Zufall im Rechner

Ausarbeitung im Rahmen der Lehrveranstaltung
„Informationssicherheit III“

Fachhochschule Bonn-Rhein-Sieg
WS 2001/2002

Fachhochschule
Bonn-Rhein-Sieg

Autor:

Jochen Rondorf
(Jochen.Rondorf@gmx.de)

Bonn, im Januar 2002

Inhaltsverzeichnis

1 Einleitung

1.1 Abstract

Es gibt zwei Arten von Zufall: „echten“ und computergenierten Zufall. Beispiele für echten Zufall, deren Daten aus nicht-deterministischen Quellen stammen, sind Münzwerfen („Kopf“ oder „Zahl“), Ziehung der Lottozahlen mit einer Maschine oder Würfeln. Im Bereich der Physik können auch Messungen im Zusammenhang mit der Erzeugung und des Registrierens von Zählimpulsen, z.B. beim radioaktiven Zerfall, echten Zufall erzeugen, im Computerbereich können Tastaturanschläge und Mausbewegungen mit in die Berechnungen einfließen.

Manchmal ist es zu aufwendig, reale Zufallszahlen zu erzeugen. Dann verwendet man sogenannte Pseudozufallsgeneratoren. Dies ist „ein Algorithmus, der aus einer kurzen Folge von Zufallszahlen, eine lange Folge berechnet, die zufällig ‚aussieht’, die also nicht in Polynomzeit von einer wirklichen zufälligen Folge unterschieden werden kann.“ [Buch99, S.90]. Computererzeugte Zufallszahlen sind periodisch. Das heißt, nach einer gewissen Zeit folgen die Zahlen in einer schon zuvor produzierten Reihenfolge. Hieraus folgt, dass Zufallszahlen theoretisch vorhersehbar sind.

1.2 Motivation: Anwendungsgebiete von Zufallszahlen

Zufallszahlen brauchen wir heute in verschiedenen Bereichen. Anwendung finden Pseudozufallsgeneratoren bei statistischen Simulationen, numerischen Analysen, unvoreingenommene Entscheidungsfindungen (auch für optimale Strategien bei Computerspielen) und in der Computerprogrammierung. Der letztgenannte Punkt ist sehr umfangreich. Hier denke man insbesondere an kryptografische Algorithmen, Tests von Effektivität und Effizienz von Programmen und Computerspiele mit Würfeln, Rouletterädern oder gemischten Spielkarten.

Der aus der Public-Key-Verschlüsselung bekannte RSA-Algorithmus „benötigt große Primzahlen, deren Erzeugung ein großes Problem darstellt. Die Ursache liegt darin, dass es außer der versuchsweisen Division mit allen möglichen Teilern (Brute-Force) keinen diskreten Nachweis gibt, ob eine Zahl prim ist oder nicht. Der Aufwand für einen Primzahltest wächst mit größer werdender Stellenzahl gigantisch. Daher beschränkt man sich auf das Auffinden von Pseudoprimzahlen. Eine Pseudoprimzahl ist nur mit einer gewissen statischen Sicherheit prim. Zuerst wählt man eine große Zufallszahl. Danach wendet man verschiedene Tests an, um zu überprüfen, ob die Zahl prim ist oder nicht. Bei einem negativen Test fängt man neu an.“ [Pseu01].

Für kryptografischen Verfahren (Schlüsselgenierung bei Public-Key-Verschlüsselung, Erzeugung von Onetime-Pads) bedient man sich heutzutage aus sicherheitstechnischen Gründen echtem Zufall.

1.3 Aufbau der Studienarbeit

Diese Studienarbeit ist so aufgebaut, dass in Kapitel 2 die Eigenschaften von guten Zufallszahlen sowie deren Bedeutung für sicherheitsrelevante Programme beschrieben werden. Im Weiteren werden verschiedene Tests vorgestellt, mittels derer die Güte von Zufallszahlenquellen ermittelt werden kann.

In Kapitel 3 wird auf die zwei wichtigsten Typen von heutigen Pseudozufallsgeneratoren eingegangen. Dies sind der lineare Kongruenzgenerator nach Lehmer und das lineare Schieberegister mit Rückkopplung. Andere Generierungstechniken sowie Varianten der zwei vorgestellten Verfahren existieren, sind aber nicht Teil dieser Arbeit.

Kapitel 4 beschäftigt sich mit dem praktischen Einsatz von Zufallszahlen und deren Bedeutung bezüglich Sicherheit. Die Initial Sequence Number bei TCP/IP-Verbindungen wird als Beispiel für sicherheitsrelevanten, computergenerierten Zufall vorgestellt. Echter Zufall hingegen wird für das webbasierte Home-Banking benötigt.

Kapitel 5 ist in zwei Teile gegliedert. Im ersten werden bestehende Implementierungen von Zufallsgeneratoren vorgestellt. Die Java-Klasse „Random“ produziert Pseudozufallszahlen, das Unix-Device /dev/random generiert echten Zufall. Hierbei wird ausschließlich auf die Funktionsweise eingegangen. Es findet keine zeilenweise Erläuterung der Quellen o.ä. statt. Diese stehen aber bei Bedarf im Internet zur Verfügung und können eingesehen werden; der Quellcode von „java.util.Random“ und „java.lang.Math“ findet sich zudem im Anhang dieser Arbeit. Im zweiten Teil wird der Quellcode einer eigenen Implementierung eines einfachen Pseudozufallsgenerators sowie eines kleinen Zufallstester-Programms vorgestellt und kurz erläutert.

Eine knapp dreiseitige Zusammenfassung dieser Arbeit findet sich in Kapitel 6.

2 Anforderungen an „gute“ Zufallszahlen

2.1 Eigenschaften

Um einen Pseudozufallsgenerator bauen zu können, müssen zunächst die Anforderungen an einen solchen Generator betrachtet werden. Wann eine Zahl „zufällig“ ist, mag auf den ersten Blick einfach zu erkennen sein. Schauen wir uns folgende Zahlen aus [ZVE98] an:

- 1111111... (Nicht zufällig, da es nur eine mögliche Ziffer gibt.)
- 123123123... (Nicht zufällig, da periodisch vorhersagbar ist.)
- 141592653 (Nicht zufällig, da Bildungsregel ersichtlich: PI)
- 937393793... (Nicht zufällig, da nur die Ziffern 9, 3 und 7 vorkommen.)
- 1234567890... (Nicht zufällig, da zwar alle Ziffern statistisch gleich häufig vorkommen, aber die Wahrscheinlichkeit, das die 8 hinter der 7, die 7 hinter der 6, usw. kommt, 100% ist.)

Ganz so einfach ist eine Definition aber nicht. Nach [Clo97] gibt es fünf Eigenschaften von „guten“ Zufallszahlen. Diese sollen nämlich

1. gleichverteilt,
2. unabhängig,
3. mit großer Periode errechnet,
4. reproduzierbar und
5. einfach und schnell (effizient) berechenbar

sein. „Gleichverteilung“ bedeutet, dass keine Zufallszahl häufiger vorkommt als eine andere. Nehmen wir an, dass wir Zufallszahlen von 1 bis n produzieren wollen. Die Wahrscheinlichkeit, dass Ereignis i eintritt (also Zufallszahl i ausgegeben wird) sei p_i. Bei einer Gleichverteilung sind alle p_i gleich. Mit „unabhängig“ ist hier nicht die statistische Unabhängigkeit gemeint, sondern die Unabhängigkeit bezüglich zweier Ereignisse (Zufallszahlen). Zwei Ereignisse A und B heißen unabhängig, wenn die bedingte Wahrscheinlichkeit des Eintretens von A (unter der Gewissheit, dass B eingetreten ist) gleich der Wahrscheinlichkeit des Eintretens von A ist: P(A|B) = P(A). Alternativ: P(A und B tritt ein) = P(A) * P(B).

Da wir nur zyklische Zufallszahlenreihen generieren können, benötigen wir eine „große Periode“ für die Reihe. Wir gehen davon aus, dass praktisch in einem Programm nie so viele Zufallszahlen benötigt werden, dass der zweite Zyklus ausgegeben wird. „Reproduzierbarkeit“ bedeutet, dass ein Pseudozufallsgenerator bei mehreren Läufen mit gleichen (Start-)Parametern identische Zufallszahlenreihen ausgibt. Der beste Zufallszahlengenerator nutzt nichts, wenn er ineffizient programmiert ist und die Generierung einer einzigen Zufallszahl mehrere Sekunden dauert.

Die Eigenschaften 3-5 sind relativ einfach zu realisieren. Für 1-2 benötigen wir Prüfprogramme. Eine kleine Java-Testsuite wird in Kapitel 5.4 vorgestellt.

2.2 Sicherheitsaspekte

Für sicherheitsrelevante Programme, wie einige Beispiele in Kapitel 4 aufgeführt sind, ist es unumgänglich, unvorhersehbare Zufallszahlen erzeugen zu können. Diese sind mit einem Computer aber nur schwer zu generieren. Das liegt daran, weil Rechner so gebaut sind, dass sie nur eine Menge genau definierter Kommandos ausführen können, die bei nochmaligem Aufruf nach genau demselben Schema abgearbeitet werden und dieselben Ergebnisse produzieren. Dadurch kann jeder fixe Algorithmus dazu benutzt werden, genau dieselben Ergebnisse auf einem anderen Computer nachzustellen. Damit lassen sich effektiv Ergebniswerte vorauszubestimmen - vorausgesetzt, dass das „Innenleben“ des Algorithmus, d.h. alle Variablenbelegungen und der Aufbau der Kontrollstrukturen, rekonstruierbar ist. Selbst wenn die Zufallsgeneratorfunktion des Zielrechners unbekannt ist, wird das Programm früher oder später identische Zufallssequenzen hervorbringen, weil die Anzahl der inneren Zustände des Algorithmus begrenzt ist. Dies liegt unter anderem daran, dass der Computer zwar mit beliebiger, aber nur mit fester, zur Kompilierzeit festgelegter Präzision rechnen kann. Glücklicherweise wiederholt sich die Sequenz bei geeigneter Parameterwahl nur nach großen Zeiträumen und Zufallsmengen. Trotzdem befinden sich heutzutage immer noch Pseudozufallsgeneratoren mit Perioden von 500 Elementen oder weniger im Einsatz, die Angriffsfläche für Hacker bieten.

„Echte“ Zufallszahlen können von www.random.org heruntergeladen werden. Sie sind hauptsächlich für Simulationen gedacht. Es wäre nicht besonders intelligent, diese Zahlen in ein sicherheitsrelevantes Programm zu kodieren, da diese Zahlensequenzen öffentlich zugänglich sind und so jedem Angreifer zur Verfügung stehen.

2.3 Tests zur Gleichverteilung

2.3.1 Entropie

Die Gleichverteilung von Zufallszahlen kann man über die Entropie beschreiben. Um Entropie mathematisch zu beschreiben, betrachten wir einen Zufallsgenerator, der n verschiedene Zahlen ausgibt, als Informationsquelle mit endlichem Alphabet $A = (a_1, a_2, ..., a_n)$. Die Wahrscheinlichkeit des Generators, die Zahl a_i auszugeben, nennen wir p_i. Dann ist die negierte Summe über $p_i * \log_2(p_i)$ die Entropie dieser Informationsquelle. ($\log_2(x)$ für $0<x<1$ ist negativ, somit muss die Summe negiert werden, um ein positives Ergebnis zu erhalten.) Sie gibt ein Maß für den "Informationsgehalt" der Quelle.

Anschaulich kann Informationsgehalt etwa in folgendem Beispiel erklärt werden: Die Wahrscheinlichkeit eines Flugzeugunglücks ist statistisch geringer als die Wahrscheinlichkeit für einen Autounfall. Wird in den Nachrichten von einem Flugzeugunglück berichtet, ist der Informationsgehalt dieser Meldung größer.

Im Fall der Gleichverteilung, d.h. alle p_i sind gleich, ist die Entropie gleich $\log_2(n)$ und maximal. In diesem Fall haben wir keine Information über das nächste ausgesendete Zeichen, wenn nur genau ein Zeichen ausgegeben werden kann (also etwa $p_i = 1$ und

alle anderen $p_i = 0$). Hier ist die Entropie gleich Null (mit der vereinbarten Rechenregel $0 * \log_2 0 = 0$). Die Quelle hat keinen Informationsgehalt, weil wir nicht wissen, welche Information als nächstes kommt. Damit gibt die Entropie wieder, wie "überraschend" ein Zeichen kommen kann, und beim Zufall wollen wir, dass die "Überraschung" maximal ist.
Eine wirkliche Zufallszahl, die theoretisch unendlich lang ist, passt in kein statistisches Muster. Ihre Entropie ist gleich Null. So ist es auch unmöglich, sie zu komprimieren, d.h. sie durch irgendwelche Bildungsregeln oder statistische Methoden zu verkleinern. „Findet man die richtigen Bildungsregeln für die Daten, lassen sie sich verkleinern. (Je) kleiner die Entropie ist, desto schwieriger wird es, Daten zu komprimieren. Solange die Datenmenge endlich ist, kann keine genaue Aussage über die Entropie gemacht werden, wenn sie jedoch unendlich groß ist, so dauert auch eine Überprüfung unendlich lange.“ [ZVE98].

2.3.2 x^2-Test

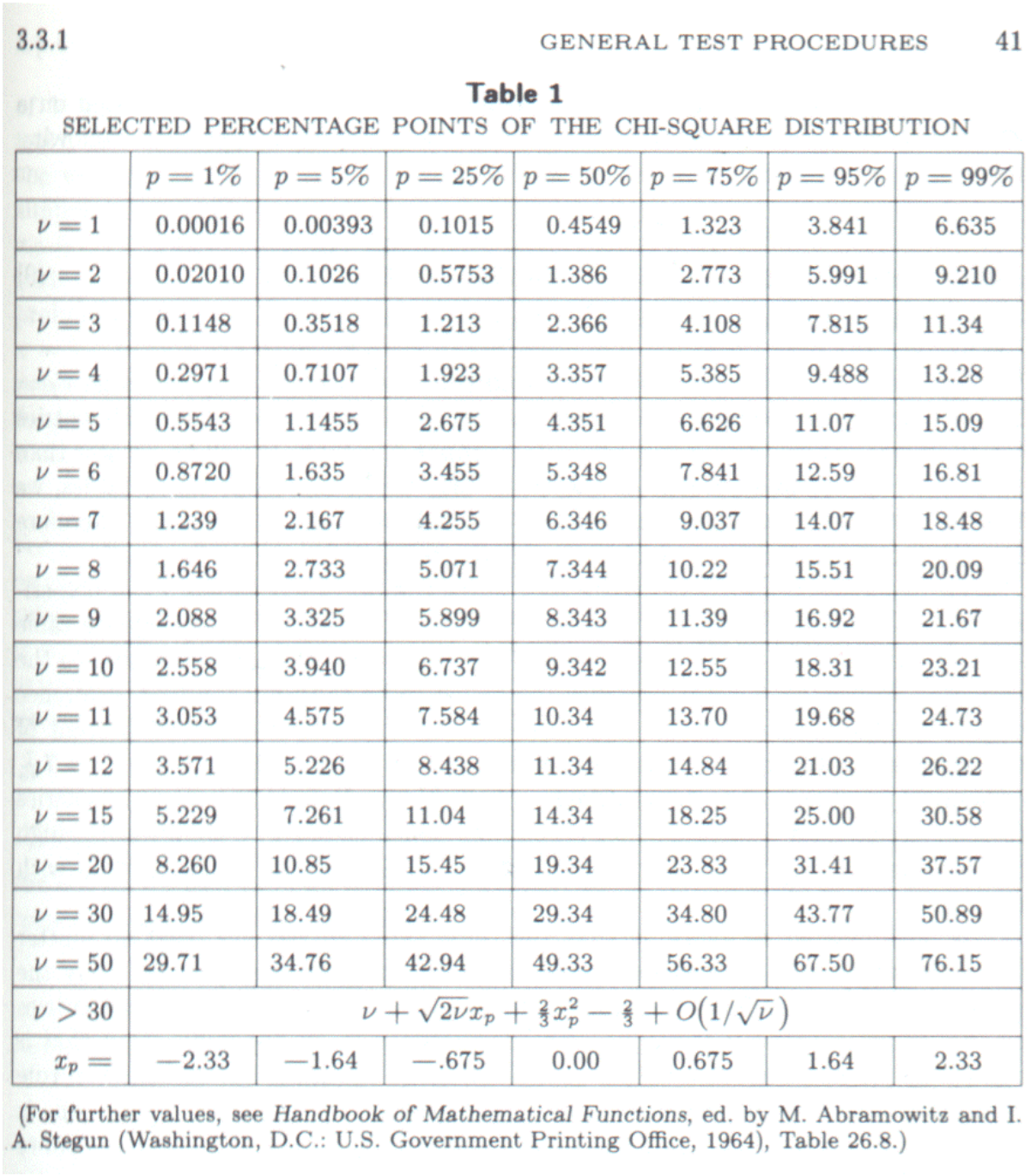
Mathematisch wird dieser Test bewerkstelligt, indem die Summe der Quadrate der einzelnen Häufigkeiten Y_s durch die zu erwartende Häufigkeit $n*p_s$ dividiert wird. (Das n kann vor das Summenzeichen gezogen werden.) Danach wird die Größe der Folge subtrahiert:

$$V = \frac{1}{n} \sum_{1 \le s \le k} \left(\frac{Y_s^2}{p_s} \right) - n$$

Die Herleitung der Formel kann in [Knut98, S. 40] nachgelesen werden. Ein Beispiel soll die Prozedur veranschaulichen: Man nehme die Summe zweier Würfel und notiere die Wahrscheinlichkeit der Augenzahlen (für n = 144):

Summe der Würfelaugen = s	2	3	4	5	6	7	8	9	10	11	12
tatsächliche Anzahl = Y_s	2	4	10	12	22	29	21	15	14	9	6
erwartete Anzahl = np_s	4	8	12	16	20	24	20	16	12	8	4

Es ergibt sich V = … = 7,14583…
Um etwas mit diesem Wert anfangen zu können, ist diese Tabelle nötig:



3.3.1 GENERAL TEST PROCEDURES 41

Table 1

SELECTED PERCENTAGE POINTS OF THE CHI-SQUARE DISTRIBUTION

	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
$\nu = 1$	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
$\nu = 2$	0.02010	0.1026	0.5753	1.386	2.773	5.991	9.210
$\nu = 3$	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
$\nu = 4$	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
$\nu = 5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
$\nu = 6$	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
$\nu = 7$	1.239	2.167	4.255	6.346	9.037	14.07	18.48
$\nu = 8$	1.646	2.733	5.071	7.344	10.22	15.51	20.09
$\nu = 9$	2.088	3.325	5.899	8.343	11.39	16.92	21.67
$\nu = 10$	2.558	3.940	6.737	9.342	12.55	18.31	23.21
$\nu = 11$	3.053	4.575	7.584	10.34	13.70	19.68	24.73
$\nu = 12$	3.571	5.226	8.438	11.34	14.84	21.03	26.22
$\nu = 15$	5.229	7.261	11.04	14.34	18.25	25.00	30.58
$\nu = 20$	8.260	10.85	15.45	19.34	23.83	31.41	37.57
$\nu = 30$	14.95	18.49	24.48	29.34	34.80	43.77	50.89
$\nu = 50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15
$\nu > 30$	$\nu + \sqrt{2\nu}x_p + \frac{2}{3}x_p^2 - \frac{2}{3} + O(1/\sqrt{\nu})$						
$x_p =$	−2.33	−1.64	−.675	0.00	0.675	1.64	2.33

(For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U.S. Government Printing Office, 1964), Table 26.8.)

Abbildung 1: Ausgewählte Prozentpunkte einer Chi-Quadrat-Verteilung, aus [Knut98, S. 41]

Wir haben 11 verschiedene Werte für s, nämlich 2-12 Würfelaugen. Der „Freiheitsgrad“ ist 1 weniger, also 10, weil sich die Wahrscheinlich des 11. Wertes durch die vorherigen Werte 1-10 ergibt, damit insgesamt die Summe 1 herauskommt. Wir schauen also in die Zeile der Tabelle mit v = 10 und erkennen, dass mit einer Wahrscheinlichkeit von 95% V einen Wert <= 18,31 hat, also mit 5%iger Wahrscheinlichkeit darüber liegt. Für unseren Wert V = 7,145… stellen wir fest, dass er zwischen 25% und 50% eingegliedert wird. Interpoliert würde man grob über den Daumen gepeilt etwa auf 40% kommen.

Die Formel wird bei gleichen Wahrscheinlichkeitsverteilungen, wie dies ja für Zufallszahlenreihen gewünscht ist, einfacher, weil p_i konstant ist.
Eine Implementierung des Chi-Quadrat-Tests für Zufallszahlen kann in Kapitel 5.4.2 betrachtet werden.

2.4 Tests zur Unabhängigkeit

Die in Kapitel 2.3 vorgestellten Verfahren testen nur die Gleichverteilung der auftretenden Zahlen. Die Reihenfolge ist völlig unerheblich und ändert nichts am Ergebnis des Tests. Dieses Manko beheben die Unabhängigkeitstest.

2.4.1 Spektraltest

Der Spektraltest ist ein sehr mächtiger und daher auch sehr komplexer und schwer verständlicher Zufallszahlenreihentest. Knuth braucht in [Knut98, S. 89-113] genau 25 Seiten, um ihn zu beschreiben. Die Bedeutung des Spektraltests ist sehr groß und er wird daher oft eingesetzt. Den Spektraltest in Gänze vorzustellen, würde den Rahmen dieser Arbeit sprengen. Trotzdem möchte ich kurz auf die Idee dahinter eingehen. Auf eine Implementierung in Kapitel 5.4 wird aus Komplexitätsgründen verzichtet.
Der von Knuth beschriebene Spektraltest richtet sich ausschließlich auf Zufallszahlenreihen, die mittels des LCG (siehe Kapitel 3.1) generiert wurden. Er untersucht, wie stark eine Zufallszahl von ihrem Vorgänger abhängt. Ein LCG mit der Formel $X_{n+1} = (137\ X_n + 187) \bmod 256$ ergibt folgendes Diagramm, wenn man auf der X-Achse X_n, auf der Y-Achse X_{n+1} und für 3D-Darstellung auf der Z-Achse X_{n+2}) aufträgt:

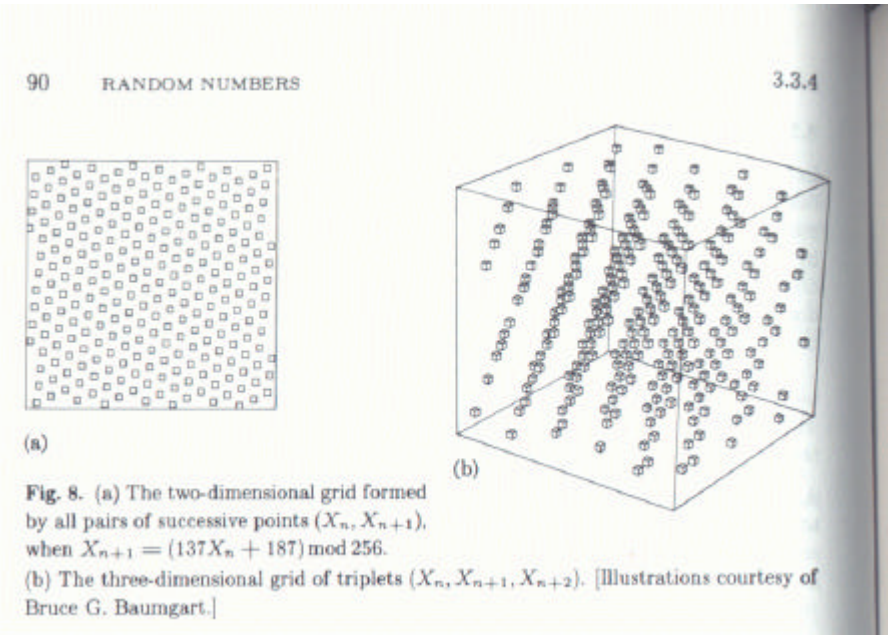


Abbildung 2: Abhängigkeit von der Vorgängerzahl bzw. den Vorgängerzahlen beim LCG in 2D und 3D, aus [Knut98, S. 90]

Der benutzte Generator mit Modulus 256 wird sicherlich nicht wirklich zufällig sein. Er lässt sich aber gut zeichnen und macht die Testidee verständlicher.
Man erkennt einige parallele Linien etwa im Winkel 30°, die teilweise noch den gleichen Abstand zueinander aufweisen. Sollte sich die Existenz solcher Muster auch bei größeren Moduli beweisen lassen, besteht der Generator den Spektraltest nicht.
Wer genügend Ehrgeiz mitbringt, sei an dieser Stelle für Genaueres auf [Knut98, S. 89-113] verwiesen.

2.4.2 Run-Test

Der Run-Test ist an sich nicht sehr komplex, wohl aber seine Interpretierung. Man überprüft hiermit, wie viele Zahlen hintereinander ohne Unterbrechung aufwärts ansteigen („run-up“) bzw. abwärts abfallen („run-down“). Betrachten wir folgende Zufallszahlenreihe:

```
1 5 9 4 8 3 4 1 9 2
 + + - + - + - + -
```

Der längste Run-Up ist 1 5 9 am Anfang mit der Länge 2. Die Reihe ist nicht sehr auffällig. Schauen wir uns folgende (präparierte) Reihe an:

```
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
 + + + + - + + + + - + + + + - + + + +
```

Mit dieser Methode kann ich trotz etwa eines bestandenen Entropietests mit maximaler(!) Entropie eine schlechte Zufallsreihenfolge nachweisen. Die erste Überlegung wäre, auf die Run-Längen einen Chi-Quadrat-Test zu fahren. Knuth spricht sich eindeutig dagegen aus, da er meint, dass Run-ups und Run-downs nicht unabhängig aufeinander folgen. Auf einen langen Run würde höchstwahrscheinlich ein kurzer folgen und umgekehrt. Er empfiehlt in Tabellen nachzuschauen, die ausschließlich für Run-Tests angefertigt wurden, wie gut (relativ wahrscheinlich) oder schlecht (relativ unwahrscheinlich) eine Zufallsfolge ist.
In [Knut98, S. 65] ist eine Tabelle abgedruckt, die in diese Arbeit nicht übernommen worden ist.

3 Verfahren von Pseudozufallsgeneratoren

3.1 Lineare Kongruenzgeneratoren

Eine bekannte Art und Weise, Zufallszahlen zu generieren, sind die linearen Kongruenzgeneratoren (LCGs) nach Lehmer. Sie sind nach der Formel

$$X_n = (a * X_{n-1} + b) \bmod m$$

$$\text{mit } ggt(b, m) = 1,\ 0 < a < m,\ 0 < b < m$$

X_0 = Startwert
a = Multiplikator
b = Inkrement
m = Modulus

aufgebaut.

Da insgesamt modulo m gerechnet wird, sind die Parameter a und b kleiner m zu wählen. (Mathematisch korrekt wären auch größere a und b, aber die würden die Laufzeit des Algorithmus verlangsamen.)

3.1.1 Parameter b und m

Der Modulus m sollte sehr hoch gewählt werden, da die Periode der Zufallszahlenreihe höchstens m betragen kann. Dies liegt daran, dass die Zufallszahl nur von der vorherigen rechnerisch abhängt. Eine Zahl, die das zweite Mal in der Reihe vorkommt, ist damit automatisch Beginn der Periode. Für $m > 10^8$ spricht Knuth von zufriedenstellenden Perioden.

Die Parameter b und m sind relativ prim zueinander zu wählen, d.h. ggt(b, m) = 1. Ein größter gemeinsamer Teiler > 1 würde die Periode der Zufallszahlenreihe verkleinern. Zudem muss (a-1) Vielfaches von p für jeden Primfaktor p von m gelten. Wenn m durch 4 teilbar ist, dann ist es auch (a-1). Der Beweis wird hier nicht geführt und ist bei Bedarf in [Knut98, S. 16] nachzulesen.

3.1.2 Random-Seed

Wie in 1.1 bereits beschrieben, ist ein Zufallsgenerator ist „ein Algorithmus, der aus einer kurzen Folge von Zufallszahlen, eine lange Folge berechnet (...)“. Die „kurze Folge“ nennt man auch Random-Seed (engl. = Zufallssamen, Zufallskern). In aktuellen Implementierungen von Pseudozufallsgeneratoren nimmt man fast immer die Unix-Repräsentation der Systemzeit, nämlich die Anzahl der Sekunden bzw. besser noch der Millisekunden seit dem 1.1.1970 als Integerwert. Hierdurch kann sicher gestellt werden, dass ein Zufallsgenerator bei jedem Neustart praktisch eine völlig neue Zufallszahlenreihe produziert.

In 2.1 haben wir als Anforderung an einen Zufallszahlengenerator festgestellt, dass er u.a. reproduzierbar sein soll. Wenn wir einen Pseudozufallsgenerator mit demselben Ran-

dom-Seed füttern, dann erhalten wir auch immer eine identische Zufallszahlenreihe. Bei einem linearen Kongruenzgenerator nimmt man den Random-Seed (eventuell mit einfachen Operationen manipuliert) als X_0-Wert.

3.1.3 Unabhängigkeit

Damit die Zahlen des Generators unabhängig sind, darf nicht x_n, sondern nur ein Teil davon ausgegebenen werden, etwa eine geeignete Anzahl der niederwertigsten Bits, nur jedes zweite Bit, oder zwei Bits paarweise XOR-verknüpft. Durch diese Verkleinerung des Ausgabebereichs, vergrößert sich die „Unabhängigkeit“.

Angenommen es würde x_n komplett ausgegeben, dann würde z.B. nach der Zahl 100 stets die 1037 ausgeliefert. Wenn nur jedes zweite Bit zurückgegeben wird, dann folgt stattdessen auf 10 die 35. Wenn jetzt irgendwann noch einmal die 10 kommt, dann muss nicht zwangsläufig als nächste Zahl die 35 kommen, da die 10 entstanden sein könnte, indem aus der Zahl 204 jedes zweite Bit genommen wurde. Bei x_n = 204 wird ganz sicher der LCG nicht 1037 ausgeben.

3.1.4 Beispielparameter

Zum Selbstbau (für Programmiersprachen ohne Random-Library) gebe ich an dieser Stelle drei Beispiel-Generatoren an. Die ersten beiden werden in [Knut98, S. 44] als „zufriedenstellend“ beurteilt, der dritte ist „Generator von SIMSCRIPT II.5“. [Zan97]. Alle drei weisen eine maximale Periode von m-1 auf:

Generator 1: a = 3141592653, b = 2718281829, m = 2^{35} = 34359738368

Generator 2: a = 23, b = 0, m = 10^8 + 1 = 100000001

Generator 3: a = 630360016, b = 0, m = 2^{31}-1= 2147483647

In Java stehen für den Datentyp “long” 8 Byte und für „int“ 4 Byte zur Verfügung. Wer ohne Langzahlarithmetik auskommen möchte, bedient sich am besten folgender Parameter, die aus Unix bekannt sind ([Zan97]):

Generator 4: a = 1103515245, b = 12345, m = 2^{32} = 4294967296

Hierbei ist der Modulus mit 2^{32} so gewählt, dass seine Berechnung durch Shiften sehr schnell durchgeführt werden kann, allerdings auf Kosten der Periodenlänge, die sich auf (m/4) verkürzt. Man muss mit dem unter C bekannten Datentyp „unsigned int“ arbeiten, der negative Zahlen ausschließt, um genug Speicherplatz für die Multiplikation zu erhalten. In C++ findet sich unter http://www.iwr.uni-heidelberg.de/~Peter.Bastian/inf1/kap3a.ps.gz, S. 97 ein Beispielzufallszahlengenerator ohne lange Arithmetik.

3.2 Lineares Schieberegister mit Rückkopplung

Auch Schieberegister mit Rückkopplung werden zur Erzeugung von Pseudozufallszahlen verwendet. In der Literatur sind vielseitige Varianten von rückgekoppelten Schieberegis-

terfolgen zu finden. Eine von ihnen wird hier vorgestellt. Die Arbeitsweise des dargestellten Schieberegisters kann folgendermaßen beschrieben werden:

- „Immer wenn ein Pseudozufallsbit benötigt wird, selektiert man das niederwertigste Bit des Schieberegisters b1, das zum Ausgabebit wird.
- Das so gewonnene Ausgabebit wird mit einem Maskenwort (das gleich viele Bits wie das Schieberegister hat) UND verknüpft.
- Das Ergebnis dieser UND-Verknüpfung wird mit den einzelnen Bits des Schieberegisters XOR verknüpft. (Wenn das Ausgabebit 1 ist, werden die Maskenbits auf die XOR-Verknüpfung durchgeschaltet, ist das Ausgabebit 0, werden die Maskenbits ausgeblendet.)
- Das Ergebnis dieser XOR-Verknüpfung wird wieder in die Schieberegisterbits eingespeichert.
- Nach Ausführung dieser Rückkopplungsfunktion werden alle Bits des Schieberegisters um eine Stelle nach rechts geschoben. Das höchstwertige Bit erhält den Wert des Ausgabebits.“ [Pseu01].

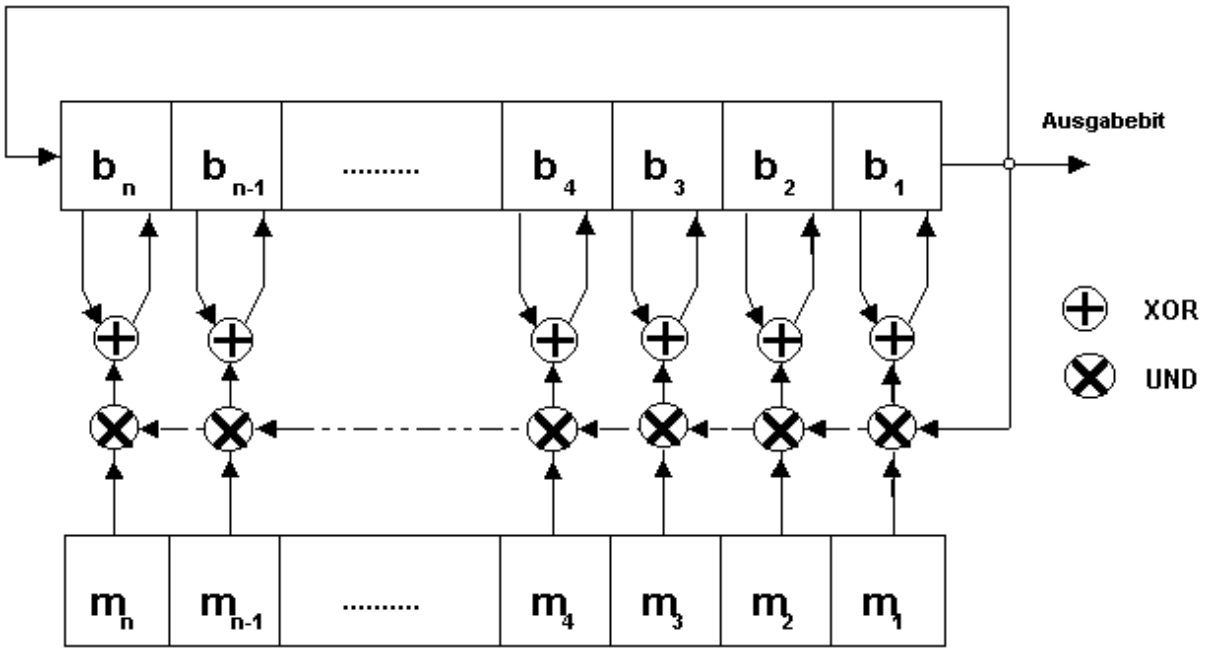


Abbildung 3: Schieberegister mit Rückkopplungsfunktion, aus [Pseu01]

Das Schieberegister lässt sich auch mit Formeln beschreiben:

Ausgabebit = b_1

b_1 = (m_2 AND Ausgabebit) XOR b_2

b_2 = (m_3 AND Ausgabebit) XOR b_3

b_3 = (m_4 AND Ausgabebit) XOR b_4

...

b_{n-1} = (m_n AND Ausgabebit) XOR b_n

$b_n = b_1$

Das Maskenbit m_1 ist, wie aus der Formelaufstellung ersichtlich, belanglos. Das Ergebnis bleibt das gleiche, ob m_1 0 ist oder 1. Wenn alle Bits des Schieberegisters b_1 bis einschließlich b_n den Wert 0 haben, sind alle folgenden Werte des Schieberegisters ebenfalls 0. Hat das Schieberegister n Bits, so können maximal (2n-1) von Null verschiedene Zustände auftreten, die periodisch durchlaufen werden.
Die Anzahl der Zustände werden durch die Maskenbits m_2 bis m_n bestimmt.
Sind sehr viele Bits des Maskenwortes 0, und nur wenige 1, so spricht man von einer "dünn besetzten" Rückkopplungsfunktion. "Dicht besetzte" Rückkopplungsfunktionen sind kryptografisch schwerer zu knacken als "dünn besetzte". Dies lässt sich ansatzweise folgendermaßen erklären: Wenn alle Maskenbits den Wert 0 haben, werden die Bits im Schieberegister im Ring geschoben. Damit können im Schieberegister maximal (n-1) verschiedene Zustände auftreten. Das heißt nach nur wenigen Durchläufen lässt sich ohne Kenntnis der Rückkopplungsfunktion auf das folgende Pseudozufallsbit schließen. Je "dicht besetzter" die Rückkopplungsfunktion, desto stärker werden die einzelnen Bits bei jedem Durchlauf "verwürfelt". Diese Aussage ist allerdings mit Vorsicht zu genießen, es gibt auch durchaus "dicht besetzte" Rückkopplungsmasken, die eine nur geringe Periode aufweisen. Als besonders kritisch erweisen sich gleichmäßige Masken, z.B. „001001001...“.
An dieser Stelle können die Sicherheitsaspekte nur angerissen werden. Als weiterführende und vertiefende Literatur sei hier auf [Schn96] verwiesen.

4 Beispiele für Sicherheit mit Zufallszahlen

In diesem Kapitel soll deutlich gemacht werden, wie wichtig Zufallszahlen und deren Generierung für sicherheitsrelevante Programme sind. In allen Fällen geht es darum, dass ein potentieller Angreifer den Algorithmus und die internen Werte eines schwachen Pseudozufallsgenerators herausfinden kann, und somit die nächste Zufallszahl in einer Sequenz errechnen kann. Bei guten Pseudozufallsgeneratoren bzw. echten Zufallsgeneratoren ist es dem Angreifer nicht möglich, die nächste Zahl zu errechen. Hier bleibt ihm nur die Brute-Force-Attacke übrig. Diese wird ihn aber aufgrund der sehr großen Anzahl von Möglichkeiten und der damit verbundenen sehr geringen Wahrscheinlichkeit für einen „Volltreffer“ nicht zufrieden stellen. Ein Programm mit einem Zufallsgenerator, der „unvorhersehbare“ Zufallszahlen erzeugt, kann somit als „sicher“ eingestuft werden.

4.1 TCP/IP Inital Sequence Number (Pseudozufall)

Für eine TCP/IP-Verbindung von einem Client zu einem Host, generiert der Host eine sogenannte Initial Sequence Number (ISN). Diese wird benötigt, um jedes Packet einer Verbindung nachverfolgen zu können und um sicherzustellen, dass die Verbindung richtig fortgeführt werden kann. Sowohl Client als auch Host benutzen diese Sequenznummern innerhalb von TCP/IP-Verbindungen.

Seit 1985 sind Spekulationen entfacht worden, dass ein Angreifer durch Erraten der nächsten ISN eine unidirektionale Verbindung zu einem Host aufbauen könne. Mittels IP-Spoofing[1] der Quell-IP-Adresse wäre es dem Hacker möglich, mit dem Host zu kommunizieren, obwohl er keine Empfangsbestätigungspakete („acknowledgement pakets“), die die ISN beinhalten, empfangen kann, da diese an die gespoofte IP geschickt werden. Es wurde festgelegt, dass jedem Stream eine eindeutige, zufällige Sequenznummer zugeordnet wird, um die Integrität einer TCP/IP-Verbindung zu gewährleisten. Das TCP-Sequenznummernfeld kann einen 32-Bit-Wert beinhalten. Die RFC-Spezifikationen schreiben einen Wert von 31 Bit vor.

Ein Angreifer, der beabsichtigt, eine Verbindung von einer vorgetäuschten IP-Adresse aufzubauen oder eine existierende Verbindung zu kompromittieren, indem er manipulierte Pakete in den TCP-Stream einschleust, muss die ISN wissen. Aus der Tatsache heraus, dass dem Internet offene Strukturen zugrunde liegen und die meisten Protokolle keine kryptografischen Mechanismen benutzen, um Datenintegrität zu gewährleisten, ist es wichtig, dass TCP/IP-Implementierungen der Gestalt designt werden, dass es einem Angreifer unmöglich ist, die ISN vorherzusehen und eine „Blind Spoofing Attack“ zu fahren.

[1] **IP-Spoofing** ist eine Technik, um nicht autorisierten Zugang zu Computern zu erlangen, wobei der Eindringling eine Nachricht schickt und sie so manipuliert, als käme sie von einem vertrauenswürdigen Host. Um sich IP-Spoofing zu Nutze zu machen, muss der Hacker zunächst mit diversen Techniken eine IP-Adresse eines vertrauenswürdigen Hosts heraus finden und dann die Header seiner eigenen IP-Pakete so manipulieren, dass sie vortäuschen, von diesem Host zu stammen. Aktuelle Router und Firewalls bieten Schutzmechanismen gegen IP-Spoofing-Attacken.

Wie in Kapitel 2.2 bereits erwähnt, ist es schwer, computererzeugten Zufall unvorhersehbar zu machen. Um diesem Mangel Herr zu werden, hat man sich überlegt, eine externe Zufalls- oder Entropiequelle anzuzapfen, die meist über Tasturanschlägeintervalle, spezielle I/O-Interrupts oder andere Parameter, die dem Angreifer nicht zugänglich sind, genährt wird. Diese Lösung - kombiniert mit einer guten Hashfunktion, die 32- oder 31-Bit-Daten ohne Korrelation mit nachfolgenden Ergebnissen, die für Hacker nützliche Informationen über das Innenleben des PRNG-Funktion offenbaren könnte - funktioniert hervorragend, um exzellente TCP-Sequenzgeneratoren zu bauen. Leider werden die TCP-ISN-Generatoren selten unter Berücksichtigung dieser Überlegungen implementiert, und wenn doch gibt es eine Menge Bugs und Fehler, die zu errechenbaren ISNs führen können.
Näheres kann in [Zale01] nachgelesen werden.

4.2 Onetime-Passwort bei SSL (echter Zufall)

Zum webbasierten Online-Banking hat sich die Java-Applet-Technologie im Verlauf der letzten Jahre durchgesetzt. Am Beispiel des Internetbanking-Applets der Kreissparkasse Siegburg soll verdeutlicht werden, dass sich hier die Software-Autoren nicht auf Pseudozufall verlassen, sondern „echten“ Zufall via Mausbewegungen und Tastatureingaben erzeugen. Die Intervalle zwischen Maus- und Tastatureingaben werden dazu benutzt, einen guten Random-Seed für einen PRNG zu generieren. Dieser erzeugt dann ein Oneway-Passwort, das zur symmetrischen Verschlüsselung der Kommunikation zwischen Bankkunde und Rechenzentrum verwendet wird. Einem potientiellen Angreifer bleibt nur das Erraten des Random-Seeds via Brute-Force übrig, um die Kommunikation belauschen und gegebenenfalls sogar manipulieren zu können.
Eine weitere Möglichkeit bietet sich dem Hacker mit der „Man-in-the-middle“-Attacke. Um diese zu verhindern, ist es aus Sicht der Bank sinnvoll, eine Zertifikat-basierte Authentifikation des Onlinebanking-Rechners gegenüber dem Bankkunden zu ermöglichen. Dieses Szenario wird hier nicht weiter beschrieben. In [Schn96] findet sich eine ausführliche Darstellung der Problematik.

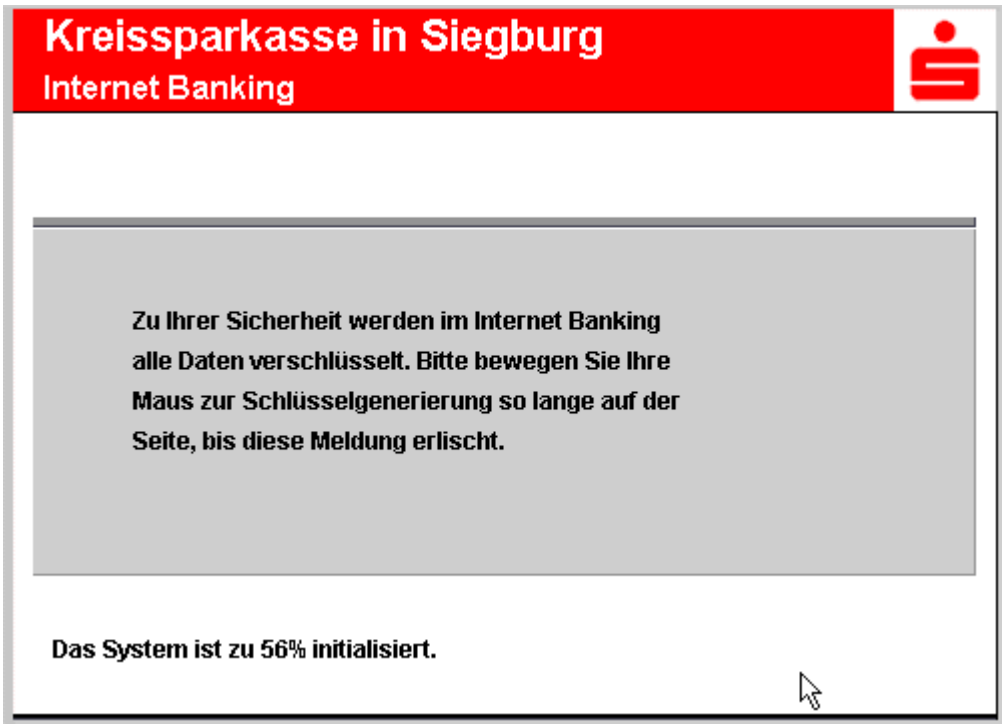


Abbildung 4: Echter Zufall beim Homebanking-Applet der Kreissparkasse Siegburg

5 PRNGs und Tests auf Zufallszahlen

5.1 Java: java.util.Random

Random.java aus dem java.util.*-Package ist ein Pseudorandomgenerator aus der Java-API. Die Funktion Math.random() legt beim ersten Start eines Java-Programms eine neue Instanz der Random-Klasse an bzw. benutzt in späteren Aufrufen die bereits instanzierte Klasse und ruft deren nextDouble()-Methode auf.
Random.java implementiert den in [Knut98, Kapitel 3.2.1] vorgestellten Algorithmus. Dieser stellt einen Lehmer’schen linearen Kongruenzgenerator dar, der anstatt der relativ aufwendig zu berechnenden Modulus-Operation mit Long-Werten, mit Bitschiebeoperationen auskommt. Der Modulus wird hierbei so geschickt wählt, dass Bits, deren Wertigkeit über dem Modulus läge, aus dem „Bitraster“ fallen. Dieses Verfahren ähnelt dem des linearen Schieberegisters aus Kapitel 3.2. Dies wird bewerkstelligt, indem alle Berechnungen mit einem „& mask“ beendet werden. Die final long Variable „mask“ ist auf mask = (1L << 48) – 1 gesetzt. Das heißt, die ersten 47 Bits sind auf 1, die restlichen auf 0 gesetzt. Die eigentliche Randomfunktion, die in vier Zeilen Code passt, ist

```
synchronized protected int next(int bits) {
    long nextseed = (seed * multiplier + addend) & mask;
    seed = nextseed;
    return (int)(nextseed >>> (48 - bits));
}
```

Wir sehen, dass hier mit Multiplikation, Addition und der oben beschriebenen geschickten Modulus-Operation wie beim LCG ausgekommen wird. Zur Sicherheit wird der randomseed noch mit einer xor-Operation verschleiert:

```
synchronized public void setSeed(long seed) {
    this.seed = (seed ^ multiplier) & mask;
}
```

Der long-Übergabewert ist meist die Anzahl der Millisekunden seit dem 1.1.1970 - bekannt durch die interne Darstellung der Unix-Systemzeit. Die Methode nextDouble()

```
public double nextDouble() {
    long l = ((long)(next(26)) << 27) + next(27);
    return l / (double)(1L << 53);
}
```

baut in der long-Variablen l eine 53-Bit-Zufallszahl auf, für die zwei Aufrufe von next() nötig sind, da dort nur je 47 Zufallsbits (siehe „mask“) generiert werden. Die Differenz von 41 zwischen 2 * 47 = 94 und 53 entspricht dem in Kapitel 3.1.3 beschriebenen Unabhängigkeitszuwachs, der bei LCGs nötig ist. L wird abschließend durch 2^{53} geteilt, damit der Rückgabewert des Doubles zwischen 0 und 0,99999... liegt, was der „Uniform Distribution“ im Intervall [0,1[entspricht.
PRNGs haben üblicherweise einen Status, der intern ist und sich bei jedem Aufruf ändert, der aber nicht identisch mit dem Rückgabewert ist. Dadurch kann erreicht werden, dass nicht immer z.B. auf die 17 die 235 folgt, denn der nächste Wert ist noch abhängig vom internen Zustand. In dieser Java-Implementation wird dieser Zustand durch die Variable „seed“ wiedergespiegelt, die bei jeder ausgegebenen Zufallszahl verändert wird, bei nextDouble() durch den zweifachen Aufruf von next() sogar zweimal.
Die Quelltexte von java.lang.Math und java.util.Random sind im Anhang dieser Studienarbeit zu finden. Weitere Informationen liefert die Java-API-Dokumentation [Java01].

5.2 Unix: /dev/(u)random

In der Unixwelt sind /dev/random und /dev/urandom seit langer Zeit bekannt. In der nachfolgenden Kurzvorstellung der Devices beziehe ich mich auf die Linux Manpage zu random, die z.B. unter [Ts97] eingelesen werden kann.
Seit Linux-Kernel 1.3.30 bieten die Gerätedateien /dev/random (major device number 1, minor device number 8) und /dev/urandom (major device number 1, minor device number 9) eine Schnittstelle zum Zufallszahlengenerator des Kernels. Der Zufallszahlengenerator sammelt Umgebungsrauschen von Gerätetreibern und anderen Quellen in einem Entropie-Pool. Hieraus werden die Zufallszahlen generiert. Bei Lesevorgängen auf die Gerätedatei /dev/random werden nur so viele Zufallsbytes ausgegeben, wie im Entropie-Pool zur Verfügung stehen. Damit genügt dieses Device hohen Ansprüchen bezüglich qualitativ gutem Zufall, etwa für Onetime-Pads oder Schlüsselerstellung für kryptografische Verfahren. Sollten sich keine Bytes mehr im Entropie-Pool befinden, wird /dev/random solange geblockt, bis wieder genug Entropie gesammelt wurde.
Im Gegensatz hierzu liefert die Gerätedatei /dev/urandom so viele Bytes aus, wie angefordert wurden. Für den Fall, dass im Entropiespeicher nicht genügend Bytes vorrätig sind, sind die ausgelieferten Werte theoretisch anfällig für Hackerattacken, da sie mit Pseudozufallsgeneratoren erstellt wurden. Wenn man also Wert auf echte Zufallszahlen in seinen Applikationen legt, sollte man ausschließlich /dev/random verwenden.
Die Manpage stellt im Weiteren vor, wie mittels des Unix-Kommandos mknod eine noch nicht vorhandene Zufallsgerätedatei angelegt werden kann. Zudem geht sie auf das Problem ein, dass am Anfang eines Systemstarts der Entropie-Pool (fast) leer ist. Hierzu wird als Tipp mitgegeben, den Entropie-Pool bei jedem Shutdown in eine Datei zu schreiben und bei jedem Systemboot aus dieser Datei wieder in den Entropie-Pool zurückzu-

schreiben. Dies geschieht am einfachsten mit dem Unix-Kommando dd. Aktuelle Linux-Distributionen haben diese Funktionalität in ihren Init-Skripts im Regelfall bereits implementiert.

5.3 Implementierung eines eigenen einfachen PRNG

Mit „RandomGenerator.java“ wurde ein einfacher Pseudozufallsgenerator entwickelt. Der Anhang dieser Arbeit enthält den Sourcecode. Mit „java RandomGenerator <number> <from> <to>” kann ich <number> eigene Zufallszahlen von <from> bis <to> erstellen lassen, die auf Stdout/System.out ausgegeben werden. Die Implementierung erfolgte nach dem Lehmer’schen linearen Kongruenzgenerator.

5.4 Implementierung eines eigenen PZG-Testprogramms

Der Java-Sourcecode der kleinen, selbst entwickelten Testsuite „RandomTester.java“ ist ebenfalls im Anhang einzusehen. Das Testprogramm erwartet mit Aufruf „java RandomTester [filename]“ eine Datei mit Zufallszahlen, wobei jede Zufallszahl in einer eigenen Zeile steht. Zeilen mit „#“ am Anfang oder leere Zeilen werden ignoriert. Mit „java RandomTester <number> <from> <to>” kann ich <number> eigene Zufallszahlen von <from> bis <to> erstellen lassen. „java RandomTester” entspricht „java RandomTester 1000 1 100”. Andere Aufrufe führen zu einer Fehlermeldung. RandomGenerator und RandomTester spielen gut zusammen. Folgende Programmabfolge verbindet beide Programme:

```
java RandomGenerator 10000 0 99 > randoms.txt
java RandomTester randoms.txt
```

Die Testsuite beherrscht den Entropie-Test, den Chi-Quadrat-Test, den Spektraltest und den Run-Test. Die einzelnen Tests werden im Folgenden näher beschrieben.

5.4.1 Entropie-Test

Um die Entropie eines unbekannten Generators zu messen, müssen wir ein Histogramm erstellen, da wir ja die Wahrscheinlichkeiten p_i nicht direkt bestimmen können. Danach fassen wir die relativen Häufigkeiten als Wahrscheinlichkeiten auf. Nehmen wir an, wir würden n Zufallszahlen im Bereich zwischen min und max vorliegen haben. Wir gehen nun die Zufallszahlen der Reihe nach durch. Für jede Zufallszahl haben wir einen Basket b_i angelegt, den wir am Anfang auf Null setzen und beim Auftreten der entsprechenden Zahl um eins erhöhen. Dann ergibt sich als Wahrscheinlichkeit $p_i = b_i / n$ für min <= i <= max. Die Summe über alle p_i sollte somit 1 ergeben.
Die Entropie errechnet sich nach der Formel aus 2.3.1 wie folgt:

$$\text{Entropie} = -\sum p_i * \log_2(p_i)$$

Die maximale Entropie von $\log_2$(min-max+1) ergäbe sich bei einer Gleichverteilung der Zufallszahlen mit p_i = (min-max+1)/n für min <= i <= max.
Wir sehen sofort, dass die Entropie nicht die einzige wichtige Maßzahl zur Beurteilung einer Zufallszahlenreihe ist: Ein Generator, der immer 1 2 3 4 5 1 2 3 4 5 1 2 3 ... ausgibt, hat nach dem Verfahren zwar maximale Entropie, aber die Ergebnisse sind nicht unabhängig voneinander. Also benötigt man weiter gehende Tests, die Entropie sagt nur etwas darüber, wie "gleichmäßig" die Zahlen verteilt sind. Die Reihenfolge der Zahlen spielt keine Rolle und wird gänzlich ignoriert.

5.4.2 x^2-Test

Der Chi-Quadrat-Test prüft, ob die erzeugten Zufallszahlen sinnvoll verteilt sind oder nicht, indem er die Abweichung der gemessenen Verteilung von einer angenommenen Gleichverteilung ermittelt. Erzeugt ein Zufallsgenerator n positive Zahlen mit k möglichen Werten, so ist zu erwarten, dass jeder Wert in etwa (n/k) mal auftaucht. Jedoch sollten die einzelnen Häufigkeiten nicht identisch sein.
Wie beim Entropietest wird auch hier mit Baskets gearbeitet. Statt einer logarithmischen Operation wird hier mit dem Quadrat der Zahl gearbeitet.

Die Formel lautet: $$Chi = \frac{k}{n} \sum_{1 \le i \le k} \left(b_i^2 \right) - n$$

Das Verfahren ist gut, wenn |Chi – k| <= 2 * sqrt(k) gilt.

5.4.3 Spektraltest

Aufgrund der Komplexität des Spektraltests wird an dieser Stelle auf eine Eigenimplementierung verzichtet. Eine komplette Testsuite, die u.a. auch den Spektraltest beinhaltet, mit veröffentlichten Quelltexten kann unter http://csrc.nist.gov/rng/ heruntergeladen werden.

5.4.4 Run-Test

Die Implementierung meines Run-Tests ermittelt lediglich den längsten Run-up und den längsten Run-down. Die Interpretation dieser Werte sei dem Leser überlassen. Als grobe Richtwerte sollten die Ergebnisse für längere Zufallsreihen aber zwischen 4-6 liegen. 2 ist zu klein und über 10 sollte nur ein einziger Test aus vielen anderen 1000 kommen.

6 Zusammenfassung

6.1 theoretischer Teil

Es gibt zwei Arten von Zufall: **„echten“ und computergenierten Zufall.** Beispiele für echten Zufall, deren Daten aus nicht-deterministischen Quellen stammen, sind Münzwerfen („Kopf“ oder „Zahl“), Ziehung der Lottozahlen mit einer Maschine oder Würfeln. Im Bereich der Physik können auch Messungen im Zusammenhang mit der Erzeugung und des Registrierens von Zählimpulsen, z.B. beim radioaktiven Zerfall, echten Zufall erzeugen, im Computerbereich können Tastaturanschläge und Mausbewegungen mit in die Berechnungen einfließen. Manchmal ist es zu aufwendig, reale Zufallszahlen zu erzeugen. Dann verwendet man sogenannte Pseudozufallsgeneratoren. Dies ist ein Algorithmus, der aus einer kurzen Folge von Zufallszahlen, eine lange Folge berechnet, die zufällig „aussieht“, die also nicht in Polynomzeit von einer wirklichen zufälligen Folge unterschieden werden kann. Computererzeugte Zufallszahlen sind periodisch. Das heißt, nach einer gewissen Zeit folgen die Zahlen in einer schon zuvor produzierten Reihenfolge. Hieraus folgt, dass Zufallszahlen theoretisch vorhersehbar sind. Zufallszahlen brauchen wir heute in verschiedenen Bereichen. Anwendung finden Pseudozufallsgeneratoren bei statistischen Simulationen, numerischen Analysen, unvoreingenommene Entscheidungsfindungen (auch für optimale Strategien bei Computerspielen) und in der Computerprogrammierung. Der letztgenannte Punkt ist sehr umfangreich. Hier denke man insbesondere an kryptografische Algorithmen, Tests von Effektivität und Effizienz von Programmen und Computerspiele mit Würfeln, Rouletterädern oder gemischten Spielkarten. Für kryptografische Verfahren (Schlüsselgenierung bei Public-Key-Verschlüsselung, Erzeugung von Onetime-Pads) bedient man sich heutzutage aus sicherheitstechnischen Gründen echtem Zufall.

Es gibt fünf **Eigenschaften von „guten“ Zufallszahlen**. Diese sollen nämlich gleichverteilt, unabhängig, mit großer Periode errechnet, reproduzierbar und einfach und schnell (effizient) berechenbar sein. Für sicherheitsrelevante Programme ist es unumgänglich, unvorhersehbare Zufallszahlen erzeugen zu können. Diese sind mit einem Computer aber nur schwer zu generieren. Das liegt daran, weil Rechner so gebaut sind, dass sie nur eine Menge genau definierter Kommandos ausführen können, die bei nochmaligem Aufruf nach genau demselben Schema abgearbeitet werden und dieselben Ergebnisse produzieren. Dadurch kann jeder fixe Algorithmus dazu benutzt werden, genau dieselben Ergebnisse auf einem anderen Computer nachzustellen.

Die Gleichverteilung von Zufallszahlen kann man über die **Entropie** beschreiben. Um Entropie mathematisch zu beschreiben, betrachten wir einen Zufallsgenerator, der n verschiedene Zahlen ausgibt, als Informationsquelle mit endlichem Alphabet $A = (a_1, a_2, ..., a_n)$. Die Wahrscheinlichkeit des Generators, die Zahl a_i auszugeben, nennen wir p_i. Dann ist die negierte Summe über $p_i * \log_2(p_i)$ die Entropie dieser Informationsquelle. ($\log_2(x)$ für $0<x<1$ ist negativ, somit muss die Summe negiert werden, um ein positives Ergebnis

zu erhalten.) Sie gibt ein Maß für den "Informationsgehalt" der Quelle. Die Quelle hat keinen Informationsgehalt, weil wir nicht wissen, welche Information als nächstes kommt. Damit gibt die Entropie wieder, wie "überraschend" ein Zeichen kommen kann, und beim Zufall wollen wir, dass die "Überraschung" maximal ist.

Mathematisch wird der **x^2-Test** (Chi-Quadrat-Test) bewerkstelligt, indem die Summe der Quadrate der einzelnen Häufigkeiten Y_s durch die zu erwartende Häufigkeit $n*p_s$ dividiert wird. (Das n kann vor das Summenzeichen gezogen werden.) Danach wird die Größe der Folge subtrahiert: $V = \frac{1}{n} \sum_{1 \le s \le k} \left(\frac{Y_s^2}{p_s} \right) - n$

Der **Spektraltest** ist ein sehr mächtiger und daher auch sehr komplexer und schwer verständlicher Zufallszahlenreihentest. Die Bedeutung des Spektraltests ist sehr groß und er wird daher oft eingesetzt. Der von Knuth beschriebene Spektraltest richtet sich ausschließlich auf Zufallszahlenreihen, die mittels des LCG generiert wurden. Er untersucht, wie stark eine Zufallszahl von ihrem Vorgänger abhängt. Ein LCG mit etwa der Formel $X_{n+1} = (137\ X_n + 187) \bmod 256$ ergibt ein Diagramm, dass parallele Linien etwa im Winkel 30° aufweist, die teilweise noch im gleichen Abstand zueinander stehen. Sollte sich die Existenz solcher Muster auch bei größeren Moduli beweisen lassen, besteht der Generator den Spektraltest nicht.

Mittels des **Run-Tests** überprüft man, wie viele Zahlen hintereinander ohne Unterbrechung aufwärts ansteigen („run-up“) bzw. abwärts abfallen („run-down“). Mit dieser Methode kann ich trotz etwa eines bestandenen Entropietests mit maximaler Entropie eine schlechte Zufallsreihenfolge nachweisen. Zur Interpretation des Run-Tests ist es nötig, in Tabellen nachzuschauen, die ausschließlich für Run-Tests angefertigt wurden, wie gut (relativ wahrscheinlich) oder schlecht (relativ unwahrscheinlich) eine Zufallsfolge ist.

Eine bekannte Art und Weise, Zufallszahlen zu generieren, sind die **linearen Kongruenzgeneratoren** (LCGs) nach Lehmer. Sie sind nach der Formel

$X_n = (a * X_{n-1} + b) \bmod m$ mit $ggt(b, m) = 1,\ 0 < a < m,\ 0 < b < m$

X_0 = Startwert, a = Multiplikator, b = Inkrement, m = Modulus

aufgebaut. Der Modulus m sollte sehr hoch gewählt werden, da die Periode der Zufallszahlenreihe höchstens m betragen kann. Dies liegt daran, dass die Zufallszahl nur von der vorherigen rechnerisch abhängt. Eine Zahl, die das zweite Mal in der Reihe vorkommt, ist damit automatisch Beginn der Periode. Für $m > 10^8$ spricht Knuth von zufriedenstellenden Perioden. Damit die Zahlen des Generators unabhängig sind, darf nicht x_n, sondern nur ein Teil davon ausgegebenen werden, etwa eine geeignete Anzahl der niederwertigsten Bits, nur jedes zweite Bit, oder zwei Bits paarweise XOR-verknüpft. Durch diese Verkleinerung des Ausgabebereichs, vergrößert sich die „Unabhängigkeit“. Wer ohne Langzahlarithmetik einen LCG nachbauen möchte, bedient sich am besten der Parameter a = 1103515245, b = 12345, $m = 2^{32} = 4294967296$. Hierbei ist der Modulus mit

2^{32} so gewählt, dass seine Berechnung durch Shiften sehr schnell durchgeführt werden kann, allerdings auf Kosten der Periodenlänge, die sich auf (m/4) verkürzt.
Auch **Schieberegister mit Rückkopplung** werden zur Erzeugung von Pseudozufallszahlen verwendet. In der Literatur sind vielseitige Varianten von rückgekoppelten Schieberegisterfolgen zu finden. Die Arbeitsweise des dargestellten Schieberegisters kann folgendermaßen beschrieben werden: Immer wenn ein Pseudozufallsbit benötigt wird, selektiert man das niederwertigste Bit des Schieberegisters b1, das zum Ausgabebit wird. Das so gewonnene Ausgabebit wird mit einem Maskenwort (das gleich viele Bits wie das Schieberegister hat) UND-verknüpft. Das Ergebnis dieser UND-Verknüpfung wird mit den einzelnen Bits des Schieberegisters XOR verknüpft. Das Ergebnis dieser XOR-Verknüpfung wird wieder in die Schieberegisterbits eingespeichert. Nach Ausführung dieser Rückkopplungsfunktion werden alle Bits des Schieberegisters um eine Stelle nach rechts geschoben. Das höchstwertige Bit erhält den Wert des Ausgabebits.

6.2 praktischer Teil

Für eine **TCP/IP-Verbindung** von einem Client zu einem Host, generiert der Host eine sogenannte **Initial Sequence Number** (ISN). Diese wird benötigt, um jedes Packet einer Verbindung nachverfolgen zu können und um sicherzustellen, dass die Verbindung richtig fortgeführt werden kann. Sowohl Client als auch Host benutzen diese Sequenznummern innerhalb von TCP/IP-Verbindungen. Seit 1985 sind Spekulationen entfacht worden, dass ein Angreifer durch Erraten der nächsten ISN eine unidirektionale Verbindung zu einem Host aufbauen könne. Mittels IP-Spoofing der Quell-IP-Adresse wäre es dem Hacker möglich, mit dem Host zu kommunizieren, obwohl er keine Empfangsbestätigungspakete („acknowledgement pakets“), die die ISN beinhalten, empfangen kann, da diese an die gespoofte IP geschickt werden. Es wurde festgelegt, dass jedem Stream eine eindeutige, zufällige Sequenznummer zugeordnet wird, um die Integrität einer TCP/IP-Verbindung zu gewährleisten. Das TCP-Sequenznummernfeld kann einen 32-Bit-Wert beinhalten. Die RFC-Spezifikationen schreiben einen Wert von 31 Bit vor. Ein Angreifer, der beabsichtigt, eine Verbindung von einer vorgetäuschten IP-Adresse aufzubauen oder eine existierende Verbindung zu kompromittieren, indem er manipulierte Pakete in den TCP-Stream einschleust, muss die ISN wissen.
Der Zufallszahlengenerator der Unix-Devices **/dev/random und /dev/urandom** sammelt Umgebungsrauschen von Gerätetreibern und anderen Quellen in einem Entropie-Pool. Hieraus werden die Zufallszahlen generiert. Bei Lesevorgängen auf die Gerätedatei /dev/random werden nur so viele Zufallsbytes ausgegeben, wie im Entropie-Pool zur Verfügung stehen. Damit genügt dieses Device hohen Ansprüchen bezüglich qualitativ gutem Zufall, etwa für Onetime-Pads oder Schlüsselerstellung für kryptografische Verfahren. Sollten sich keine Bytes mehr im Entropie-Pool befinden, wird /dev/random solange ge-

blockt, bis wieder genug Entropie gesammelt wurde. Im Gegensatz hierzu liefert die Gerätedatei /dev/urandom so viele Bytes aus, wie angefordert wurden.

Literaturliste

[Pseu01] “Pseudozufallsgeneratoren”, Weiß C., Essig M. und Henisch S., Studienarbeit, http://rhlx01.rz.fht-esslingen.de/projects/krypto/pseudo/pseudo.html, 2001

[PRNG] “(Pseudo-)Zufallsgeneratoren“, Brachtl M. und Meixner A., http://www.cosy.sbg.ac.at/~ameixner/PRNG/, Datum unbekannt

[Buch99] „Einführung in die Kryptographie“, Buchmann J., Springer-Verlag, 1999

[Stoc00] „Stochastische Simulation“, Autor unbekannt, Vorlesungsskript, http://dsor.uni-paderborn.de/de/material/online/simweb/simweb2000/Vorlesung/stochastische/stochastische.htm, 2000

[Erze98] “Erzeugung von Zufallszahlen mit vorgegebener Verteilung und Test auf statistische Abhängigkeit”, Schenk E. und Wörz M., Studienarbeit, http://www.it.fht-esslingen.de/~schmidt/vorlesungen/inco/seminar/ss98/html/04/ict_04.html, 1998

[Knut98] “The Art of Computer Programming – Seminumerical Algorithms”, Band 2, Knuth D., Addison Wesley, 1998

[Schn96] “Angewandte Kryptographie - Einführendes Lehrbuch, Protokolle, Algorithmen, C-Programme”, Schneier B., Addison Wesley, 1996

[Zale01] “Strange Attractors and TCP/IP Sequence Number Analysis”, Zalewski M., http://razor.bindview.com/publish/papers/tcpseq.html, 2001

[Clo97] „Diskrete Simulation“, Müller-Clostermann B., Hintelmann J., Begleitmaterial zu einer Lehrveranstaltung, http://www.informatik.uni-essen.de/Lehre/Material/DiskreteSim/Skript/ds_skript_1.html, 1997

[Java01] Java™ 2 Platform, Standard Edition, v 1.3.1 API Specification, Sun Microsystems, http://java.sun.com/j2se/1.3/docs/api/, 2001

[Ts97] Linux-Manpage zu Random/Urandom, Ts T., http://www.linux.gr/cgi-bin/man2html/usr/share/man/man4/random.4.gz, 1997

[RFC1750] Request for Comments (RFC) 1750, “Randomness Recommendations for Security”, Eastlake D., Crocker S., Schiller J., z.B. http://www.rfc.net/rfc1750.html, 1994

[ZVE98] „Zufall, Vorhersagbarkeit, Entropy“, „Scut“, http://segfault.net/~scut/articles/random/, 1998

[Zan97] „Theorie des Zufalls - mathematische Grundlagen“, Zankl W. M., http://www.gambling-hall.com/zufall/zufall.htm, 1997

Anhang

Sourcecode von RandomGenerator.java

```
/*
 * Author:    Jochen Rondorf
 * Date:      2002-01-12
 * Version:   1.0
 *
 * Copyright: 2002 by J.Rondorf
 */

import java.util.*;

class RandomGenerator {
    static private int from = 1;
    static private int to = 100;
    static private int amount = 1000;
    static private MyRandom mr;

    public static void main (String args[]) {
        if (args.length == 3){
            try {
                amount = (new Integer(args[0])).intValue();
                from = (new Integer(args[1])).intValue();
                to = (new Integer(args[2])).intValue();
            } catch (NumberFormatException e) { usage(); }
        }

        if ((args.length != 0) && (args.length != 3)) usage();

        System.out.println("# "+amount+" x from "+from+" to "+to);
        for (int i = 0; i < amount; i++) {
            int r = rndFromTo(from, to+1);
            System.out.println(r);
        }
    }

    static void usage(){
        System.err.println("Usage: java RandomGenerator [<number> <from> <to>]");
       System.exit(1);
    }

    public static int rndFromTo(int from, int to){
        if (mr == null) mr = new MyRandom();
        double nd = mr.nextDouble();
        return from + (int)(nd * (to-from));
    }

    // inner class: PRNG as simple alternative for java.util.Random
    static class MyRandom {
        final static private long a = 3141592653L;
        final static private long b = 2718281829L;
        final static private long m = 34359738368L; // m = 2^35
        final static private long mask = (1L << 28) - 1;

        private long x;

        MyRandom(long x0){
            x = x0;
        }

        MyRandom(){
            long ctm = System.currentTimeMillis();
            x = ctm % m;
        }

        public double nextDouble(){
            // next step
            double nextX = ((double)a * x + b) % m;
            x = (long)nextX;

            // decrease modulo to add some independence
            return (double)(x & mask ) / (mask + 1);
        }
    }
}
```

Sourcecode von RandomTester.java

```
/*
 * Author:    Jochen Rondorf
 * Date:      2002-01-12
 * Version:   1.0
 *
 * Copyright: 2002 by J.Rondorf
 */

import java.util.*;
import java.io.*;

class RandomTester {
    private static int amountRandoms;   // length of al
    private static int smallestRandom;  // min random found
    private static int biggestRandom;   // max random found
    private static HashMap hm;          // baskets for randoms
    private static int[] randoms;       // random array

    public static void main (String args[]) {
        int params = args.length;
        switch (params) {
            case 0:
                // example values
                amountRandoms = 1000;
                smallestRandom = 1;
                biggestRandom = 100;

                // no filename, create own randoms
                createOwnRandoms();
              break;

            case 1:
                // filename available, create randoms from file
                getRandomsFromFile(args[0]);
               break;

           case 3:
                // example values
                try {
                    amountRandoms = (new Integer(args[0])).intValue();
                    smallestRandom = (new Integer(args[1])).intValue();
                    biggestRandom = (new Integer(args[2])).intValue();
                } catch (NumberFormatException e) {
                    usage();
                }

                // no filename, create own randoms
                 createOwnRandoms();
              break;

           default:
               usage();
      }

        makeEntropyTest();
        makeChiSquareTest();
        makeRunTest();
    }

    static void usage(){
        System.err.println("Usage: java RandomTester [<filename>]");
        System.err.println("       java RandomTester <number> <from> <to>");
        System.exit(1);
    }

    static void makeRunTest(){
        int longestDown = 0;
        int longestUp = 0;
        int last = randoms[0];
        int down = 0;
        int up = 0;
        int posUp = 0;
        int posDown = 0;
        for (int i = 1; i < randoms.length; i++){
            if (randoms[i] <= last){
                down++;
```

```
                if (up > longestUp) {
                    longestUp = up;
                    posUp = i;
                }
                up=0;
            }
            if (randoms[i] >= last){
                up++;
                if (down > longestDown){
                    longestDown = down;
                    posDown = i;
                }
                down=0;
            }
            last = randoms[i];
        }
        System.out.println("\nlongest down-run: " + longestDown + "(" + posDown + ")");
        System.out.println("longest up-run: " + longestUp + "(" + posUp + ")");
    }

    static void makeEntropyTest(){
        double entro = 0;
        double warSum = 0;
        int c = 0;
        for (int i = smallestRandom; i <= biggestRandom; i++){
            double war;
            try {
                war = ((double)((Integer)hm.get(new Integer(i))).intValue()) / amountRandoms;
                if (c++ < 20) // only print first 20
                    System.out.println(i + " -> " + hm.get(new Integer(i)) + " ("+ war + ")");
            } catch (NullPointerException npe) {
                war = 0d;
            }
            warSum += war;
            entro -= (war == 0) ? 0d : war * log(2, war);
        }

        entro = Math.round(1000*entro)/1000d;
        System.out.println("\n\nEntropy: " + entro);

        double war = 1d / amountRandoms;
        int basketSize = (biggestRandom - smallestRandom + 1);
        double mpe = log(2, basketSize);
        mpe = Math.round(1000*mpe)/1000d;
        System.out.println("Max possible Entropy: " + mpe);
    }

    static void makeChiSquareTest(){
        double chi = 0;
        int c = 0;
        for (int i = smallestRandom; i <= biggestRandom; i++){
            int times;
            try {
                times = ((Integer)hm.get(new Integer(i))).intValue();
                c++;
            } catch (NullPointerException npe) {
                times = 0;
            }
            chi += times * times;
        }
        double chiResult = chi*c/(double)amountRandoms - amountRandoms;
        String result;
        if (chiResult < c) chiResult = 2*c - chiResult;
        if ((chiResult - c) < (2*Math.sqrt(c)))
           result = "good";
        else
           result = "bad";

        System.out.println("\nChi-Square: " + chiResult + " (" + result + ")");
    }

    static void createOwnRandoms(){
        randoms = new int[amountRandoms];
        hm = new HashMap();
       for (int i = 0; i < amountRandoms; i++){
            int r = (int)rndFromTo(smallestRandom, biggestRandom);
            randoms[i] = r;
            Integer s = new Integer(r);
```

```
            Integer n = (hm.containsKey(s)) ? (Integer)(hm.get(s)) : new Integer(0);
            n = new Integer(n.intValue()+1);
            hm.put(s, n);
        }
    }

    public static void getRandomsFromFile (String filenameString){
        File filename = new File(filenameString);
        FileInputStream in = null;
        String line="";
        hm = new HashMap();
        ArrayList al = new ArrayList();
        biggestRandom = 0;
        smallestRandom = 0;
        amountRandoms = 0;
        try {
            in = new FileInputStream(filename);
            BufferedReader bor = new BufferedReader(new InputStreamReader(in));
            do {
                line = bor.readLine().trim();
                if ((line != null) && (line != "") && !line.startsWith("#"))  {
                    Integer s = new Integer(line);
                    al.add(s);
                   Integer n = (hm.containsKey(s)) ? (Integer)(hm.get(s)) : new Integer(0);
                    n = new Integer(n.intValue()+1);
                    hm.put(s, n);

                    // count amount
                    amountRandoms++;

                    // check for max and min and save
                    int ss = s.intValue();
                    if (biggestRandom < ss) biggestRandom = ss;
                    if (smallestRandom > ss) smallestRandom = ss;
                }
            } while (line != null);
        }
        catch (FileNotFoundException e) {
            System.err.println(e);
            System.exit(1);
        }
        catch (Exception e) {}
        finally {
            try { in.close(); } catch (Exception e) {}
        }

        // build randoms array, after got known, how many there are
        randoms = new int[amountRandoms];
        Iterator iter = al.iterator();
        int c = 0;
        while (iter.hasNext()) { // can't get over amountRandoms
            Integer t = (Integer)iter.next();
            randoms[c++] = t.intValue();
        }
       }

    public static double log(int base, double value){
        return Math.log(value)/Math.log(base);
    }

    public static double rndFromTo(int from, int to){
        return (double)from + Math.random() * (++to-from);
    }
}
```

Sourcecode von java.lang.Math

```
/*
 * @(#)Math.java    1.39 99/04/22
 *
 * Copyright 1994-1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
```

```
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

package java.lang;
import java.util.Random;

/**
 * The class <code>Math</code> contains methods for performing basic
 * numeric operations such as the elementary exponential, logarithm,
 * square root, and trigonometric functions.
 * <p>
 * To help ensure portability of Java programs, the definitions of
 * many of the numeric functions in this package require that they
 * produce the same results as certain published algorithms. These
 * algorithms are available from the well-known network library
 * <code>netlib</code> as the package "Freely Distributable
 * Math Library" (<code>fdlibm</code>). These algorithms, which
 * are written in the C programming language, are then to be
 * understood as executed with all floating-point operations
 * following the rules of Java floating-point arithmetic.
 * <p>
 * The network library may be found on the World Wide Web at:
 * <blockquote><pre>
 *   <a href="http://metalab.unc.edu/">http://metalab.unc.edu/</a>
 * </pre></blockquote>
 * <p>
 * The Java math library is defined with respect to the version of
 * <code>fdlibm</code> dated January 4, 1995. Where
 * <code>fdlibm</code> provides more than one definition for a
 * function (such as <code>acos</code>), use the "IEEE 754 core
 * function" version (residing in a file whose name begins with
 * the letter <code>e</code>).
 *
 * @author  unascribed
 * @version 1.39, 04/22/99
 * @since   JDK1.0
 */

public final class Math {

    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    [...]

    private static Random randomNumberGenerator;

    /**
     * Returns a random number greater than or equal to <code>0.0</code>
     * and less than <code>1.0</code>. Returned values are chosen
     * pseudorandomly with (approximately) uniform distribution from that
     * range.
     * <p>
     * When this method is first called, it creates a single new
     * pseudorandom-number generator, exactly as if by the expression
     * <blockquote><pre>new java.util.Random</pre></blockquote>
     * This new pseudorandom-number generator is used thereafter for all
     * calls to this method and is used nowhere else.
     * <p>
     * This method is properly synchronized to allow correct use by more
     * than one thread. However, if many threads need to generate
     * pseudorandom numbers at a great rate, it may reduce contention for
     * each thread to have its own pseudorandom-number generator.
     *
     * @return  a pseudorandom <code>double</code> greater than or equal
     * to <code>0.0</code> and less than <code>1.0</code>.
     * @see     java.util.Random#nextDouble()
     */
    public static synchronized double random() {
        if (randomNumberGenerator == null)
            randomNumberGenerator = new Random();
        return randomNumberGenerator.nextDouble();
    }
    [...]
}
```

Sourcecode von java.util.Random

```
/*
 * @(#)Random.java  1.27 99/04/22
 *
 * Copyright 1995-1999 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

package java.util;

/**
 * An instance of this class is used to generate a stream of
 * pseudorandom numbers. The class uses a 48-bit seed, which is
 * modified using a linear congruential formula. (See Donald Knuth,
 * <i>The Art of Computer Programming, Volume 2</i>, Section 3.2.1.)
 * <p>
 * If two instances of <code>Random</code> are created with the same
 * seed, and the same sequence of method calls is made for each, they
 * will generate and return identical sequences of numbers. In order to
 * guarantee this property, particular algorithms are specified for the
 * class <tt>Random</tt>. Java implementations must use all the algorithms
 * shown here for the class <tt>Random</tt>, for the sake of absolute
 * portability of Java code. However, subclasses of class <tt>Random</tt>
 * are permitted to use other algorithms, so long as they adhere to the
 * general contracts for all the methods.
 * <p>
 * The algorithms implemented by class <tt>Random</tt> use a
 * <tt>protected</tt> utility method that on each invocation can supply
 * up to 32 pseudorandomly generated bits.
 * <p>
 * Many applications will find the <code>random</code> method in
 * class <code>Math</code> simpler to use.
 *
 * @author  Frank Yellin
 * @version 1.27, 04/22/99
 * @see     java.lang.Math#random()
 * @since   JDK1.0
 */
public
class Random implements java.io.Serializable {
    /** use serialVersionUID from JDK 1.1 for interoperability */
    static final long serialVersionUID = 3905348978240129619L;

    /**
     * The internal state associated with this pseudorandom number generator.
     * (The specs for the methods in this class describe the ongoing
     * computation of this value.)
     *
     * @serial
     */
    private long seed;

    private final static long multiplier = 0x5DEECE66DL; // 25214903917
    private final static long addend = 0xBL;             // 11
    private final static long mask = (1L << 48) - 1;     // 281474976710655


    /**
     * Creates a new random number generator. Its seed is initialized to
     * a value based on the current time:
     * <blockquote><pre>
     * public Random() { this(System.currentTimeMillis()); }</pre></blockquote>
     *
     * @see     java.lang.System#currentTimeMillis()
     */
    public Random() { this(System.currentTimeMillis()); }

    /**
     * Creates a new random number generator using a single
     * <code>long</code> seed:
```

```
     * <blockquote><pre>
     * public Random(long seed) { setSeed(seed); }</pre></blockquote>
     * Used by method <tt>next</tt> to hold
     * the state of the pseudorandom number generator.
     *
     * @param   seed   the initial seed.
     * @see     java.util.Random#setSeed(long)
     */
    public Random(long seed) {
        setSeed(seed);
    }

    /**
     * Sets the seed of this random number generator using a single
     * <code>long</code> seed. The general contract of <tt>setSeed</tt>
     * is that it alters the state of this random number generator
     * object so as to be in exactly the same state as if it had just
     * been created with the argument <tt>seed</tt> as a seed. The method
     * <tt>setSeed</tt> is implemented by class Random as follows:
     * <blockquote><pre>
     * synchronized public void setSeed(long seed) {
     *       this.seed = (seed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
     *       haveNextNextGaussian = false;
     * }</pre></blockquote>
     * The implementation of <tt>setSeed</tt> by class <tt>Random</tt>
     * happens to use only 48 bits of the given seed. In general, however,
     * an overriding method may use all 64 bits of the long argument
     * as a seed value.
     *
     * @param   seed   the initial seed.
     */
    synchronized public void setSeed(long seed) {
        this.seed = (seed ^ multiplier) & mask;
       haveNextNextGaussian = false;
    }

    /**
     * Generates the next pseudorandom number. Subclass should
     * override this, as this is used by all other methods.<p>
     * The general contract of <tt>next</tt> is that it returns an
     * <tt>int</tt> value and if the argument bits is between <tt>1</tt>
     * and <tt>32</tt> (inclusive), then that many low-order bits of the
     * returned value will be (approximately) independently chosen bit
     * values, each of which is (approximately) equally likely to be
     * <tt>0</tt> or <tt>1</tt>. The method <tt>next</tt> is implemented
     * by class <tt>Random</tt> as follows:
     * <blockquote><pre>
     * synchronized protected int next(int bits) {
     *       seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
     *       return (int)(seed >>> (48 - bits));
     * }</pre></blockquote>
     * This is a linear congruential pseudorandom number generator, as
     * defined by D. H. Lehmer and described by Donald E. Knuth in <i>The
     * Art of Computer Programming,</i> Volume 2: <i>Seminumerical
     * Algorithms</i>, section 3.2.1.
     *
     * @param   bits random bits
     * @return  the next pseudorandom value from this random number generator's sequence.
     * @since   JDK1.1
     */
    synchronized protected int next(int bits) {
        long nextseed = (seed * multiplier + addend) & mask;
        seed = nextseed;
        return (int)(nextseed >>> (48 - bits));
    }

    private static final int BITS_PER_BYTE = 8;
    private static final int BYTES_PER_INT = 4;

    /**
     * Generates random bytes and places them into a user-supplied
     * byte array.  The number of random bytes produced is equal to
     * the length of the byte array.
     *
     * @param bytes  the non-null byte array in which to put the
     *               random bytes.
     * @since   JDK1.1
     */
```

```
    public void nextBytes(byte[] bytes) {
       int numRequested = bytes.length;

       int numGot = 0, rnd = 0;

       while (true) {
           for (int i = 0; i < BYTES_PER_INT; i++) {
                    if (numGot == numRequested)
                        return;

                    rnd = (i==0 ? next(BITS_PER_BYTE * BYTES_PER_INT)
                                : rnd >> BITS_PER_BYTE);
                    bytes[numGot++] = (byte)rnd;
           }
       }
    }

    /**
     * Returns the next pseudorandom, uniformly distributed <code>int</code>
     * value from this random number generator's sequence. The general
     * contract of <tt>nextInt</tt> is that one <tt>int</tt> value is
     * pseudorandomly generated and returned. All 2<font size="-1"><sup>32
     * </sup></font> possible <tt>int</tt> values are produced with
     * (approximately) equal probability. The method <tt>nextInt</tt> is
     * implemented by class <tt>Random</tt> as follows:
     * <blockquote><pre>
     * public int nextInt() {  return next(32); }</pre></blockquote>
     *
     * @return  the next pseudorandom, uniformly distributed <code>int</code>
     *          value from this random number generator's sequence.
     */
    public int nextInt() {  return next(32); }

    /**
     * Returns a pseudorandom, uniformly distributed <tt>int</tt> value
     * between 0 (inclusive) and the specified value (exclusive), drawn from
     * this random number generator's sequence.  The general contract of
     * <tt>nextInt</tt> is that one <tt>int</tt> value in the specified range
     * is pseudorandomly generated and returned.  All <tt>n</tt> possible
     * <tt>int</tt> values are produced with (approximately) equal
     * probability.  The method <tt>nextInt(int n)</tt> is implemented by
     * class <tt>Random</tt> as follows:
     * <blockquote><pre>
     * public int nextInt(int n) {
     *     if (n<=0)
     *       throw new IllegalArgumentException("n must be positive");
     *
     *     if ((n & -n) == n)  // i.e., n is a power of 2
     *         return (int)((n * (long)next(31)) >> 31);
     *
     *     int bits, val;
     *     do {
     *         bits = next(31);
     *         val = bits % n;
     *     } while(bits - val + (n-1) < 0);
     *     return val;
     * }
     * </pre></blockquote>
     * <p>
     * The hedge "approximately" is used in the foregoing description only
     * because the next method is only approximately an unbiased source of
     * independently chosen bits.  If it were a perfect source of randomly
     * chosen bits, then the algorithm shown would choose <tt>int</tt>
     * values from the stated range with perfect uniformity.
     * <p>
     * The algorithm is slightly tricky.  It rejects values that would result
     * in an uneven distribution (due to the fact that 2^31 is not divisible
     * by n). The probability of a value being rejected depends on n.  The
     * worst case is n=2^30+1, for which the probability of a reject is 1/2,
     * and the expected number of iterations before the loop terminates is 2.
     * <p>
     * The algorithm treats the case where n is a power of two specially: it
     * returns the correct number of high-order bits from the underlying
     * pseudo-random number generator.  In the absence of special treatment,
     * the correct number of <i>low-order</i> bits would be returned.  Linear
     * congruential pseudo-random number generators such as the one
     * implemented by this class are known to have short periods in the
     * sequence of values of their low-order bits.  Thus, this special case
```

```
     * greatly increases the length of the sequence of values returned by
     * successive calls to this method if n is a small power of two.
     *
     * @parameter n the bound on the random number to be returned.  Must be
     *         positive.
     * @return  a pseudorandom, uniformly distributed <tt>int</tt>
     *          value between 0 (inclusive) and n (exclusive).
     * @exception IllegalArgumentException n is not positive.
     * @since JDK1.2
     */

    public int nextInt(int n) {
        if (n<=0)
            throw new IllegalArgumentException("n must be positive");

        if ((n & -n) == n)  // i.e., n is a power of 2
            return (int)((n * (long)next(31)) >> 31);

        int bits, val;
        do {
            bits = next(31);
            val = bits % n;
        } while(bits - val + (n-1) < 0);
        return val;
    }

    /**
     * Returns the next pseudorandom, uniformly distributed <code>long</code>
     * value from this random number generator's sequence. The general
     * contract of <tt>nextLong</tt> is that one long value is pseudorandomly
     * generated and returned. All 2<font size="-1"><sup>64</sup></font>
     * possible <tt>long</tt> values are produced with (approximately) equal
     * probability. The method <tt>nextLong</tt> is implemented by class
     * <tt>Random</tt> as follows:
     * <blockquote><pre>
     * public long nextLong() {
     *       return ((long)next(32) << 32) + next(32);
     * }</pre></blockquote>
     *
     * @return  the next pseudorandom, uniformly distributed <code>long</code>
     *          value from this random number generator's sequence.
     */
    public long nextLong() {
        // it's okay that the bottom word remains signed.
        return ((long)(next(32)) << 32) + next(32);
    }

    /**
     * Returns the next pseudorandom, uniformly distributed
     * <code>boolean</code> value from this random number generator's
     * sequence. The general contract of <tt>nextBoolean</tt> is that one
     * <tt>boolean</tt> value is pseudorandomly generated and returned.  The
     * values <code>true</code> and <code>false</code> are produced with
     * (approximately) equal probability. The method <tt>nextBoolean</tt> is
     * implemented by class <tt>Random</tt> as follows:
     * <blockquote><pre>
     * public boolean nextBoolean() {return next(1) != 0;}
     *
     * @return  the next pseudorandom, uniformly distributed
     *          <code>boolean</code> value from this random number generator's
     *       sequence.
     * @since JDK1.2
     */
    public boolean nextBoolean() {return next(1) != 0;}

    /**
     * Returns the next pseudorandom, uniformly distributed <code>float</code>
     * value between <code>0.0</code> and <code>1.0</code> from this random
     * number generator's sequence. <p>
     * The general contract of <tt>nextFloat</tt> is that one <tt>float</tt>
     * value, chosen (approximately) uniformly from the range <tt>0.0f</tt>
     * (inclusive) to <tt>1.0f</tt> (exclusive), is pseudorandomly
     * generated and returned. All 2<font size="-1"><sup>24</sup></font>
     * possible <tt>float</tt> values of the form
     * <i>m x </i>2<font size="-1"><sup>-24</sup></font>, where
     * <i>m</i> is a positive integer less than 2<font size="-1"><sup>24</sup>
     * </font>, are produced with (approximately) equal probability. The
     * method <tt>nextFloat</tt> is implemented by class <tt>Random</tt> as
```

```
     * follows:
     * <blockquote><pre>
     * public float nextFloat() {
     *      return next(24) / ((float)(1 << 24));
     * }</pre></blockquote>
     * The hedge "approximately" is used in the foregoing description only
     * because the next method is only approximately an unbiased source of
     * independently chosen bits. If it were a perfect source or randomly
     * chosen bits, then the algorithm shown would choose <tt>float</tt>
     * values from the stated range with perfect uniformity.<p>
     * [In early versions of Java, the result was incorrectly calculated as:
     * <blockquote><pre>
     * return next(30) / ((float)(1 << 30));</pre></blockquote>
     * This might seem to be equivalent, if not better, but in fact it
     * introduced a slight nonuniformity because of the bias in the rounding
     * of floating-point numbers: it was slightly more likely that the
     * low-order bit of the significand would be 0 than that it would be 1.]
     *
     * @return  the next pseudorandom, uniformly distributed <code>float</code>
     *          value between <code>0.0</code> and <code>1.0</code> from this
     *          random number generator's sequence.
     */
    public float nextFloat() {
        int i = next(24);
        return i / ((float)(1 << 24));
    }

    /**
     * Returns the next pseudorandom, uniformly distributed
     * <code>double</code> value between <code>0.0</code> and
     * <code>1.0</code> from this random number generator's sequence. <p>
     * The general contract of <tt>nextDouble</tt> is that one
     * <tt>double</tt> value, chosen (approximately) uniformly from the
     * range <tt>0.0d</tt> (inclusive) to <tt>1.0d</tt> (exclusive), is
     * pseudorandomly generated and returned. All
     * 2<font size="-1"><sup>53</sup></font> possible <tt>float</tt>
     * values of the form <i>m x </i>2<font size="-1"><sup>-53</sup>
     * </font>, where <i>m</i> is a positive integer less than
     * 2<font size="-1"><sup>53</sup></font>, are produced with
     * (approximately) equal probability. The method <tt>nextDouble</tt> is
     * implemented by class <tt>Random</tt> as follows:
     * <blockquote><pre>
     * public double nextDouble() {
     *       return (((long)next(26) << 27) + next(27))
     *           / (double)(1L << 53);
     * }</pre></blockquote><p>
     * The hedge "approximately" is used in the foregoing description only
     * because the <tt>next</tt> method is only approximately an unbiased
     * source of independently chosen bits. If it were a perfect source or
     * randomly chosen bits, then the algorithm shown would choose
     * <tt>double</tt> values from the stated range with perfect uniformity.
     * <p>[In early versions of Java, the result was incorrectly calculated as:
     * <blockquote><pre>
     *  return (((long)next(27) << 27) + next(27))
     *      / (double)(1L << 54);</pre></blockquote>
     * This might seem to be equivalent, if not better, but in fact it
     * introduced a large nonuniformity because of the bias in the rounding
     * of floating-point numbers: it was three times as likely that the
     * low-order bit of the significand would be 0 than that it would be
     * 1! This nonuniformity probably doesn't matter much in practice, but
     * we strive for perfection.]
     *
     * @return  the next pseudorandom, uniformly distributed
     *          <code>double</code> value between <code>0.0</code> and
     *          <code>1.0</code> from this random number generator's sequence.
     */
    public double nextDouble() {
        long l = ((long)(next(26)) << 27) + next(27);
        return l / (double)(1L << 53);
    }

    private double nextNextGaussian;
    private boolean haveNextNextGaussian = false;

    /**
     * Returns the next pseudorandom, Gaussian ("normally") distributed
     * <code>double</code> value with mean <code>0.0</code> and standard
     * deviation <code>1.0</code> from this random number generator's sequence.
```

```
     * <p>
     * The general contract of <tt>nextGaussian</tt> is that one
     * <tt>double</tt> value, chosen from (approximately) the usual
     * normal distribution with mean <tt>0.0</tt> and standard deviation
     * <tt>1.0</tt>, is pseudorandomly generated and returned. The method
     * <tt>nextGaussian</tt> is implemented by class <tt>Random</tt> as follows:
     * <blockquote><pre>
     * synchronized public double nextGaussian() {
     *    if (haveNextNextGaussian) {
     *            haveNextNextGaussian = false;
     *            return nextNextGaussian;
     *    } else {
     *            double v1, v2, s;
     *            do {
     *                    v1 = 2 * nextDouble() - 1;   // between -1.0 and 1.0
     *                    v2 = 2 * nextDouble() - 1;   // between -1.0 and 1.0
     *                    s = v1 * v1 + v2 * v2;
     *            } while (s >= 1);
     *            double multiplier = Math.sqrt(-2 * Math.log(s)/s);
     *            nextNextGaussian = v2 * multiplier;
     *            haveNextNextGaussian = true;
     *            return v1 * multiplier;
     *    }
     * }</pre></blockquote>
     * This uses the <i>polar method</i> of G. E. P. Box, M. E. Muller, and
     * G. Marsaglia, as described by Donald E. Knuth in <i>The Art of
     * Computer Programming</i>, Volume 2: <i>Seminumerical Algorithms</i>,
     * section 3.4.1, subsection C, algorithm P. Note that it generates two
     * independent values at the cost of only one call to <tt>Math.log</tt>
     * and one call to <tt>Math.sqrt</tt>.
     *
     * @return  the next pseudorandom, Gaussian ("normally") distributed
     *          <code>double</code> value with mean <code>0.0</code> and
     *          standard deviation <code>1.0</code> from this random number
     *          generator's sequence.
     */
    synchronized public double nextGaussian() {
        // See Knuth, ACP, Section 3.4.1 Algorithm C.
        if (haveNextNextGaussian) {
           haveNextNextGaussian = false;
           return nextNextGaussian;
         } else {
            double v1, v2, s;
            do {
                v1 = 2 * nextDouble() - 1; // between -1 and 1
               v2 = 2 * nextDouble() - 1; // between -1 and 1
                 s = v1 * v1 + v2 * v2;
            } while (s >= 1);
            double multiplier = Math.sqrt(-2 * Math.log(s)/s);
            nextNextGaussian = v2 * multiplier;
            haveNextNextGaussian = true;
            return v1 * multiplier;
         }
    }
}
```