# Zusatzaufgabe 2 für Informatiker

**Bearbeitungszeit:** vier Wochen (bis Montag, den 21. Juni 2021)

**Mathematischer Hintergrund:** Primzahltest

**Elemente von C++:** Bitoperationen

---

## Aufgabenstellung

Für Public-Key-Verschlüsselungsverfahren (z.B. RSA-Verfahren) werden zufällige große Primzahlen benötigt. Zu diesem Zweck erzeugt man Zahlen in der gewünschten Größenordnung (realistisch $> 10^{75}$) und testet dann, ob es sich um Primzahlen handelt. Zum Ausführen dieses Tests werden im Text die folgenden Methoden beschrieben:

- Probedivision (Sieb des Eratosthenes)
- Fermat-Test
- Miller-Rabin-Test.

Schreiben Sie ein Programm, welches die Primalität einer gegebenen Zahl mit Hilfe dieser Methoden beweisen, vermuten bzw. widerlegen kann.

## Generelle Vorgehensweise

Für die verschiedenen Methoden

```
enum Method { DIVISION, FERMAT, MILLER_RABIN };
```

werden Ihnen `num_examples` natürliche Zahlen zur Verfügung gestellt, die Sie auf *prim* testen sollen. Da diese Zahlen sehr groß sein können, sollen Sie mit dem vordefinierten Typ `lint` und nicht mit `unsigned int` arbeiten, d.h. alle Funktionsköpfe sind mit diesem Typ zu schreiben, sofern es sich um Zahlen handelt, die in der Größenordnung der Primzahlen liegen können. Durch den Aufruf von

```
lint getCandidate(Method method, unsigned int ex);
```

erhalten Sie je Methode und je Beispiel einen Kandidaten zum Testen. Wie Sie Ihr Ergebnis dann auf Richtigkeit prüfen können (und sollen) wird im folgenden bei der jeweiligen Methode beschrieben.
Die für Sie wichtigen Definitionen und Funktionsköpfe stehen in `a3.h`.

## Probedivision

Um festzustellen, daß eine gegebene Zahl $n$ *nicht prim* ist reicht es offenbar aus, mindestens einen Faktor ($> 1, \neq n$) von $n$ zu finden. Umgekehrt ist eine Zahl dann *prim*, wenn keine der Primzahlen $p \leq \sqrt{n}$ die Zahl $n$ (ohne Rest) teilt. Es stellt sich daher die Frage, wie man Primzahlen bis zu einer vorgegebenen Schranke einfach berechnen kann. Die Idee des *Sieb des Eratosthenes* ist dabei, ein Feld einer vorgegebenen Länge von Markern zu generieren, die darüber Aufschluß geben, ob die Zahl korrespondierend zum jeweilige Index prim ist oder nicht, z.B.

```
bool Sieb[100]; // (Sieb[11]==true) bzw. (Sieb[11]) bedeutet: 11 ist
    prim
```

Dazu geht man wie folgt vor:

- Zu Anfang setzt man alle Zahlen des Feldes bis auf 0 und 1 auf *prim*.
- Angefangen bei 2, setzt man alle Vielfachen (innerhalb des Feldes) auf *nicht prim*, jedoch ohne die aktuelle Zahl selber.
- Nun wählt man als neue aktuelle Zahl die nächste Primzahl im Feld, bis das das Ende des Feldes erreicht ist. Dabei ist die nächste Primzahl die Zahl, deren Marker auf *prim* steht.

Es bleibt offenbar ein Feld von Primzahlen übrig, mit denen man die Probedivision ausführen kann.
Zu diesem Zweck existiert eine Klasse für dieses Sieb und Sie brauchen es nur noch zu füllen. Die Klasse heißt `Sieve` und Variablen diesen Typs sind so initialisiert, daß alle Zahlen auf *prim* stehen. Bitte lesen und verstehen Sie die Operatoren in der Klasse[1]. Der Zugriff auf ein solches Sieb könnte wie folgt aussehen:

```
Sieve sieve;          // Achtung: alle prim
// ...
sieve[1] = false;     // 1 ist keine Primzahl
// ...
for (lint i=2; i<sieve.length(); ++i) {
    // Schleife ueber das Sieb ...
}
```

Legen Sie eine Variable vom Typ `Sieve` an und programmieren Sie das Sieb des Eratosthenes. Rufen Sie dann zum Test die Funktion

```
void checkSieve(const Sieve& sieve);
```

mit Ihrem Sieb auf.
Wenn das Sieb in Ordnung ist, programmieren Sie eine Funktion

```
lint trialDivision(lint n, const Sieve& sieve);
```

die mittels Ihres Siebs eine Zahl `n` testet und entweder den kleinsten gefundenen Faktor ($> 1$) der Zahl zurückgibt, oder 1, wenn Sie keinen Faktor gefunden haben. Mit diesem Rückgabewert rufen sie dann

```
void checkSolution(Method method, unsigned int ex, const lint f);
```

mit der Methode `DIVISION` und der jeweiligen Beispielnummer auf. Den Programmrahmen finden Sie (ohne Funktionalität) in `a3.cpp`.

## Fermat-Test

Dieser Primzahltest basiert auf dem kleinen Satz von Fermat:

*Ist $n$ eine Primzahl, so gilt $a^{n-1} = 1$ mod $n$ für alle $a \in Z$ mit gcd$(a, n) = 1$ .*[2]

[1] Lassen Sie sich nicht durch die Angabe `template` stören, das ist noch nicht so wichtig. Die eigentliche Typdefinition findet nach der Klassenbeschreibung in `using` statt.

[2] `gcd` berechnet den größten gemeinsame Teiler (greatest common divisor).

Mit diesem Theorem hat man die Möglichkeit festzustellen, ob eine Zahl faktorisierbar ist. Dazu wählt man eine Basis $a \in \{1, \ldots, n-1\}$ mit $\gcd(a, n) = 1$ und berechnet $y = a^{n-1} \bmod n$. Ist $y \neq 1$ so kann $n$ keine Primzahl sein.
Allerdings ist für den Fall $y = 1$ auch nicht sicher, daß $n$ eine Primzahl ist. Die Zahlen, für die $y = 1$ für alle $a$ mit obiger Eigenschaft gilt, heißen *Carmichael-Zahlen*. Die Carmichael-Zahlen unter 100000 lauten

$$561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973, 75361.$$

Ihre Aufgabe ist es nun, für alle Beispielzahlen $n$ und für alle Primzahlen (Basen) $a$ aus Ihrem Sieb, die kleiner als `maxPrimeNumber` sind und $n$ nicht direkt teilen[3], den Wert

$$y = a^{n-1} \bmod n$$

in einer Funktion

```
lint fermatTest(lint n, lint a);
```

zu berechnen. Ist für eine Basis der Wert $y \neq 1$, so kann $n$ keine Primzahl sein, ansonsten kann man es nicht beurteilen. Merken Sie sich, ob ein $a$ dazu geführt hat, daß $n$ keine Primzahl ist. Rufen Sie weiter nach jeden (Fermat)Test die Funktion

```
void checkSolution(Method method, unsigned int ex, const lint f,
   const lint basis = 0);
```

mit der Methode `FERMAT`, der jeweiligen Beispielnummer, Ihrem errechneten Wert $y$ und dem benutzten $a$ auf. Haben Sie alle angegebenen $a$'s getestet, rufen Sie für dieses Beispiel abschließend nochmals die Funktion `checkSolution` auf. Diesmal geben sie als Basis $a = 0$ an und übergeben in $y$, ob sie der Meinung sind, daß $n$ eine Primzahl ist ($y = 1$) oder nicht ($y = 0$).
Für die Berechnung in `fermatTest` benötigen Sie noch eine Funktion zur schnellen Berechnung von $y$, nämlich `fastpow`. Eine Beschreibung dazu erfolgt im Anschluß an die Beschreibung der Methoden.

### Rabin-Miller–Test

Dieser Primzahltest beruht auf einer Erweiterung des kleinen Satzes von Fermat:

*Ist $n$ eine Primzahl und ist $a$ eine zu $n$ teilerfremde natürliche Zahl, so gilt entweder i)*

$$a^d = 1 \ mod \ n$$

*oder es existiert ein $r \in \{0, 1, \ldots, s-1\}$ mit ii)*

$$a^{2^r d} = n - 1 \ mod \ n.$$

Dabei ist

$$s = \max\{r \in N \ : \ 2^r \text{ teilt } n-1\}, \quad d = \frac{n-1}{2^s}.$$

Findet man also eine Zahl $a$, die zu $n$ teilerfremd ist und für die weder i) noch ii) für ein $r$ gilt, so ist $n$ keine Primzahl sondern zusammengesetzt. Eine solche Zahl $a$ heißt *Zeuge* gegen die Primalität von $n$.
Ihre Aufgabe in diesem Teil ist die Bereitstellung der Funktionen

```
lint millerRabinTest(lint n, lint a, lint d, lint r);
void getDS(lint n, lint& d, lint& s);
```

[3] Eigentlich muß hier auf `gcd(a,n)=1` getestet werden. Da aber alle zu benutzenden $a$'s Primzahlen sind, gilt `gcd(a,n)`$\neq 1$ genau dann, wenn $a$ direkt $n$ teilt.

die

$$y = a^{2^r d} \bmod n$$

und die zugehörigen $d$ und $s$ berechnen. Sie sollen auch hier die Funktion `fastpow` benutzen.
Gehen Sie analog zu Aufgabenteil 2 vor und ermitteln Sie, ob es sich bei den Beispielzahlen um Primzahlen handelt. Beachten Sie, daß entweder i) oder ii) gelten kann und testen Sie für den Fall ii) alle relevanten $r$. Rufen Sie zum Test

```
void checkSolution(Method method, unsigned int ex, const lint f,
   const lint basis = 0, const lint r = 0);
```

auf und am Ende wiederum `checkSolution` mit $a = 0$ und $y$ je nach Testergebnis.

### Schnelle Potenzberechnung

Angenommen es ist auszurechnen:

$$a^x \bmod n.$$

Weiter sei

$$x = \sum_{i=0}^{k} x_i \, 2^i$$

die Binärentwicklung von $x$. Dabei sind $x_i \in \{0, 1\}$. Wegen

$$a^x = a^{\sum x_i 2^i} = \prod_{i=0}^{k} (a^{2^i})^{x_i} = \prod_{0 \le i \le k, x_i = 1} a^{2^i}$$

kann man das Produkt nun aus der Multiplikation der Quadrate $a^{2^i}$ berechnen, für die $x_i = 1$ gilt. Bei der Umsetzung ist auszunutzen, daß

$$a^{2^{i+1}} = (a^{2^i})^2$$

gilt. Die Modulo-Rechnung kann auf jeden Faktor $a^{2^i}$ einzeln angewandt werden. Dies ermöglicht insbesondere bei realistischen Größenordnungen erst die Berechnung.

## Literatur

[1] J. Buchmann. *Einführung in die Kryptographie.* Springer Verlag, Heidelberg, 1999.