Praktikum V – Zufall

GTI SoSe 2017 Prof. A. Siebert, T. Franzke

Aufgabe 1. Stochastische Matrizen

Joe Naturhansl verbringt seine Freizeit mit Radln, Klettern, Biergarting und Fensterln, und zwar nach festem Ritual:

- Nach dem Radln geht er in 20% der Fälle nochmals Radln, in 30% der Fälle Klettern und in 50% der Fälle Biergarting.
- Nach dem Klettern geht er in der Hälfte der Fälle Radln, sonst Biergarting.
- Nach dem Biergarting geht er immer Fensterln.
- Nach dem Fensterln geht er fast immer (zu 90%) Radln, sonst Klettern.

a. Über sein ganzes Leben betrachtet, wieviel Zeit verbringt Joe anteilig mit seinen vier Aktivitäten?

b. Wenn Joe gerade Fensterln war, wie groß ist dann die Wahrscheinlichkeit p, dass seine übernächste Aktivität Radln ist?

Aufgabe 2. Miller-Rabin Test

a. Implementieren Sie zunächst die schnelle Modulare Potenzierung $x = a^e \mod m$ gemäß Skript 03 / Folie 65 für x, a, e, m vom Datentyp Integer.

Zum Laufzeitvergleich ist hier der Code für die langsame Modulare Potenzierung (siehe auch Datei `RabinAux.java`):

```
int powModIntSlow(int a, int expo, int modu) {
    long result = a % modu;
    for (int i=2; i<=expo; i++)
        result = (result*a) % modu;
    return (int) result;
}
```

Messen Sie die Laufzeiten für die langsame und für die schnelle Modulare Potenzierung mit $a = 4711, e = 1234567890, m = 2^{31} - 1$.
Die Zeitmessung können Sie vornehmen mit

```
long startTime, diffTime;
startTime = System.nanoTime();
* * * your terrific code here * * *
diffTime  = (System.nanoTime() - startTime)/1000000;
System.out.println("run time: " + diffTime + " ms");
```

b. Für praxistaugliche Primzahlentests muß die Modulare Potenzierung für große Zahlen (d.h. Parameter und Ergebnis $> 2^{63}$) funktionieren.
Implementieren Sie deshalb die schnelle Modulare Potenzierung nun für den Datentyp BigInteger, also eine Methode
`BigInteger powModBig(BigInteger a, BigInteger expo, BigInteger modu)`.

Check: $\text{rs129}^{\text{cm1000}} \mod \text{cm16} = 2612875580173417$.
Die Werte von rs129, cm16 und cm1000 finden Sie in `RabinAux.java`.

c. Implementieren Sie den (für größere Carmichael-Zahlen unzuverlässigen) probabilistischen Primzahltest gemäß Skript 03 / Folie 63 für Datentyp BigInteger.

Für den Test der Primzahl n verlangt der Pseudo-Code die Auswahl einer zufälligen Basis a zwischen 2 und n-1.
Ein Linearer Kongruenzgenerator lässt sich für BigInteger kaum sinnvoll einsetzen, da es schwierig ist, geeignete Werte für den Multiplikator und den Modulus zu bestimmen. Deshalb ist es sinnvoll, einen zufälligen BigInteger-Wert ($0 \leq$ result $\leq$ n-1) mit Hilfe eines zufälligen Bitfeldes zu bestimmen, wie folgt:

```
BigInteger nextBigInt(BigInteger n) {
    Random rng = new Random();
    int nBitLen = n.bitLength();

    BigInteger result = new BigInteger(nBitLen, rng);
    while (result.add(BigInteger.ONE).compareTo(n) >= 0) {
        result = new BigInteger(nBitLen, rng);
    }
    return result;
}
```

Hier wird zunächst eine Zufallszahl der erforderlichen Bitlänge erstellt. Falls die erzeugte Zahl größer sein sollte als n-1, so wird die Generierung wiederholt. Die erwartete Anzahl der Wiederholung ist 2 (es droht hier keine Endlosschleife).

Testen Sie ihren Probabilistischen Primzahltest mit `NUMBER_OF_TRIALS=20`, also maximal 20 verschiedenen Werten für die Basis a. Bei den kleineren ($\leq$ 5-stelligen) Zahlen sollten keine Fehler auftreten, auch nicht bei den Carmichael-Zahlen (561, 1105, ... 41041) aus jenem Bereich.
Sie sollten aber beobachten können, dass Ihr Test für die größeren Carmichael-Zahlen (cm16, cm1000) allerhöchstwahrscheinlich den falschen Wert ("ist Primzahl") liefert.

d. [Sniff, jetzt kommt die eigentliche Aufgabe, einer der Höhepunkte Ihrer illustren GTI-Praktikumskarriere. Für alle diejenigen, die in das Praktikum V bereits mindestens vier Stunden Arbeit gesteckt haben, ist dieser Teil jedoch freiwillig.]

Implementieren Sie den Miller-Rabin-Primzahltest gemäß Skript 03 / Folie 67 für den Datentyp BigInteger.

Die Java-Methode `BigInteger.isProbablePrime()` ist eine Umsetzung des Miller-Rabin-Tests. Sie sollten Ihre Implementierung bezüglich der Ergebnisse und Laufzeiten für zahlreiche große Zahlen vergleichen.

Für relativ kleine Zahlen liegt die Laufzeit meiner Implementierung etwa um einen Faktor 5 über der Laufzeit von `isProbablePrime()`. Für größere Eingabezahlen wird der Unterschied noch größer.
Zum Praxiseinsatz empfiehlt es sich offensichtlich, einige Optimierungen vorzunehmen. Insbesondere ist zu empfehlen, zunächst mit den etwa 1000 kleinsten Primzahlen (welche in einer Liste bereit gehalten werden) herkömmliche Probedivision durchzuführen, ehe man auf den eigentlichen Miller-Rabin-Test umsattelt.