Hochschule für Technik und Wirtschaft Berlin

Algorithms and Optimizations

Lehrbeauftragter: Dr. Hermann Thiel

Wintersemester 2015 / 2016

Teil 3: Spezielle Algorithmen

Heute: Probabilistische / Randomisierte Algorithmen

Was berechnet die folgende Methode?

```
public static double montecarlo(int nrIter){
    Random r = new Random();
    r.setSeed(0);
    int count = 0;
    float x, y, z;
    for (int i=0; i<nrIter; i++){
        x = r.nextFloat();
        y = r.nextFloat();
        z = x*x+y*y;
        if (z<=1) count++;
    }
    return (float) count/nrIter*4;
}
```

Probabilistische / randomisierte Algorithmen

- Monte-Carlo-Verfahren
- In vielen Problemen, in denen die rechnerische Komplexität zu groß ist, z.B. TSP
- In der Kryptographie: Stream-Ciphers, DH, RSA
- Primzahlerzeugung

Zufallszahlen

Natürlich können auf einer deterministischen Maschine keine echten Zufallszahlen erzeugt werden. Donald Knuth zitiert John von Neumann:

„Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.“

Pseudozufallszahlen-Generatoren

Sollen Folgen von Nullen und Einsen ausgeben,

Die „zufällig aussehen“, d.h. diese Folgen sollen alle statistischen Tests für Zufälligkeit bestehen:

- Die Periode sollte groß genug sein
- Die Anzahl der erzeugten Nullen und die Anzahl der erzeugten Einsen

  sollte ungefähr gleich groß sein;

  für Folgen gleicher Zahlen sollte gelten:

  Die Hälfte solcher Folgen sollte die Länge 1 haben, ein Viertel

  die Länge 2, ein Achtel die Länge 3 usw.
- Die Zufallsfolgen sollen nicht komprimierbar sein.

Diese und weitere Eigenschaften kann man empirisch bestimmen und mir den statistischen Erwartungen vergleichen (z.B. Chi – Quadrat-Test).

Kryptographie

Kryptographieverfahren verlangen von einer Zufallszahlenfolge viel mehr als andere Anwendungen. Kryptographische Pseudozufallszahlen sollen sich nicht nur statistisch zufällig verhalten, sondern auch *unvorhersagbar* sein.

Die Forderung „Kryptographisch sichere Pseudozufallszahlen sollten nicht komprimierbar sein“ gilt natürlich nur solange, wie der Schlüssel zu dem zur Erzeugung verwendeten Verfahren unbekannt ist ...

Der Schlüssel ist normalerweise der Anfangs - (oder Initialisierungs / seed) Status des Generators. Damit werden Zufallszahlengeneratoren Gegenstand von Entschlüsselungsangriffen. Zufallszahlengeneratoren sicher zu machen ist also ein wichtiges Kapitel der Kryptographie.

Real Random

Gibt es Zufall? Was ist eine Zufallsfolge? Woran erkennt man die Zufälligkeit einer Folge? Wann ist eine Folge „zufälliger“ als eine andere?

Die Quantenmechanik behauptet, dass es in der Realität Zufall gibt. Aber wie kommt Zufall in eine deterministische Maschine?

Hier soll eine weitere Eigenschaft für echten Zufall gefordert werden:

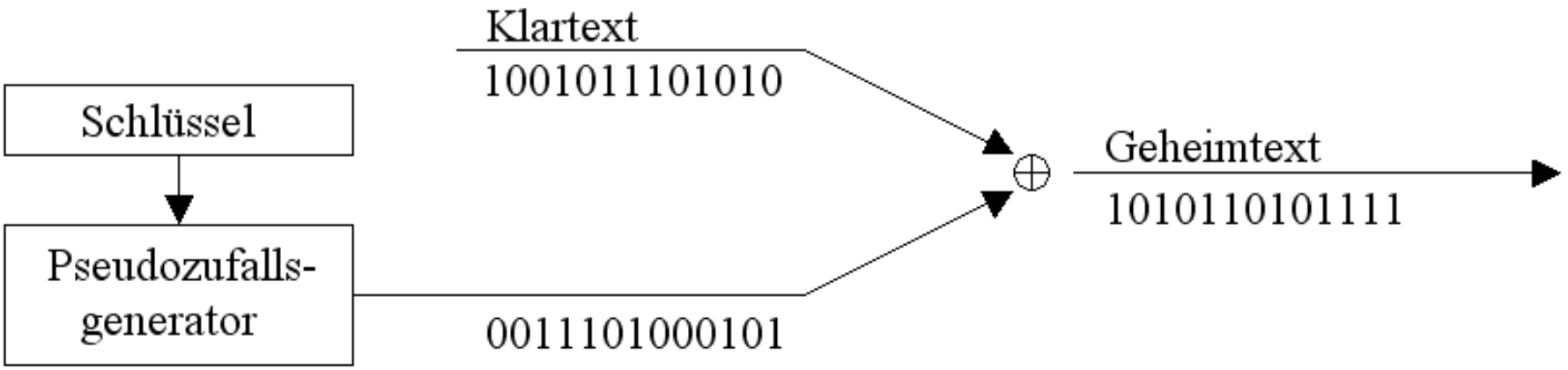
Eine Folge von Zufallszahlen soll man nicht reproduzieren können.

Quellen von echtem Zufall: Radioaktiver Zerfall, allg. Quantenphänomene, Zener Dioden, ...

... und nicht ganz so echte: Mausbewegungen, Systemzeit, ...

Eine Anwendung sind Stream-Ciphers.

Ein einfaches Modell:



Realisierungen von Zufallsgeneratoren

- Wiederholtes Verschlüsseln einer Zeichenfolge mit DES / AES und einem gegebenen Schlüssel ergibt eine zufällig aussehende Ausgabefolge
- arithmetische Generatoren
- Schieberegister

LCG: Linear Congruential Generators

- Die Folge der Pseudozufallszahlen erhält man durch:

$$x_{n+1} \equiv (ax_n + b) \bmod m$$

- *a* heißt der Multiplikator (multiplier), *b* Inkrement (increment) und *m* der Modul (modulus). Der Schlüssel oder seed ist der Wert x_0.
- Die Periode ist kleiner oder gleich *m*. Sind *a* und *b* geeignet gewählt, sodass die Periode *m* ist, heißt der Generator ein *maximal period generator*.
- Es gibt Tabellen mit geeigneten Werten für *a*, *b* und *m*.
- LCR's sind zum Verschlüsseln ungeeignet, sie sind vorhersagbar.
- LCR's sind schnell.
- LCR's haben gute statistische Eigenschaften

Es gibt weitere Zufallsgeneratoren für die Kryptographie:

Blum-Micali,

RSA-Generator,

BBS (Blum, Blum und Shub) / quadratic residue generator,

LFSR (Linear Feedback Shift Registers,

FCSR (Feedback with Carry Shift Registers,

Nonlinear – Feedback Shift Registers

RSA: Rivest, Shamir, Adleman

1. Alice wählt zwei große Primzahlen $p \neq q$ und berechnet $n = pq$

   Die Primzahlen können z.B. jeweils 1024 bit lang sein.

2. Alice wählt eine weitere ganze Zahl *s* mit ggT(s, (p-1)(q-1)) = 1

3. Alice berechnet daraus eine ganze Zahl *t* mit

   $$st \equiv 1 \bmod (p-1)(q-1)$$

4. Alice veröffentlicht n und s und hält p, q und t geheim.

5. Will Bob eine Nachricht k an Alice schicken, so holt er sich Alice Schlüssel (n, s), berechnet $C = k^s \bmod n$ und schickt C an Alice.

6. Alice kann *C* mit ihrem geheimen Schlüssel (n, t) entschlüsseln:

   $$k = C^t \bmod n$$

Beispiel

Große Primzahlen

Public – key Algorithmen benötigen Primzahlen.
Ein Netzwerk benötigt sehr viele Primzahlen.

1. Findet man genug Primzahlen?
   Ja! Es gibt ungefähr 10^{151} Primzahlen mit einer Länge von 512 Bits oder kürzer.
   Für Zahlen nahe n ist die Wahrscheinlichkeit für eine Zufallszahl prim zu sein ungefähr eins zu $\ln(n)$. Die Anzahl der Primzahlen kleiner n ist ungefähr $n/\ln(n)$.

2. Könnte es nicht sein, dass zwei verschiedene Personen zufälligerweise die gleiche Primzahl auswählen, obwohl sie dies nicht wollen?
   Nein! Wenn man unter 10^{151} Primzahlen wählen kann, wird dies nicht geschehen.

3. Könnte man nicht eine Datenbank mit allen in Frage kommenden Primzahlen aufbauen? Und diese dann dazu benutzen, public – key Algorithmen zu knacken?
   Nein! Es gibt einfach zu viele.

4. Man muss allerdings darauf achten, dass die Zufallsgeneratoren, mit denen man die Primzahlen auswählt, kryptographisch sicher, d. h. insbesondere unvorhersagbar sind!

Der Satz von Fermat

Für jede Primzahl p und jede zu p teilerfremde Zahl a gilt:

$$a^{p-1} \equiv 1 \bmod p$$

Der Fermat-Test prüft, ob eine gegebene Zahl n eine Primzahl ist:

Dazu wählt man eine ganze Zahl a mit $1 < a < n$; ist nun

a^{n-1} nicht kongruent 1 mod n, so ist n keine Primzahl.

Ist $a^{n-1} \equiv 1 \bmod n$, so heißt n *pseudoprim* bezüglich a.

Die sog. Carmichael-Zahlen sind pseudoprim bezüglich jeder Zahl a, für die ggT(a, n) = 1 gilt. Es gibt Carmichael-Zahlen, die keine Primzahlen sind.

Beispiele

Rabin-Miller-Test

Für die ungerade natürliche Zahl n sei $n-1=s\cdot 2^t$, wobei s ungerade ist und

$t > 0$. Gelten für die ganze Zahl a mit $1 < a < n$ die Aussagen

$$a^s \not\equiv 1 \bmod n \text{ und } a^{s\cdot 2^r} \not\equiv -1 \bmod n \text{ für } 0 \le r < t ,$$

so ist n nicht prim.

In diesem Fall heißt a ein *Zeuge* für den Rabin-Miller-Test für n . Ist dies nicht der Fall, so *besteht* n *für* a *den Rabin-Miller-Test*.

Es gilt der folgende

Satz (Rabin-Monier)

Ist eine natürliche Zahl $n > 9$ zusammengesetzt und ungerade, so besteht sie den Rabin-Miller-Test für höchstens $\phi(n)/4$ aller Basen $0 < a < n$. (ϕ ist die Eulersche ϕ – Funktion)

Anders formuliert bedeutet dies: Mehr als 75% der Basen sind Zeugen (für die Zusammengesetztheit).

Oder: Die Wahrscheinlichkeit für ein *falsch-positives* Resultat des Rabin-Miller-Tests ist höchstens ¼ (falsch-positiv bedeutet dabei, dass die Zahl n vom Test als Primzahl bezeichnet wird, obwohl sie dies nicht ist).

Dateien: *Primzahlerzeugung.pdf,* Bayes bei Rabin-Miller (moodle)

Hausaufgabe 6

1. Wieso kann eine natürliche Zahl n nicht pseudoprim bezüglich einer Zahl a sein, wenn $ggT(a, n) \neq 1$ gilt?

2. Implementieren Sie den Algorithmus von Rabin und Miller; einen Pseudocode dazu finden Sie in moodle. Testen Sie Ihre Implementierung, indem Sie diese mit der java-BigInteger-Methode *isProbablePrime* vergleichen. Wie viele Primzahlen $< 1.000.000$ findet Ihr Programm? Mit welchem Parameter k?

3. Finden Sie eine 32-bit-Zahl, die bei *einem* ($k = 1$) Durchlauf des Miller-Rabin-Testes nicht als zusammengesetzt erkannt wird, obwohl sie das aber ist.

4. Finden Sie mit Ihrer Implementierung des Rabin-Miller-Algorithmus eine Zufalls(pseudo)primzahl mit 512 bit.