffie_Hellman
(Abruf am 31. 07. 2010)

[9] Wikipedia: *RSA-Kryptosystem,*
http://de.wikipedia.org/wiki/RSA-Kryptosystem
(Abruf am 31. 07. 2010)

[10] Wikipedia: *RSA-129*,
http://de.wikipedia.org/wiki/RSA-129
(Abruf am 31. 07. 2010)

[11] Wikipedia: *Erweiterter euklidischer Algorithmus*,
http://de.wikipedia.org/wiki/Erweiterter_euklidischer_Algorithmus
(Abruf am 03. 08. 2010)

[12] Wikipedia: *SHA-2*,
http://en.wikipedia.org/wiki/SHA-256
(Abruf am 27. 09. 2010)

[13] Wikipedia: *MD5*,
http://en.wikipedia.org/wiki/MD5
(Abruf am 28. 09. 2010)

[14] Wikipedia: *SHA-3*,
http://en.wikipedia.org/wiki/Sha-3
(Abruf am 02. 10. 2010)

[15] Wikipedia: *Toom-Cook multiplication*,
http://en.wikipedia.org/wiki/Toom-Cook_multiplication
(Abruf am 23. 11. 2010)

[16] Microsoft: *Visual Studio Express Downloads*,
http://www.microsoft.com/express/Downloads
(Abruf am 17. 07. 2010)

[17] Microsoft: *FxCop 10.0*,
http://www.microsoft.com/downloads/details.aspx
?FamilyID=917023f6-d5b7-41bb-bbc0-411a7d66cf3c
(Abruf am 17. 07. 2010)

[18] Microsoft: *Operator overloads have named alternates*,
http://msdn.microsoft.com/library/ms182355.aspx
(Abruf am 17. 07. 2010)

[19] Microsoft: *Random Class*,
http://msdn.microsoft.com/library/system.random
(Abruf am 02. 08. 2010)

[20] Microsoft: *INotifyPropertyChanged Interface*,
http://msdn.microsoft.com/system.componentmodel.inotifypropertychanged
(Abruf am 05. 10. 2010)

[21] Microsoft: *ICommand Interface*,
http://msdn.microsoft.com/library/system.windows.input.icommand
(Abruf am 05. 10. 2010)

[22] Microsoft: *BackgroundWorker Class*,
http://msdn.microsoft.com/library/system.componentmodel.backgroundworker
(Abruf am 05. 10. 2010)

[23] Microsoft: *Parallel Programming in the .NET Framework*,
http://msdn.microsoft.com/en-us/library/dd460693.aspx
(Abruf am 23. 11. 2010)

[24] Josh Smith: *WPF Apps With The MVVM Design Pattern*,
http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
(Abruf am 05. 10. 2010)

[25] Alireza Shirazi: *How To Get Hardware Information*,
http://www.codeproject.com/KB/system/GetHardwareInformation.aspx
(Abruf am 05. 10. 2010)

[26] Arndt Brünner: *Primzahlen*,
http://www.arndt-bruenner.de/mathe/scripts/primtab.htm
(Abruf am 05. 10. 2010)

[27] Bertrand Le Roy: *Immutability in C#*,
http://weblogs.asp.net/bleroy/archive/2008/01/16/immutability-in-c.aspx
(Abruf am 17. 07. 2010)

[28] Inbar Gazit: *Introducing: System.Numeric.BigInteger*,
http://blogs.msdn.com/b/bclteam/archive/2007/01/16/
introducing-system-numeric-biginteger-inbar-gazit.aspx
(Abruf am 20. 07. 2010)

[29] Melitta Andersen: *Where did BigInteger go?*,
http://blogs.msdn.com/b/bclteam/archive/2008/01/04/
where-did-biginteger-go-melitta-andersen.aspx
(Abruf am 20. 07. 2010)