Aufgabenblatt 1
IT-Security
Angewandte Informatik
WS 2018/19

Lernziele – 8 Punkte

- Bibliothek BigInt
- (Schnelle) Algorithmen für Multiplikation und Division

Für dieses Aufgabenblatt benötigen Sie **viel Zeit**.

Aufgaben

Es soll der Beginn einer Bibliothek für Algorithmen mit beliebig hoher Genauigkeit für ganze Zahlen erstellt werden. In diesem Aufgabenblatt wird der erste Teil dazu erstellt; in späteren Aufgabenblättern werden diese Routinen benutzt und eventuell ergänzt, d.h. wer die letzten Aufgabenblätter bearbeiten möchte, muss dieses Blatt in jedem Falle erfolgreich lösen.

Als Programmiersprachen sind möglich: Java (empfohlen), C, C++, C#, PHP, JavaScript (Node.js), Python, Pascal/Delphi. Auf keinen Fall dürfen fertige Bibliotheken, z.B. BigInteger bei Java, bei der Lösung benutzt werden. Selbstverständlich sind diese Bibliotheken zum Testen benutzbar.

Wichtig ist: Testen, Testen und Testen. Dazu können Sie ein Framework, wie z.B. junit, cunit etc. und die gerade erwähnten Bibliotheken benutzen. In jedem Falle muss mindestens mit den für diese Blätter zur Verfügung gestellten Testdaten getestet werden. Diese Testdaten liegen als verschiedene Dateien mit demselben syntaktischen Aufbau vor, so dass sich die Realisierung eines "Testtreibers", der diese Dateien einliest, und die Tests automatisiert durchführt, anbietet.

Für die Realisierung sind einige Entscheidungen zu treffen:

(I) Als erstes ist eine Entscheidung zu treffen, wie viele Bits breit eine Stelle in einer Integer-Variablen (Zelle) sein soll bzw. die Basis oder der Modulo-Wert für eine Zelle. Für 32bit-Maschinen wäre dies 16, für 64bit-Maschinen 32. Die benutzten Bits der Zellen sollten immer eine 2er-Potenz sein. BCD-Arithmetik oder ähnlich sollen nicht benutzt werden.

Zu dieser Entscheidung gehört auch die Festlegung der Datentypen Cell, Cell2 sowie die Definition und Benennung verschiedener Konstanten, wie z.B. mask.

(II) Dann ist eine Datenstruktur zu definieren, die (a) den Wert als Array von nat (unsigned int), (b) eine Angabe über die Array-Größe und (c) eine Angabe darüber, welches Array-Element von der höchsten Wertigkeit betrachtet, einen Wert ungleich 0 hat (d.h. alle höherwertigen Zellen haben den Wert 0), beinhaltet. Orientieren Sie sich am besten an den Foliensätzen.

(III) Die nächste Entscheidung betrifft die Fehlerbehandlung: Wird beim Overflow, Underflow eine Exception geworfen oder ein Errorcode geliefert? Im letzteren Fall muss z.B. der Aufrufer als Return-Wert den Errorcode vom Ergebnis unterscheiden können. Diese Entscheidung hat Einfluss auf die Signaturen der Methoden.

(IV) In den Foliensätzen wurden (meist) die Routinen als Funktionen ohne Seiteneffekte definiert. Dazu führt dazu, dass BigInts sehr häufig kopiert werden müssen. Die Alternative dazu besteht darin, Prozeduren mit einem In-/Output-Parameter (Call by Reference) zu definieren. Das entsprechende BigInt-Objekt wird dann per Adresse übergeben und in der Prozedur verändert. Das führt zu anderen Signaturen als unten angeben und natürlich auch zu einem anderen Programmierstil. Im Falle von Java wird die Benutzung von Adressen durch Objekte vorgegeben.

(V) Als Letztes müssen die Signaturen der jeweiligen Programmiersprache angepasst werden. Hier fließen die obigen Entscheidungen ein. Wer gerne in Java (oder C#) programmiert, kann sich an die Signaturen von Java anlehnen (Siehe z.B. Web-Link Nr. 7). Dann sieht die Parameterversorgung etwas anders aus. Die folgenden Signaturen orientieren sich an einem prozeduralen Ansatz. Es wird sich in späteren Aufgabenblättern herausstellen, dass diese Bibliothek erweitert werden muss.

Im Folgenden werden die Signaturen der Routinen in einer Freistilnotation angegeben; diese müssen dann entsprechend der Programmiersprache umgeschrieben werden, z.B. als Klassen oder Module. Das betrifft auch die Fehlerbehandlung und die Return-Codes. Programmiersprachen, die eine eigene Definition von + - etc. erlauben, z.B. Python, sollten davon Gebrauch machen und die Signaturen entsprechend ändern.

Nun die Kernfunktionen, die bis auf die Karasuba-Version in diesem Aufgabenblatt realisiert werden müssen:

- func BigInt add(BigInt x, y) - Schulbuchaddition
- func BigInt sub(BigInt x, y) – Schulbuchsubtraktion
- func BigInt mul(BigInt x, y) – Schulbuchmultiplikation
- func BigInt mulKara(BigInt x, y) – Karazuba-Multiplikation. Diese Routine ist für dieses Aufgabenblatt optional. Wer diese Routine realisiert, möge sie auch gleich in mul() einbauen.
- func BigIntMod moddiv(BigInt x, y) – Schulbuchdivision

  Der Returnwert vom Typ BigIntMod ist eine Struktur bestehend aus dem Divisionsergebnis und dem Rest.

Nun kommen die Konvertierungsroutinen - dies sind zugleich auch die Erzeugungsroutinen für die benötigten Daten bzw. Objekte:

- func BigInt toBigInt(String x) – Konvertieren einer langen Dezimalzahl in einem String nach BigInt, auch negative Zahlen mit Vorzeichen
- func String toString(BigInt x) – Konvertieren einer BigInt-Zahl nach einem Dezimalzahl-String, auch negative Zahlen mit Vorzeichen
- func BigInt toBigInt16(String x) – Konvertieren einer langen Hexadezimalzahl in einem String nach BigInt, auch negative Zahlen
- func String toString16(BigInt x) – Konvertieren einer BigInt-Zahl nach einem Hexadezimalzahl-String, auch negative Zahlen mit Vorzeichen

  Die beiden Hex-Versionen werden zum Testen benötigt.

Nun die absehbaren Hilfsroutinen:

- func BigInt reduce(BigInt x) – reduziert die Zahl x auf die Anzahl der Zellen, die einen Wert ungleich 0 realisieren. Diese Routine sollte intern immer aufgerufen werden, wenn die Zahl der relevanten Zellen sich verkleinert haben könnte.
- func BigInt resize(BigInt x, nat sz) – vergrößert den relevanten Teil der Zahl x auf die Größe sz. Falls sz kleiner als der vorhandene relevanten Teil ist, beginnt eine Fehlerbehandlung. resize() ist gewissermaßen das Gegenteil zu reduce().
- func BigInt Cell div10(BigInt x) – Division mit 10 mit Liefern des Restes.
- func BigInt mul10(BigInt x) – Multiplikation mit 10.
- func BigInt mulCell(BigInt a, Cell b) - Multiplikation mit b
- func BigInt shiftLeft(BigInt a, nat cnt) – Verschieben um cnt Bits nach links
- func BigInt addCell2(BigInt a, nat index, Cell2 b)
- BigInt invers(BigInt a) – Bilden des Inversen bzgl. der Addition; diese Routine wird für die Ein-/Ausgabe bzw. Konvertieren von/nach Strings benötigt.
- func BigInt copy(BigInt x, nat b>=0) – kopiert die unteren b Bits von x als Return-Wert. Ist b gleich 0, dann wird die gesamte Zahl kopiert. Diese Routine wird für die Karazuba-Multiplikation benötigt. Mit b=0 wird die Zahl geklont.

Dann noch die Allokations- und Freigaberoutinen:

- func BigInt newBigInt(nat sz>0, int init) – Anlegen eines neuen BigInt-Objekts mit der Mindestgröße sz und der Initialisierung mit int (32 bit). Der relevante Teil wird mit init initialisiert.
- func BigInt newBigInt(nat sz>0, BigInt init) - Anlegen eines neuen

BigInt-Objekts mit der Mindestgröße sz und der Initialisierung mit einem anderen BigInt init.

- proc freeBigInt(BigInt x) – Freigeben des Objekts x; dies ist nur für Sprachen wie C/C++ oder ähnlich notwendig.

Der Sinn bzw. Zweck dieser Routinen ergibt sich aus den kryptographischen Algorithmen. Bitte beachten Sie folgendes Prinzip: **Es kommt auf Korrektheit und nicht auf Performanz an**. Lieber langsam und korrekt statt schnell und (manchmal) inkorrekt. Der Sinn der Aufgaben besteht darin, dass Sie die Algorithmen durch Programmieren verstehen.

Noch ein paar Tipps:

- Testen Sie beginnend mit den Hilfsroutinen aufwärts, jeweils immer nur eine Routine allein. Dann setzen Sie daraus eine komplexere Routine zusammen und testen diese etc.
- Damit die Strukturen BigInt bei der Parameterübergabe nicht kopiert zu werden brauchen, arbeiten Sie wie bei Java intern mit Pointern (gilt für C/C++).
- Um das Leben einfacher zu gestalten legen Sie physikalisch alle BigInt-Arrays gleichlang mit 2048 bit (oder mehr) an; das sollte für den Hausgebrauch reichen. Die Abfrage auf Overflow und Underflow müssen Sie in jedem Fall programmieren. Dadurch ersparen Sie sich eine physikalische Vergrößerung mit malloc() o.ä. Sie können natürlich dynamisch den Speicher vergrößern bzw. verkleinern.
- Einige Routinen verkürzen den relevanten Teil der Zahlen (Bereich der unteren Zellen ungleich 0), so dass dann die Routine reduce() aufgerufen werden sollte.
- Bei den Grundrechenarten ist es hilfreich, dass beide Operanden gleich lang sind. Daher erweitern Sie den kürzeren Operanden auf die relevante Größe des längeren Operanden, auch wenn darunter die Performanz leidet. Dazu dient resize().

Links

1. https://de.wikipedia.org/wiki/GNU_Multiple_Precision_Arithmetic_Library
2. https://gmplib.org/
3. https://github.com/MikeMcl/big.js/
4. http://www.php.net/manual/en/book.bc.php
5. http://php.net/manual/en/book.gmp.php
6. https://de.wikipedia.org/wiki/Karazuba-Algorithmus
7. https://dbs.cs.uni-duesseldorf.de/lehre/docs/java/javabuch/html/-

k100114.html

8. https://www.dpunkt.de/java/Referenz/Das_Paket_java.math/3.html
9. http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_12_006.htm

Abnahme

Zur Abnahme des "Moduls" BigInt gehören folgende Dateien:

- Testfälle mit Reports,
- Source-Code und
- Projektdateien, z.B. nbproject oder make-Dateien etc.

Die Tests müssen der minimalen Bedingung genügen, dass alle Programmpfade einmal durchlaufen werden. Dazu gehören auch die Testfälle, die zu dieser Veranstaltung zur Verfügung gestellt wurden.

Die Vorführung besteht in folgenden Ablauf: (1) Source-Code-Begutachtung, (2) Übersetzung und (3) Testlauf.

Auch diese Aufgabe kann per Email abgegeben werden, dann mit allen Sourcen, Übersetzungsdateien, z.B. make bzw. eclipse/netbeans-Projekte, einschließlich der Tests. Alles muss ohne weiteres auf dem Rechner des Dozenten laufen können (Java, C++, Linux, CentOS 6.10). Programmieren Sie also portabel. Im Falle von Python, Ruby oder Lua bitte noch die notwendige Laufzeitumgebung mitliefern.