Zufall

So ein Zufall?

- Zufällige Ereignisse sind nicht kausal erklärbar (d.h. nicht eindeutig vorherbestimmt und nicht vorhersagbar).
- Zufällige Ereignisse sind voneinander unabhängig.
- Zufällige Daten sind nicht verlustfrei komprimierbar.

Hmm, sind Rechner nicht prinzipiell programmiert und somit vorhersagbar?
Kann Zufall in Algorithmen überhaupt von Vorteil sein?

Alles läuft hier nach Fahrplan
der Zufall ist unser Feind
Die Toten Hosen – 1000 Gute Gründe

*Sowohl bei der Erklärung vergangener Ereignisse als auch bei der Vorhersage der Zukunft konzentrieren wir uns auf die **kausale** Rolle von Fähigkeiten und vernachlässigen die Rolle des Zufalls.*
Daher sind wir anfällig für die Illusion der Kontrolle.
Daniel Kahneman –
Schnelles Denken, Langsames Denken

Zufall

Die **Minimale Beschreibungslänge** (J. Rissanen, 1978)

Ein Ansatz zur verlustfreien Kompression von Zahlen besteht darin, diese mit einem **möglichst kurzen Programm** zu beschreiben.

Das kürzeste Programm (egal in welcher Sprache) definiert dann die minimale Beschreibungslänge (**MDL**, *minimum description length*, auch Kolmogorow-Komplexität genannt).

Wenn die MDL genauso lang ist wie die Zahlenfolge selbst, dann ist die Zahlenfolge zufällig.

► *Wie könnte eine Zahlenfolge aussehen, die sich sehr kompakt mit einem kurzen Programm darstellen lässt?*

Zufall

FAKULTÄT INFORMATIK

Die MDL ist nicht berechenbar.

Man kann also kein Programm schreiben, das bei beliebigen Daten D garantiert immer das kürzeste Programm erzeugt, welches D generiert.

► *Warum ist die MDL nicht berechenbar? Es gibt doch nur* endlich *viele Programme, welche eine Länge kleiner gleich der Länge der gegebenen Zahlenfolge haben!*

Vergleiche die MDL mit Ockhams Rasiermesser:
Die kürzeste / einfachste korrekte Erklärung ist die beste („Sparsamkeitsprinzip“).

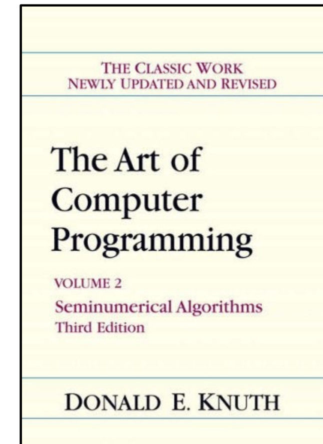
Zufallszahlen

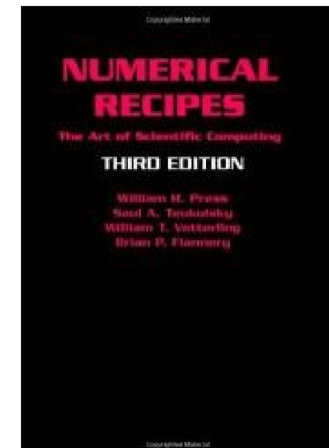
Zufallszahlengeneratoren erzeugen **repräsentative Stichproben** aus **Wahrscheinlichkeitsverteilungen**.

Wegen der deterministischen Natur von Programmen spricht man hier auch von **Pseudo-Zufallszahlen**.

Die altvorderen Standardwerke:

- D. Knuth, *The Art of Computer Programming, Vol. 2*
- W. Press et al., *Numerical Recipes* (numerical.recipes)
- Park/Miller: *Random Number Generators: Good ones are hard to find*, CACM Oct.1988.





> *Random numbers should not be generated with a method chosen at random.*
>
> Donald Knuth

Zufallszahlen

FAKULTÄT INFORMATIK

Zufallszahlen können auf verschiedene Arten **verteilt** sein.

Gleichverteilung: Alle Zahlen (= **Ereignisse**) haben dieselbe **Eintrittswahrscheinlichkeit**.

Ein Würfel sollte **fair** sein, d.h. jede Zahl von 1 bis 6 sollte dieselbe uniforme Eintrittswahrscheinlichkeit $p_i = 1/6$ haben.

Die **Beobachtung** eines Würfels liefert die tatsächlichen (**empirischen**) **Häufigkeiten**.

Teilt man die Häufigkeiten H_i der einzelnen Ereignisse durch die Gesamtzahl n der Ereignisse, so erhält man die **relativen Häufigkeiten** $h_i = H_i / n$, mit $0 \leq h_i \leq 1$.

Für $n \rightarrow \infty$ sollte $h_i \rightarrow p_i$ gelten.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Lineare Kongruenzgeneratoren (**LKG**s, D. Lehmer, 1949)

LKGs sollen gleichverteilte positive Ganzzahlen generieren.
Sie erzeugen iterativ eine Zahl aus der vorherigen Zahl mittels

$$z_{i+1} = a \cdot z_i \mod m$$

Bei dieser Methode benötigt man also

- einen **Multiplikator** a, $1 < a < m$, z.B. $a = 16807$
- einen **Modulus** m, $m > 0$, z.B. $m = 2^{31}-1 = 2147483647$
- einen **Startwert** (*seed*) z_0, $0 < z_0 < m$.

Bei einem Startwert $z_0 = 1$ ergibt sich mit den obigen Parametern die Folge 16807, 282475249, 1622650073, 984943658, ...

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Der LKG verwendet ausschließlich Ganzzahlen.

Möchte man reelle Zufallszahlen aus dem Intervall]0, 1] haben, so gibt man in jeder Iteration den Wert

$$y_i = z_i/(m-1)$$

aus. Zusätzliche Arbeit: Umwandlung in den Datentyp double sowie Gleitpunkt-Division.

Obige Folge lautet dann 0.0000000004656613, 0.000007826369, 0.1315378, 0.75560534, 0.45865014, ...

► *Wie generiert man gleichverteilte reelle Zufallszahlen aus dem Intervall*]17, 19] ?

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Lineare Kongruenzgeneratoren sind **periodisch**: Es wird immer dieselbe Reihenfolge von endlich vielen Zahlen durchlaufen.

Konsequenz: Folgen wie z.B. .. 5, 3, 5, 4, .. oder .. 5, 3, 4, 4, 1, .. sind mit einfachen LKGs direkt nicht möglich.

Die **maximale Periodenlänge** ist m-1, wenn alle Zahlen größer 0 und kleiner m (in beliebiger Reihenfolge) durchlaufen werden.
Aber Vorsicht: wesentlich kürzere Perioden sind möglich.

Sei z.B. m = 11, a = 4. Die maximale Periodenlänge wäre 11-1 = 10.
Aber mit $z_0 = 1$ ergibt sich die Folge 1, 4, 5, 9, 3, 1, ... der Länge 5.

Noch übler sieht es mit m = 8, a = 4 aus. Mit $z_0 = 1$ ergibt sich die Folge 1, 4, 0, 0, ... Diese Parameter sind also unbrauchbar.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

In der Praxis wählt man den Modulus m möglichst groß, um eine lange Periode zu erhalten. Der Datentyp bestimmt die Obergrenze, z.B. `Integer.MAX_VALUE` = $2^{31}-1$.

Die maximale Periodenlänge wird erreicht, wenn man m derart als Primzahl wählt, dass für alle Primfaktoren q von m-1 gilt:

$$a^{(m-1)/q} \bmod m \neq 1.$$

Für a kommen i.d.R. zahlreiche Werte in Betracht.

Beim LKG mit a = 4, m = 11 haben wir

- $4^{(11-1)/5} \bmod 11 = 5 \neq 1$ und
- $4^{(11-1)/2} \bmod 11 = 1$.

$\Rightarrow$ Die maximale Periodenlänge wird nicht erreicht.

Die Wahl des Startwertes $0 < z_0 < m$ ist unkritisch. Man kann z.B. die Systemzeit wählen, wenn man laufend ein neues z_0 haben möchte.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Bei der Implementierung der Formel $z_{i+1} = a \cdot z_i \bmod m$ sollte ein Überlauf vermieden werden.

Die folgende naive – *fehlerhafte* – Java-Implementierung durch

```
int a = 16807;
int m = 0x7fffffff;  // = 2147483647 = 2^31-1
int randomNumber = 1;
for (int i=1; i<5; i++) {
     randomNumber *= a % m;
     System.out.println(randomNumber);
}
```

führt zur Ausgabe von 16807, 282475249, 1622647863 (falsch!), ...
Der 32-bit Datentyp `int` kann das Ergebnis der Multiplikation von 16807 mit 282475249 nicht fassen.

Eine Fehlermeldung des Laufzeitsystems (der JVM) wäre beruhigend – müsste aber mit hohem Rechenzeitaufwand bezahlt werden!

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Uns interessiert aber nicht das Ergebnis der Multiplikation $a \cdot z_i$, sondern nur dessen Rest modulo m, welcher immer kleiner m ist.

Zur Vermeidung des Überlaufs zerlegt man m gemäß $m = a \cdot q + r$, also $q = m \text{ div } a$, $r = m \text{ mod } a$. Ist $r < q$, so gilt

$$a \cdot z_i \text{ mod } m = a \cdot (z_i \text{ mod } q) - r \cdot (z_i \text{ div } q)$$

Ist dieser Ausdruck kleiner Null, so muss noch m dazu addiert werden.

div bezeichnet die ganzzahlige Division, also z.B. 19 div 6 = 3.

Bsp.: Für $m = 2^{31}-1 = 2147483647$ und $a = 16807$ ergibt sich $q = 127773$ und $r = 2836$, mit $r < q$. Für $z_i = 282475249$ ist dann $z_{i+1} =$

```
16807·282475249 mod 2147483647 =
16807·(282475249 mod 127773) - 2836·(282475249 div 127773) =
16807·96919 - 2836·2210  =  1628917633 - 36014364  =  1622650073.
```

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Eine Variante des LKG verwendet zusätzlich ein **Inkrement** c:

$$z_{i+1} = a \cdot z_i + c \mod m$$

- Die Addition verursacht zusätzlichen Rechenaufwand.
- Die maximale Periodenlänge ist m (die 0 ist jetzt zulässig).
- Es ist nicht mehr notwendig, dass m eine Primzahl ist.
- Es gibt eine größere Anzahl von Kombinationen aus a und m, welche die maximale Periodenlänge erreichen.
- Die erzeugten Sequenzen sind oft „zufälliger“ als bei $c = 0$.

Eine maximale Periodenlänge allein ist noch nicht hinreichend, um brauchbare Zufallszahlen zu generieren, wie man am Beispiel $a = 1$, $c = 1$ erkennt.

Zufallszahlengenerierung

Auch die Java-Klasse `Random` verwendet einen LKG mit Inkrement c. Die Parameter sind

$m = 2^{48}$

$a = 25214903917 = 0x5DEECE66D$

$c = 11 = 0xB$

Von den 48 bit von z_i werden die ersten 32 bit (47...16) als Zufallszahl hergenommen. Aus der Java-API:

The method next is implemented by class Random by atomically updating the seed to

(seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)

Die Klasse Random hat die kryptographisch stärkere Unterklasse `SecureRandom`, siehe weiter unten.

Zufallszahlengenerierung

Die Periodenlänge eines LKG lässt sich deutlich verlängern, wenn der Wert von z_{i+1} von mehr als nur einem Vorgänger abhängt.

Der einfachste solch additiver Generator wäre der Fibonacci-Generator

$$z_{i+1} = (z_i + z_{i-1}) \bmod m$$

welcher jedoch unbefriedigende Zahlenfolgen liefert.

Etwas besser sind die **verzögerten Generatoren** vom Typ

$$z_{i+1} = (z_i + z_{i-k}) \bmod m \quad \text{für } k > 1.$$

Folgende Variante liefert recht brauchbare Zufallszahlen:

$$z_i = (z_{i-24} + z_{i-55}) \bmod m$$

Zur Implementierung hält man eine zyklische Liste der letzten 55 z_i vor. Die Periodenlänge ist beachtliche $2^{e-1}(2^{55}-1)$ mit $m = 2^e$ und z.B. $e = 30$.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Eine Verallgemeinerung der verzögerte Generatoren sind die **additiv-multiplikativen Generatoren**

$$z_i = (a_1 \cdot z_{i-1} + \ldots + a_k \cdot z_{i-k}) \mod m$$

Hiermit kann man die Periodenlänge m^k-1 erreichen.

Der Modulus m wird als Primzahl gewählt.

Für $k = 1$ ergibt sich als Sonderfall der Standard-LKG.

Eine gute Wahl von $a_1 \ldots a_k$ erfordert beträchtlichen analytischen und rechnerischen Aufwand.

Die Berechnung der Zufallszahlen ist hier relativ teuer. Die Qualität der erzeugten Zufallszahlen ist bei gut gewählten a_i ziemlich hoch.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Es gibt zahlreiche „Tricks“, Zahlenfolgen „noch zufälliger“ zu machen.

** * * Warnung: Die meisten davon sind schlecht!!! * * **

Was i.d.R. aber funktioniert:

- „Wegwerfen“ von Zahlen aus der erzeugten Sequenz.
- Mischen (*Shuffling*). Hier hält man z.B. eine Liste von Zufallszahlen vor und wählt zufällig einen Index aus, der angibt, welche der Zahlen aus der Liste als nächstes z_i hergenommen wird.

Noch eine Warnung: Die einzelnen Bits von guten Zufallszahlen sind nicht unbedingt als zufällige Bits brauchbar!

$\Rightarrow$ Spezielle Bit-Zufallsgeneratoren verwenden.

Zufallszahlengenerierung

Der **Mersenne-Twister (MT)** (Matsumoto und Nishimura, 1998)

Der MT erzeugt vorzeichenlose (*unsigned*) 32-bit Integer Zufallszahlen.

Seine Periodenlänge ist monumentale $2^{19937}-1$ ($\approx 10^{6000}$).

Statt eines einzigen Startwerts verwendet der MT ein Startfeld von 624 Zahlen x_i (welche z.B. mit einem LKG erzeugt werden können).

Die Folgewerte berechnen sich dann zu

$$x_{i+1} = x_{i-227} \text{ XOR } y/2 \text{ XOR } ((y \bmod 2) \cdot 2567483615)$$

mit

$$y = (x_{i-624} - (x_{i-624} \bmod 2^{31})) + x_{i-623} \bmod 2^{31}$$

Zufallszahlengenerierung

Die eigentlich Zufallszahl z wird dann aus $z_0 = x_{i+1}$ mittels der folgenden Sequenz von Bitoperationen (dem „*Twisting*“) berechnet:

```
z = z XOR (z / 2^11);
z = z XOR (z · 2^7 AND 2636928640);
z = z XOR (z · 2^15 AND 4022730752);
z = z XOR (z / 2^18)
```

Der MT ist sehr schnell, da nur Bitoperationen verwendet werden.

Java hat keinen unsigned Datentyp, so dass dort bei der Ausgabe die 32-bit Integer auf den Typ `long` abgebildet werden müssen.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Der Mersenne-Twister geschrieben mit Bitoperationen:

```
x[i+1] = x[i-227] ^ (y >>> 1) ^ mat[y & 0x1]   mit
y = (x[i-624] & 0x80000000) | (x[i-623] & 0x7fffffff)
und   mat[0] = 0,   mat[1] = 0x9908b0df
```

```
z = z ^ (z >>> 11);
z = z ^ (z <<   7 & 0x9d2c5680);
z = z ^ (z <<  15 & 0xefc60000);
z = z ^ (z >>> 18)
```

mit ^ = XOR, & = AND, | = OR, << = Linksverschiebung,
>>> = Rechtsverschiebung mit Auffüllen mit Nullen. Die „Tricks“:

- Das Ergebnis von y mod 2 kann nur 0 oder 1 sein
- $(x_{i-624} - (x_{i-624} \bmod 2^{31}))$ ist das erste Bit von x_{i-624} (mal 2^{31})
- $x_{i-623} \bmod 2^{31}$ sind die hinteren 31 Bits von x_{i-623}.

Zufallszahlengenerierung

Bsp.: Beim Twisting wird aus z = 4711 die 4849911 generiert:

$z =$	`00000000000000000001001001100111`	// 4711
$z/2^{11} =$	`00000000000000000000000000000010`	// 2
z XOR $(z/2^{11}) =$	`00000000000000000001001001100101`	// 4709
$z \cdot 2^7 =$	`00000000000010010011001010000000`	// 602752
B = 0x9d2c5680	`10011101001011000101011010000000`	// 2636928640
$(z \cdot 2^7$ AND $B) =$	`00000000000010000001001010000000`	// 529024
z XOR $(z \cdot 2^7$ AND $B) =$	`00000000000010000000000011100101`	// 524517
…		
z XOR $(z \cdot 2^{15}$ AND $C) =$	`00000000010010100000000011100101`	// 4849893
…		
z XOR $(z/2^{18}) =$	`00000000010010100000000011110111`	// 4849911

FAKULTÄT INFORMATIK

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Primzahlen der Form 2^{Primzahl}-1 heißen **Mersenne-Primzahlen** (Marin Mersenne, 1588-1648).

Eine Zahl der Form 2^n-1 kann nur dann Primzahl sein, wenn n Primzahl ist, da $2^{u \cdot v}$-1 durch 2^u-1 und durch 2^v-1 teilbar ist.

Die ersten Mersenne-Primzahlen ergeben sich aus den Exponenten 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, ...

2^{19937}-1 ist die vierundzwanzigste Mersenne-Primzahl.

Bislang (Mai 2024) sind nur 51 Mersenne-Primzahlen bekannt. Die größte ist $2^{82\,589\,933}$-1 (gefunden vom *crowdsourcing* Projekt GIMPS, *Great Internet Mersenne Prime Search*).

Die größten bekannten Primzahlen sind Mersenne-Primzahlen, da es für sie einen recht einfachen Primzahlentest gibt, den **Lucas-Lehmer-Test**.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Der **XORShift** Zufallszahlengenerator (G. Marsaglia, 2003)

XORShift erzeugt auf einem `unsigned long` Datentyp eine Periode der Länge 2^{64}-1. Nur die 0 ist ausgeschlossen.

```
z = z ^ (z <<  21)
z = z ^ (z >>> 35)
z = z ^ (z <<   4)
```

Da nur wenige XOR- und Shiftoperationen verwendet werden, ist der XORShift sehr schnell. Die Qualität der Zufallszahlen ist recht hoch.

Die Parameter, oben (21, 35, 4), sind nicht einzigartig. Auch (13, 7, 17) erzeugen die volle Periodenlänge.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Eine beliebige Liste A lässt sich mit einem Lauf über alle Elemente von A in eine zufällige Reihenfolge bringen:

```
Algorithmus  Shuffle
Eingabe:     Zu mischendes Feld A mit |A| = n
Ausgabe:     Permutiertes Feld A

Für alle i von 1 bis n-1
    vertausche A[i] mit A[Random(i, n)]
```

Hierbei erzeugt `Random(i, n)` gleichverteilte Zufallszahlen r mit $i \leq r \leq n$.

Java verfügt über die Methode `Collections.shuffle()`.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Java: Class `SecureRandom()`

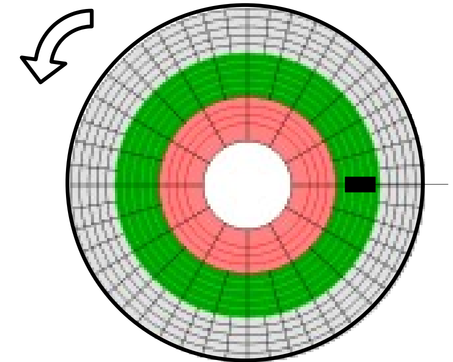
Die Klasse `SecureRandom()` stellt einen **kryptographisch starken** Zufallszahlengenerator bereit. Dies bedeutet u.a.:

- Es ist nahezu unmöglich, aus der aktuell generierten Zufallszahl auf die vorhergehende oder nachfolgende Zufallszahl zu schließen.
- Die Periode ist lang (hier: $2^{160} \approx 10^{48}$).
- Alle Startwerte sind nahezu gleichwahrscheinlich.

Die Laufzeiten der Methoden aus `SecureRandom()` sind beträchtlich höher als die von `Random()`.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK



Kritisch für die krypthographische Stärke ist die Verfügbarkeit von quasi-zufälligen Startwerten.

Mögliche Quellen hierfür:

- Positionierungszeiten von Festplatten-Leseköpfen
- Zeiten zwischen Tastenanschlägen oder Mausbewegungen
- CPU- oder Speicherauslastungen
- Rauschen von Mikrofonen und Sensoren.

Nicht sehr zufällig aber sich ständig ändernd ist die Uhrzeit.

Mehr über Zufallszahlengeneratoren unter Linux: Gutterman/Pinkas, *Analysis of the Linux Random Number Generator*, 2006 (www.pinkas.net/PAPERS/gpr06.pdf).

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Pleiten, Pech und Pannen
(eine kleine Auswahl)

- John von Neumanns *Middle Square* Methode (1946):
z_{i+1} = mittlere Ziffern aus z_i^2

- Donald Knuths *Super-random number generator* (1959):
So kompliziert, dass keiner ihn verstehen sollte...

- IBMs RANDU (ab 1960): $z_{i+1} = 65539 \cdot z_i \mod 2^{31}$

- C++ Standardroutine `rand()`. P. Hartmann (2006), *Mathematik für Informatiker*, Kapitel 22.3, Abschnitt *Anpassungstest:* Der C++ Zufallszahlengenerator generierte abwechselnd immer eine gerade und eine ungerade Zahl.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

Donald Knuths *Super-random number generator*: Fixpunkt 6065038420

Algorithm K (*"Super-random" number generator*). Given a 10-digit decimal number X, this algorithm may be used to change X to the number that should come next in a supposedly random sequence. Although the algorithm might be expected to yield quite a random sequence, reasons given below show that it is not, in fact, very good at all. (The reader need not study this algorithm in great detail except to observe how complicated it is; note, in particular, steps K1 and K2.)

K1. [Choose number of iterations.] Set $Y \leftarrow \lfloor X/10^9 \rfloor$, the most significant digit of X. (We will execute steps K2 through K13 exactly $Y+1$ times; that is, we will apply randomizing transformations a *random* number of times.)

K2. [Choose random step.] Set $Z \leftarrow \lfloor X/10^8 \rfloor \bmod 10$, the second most significant digit of X. Go to step $K(3+Z)$. (That is, we now jump to a *random* step in the program.)

K3. [Ensure $\geq 5 \times 10^9$.] If $X < 5000000000$, set $X \leftarrow X + 5000000000$.

K4. [Middle square.] Replace X by $\lfloor X^2/10^5 \rfloor \bmod 10^{10}$, that is, by the middle of the square of X.

K5. [Multiply.] Replace X by $(1001001001\,X) \bmod 10^{10}$.

K6. [Pseudo-complement.] If $X < 100000000$, then set $X \leftarrow X + 9814055677$; otherwise set $X \leftarrow 10^{10} - X$.

K7. [Interchange halves.] Interchange the low-order five digits of X with the high-order five digits; that is, set $X \leftarrow 10^5(X \bmod 10^5) + \lfloor X/10^5 \rfloor$, the middle 10 digits of $(10^{10}+1)X$.

K8. [Multiply.] Same as step K5.

K9. [Decrease digits.] Decrease each nonzero digit of the decimal representation of X by one.

K10. [99999 modify.] If $X < 10^5$, set $X \leftarrow X^2 + 99999$; otherwise set $X \leftarrow X - 99999$.

K11. [Normalize.] (At this point X cannot be zero.) If $X < 10^9$, set $X \leftarrow 10X$ and repeat this step.

K12. [Modified middle square.] Replace X by $\lfloor X(X-1)/10^5 \rfloor \bmod 10^{10}$, that is, by the middle 10 digits of $X(X-1)$.

K13. [Repeat?] If $Y > 0$, decrease Y by 1 and return to step K2. If $Y = 0$, the algorithm terminates with X as the desired "random" value. ▮

(The machine-language program corresponding to this algorithm was intended to be so complicated that a person reading a listing of it without explanatory comments wouldn't know what the program was doing.)

Quelle: D. Knuth, *The Art of Computer Programming*, Vol. 2, p.5.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

BSafe???

Das RSA-Verfahren zur kryptographischen Verschlüsselung erzeugt im ersten Schritt große Primzahlen.

Die Firma RSA Security verwendete hierzu ein BSafe genanntes Modul. Dieses verwendet einen *Dual Elliptic Curve* Zufallszahlengenerator.

Am 20.12.2013 meldete die Nachrichtenagentur Reuters unter Berufung auf Edward Snowden, dass die RSA Security 10 Millionen $ von der NSA akzeptiert habe, um in BSafe die für eine Hintertür geeigneten Parameter als Default-Einstellung von RSA-Produkten zu setzen.

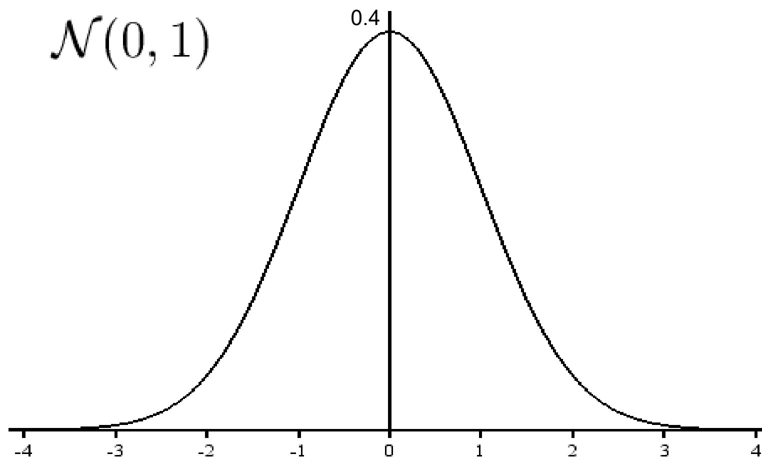
⇒ *Zufallszahlengeneratoren können politisch gesteuert sein. Manipulierte RNGs stellen eine Sicherheitslücke dar.*

Quelle: CACM May 2015 p.49.

Zufallszahlengenerierung

Es gibt zahlreiche andere wichtige Verteilungen, z.B. die **Gaußsche Normalverteilung** oder die **Exponentialverteilung**.

Zufallszahlen für solche Verteilungen werden in der Regel mittels **Transformationen** aus gleichverteilten Zufallszahlen generiert. Ein LKG kann prinzipiell also auch andere Verteilungen erzeugen.



Gaußsche Normalverteilung:

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Standardnormalverteilung mit Mittelwert $\mu = 0.0$ und Standardabweichung $\sigma = 1.0$:

$$G(x) = \frac{1}{\sqrt{2\pi}}\, e^{-x^2/2} = \mathcal{N}(0,1)$$

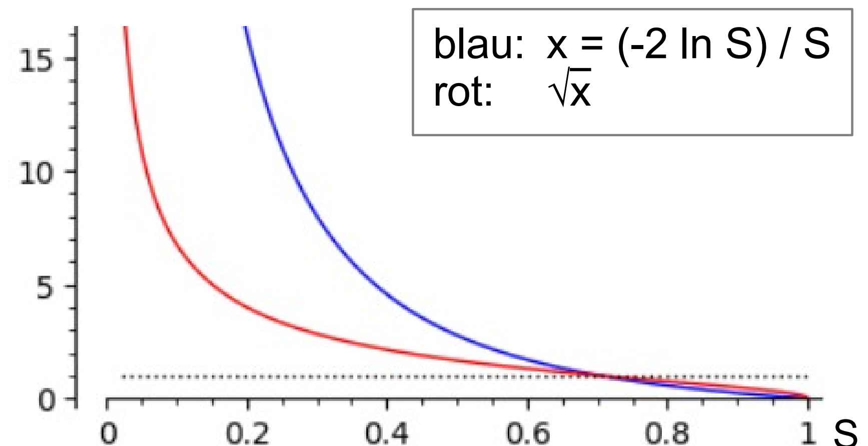
Zufallszahlengenerierung

Eine Methode zur Generierung von standardnormalverteilten Zufallszahlen stammt von Box / Muller / Marsaglia (1958):

- Generiere zwei uniforme Zufallszahlen $0 \leq v_1, v_2 \leq 1$
- Setze $v_1 = 2 \cdot v_1 - 1$, $v_2 = 2 \cdot v_2 - 1$ // $-1 \leq (v_1, v_2) \leq 1$
- Berechne $S = v_1^2 + v_2^2$
- Wiederhole die obigen Schritte solange, bis $0 < S < 1$
- Setze $x = (-2 \cdot \ln S) / S$
- Die zwei nächsten gaußverteilten Zufallszahlen sind

  $x_1 = v_1 \cdot \sqrt{x}$ und $x_2 = v_2 \cdot \sqrt{x}$

Java implementiert in der Methode `Random.nextGaussian()` ebenfalls Box-Muller-Marsaglia.



Zufallszahlengenerierung

FAKULTÄT INFORMATIK

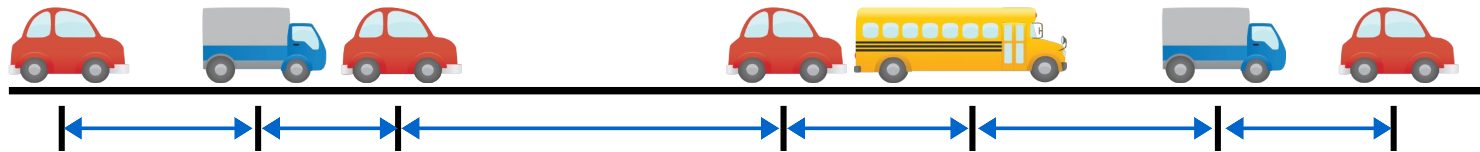
Mit Hilfe der linearen Abbildung

$$x' = \sigma \cdot x + \mu$$

kann man die standardnormalverteilten Zufallszahlen x in normalverteilte Zahlen x' mit Mittelwert μ und Standardabweichung σ umwandeln.

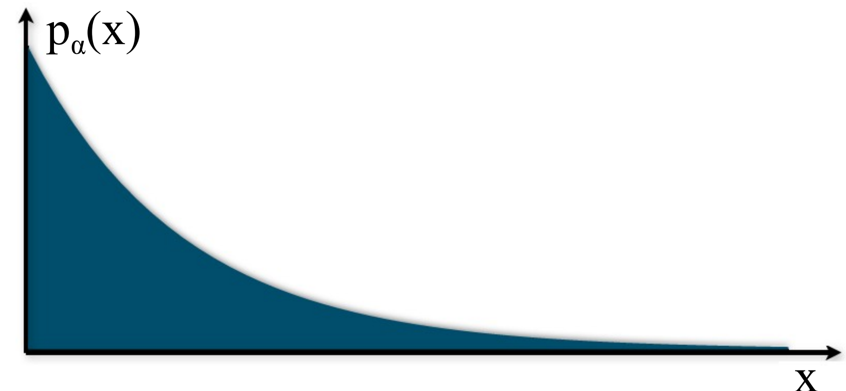
$$\mathcal{N}(0,1) \longrightarrow_{\sigma \cdot x + \mu} \mathcal{N}(\mu, \sigma^2)$$

Zufallszahlengenerierung



Die **Exponentialverteilung** mit Parameter α hat die Dichte

$$p_\alpha(x) = \begin{cases} \alpha\, e^{-\alpha x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$



Ist X eine in [0..1] gleichverteilte Zufallsvariable, dann ist Y mit

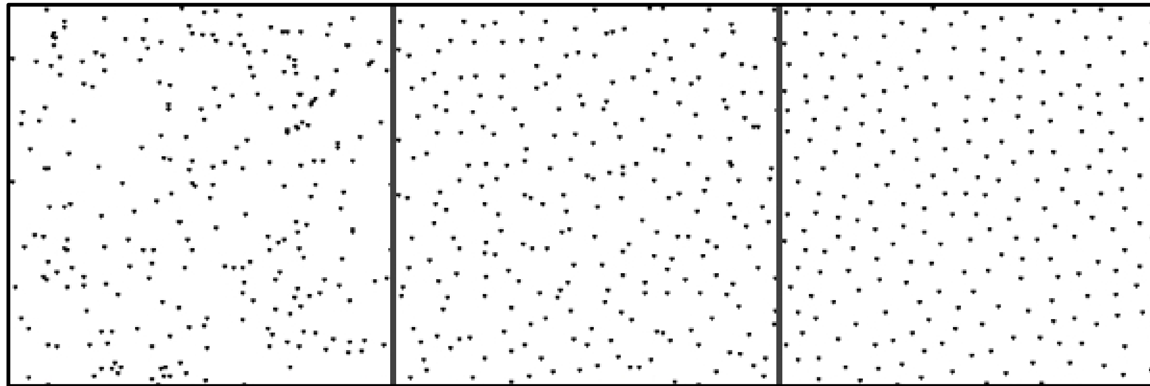
$$Y = -1/\alpha \cdot \ln X$$

exponentialverteilt.

Der nebenstehende Java Code erzeugt n exponentialverteilte Zufallszahlen mit Parameter α.

```
Random rng = new Random();
alphaInv = -1.0/alpha;
for (int i=0; i<n; i++)
    y = alphaInv *
        Math.log(rng.nextDouble());
```

Zufallszahlengenerierung



(l) uniform
(m) gitterbasiert
(r) Poisson Scheibe

Quelle: http://devmag.org.za/2009/05/03/poisson-disk-sampling/

Gesteuerter Zufall: *Poisson Disk Sampling*

In vielen Anwendungen (z.B. in der Computergraphik: Platzierung von Objekten) möchte man 2-d Zufallspunkte mit einem **Mindestabstand** haben.

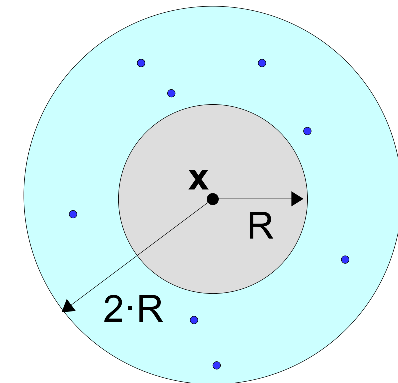
Generiert man 2-d Zufallspunkte mit uniformer Verteilung beider Koordinaten, so liegen häufig mehrere Punkte dicht beieinander.

Das Unterteilen der Ebene in ein regelmäßiges Gitter mit Setzen eines Zufallspunktes je Quadrat löst das Problem nur begrenzt.

Zufallszahlengenerierung

Beim ***Poisson Disk Sampling*** stellt man sicher, dass die Punkte mindestens eine Entfernung R zum nächsten Punkt haben.

Man wählt einen beliebigen Zufallspunkt **x**. Um **x** herum definiert man einen Ring mit innerem Radius R und äußerem Radius 2·R.



In diesem Ring erzeugt man k uniforme Zufallspunkte. Jeder dieser k Punkte, der den Mindestabstand R zu allen seinen Nachbarn hat, wird gesetzt.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

> *Der Zufall ist wie seine Brüder – der Einfall, der Unfall und der Anfall – nicht kontrollierbar.*
>
> Christoph Süß

In vielen Anwendungen sind die Eigenschaften eines Zufallszahlen-generators relativ unkritisch – manchmal aber nicht:

- Zufällige Auswahl von Personen, die dann eine bestimmte Pflicht (z.B. Schöffendienst, Wehrdienst, Wahlhelfer) übernehmen müssen.
- Zufällige Auswahl von Steuerprüfungen, Betriebsprüfungen, etc.
- Glückspiel-Automaten, bei denen Geld ein- und ausgezahlt wird.

Bei Glückspiel-Automaten wird oft vom Gesetzgeber das Einhalten von Auszahlungsquoten gefordert, d.h. ein bestimmter Anteil des eingenommenen Geldes muss in einem vorgegebenen Zeitraum als Gewinn wieder an die Spieler ausgeschüttet werden.

Zufallszahlengenerierung

FAKULTÄT INFORMATIK

True Random Number Generators (**TRNG**s)

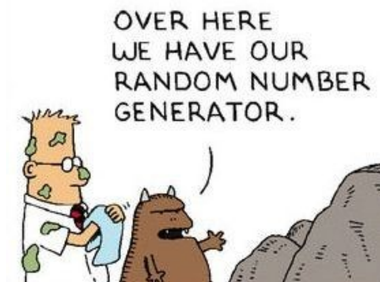
TRNGs erzeugen Zufallszahlen aus vermutlich rein zufälligen physikalische Vorgängen.

Zu diesen gehört nach dem Stand der Physik

- der Zerfall von radioaktivem Material
- thermisches Rauschen in Halbleiter-Bauteilen
- quantenoptische Effekte von Photonen o.ä.

Quantencomputer können echte Zufallszahlen algorithmisch erzeugen.

Randomness Beacons (z.B. drand) generieren öffentlich und verifizierbar Zufallszahlen, die zu vorgegebenen Zeitpunkten gemäß einem definierten Prozess dezentral erzeugt werden.



Zufallszahlentests 11011100101110111100010011010...

FAKULTÄT INFORMATIK

> *Wenn der Zufall seine Hand im Spiel hat,*
> *können regelmäßige Muster nur Illusion sein.*
> Daniel Kahneman

Tests für Zufallszahlen

Wie gut ist die von einem Zufallszahlengenerator generierte Verteilung?

Zahlreiche Tests sind hierzu entwickelt worden.

⇒ Einen einzigen, seligmachenden Test, der eindeutig sagt: „zufällig“ oder „nicht zufällig“, gibt es nicht – und kann es nicht geben.

Jeder Test versucht, irgendein Muster zu finden. Je mehr Tests man durchführt, desto weniger zufällig erscheinen die getesteten Zahlen.

Test „**Optische Inspektion**“ („*chi by eye*“): Man visualisiert die Zufallszahlen als Histogramm und schaut, ob es passen könnte.

Dieser Test ist unzureichend, da er nichts über die Abhängigkeiten der Zufallszahlen untereinander aussagt.

Zufallszahlentests

FAKULTÄT INFORMATIK

Der χ^2-Test („Chi Quadrat“, K. Pearson, 1900)

Der χ^2-Anpassungstest prüft, ob eine beobachtete (empirische) Häufigkeit einer gewünschten (erwarteten) Verteilung entspricht.

Sei N_i die Häufigkeit des Ereignisses i und n_i dessen erwartetes Vorkommen gemäß einer angenommenen Verteilung, mit
E = Anzahl der verschiedenen Ereignis-Kategorien.

Man berechnet die folgende Prüfgröße (Pearsonsche Testfunktion):

$$\chi^2 = \sum_{i=1}^{E} \frac{(N_i - n_i)^2}{n_i}$$

Tritt in χ^2 der Fall $n_i = N_i = 0$ auf, so wird dieser Term weggelassen.
Tritt hingegen $n_i = 0, N_i \neq 0$ auf, so liefert die Formel den Wert ∞
(dann ist die angenommene Verteilung wohl nicht die Richtige...).

Zufallszahlentests

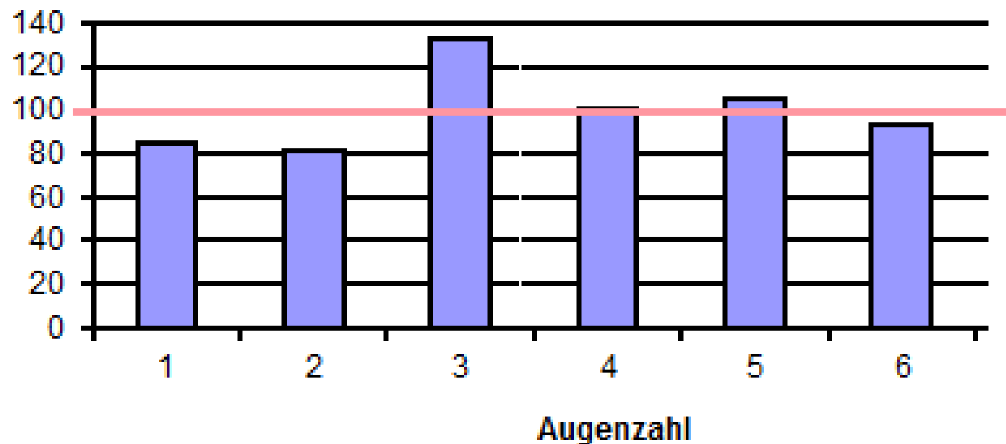
FAKULTÄT INFORMATIK

Bsp.: Ein Würfel werde 600 mal geworfen.

Die erwarteten Vorkommen sind $n_1 = n_2 = ... = n_6 = 600/6 = 100$.

Die beobachteten Häufigkeiten für jede Augenzahl seien
$N_1 = 85$, $N_2 = 81$, $N_3 = 133$, $N_4 = 101$, $N_5 = 106$, $N_6 = 94$.

Liegt eine Gleichverteilung vor?



Zufallszahlentests

FAKULTÄT INFORMATIK

Die Prüfgröße χ^2 ergibt sich zu

$$\chi^2 = (85-100)^2/100 + (81-100)^2/100 + \ldots = 17.48$$

Und was schließen wir aus diesem Wert?

Große Werte von χ^2 bedeuten große Abweichung der beobachteten Häufigkeiten von der erwarteten Verteilung.

Es wäre zwar möglich, aber sehr unwahrscheinlich, dass ein fairer Würfel z.B. 600 mal hintereinander die 6 zeigt:

$$(1/6)^{600} \approx 10^{-467} \quad \text{mit} \quad \chi^2 = 3000.0$$

Zufallszahlentests

FAKULTÄT INFORMATIK

Es wäre aber auch merkwürdig, wenn bei 600 Würfen der Würfel exakt 100 mal die 1, exakt 100 mal die 2 usw. würfelt.

$\Rightarrow$ Sehr kleine Werte von χ^2 sind ebenfalls verdächtig!

Man verwendet daher zur Qualitätsbeurteilung die Abweichung von χ^2 zum zugehörigen Wert aus der χ^2-**Verteilung**.

Diese gibt an, wie wahrscheinlich die Werte von χ^2 sind, wenn die erwartete Verteilung tatsächlich vorliegt.

Ob der konkrete Wert 17.48 gut genug ist, um die Verteilung als Gleichverteilung „durchgehen" zu lassen, ist Ansichtssache.

Zufallszahlentests

FAKULTÄT INFORMATIK

Die Statistik macht lediglich eine Aussage, wie wahrscheinlich es ist, dass bei Vorliegen einer Gleichverteilung $\chi^2 \leq 17.48$ gilt.

Letztere WS liest man aus Tabellen ab, die für verschiedene **Freiheitsgrade** f = E-1 für **Quantile** einen χ^2-Wert angeben.

Ein **p-Quantil**, $0 \leq p \leq 1$, ist ein **Lagemaß** in der Statistik:
Links vom p-Quantil liegen $100 \cdot p$ Prozent der Beobachtungswerte.

Das 0,5-Quantil ist der Median (Zentralwert).

Bsp.: 25% aller Arbeiter verdienen weniger als 2200 € pro Monat.
$\Rightarrow$ Das 0.25-Quantil („Quartil“) hat den Wert 2200 €.

Bsp.: Das 0.995-Quantil habe laut χ^2-Tabelle den Wert 16.75.
$\Rightarrow$ In 99.5% aller Fälle liegt der Wert von χ^2 unter 16.75, falls die angenommene Verteilung zutrifft.

Zufallszahlentests

FAKULTÄT INFORMATIK

Ausschnitt aus der **Quantiltabelle**, d.h. aus der Tabelle der χ^2-Verteilung als Funktion von Freiheitsgrad f und Quantil p:

f	0,900	0,950	0,975	0,990	0,995	0,999
1	2,71	3,84	5,02	6,63	7,88	10,83
2	4,61	5,99	7,38	9,21	10,60	13,82
3	6,25	7,81	9,35	11,34	12,84	16,27
4	7,78	9,49	11,14	13,28	14,86	18,47
5	9,24	11,07	12,83	15,09	16,75	20,52
6	10,64	12,59	14,45	16,81	18,55	22,46
7	12,02	14,07	16,01	18,48	20,28	24,32
8	13,36	15,51	17,53	20,09	21,95	26,12
9	14,68	16,92	19,02	21,67	23,59	27,88
10	15,99	18,31	20,48	23,21	25,19	29,59

Beim Würfel gibt es 6 Kategorien und somit f = 5 Freiheitsgrade.

In der Zeile für f = 5 liegt der Wert von $\chi^2 = 17.48$ zwischen den Quantilen 0.995 und 0.999.

Das bedeutet, die WS, dass χ^2 bei Vorliegen einer Gleichverteilung den Wert 17.48 hat, liegt zwischen 0.5% und 0.1% $\Rightarrow$ zu gering.

Eine χ^2-Quantiltabelle bis f = 40 findet man z.B. in https://de.wikibooks.org/wiki/Statistik:_Tabelle_der_Chi-Quadrat-Verteilung.

Zufallszahlentests

FAKULTÄT INFORMATIK

Für Freiheitsgrade f > 30 gilt in guter Näherung

$$\chi_p^2(f) = f + x_p\sqrt{2f} + 2/3\ x_p^2 - 2/3$$

mit $x_{0.01} = -2.33$, $x_{0.05} = -1.64$, $x_{0.5} = 0$, $x_{0.95} = 1.64$, $x_{0.99} = 2.33$.
So ist z.B. für f = 40 und p = 0.95

$$\chi_{0.95}^2(40) = 40 + 1.64 \cdot \sqrt{80} + 2/3 \cdot 1.64^2 - 2/3 = 55.795$$

Zum Vergleich: Der exakte Tabellenwert für $\chi^2_{0.95}(40)$ ist 55.76.

Faustregel zur Interpretation der Quantile der χ^2-Verteilung:
Quantil < 0.01 oder Quantil > 0.99 : Zufallszahlen zurückweisen.
Quantil < 0.05 oder Quantil > 0.95 : Zufallszahlen sind suspekt.

Knuth bezeichnet diese Regeln als suspekt...

Zufallszahlentests

FAKULTÄT INFORMATIK

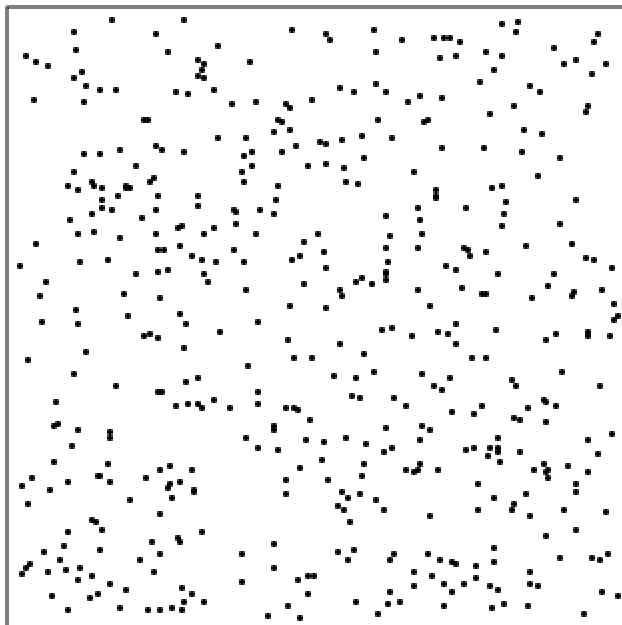
► *Eine Münze werde 1000 mal geworfen.*
Kopf erscheint 542 mal. Ist die Münze fair?

Oder anders formuliert:

Sie erzeugen mit obiger Münze Zufallsbits,
die gleichverteilt sein sollen.

Akzeptieren Sie die Zufallsbits?

Zufallszahlentests



Der Tupel-Test (*serial test*)

Den χ^2-Test haben wir oben benutzt, um eine 1-d Gleichverteilung zu analysieren.

Dieser Test lässt sich leicht auf 2-d erweitern: Man bildet aus jeweils 2 aufeinanderfolgenden Zahlen einen 2-d Punkt.

Brauchbare Zufallszahlen sollten auch im 2-d Raum gleichverteilt sind.

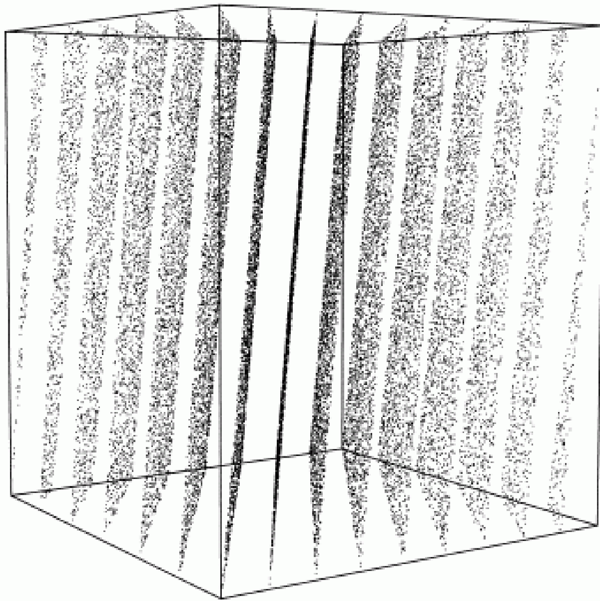
$\Rightarrow$ Wir prüfen mit dem χ^2-Test, ob 2-d Bereiche die erwarteten Anzahlen enthalten.

Die Grafik zeigt die 2-d Verteilung für den LKG

$$z_{i+1} = 65539 \cdot z_i \mod 2^{31} \qquad \text{(„RANDU“)}$$

(500 Punkte, Startwert $z_0 = 12345678$)

Zufallszahlentests



Quelle: P. Hellekalek, *Good random number generators are (not so) easy to find, 1998.*

Genauso kann man jeweils 3 Zufallszahlen zu Punkten im 3-d Raum zusammenfassen.

Die Grafik zeigt die resultierende 3-d Verteilung, wiederum für RANDU.

Diese Verteilung offenbart 15 Ebenen – definitiv keine Gleichverteilung!

Der χ^2-Test sollte bei geeigneter Wahl der Intervall-Größe dieses Problem offenbaren.

Der Mersenne-Twister ist mindestens in den ersten 623 Dimensionen gleichverteilt.

Eine leistungsstarke Verallgemeinerung des Tupel-Tests ist unter dem Namen **Spektraltest** in der Literatur zu finden.

Zufallszahlentests

FAKULTÄT INFORMATIK

Weitere Tests:

- **Lückentest** (*gap test*): Für ein Ereignis zählt man, wieviele andere Ereignisse generiert werden, bis dasselbe Ereignis wieder auftritt.
- **Sammelbildertest** (*coupon collector's test*): Hier zählt man, wieviele Ereignisse generiert werden müssen, bis jedes Ereignis mindestens einmal aufgetreten ist.
- **Maximum-von-t Test**: Man bestimmt das Maximum von Blöcken von jeweils t Zufallszahlen.
- **Lauflängentest** (*run test*): Man zählt, wieviele Zahlen hintereinander einen monoton ansteigenden Lauf bilden. Ein Lauf endet, wenn $z_i > z_{i+1}$ gilt.
- usw.

Zufallszahlentests

FAKULTÄT INFORMATIK

Bei den vorherigen Tests muss man die Verteilung der Prüfgrößen kennen. Danach kann man mit dem χ^2-Test prüfen, ob die empirische und die geforderte Verteilung übereinstimmen.

Knuth empfiehlt den Lauflängentest, allerdings mit einem Trick, der die Auswertung mit dem χ^2-Test vereinfacht:

Endet ein Lauf mit z_i, so wird z_{i+1} übersprungen und der nächste Lauf erst ab z_{i+2} gezählt.

Damit ergibt sich die WS p_r der Länge r eines Laufs zu

$$p_r = 1/r! - 1/(r+1)!$$

r könnte theoretisch sehr groß werden. Da aber p_r für größere Werte von r schnell gegen 0 geht, fasst man in der Praxis alle Lauflängen $r \geq t$ (z.B. $t = 7$) zusammen, mit $p_t = 1/t!$

Zufallszahlentests

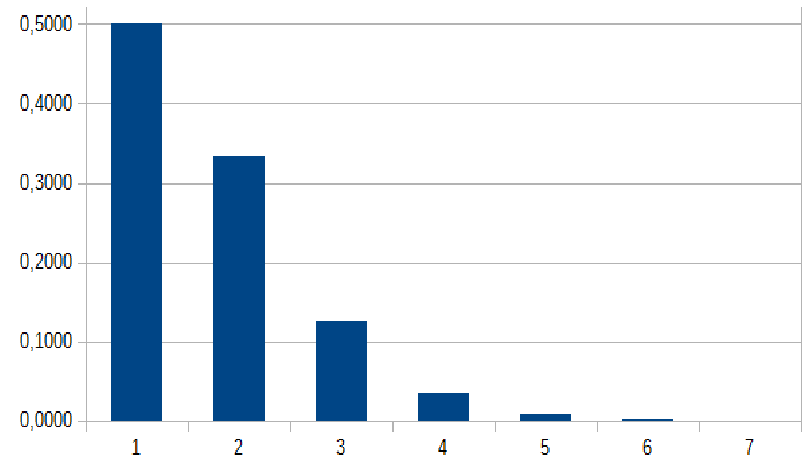
Bsp. Lauflängentest: Die beobachtete Folge mit N = 24 sei

1 6 | 1 2 3 3 4 6 6 | 5 3 4 | 1 2 9 | 8 5 | 3 6 7 | 0 4 5 6

Damit ergeben sich die folgenden Werte, mit

- L = Gesamtzahl der beobachteten Läufe = 7
- $n_r = L \cdot p_r$ = erwartete Anzahl der Läufe der Länge r
- N_r = Anzahl der beobachteten Läufe der Länge r:

r	p_r	$n_r = L \cdot p_r$	N_r
1	0.5000000	3.5000000	1
2	0.3333333	2.3333333	4
3	0.1250000	0.8750000	1
4	0.0333333	0.2333333	0
5	0.0069444	0.0486108	0
6	0.0011905	0.0083335	1
≥ 7	0.0001984	0.0013888	0



In diesem Fall ergibt sich der sehr hohe Wert $\chi^2 = 121.28$, mit f = 6.
Die Zahlen werden zurückgewiesen.

Zufallszahlentests

Ein anderer Test ergibt sich aus der Kolmogorow-Definition:

Man versucht mit einem gängigen Kompressionsprogramm die erzeugten Zahlen verlustfrei zu komprimieren.

Je größer der erzielte Kompressionsfaktor, desto weniger zufällig sind die Zahlen.

Hierbei ist wichtig, dass die Zahlen nicht in einem redundanten Format vorliegen (ASCII, 32-bit, ...).

Theoretische Tests analysieren nicht die generierten Zufallszahlen, sondern die Formeln, welche die Zufallszahlengeneratoren verwenden (sehr mathematisch...).

Zufallszahlentests

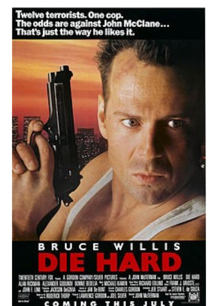
ENT

Ein kleines Werkzeug zur Beurteilung von Zufallszahlen ist ENT (www.fourmilab.ch/random). Auf Bit- oder Byte-Level werden folgende Tests durchgeführt:

- Entropie (sollte hoch sein)
- χ^2-Test (sollte brauchbares Quantil der χ^2-Verteilung liefern)
- Durchschnittswert (0.5 bei Bits, 127.5 bei *unsigned* Bytes)
- Monte Carlo Berechnung von π (dicht bei 3.141592654...)
- Autokorrelation c aufeinander folgender Zahlen ($-1 \leq c \leq 1$, c sollte bei 0 liegen)

Umfangreiche Tests („*batteries*“) gibt es u.a. unter

- http://simul.iro.umontreal.ca („TestU01“)
- http://webhome.phy.duke.edu/~rgb/General/dieharder.php



Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Probabilistische (**randomisierte, stochastische**) **Algorithmen**, deren Ablauf von Zufallszahlen abhängt, werden eingesetzt, wenn

- keine analytische Lösungen zu einem Problem bekannt sind
- die bekannten Lösungen hartnäckig (NP-schwer) sind.

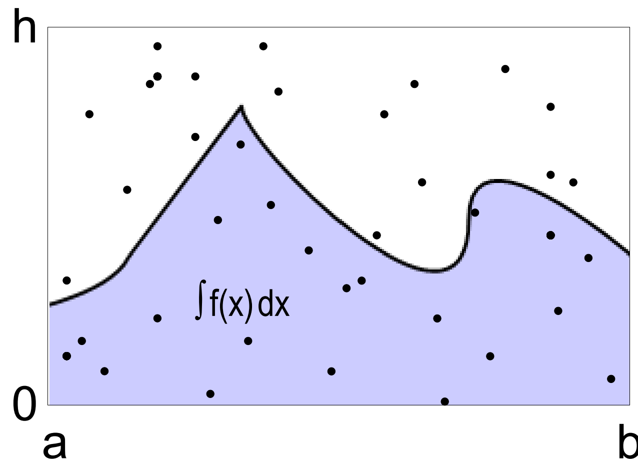
Wenn die richtige bzw. optimale Lösung nicht berechnet werden kann, so kann ein probabilistischer Algorithmus möglicherweise

- mit **hoher Wahrscheinlichkeit** die richtige Lösung oder
- eine **brauchbare Annäherung** an das Optimum berechnen.

Im weiteren Sinne kann man auch **Simulationen** sowie rechnerbasierte **(Glücks-) Spiele** als probabilistische Algorithmen auffassen.

Probabilistische Algorithmen

Lösung von Integralen mit der Monte Carlo Methode



Gegeben: Funktion f(x)

Gesucht: $\int f(x)\,dx$ im Interval [a, b]

Methode: Man „beschießt“ das umgebende Rechteck mit zufälligen, gleichverteilten Punkten.

Die Fläche ergibt sich gemäß

$$\frac{\int_a^b f(x)\,dx}{(b-a)\cdot h} = \frac{N_f}{N}$$

Rechteckfläche = (b-a) · h, mit $h \geq \max f(x)$ im Interval [a, b]

N = Versuche = Anzahl Punkte

N_f = Treffer = Anzahl der Punkte (x, y) mit $y \leq f(x)$

Je größer N, desto genauer die Lösung.

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

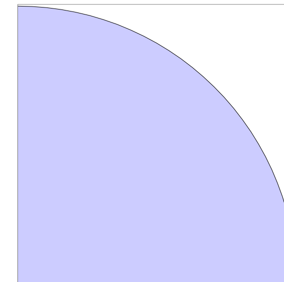
Aus der Monte Carlo Methode lässt sich sofort ein weiterer Zufallszahlentest (**Monte Carlo Pi**) konstruieren:

Man verwendet die zu testenden Zahlen, um die Fläche eines Viertelkreises (der Einfachheit halber mit Radius $r = 1$) zu bestimmen.

Umschließt man den Viertelkreis mit einem Einheitsquadrat, so ergibt sich die Monte Carlo Formel

$$\pi = 4 \cdot N_f / N$$

Sind die zu testenden Zahlenpaare zufällig, so sollte sich eine gute Annäherung an $\pi = 3.1415926$ ergeben.



Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Primzahltests

Der erste Schritt im RSA-Verschlüsselungsverfahren besteht darin, zwei große (z.B. 1024 bit lange) Primzahlen zufällig auszuwählen.

Die bekannten deterministischen Verfahren (Probedivision, Sieb des Eratosthenes, usw.) können in vertretbarer Zeit nicht berechnen, ob eine derartige Zahl ($2^{1024} \approx 10^{308}$) prim ist oder zusammengesetzt:

Bei der Probedivision müssten wir bis zu $\sqrt{10^{308}} \approx 10^{154}$ Divisionen ausführen, um sicher zu stellen, dass eine solche Zahl prim ist.

1977 veröffentlichten Rivest/Shamir/Adleman die Zahl „RSA-129“ =
114381625757888867669235779976146612010218296721242362562561842935
706935245733897830597123563958705058989075147599290026879543541.

Diese wurde erst 1994, nach 8 Monaten CPU-Laufzeit, faktorisiert als
3490529510847650949147849619903898133417764638493387843990820577 ·
32769132993266709549961988190834461413177642967992942539798288533.

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Kleiner Satz von **Fermat**: Sei p eine Primzahl und a eine ganze Zahl, welche keine Vielfache von p ist. Dann gilt

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

Bsp.: $p = 7, \; a = 6 \;\; \Rightarrow \;\; 6^{7-1} \bmod 7 = 46656 \bmod 7 = 1$
mit $46656 = 7 \cdot 6665 + 1$.

Also: $a^{p-1} \bmod p \neq 1 \;\Rightarrow$ Mit Sicherheit ist p keine Primzahl.
Der Test sagt aber nichts über die Teiler von p aus.

Die Umkehrung des Satzes von Fermat gilt *fast* immer:
Sei p eine Zahl, von der wir nicht wissen, ob sie prim ist.
Gilt $a^{p-1} \bmod p = 1$, so ist p wahrscheinlich eine Primzahl.
Wir nennen p dann eine **Basis-a-Pseudoprimzahl**.

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Die Wahrscheinlichkeit, dass der obige Test eine korrekte Antwort liefert, lässt sich weiter dadurch erhöhen, dass man ihn mit verschiedenen Werten von a durchführt.

Bsp.: Sei $p = 341$.

Test mit $a = 2$: $2^{341-1} \bmod 341 = 1$ // 341 könnte Primzahl sein

Test mit $a = 3$: $3^{341-1} \bmod 341 = 56$ // 341 ist keine Primzahl!

In der Tat gibt es die Faktorisierung $341 = 11 \cdot 31$.

Es verbleiben jedoch einige „Spielverderber“:

Diejenigen Zahlen c, für die $a^{c-1} \bmod c = 1$ gilt, obwohl c keine Primzahl ist, und zwar für alle a, die teilerfremd sind zu c.

Diese c heißen **Carmichael-Zahlen**.

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Jede Carmichael-Zahl c ist das Produkt von mindestens drei Primteilern p_i.

Satz von Korselt: Eine Zahl n ist genau dann eine Carmichael-Zahl, wenn sie quadratfrei ist (d.h. keiner der Primteiler kommt mehrfach vor = keine Quadratzahl $\neq 1$ teilt n) und für jeden Primteiler p_i gilt, dass p_i-1 die Zahl n-1 teilt.

Es gibt unendlich viele Carmichael-Zahlen. Im Bereich $[1, 10^9]$ sind es 646 solcher Zahlen, die kleinste ist 561.

Als untere Abschätzung der Anzahl C(x) der Carmichael-Zahlen im Bereich [1..x] für hinreichend großes x gilt $C(x) > x^{1/3}$.

c	Primteiler
561	3·11·17
1105	5·13·17
1729	7·13·19
2465	5·17·29
2821	7·13·31
6601	7·23·41
8911	7·19·67
10585	5·29·73
15841	7·31·73
29341	13·37·61
41041	7·11·13·41

n	1	2	3	4	5	6	7	8	9	10
$C(10^n)$	0	0	1	7	16	43	105	255	646	1547

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Algorithmus `Probabilistischer Primzahltest`
Eingabe: Ungerade Ganzzahl `n>2`
Ausgabe: `n` ist zusammengesetzt oder wahrscheinlich Primzahl

```
Für mehrere Werte von a mit a ∈{2, 3, ... n-1}
    Wenn a^(n-1) mod n ≠ 1
    Dann „n ist zusammengesetzt“; exit
„n ist wahrscheinlich Primzahl“
```

Die Auswahl der Werte für a ist offen, also probabilistisch.

Für Zahlen, die weder prim noch carmichael sind, führt der Test sehr schnell zum korrekten Ergebnis.

Bei den Carmichael-Zahlen c führen nur die Primteiler von c und ihre Vielfachen zum korrekten Ergebnis.

Verbesserung: Der **Miller-Rabin-Test** (1976).

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Für große n muss die Potenzierung a^n deutlich schneller gehen als die (n-1)-malige Multiplikation $a \cdot a \cdot a \cdot \ldots \cdot a \Rightarrow$ Wiederholte Quadrierungen:

$$a^n = \begin{cases} 1 & \text{wenn } n = 0 \\ a^{2 \cdot n/2} = (a^{n/2})^2 & \text{wenn } n \text{ gerade} \\ a \cdot a^{n-1} & \text{wenn } n \text{ ungerade} \end{cases}$$

```
Algorithmus Schnelle Potenzierung
Eingabe:    Basis a, Exponent e
Ausgabe:    erg = a^e
erg = 1
Solange e > 0
   Wenn e ungerade   // e mod 2 = 1
   Dann erg = erg*a
   a = a*a
   e = e/2  // Integer-Arithmetik
```

Bsp.: Berechnung von

$$3^{11} = 3^{2^3+2^1+2^0} = 3^8 \cdot 3^2 \cdot 3^1 = 6561 \cdot 9 \cdot 3 = 177\,147$$

a	3	9	81	6561	43046721
e	11	5	2	1	0
erg	1	3	27	27	177147

Probabilistische Algorithmen

Der Wert von a^n wird sehr schnell sehr groß (→ Überlauf).
Zum Glück müssen wir aber nur a^{n-1} mod n berechnen. Es gilt:

Ist $a' = a \bmod m$ und $b' = b \bmod m$, ⇐ a, b können groß sein.
so ist $(a' \cdot b') \bmod m = (a \cdot b) \bmod m$. ⇐ a', b' sind klein (< m).

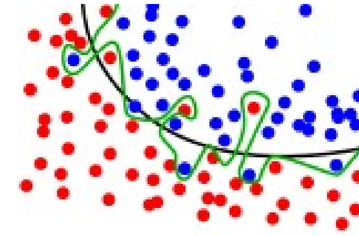
```
Algorithmus Modulare Potenzierung
Eingabe:    Basis a, Exponent e, Modul m
Ausgabe:    erg = a^e mod m

erg = 1
Solange e > 0
   Wenn e ungerade   // e mod 2 = 1
   Dann erg = (erg*a) mod m
   a = (a*a) mod m
   e = e/2  // Integer Arithmetik
```

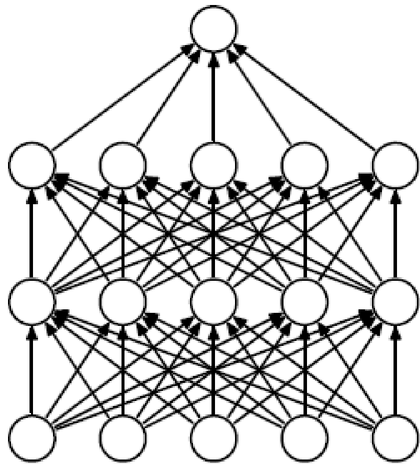
Bsp.: Berechnung von

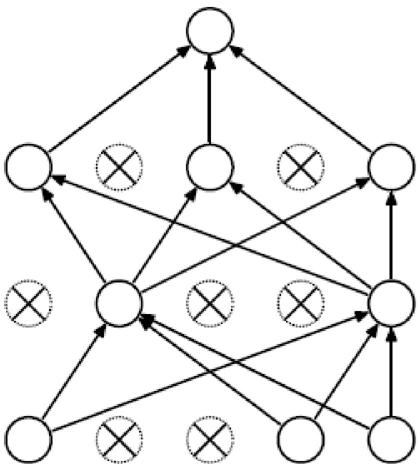
$3^{11} \bmod 7$
$= 177\,147 \bmod 7$
$= 5$

a	3	2	4	2	4
e	11	5	2	1	0
erg	1	3	6	6	5

Probabilistische Algorithmen







Randomisierung in Neuronalen Netzen: *Dropout*

Die Gewichte (= Parameter) eines NN werden gemäß den Trainingsdaten gelernt.

Dies kann zu einer **Überanpassung** (*overfitting*) mit resultierender schlechter **Generalisierung** führen: Die Erkennungsleistung wird niedrig auf Daten, die nicht zum Trainingssatz gehören.

Die *Dropout*-Lösung sieht vor, dass beim Training zufällig Neuronen vorübergehend aus dem Netzwerk ausgeschlossen werden.

Auf diese Weise kann das NN besser mit Variationen beim Eingabemuster fertig werden.

Quelle: N. Srivastava et al., *Dropout: A Simple Way to Prevent Networks from Overfitting*, JMLR 2014.

Probabilistische Algorithmen

FAKULTÄT INFORMATIK

Viele Einrichtungen tun sich schwer, auf gerechte Weise die besten Forschungsvorhaben und Bewerber auszuwählen. Dabei gäbe es eine elegante Lösung: Losverfahren.

Kommentar von Christian Gschwendtner

Exklusiv **Förderung**

Digitalisierung per Losverfahren

Peter Altmaier hatte viel vor mit seinem Digitalisierungsprogramm.

(SZ vom 22.12.2020)

Die SZ vom 31.08.2018 schlägt vor, Fördermittel für Forschungsprojekte, Nobel- und Literaturpreise, das Papst-Amt, etc. nach einem zweistufigen Verfahren zu vergeben:

Experten treffen eine Vorauswahl, danach entscheidet das Los.

Vorteile: Schneller, einfacher, gerechter – vielleicht sogar besser!

```
Algorithmus  GerechteAuswahl
Eingabe:     n auszuwählende Objekte
             k   // Anzahl vorselektierter Objekte
Ausgabe:     Ausgewähltes Objekt s

Experten selektieren k Objekte  // Vorauswahl
s = Random(1..k)
```

Gleitkommaarithmetik

Eine Quelle von scheinbar zufälligem Verhalten ist die **Gleitkommaarithmetik** mit endlich vielen Bits.

Das Problem: Es gibt (überabzählbar) unendlich viele reelle Zahlen, aber wir können nur endlich viele Gleitpunktzahlen darstellen – z.B. 2^{32} Zahlen mit einfacher und 2^{64} Zahlen mit doppelter Genauigkeit.

Welche Zahlen genau man so darstellen kann, hängt davon ab, wieviele Bits man für Exponent bzw. Mantisse spendiert.

Der **IEEE Standard 754** (1985; 2008) legt diese Aufteilung fest:

31	30							23	22																						0
0	0	1	1	0	0	1	0	1	1	0	0	0	1	1	1	1	0	1	1	0	0	0	1	1	0	1	1	1	0	0	1
VZ	Exponent								Mantisse																						

Gleitkommaarithmetik

FAKULTÄT INFORMATIK

Die Auswirkungen der Endlichkeit von Gleitkommazahlen sind eher selten (Überlauf, Unterlauf) oder vernachlässigbar (Rundungsfehler). Aber manchmal gibt es dramatische Auswirkungen – Grundgesetze der Mathematik werden möglicherweise ausgehebelt.

Die folgende Methode zeigt, dass auf einem realen Rechner (nicht aber auf einer Turingmaschine) $x + (y + z) \neq (x + y) + z$ gelten kann!

```
public static void main (String[] args) {
       float x = -1e38f;                    // 1e38f = (float) 10^38
       float y =  1e38f;
       float z =   1.0f;
       float sum1 =  x + (y  + z);
       float sum2 = (x +  y) + z;
       System.out.println("sum1: " + sum1);
       System.out.println("sum2: " + sum2);
}
```

Gleitkommaarithmetik

FAKULTÄT INFORMATIK

Also: Informatik $\neq$ Mathematik. Mathematische Äquivalenzen sind bei Berechnungen mit endlichen Resourcen nicht mehr unbedingt äquivalent.

$\Rightarrow$ Wegen begrenzter Genauigkeit robuste Berechnung anstreben!

Bsp.: Berechnung des Betrages einer komplexen Zahl z:

$$|z| = |a + b\,i| = \sqrt{(a^2 + b^2)}.$$

Naive Implementation: Überlauf bei großem a oder b. Besser:

$$|z| = \sqrt{(a^2 + b^2)} = |a|\,\sqrt{((a^2 + b^2)/a^2)} = |a|\,\sqrt{((1 + (b/a)^2)} \quad \text{bzw.}$$

$$|z| = \sqrt{(a^2 + b^2)} = |b|\,\sqrt{((a^2 + b^2)/b^2)} = |b|\,\sqrt{((1 + (a/b)^2)}.$$

Die numerisch stabilste Formel zur Berechnung von |z| lautet somit

$$|a + ib| = \begin{cases} |a|\sqrt{1 + (b/a)^2} & \text{wenn } |a| \geq |b| \\ |b|\sqrt{1 + (a/b)^2} & \text{wenn } |a| < |b| \end{cases}$$

Gleitkommaarithmetik

32-bit Gleitpunktzahlen sind zwar unpräziser als 64-bit Zahlen, aber auf 32-bit Prozessoren schneller und energiesparender.

$\Rightarrow$ Auf Mobilgeräten (fast) immer Typ `float` statt `double` verwenden und `int` oder `byte` statt `long`.

Man kann diese Idee weiter treiben und Energie sparen, indem man mit weniger Genauigkeit (< 32 bits) rechnet und Schaltkreise abschaltet.

Dies nennt sich *Inexact Design* oder auch *Probabilistic Design*.

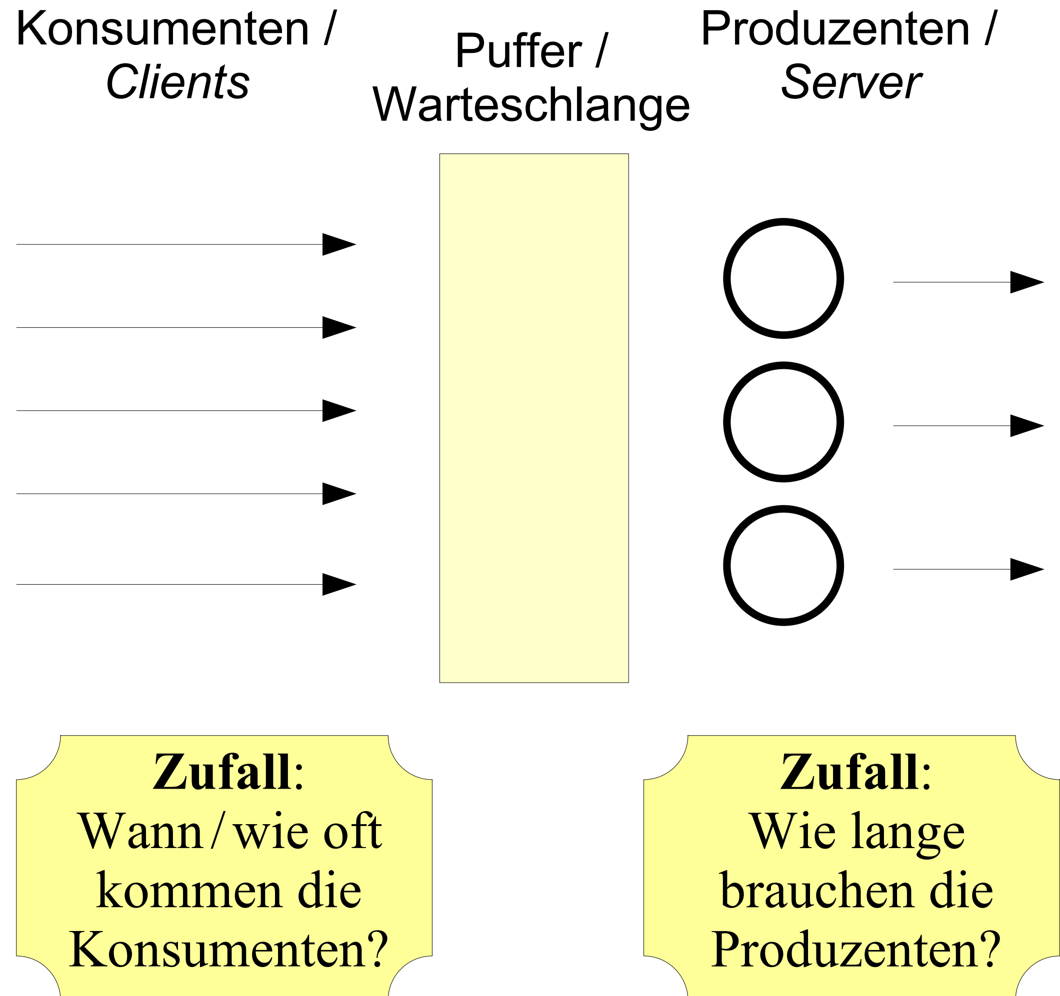
Resultierende ungenaue Ergebnisse können bei zahlreichen Applikationen (Video, Audio) sehr wohl akzeptabel sein.

Wartesysteme

Wartesysteme

In vielen Systemen konkurrieren zahlreiche **Konsumenten** (Nutzer, Kunden, *Clients*) um Resourcen, die von **Produzenten** (*Servers*) bereitgestellt werden.

Die Synchronisation der Prozesse erfolgt in der Regel über Puffer bzw. **Warteschlangen**.



Wartesysteme

FAKULTÄT INFORMATIK

Beispiele für Wartesysteme:

- Supermarktkassen
- Flughafenschalter
- Kreuzungen / Verkehrsampeln
- Tankstelle
- Arztbesuch, Impfzentrum
- Wohnheimanträge
- Fertigungsketten
- Logistikketten
- Call-Center
- Webserver
- Interprozesskommunikation
- ...

Implizite Warteschlangen (*Pipes*) in Unix/Linux zur Prozesssynchronisation:

```
ls | grep pdf | wc
```

hier mit

`ls` = list

`grep` = global regular expression / print

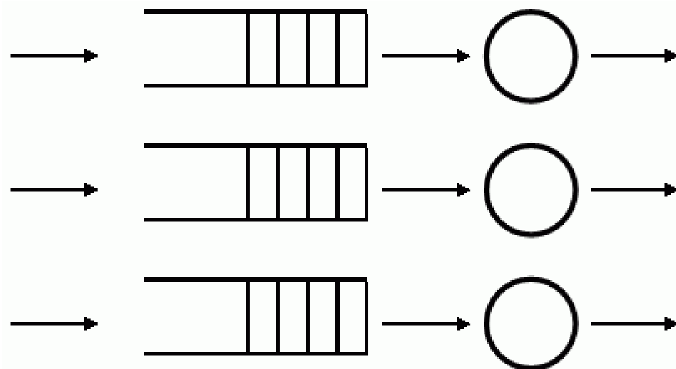
`wc` = word count

Wartesysteme

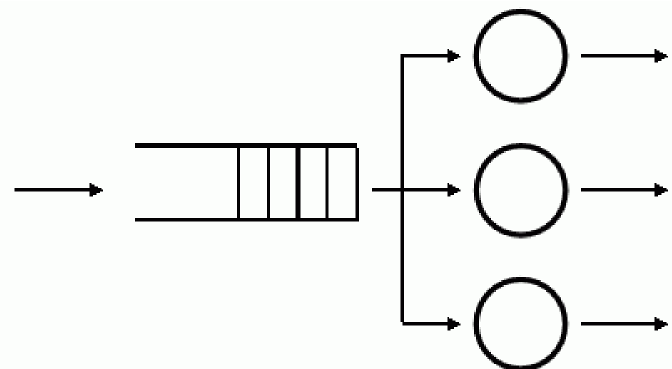
Wartesysteme können unterschiedlich organisiert sein:

- Beim Supermarkt hat man pro Bedieneinheit (= pro Kasse) eine Warteschlange, also bei n Kassen n parallele Warteschlangen.
- Beim Flughafenschalter hat man eine Warteschlange für alle Bedieneinheiten (= für alle Abfertigungsschalter) zusammen.

► *Welches dieser Wartesysteme ist besser*?



WS – Supermarktkassen



WS – Flughafenschalter

Wartesysteme

FAKULTÄT INFORMATIK

Die statische Struktur von Wartesystemen wird beschrieben durch:

- Der **Ankunftsprozess** A.

Dieser beschreibt die statistische Verteilung der **Zwischenankunftszeiten** T_A der Konsumenten.

- Der **Abfertigungsprozess** B.

Dieser beschreibt die statistische Verteilung der **Bedienzeiten** T_B der Produzenten.

- Die **Anzahl** c **der Produzenten**.

Die Zwischenankunftszeit ist die Zeitspanne, die zwischen der Ankunft zweier aufeinander folgenden Konsumenten vergeht.

Wartesysteme

FAKULTÄT INFORMATIK

Ankunfts- und Bedienprozess sind häufig **exponentialverteilt**.

Die aufgeführten Charakteristika eines WS-Systems werden oft in der verkürzten **Kendall-Notation** wie folgt zusammengefasst:

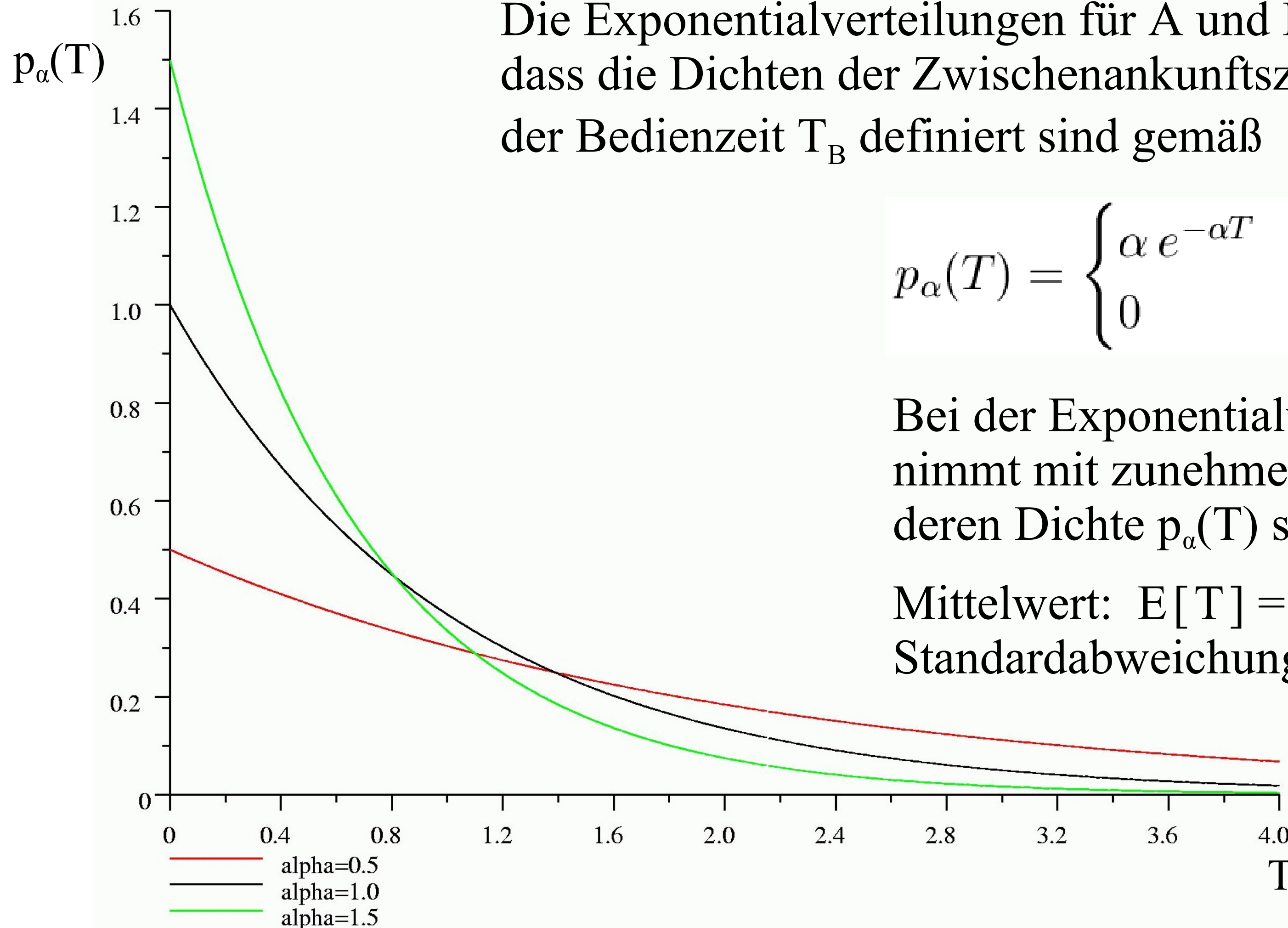
WS = A / B / c

wobei man A / B ersetzt durch Abkürzungen für die jeweilige Verteilung, z.B.

- **M** für die Exponentialverteilung
- **E** für die Erlang-Verteilung, einer Verallgemeinerung von **M**.

In dieser Notation lautet also die Beschreibung des oben grafisch dargestellten Flughafenschalter-Systems WS = **M** / **M** / **3**, wenn man Exponentialverteilungen zugrunde legt.

Wartesysteme



Die Exponentialverteilungen für A und B bedeuten, dass die Dichten der Zwischenankunftszeit T_A bzw. der Bedienzeit T_B definiert sind gemäß

$$p_\alpha(T) = \begin{cases} \alpha\, e^{-\alpha T} & \text{für } T \geq 0 \\ 0 & \text{für } T < 0 \end{cases}$$

Bei der Exponentialverteilung nimmt mit zunehmender Zeit T deren Dichte $p_\alpha(T)$ schnell ab.

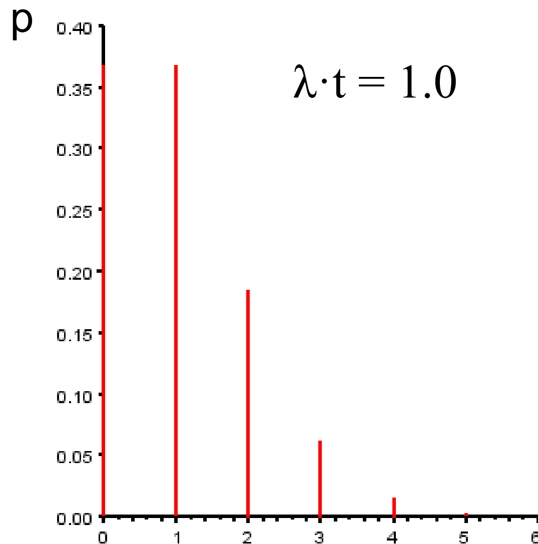
Mittelwert: $E[T] = 1/\alpha$
Standardabweichung: $\sigma = 1/\alpha$

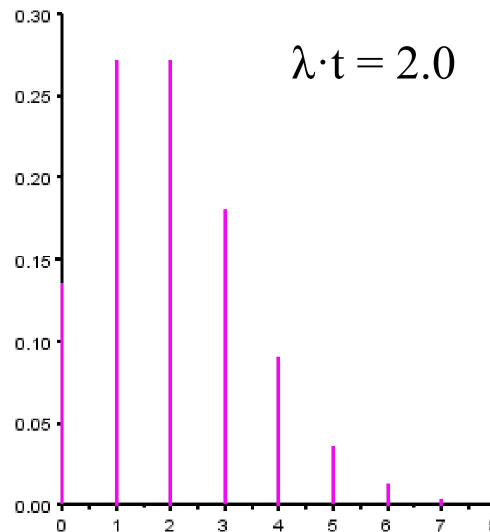
Wartesysteme

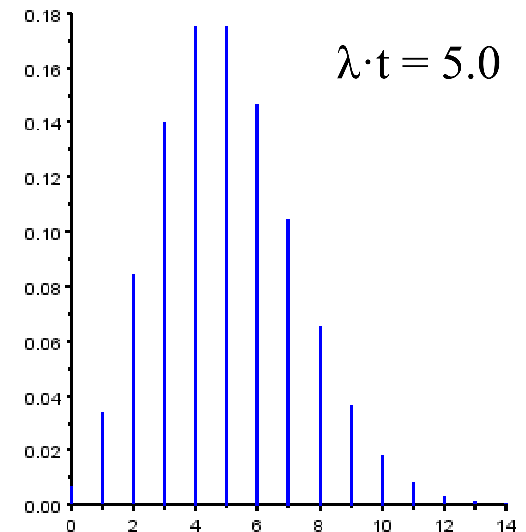
FAKULTÄT INFORMATIK

Sind die Zwischenankunftszeiten exponentialverteilt und die Ankunftsrate $\lambda = \alpha$ bekannt, so kann man die Wahrscheinlichkeit berechnen, dass im Zeitintervall t genau n Konsumenten eintreffen:

$$P(n; t, \lambda) = \frac{(\lambda t)^n \, e^{-\lambda t}}{n!}, \quad n = 0, 1, 2, \ldots$$







Wartesysteme

FAKULTÄT INFORMATIK

Die Produzenten bedienen die Konsumenten i.d.R. in der Reihenfolge ihres Eintreffens (**FCFS** = *First Come, First Served*).

Alternative **Abfertigungsstrategien** wären z.B.

- **LCFS** (*Last Come, First Served*)
- **SIRO** (*Service In Random Order*)
- **RR** (*Round Robin*).

Bei *Round Robin* wird jeder Konsument für eine feste Zeitspanne bedient. Ist der Auftrag innerhalb dieser Zeitspanne noch nicht abgearbeitet, so wird das Zwischenergebnis gespeichert und der Konsument am Ende der Warteschlange eingereiht.

Wären die Abfertigungszeiten bekannt, so würde die Strategie *Shortest First* zu verkürzter durchschnittlicher Wartezeit führen.

Wartesysteme

Dynamischen Größen zur Quantifizierung von Wartesystemen:

- N_q, die Anzahl der Konsumenten, die aktuell in der WS sind.
- N_s, die Anzahl der Konsumenten, die aktuell bedient werden.
- $N = N_q + N_s$, die aktuelle Gesamtzahl der Konsumenten im System.
- T_A, die Zwischenankunftszeiten.
- $\lambda = 1 / E[T_A]$, die Ankunftsrate.
- T_B, die Bedienzeiten.
- $\mu = 1 / E[T_B]$, die Bedienrate.
- T_W, die Wartezeit eines Konsumenten in der WS.
- $T = T_W + T_B$, die gesamte Verweildauer eines Konsumenten.

Notation: $E[x]$ bezeichnet den **Erwartungswert** (= Mittelwert) von x.

Wartesysteme

FAKULTÄT INFORMATIK

Man definiert die **Auslastung** ρ als

$$\rho = \frac{\text{mittlere Bedienzeit}}{\text{mittlere Zwischenankunftszeit}} = \frac{\text{Ankunftsrate}}{\text{Bedienrate}} = \frac{\lambda}{\mu}$$

Bei c Bedieneinheiten gilt entsprechend $\rho = \lambda / (c \cdot \mu)$.

Sinnvollerweise sollte $\rho < 1$ gelten, da sonst die Länge der Warteschlange gegen unendlich wachsen würde.

Die Auslastung bestimmt die erwartete Länge der Warteschlange:

$$E[N] = \rho / (1 - \rho)$$

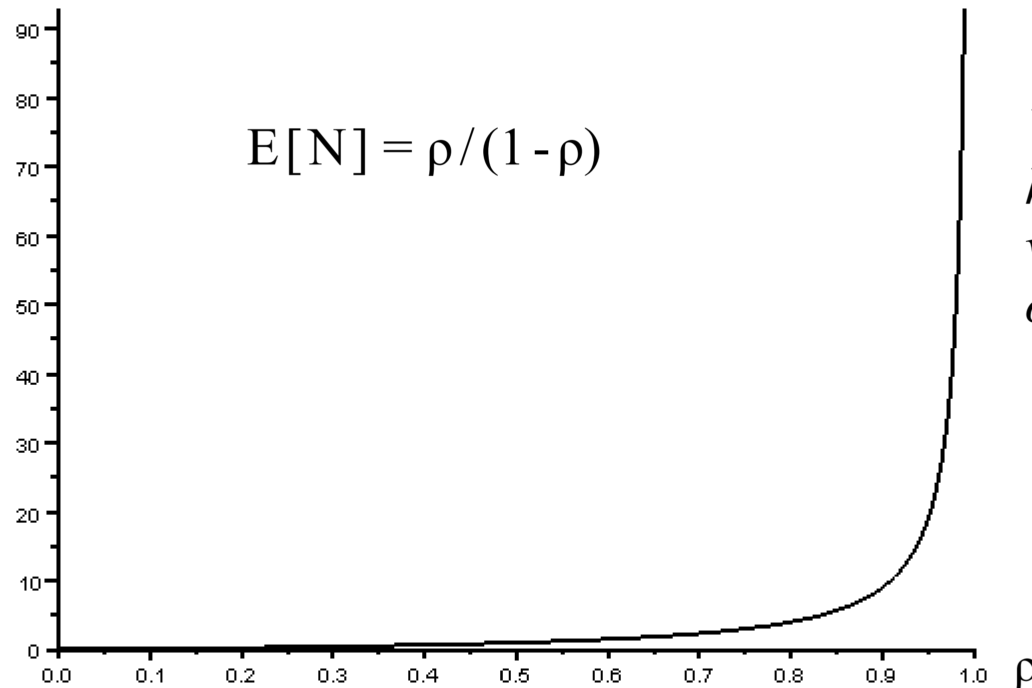
$$E[N_q] = \rho \cdot E[N] = \rho^2 / (1 - \rho)$$

Wartesysteme

Die Grafik zeigt E [N] als Funktion von ρ.

Die Funktion ist hochgradig nicht-linear. Jenseits einer Auslastung von etwa 85% steigt die Länge der Warteschlangen stark an.

⇒ Hohe Auslastungen führen zum Systemzusammenbruch!



Vorsicht: Anzugträger halten eine Auslastung von weniger als 100% oft für „ineffizient“.

Wartesysteme

FAKULTÄT INFORMATIK

Die durchschnittliche Länge der Warteschlange $E[N_q]$ ist proportional zur durchschnittlichen Verweildauer $E[T_W]$ eines Konsumenten in der Warteschlange.
Der Proportionalitätsfaktor ist die Ankunftsrate λ.

Diese Aussage ist bekannt als der **Satz von Little** (1961):

$$E[N_q] = \lambda \cdot E[T_W] \quad \text{bzw.} \quad E[N] = \lambda \cdot E[T]$$

Diese Gesetzmäßigkeit gilt

- für jegliche Verteilungen von T_A und T_B
- für jegliche Abfertigungsstrategien.

Es bleibt aber ein schwieriges Unterfangen, $E[T_W]$ oder $E[N_q]$ im nichttrivialem Fall analytisch zu bestimmen.
$\Rightarrow$ Wartesysteme werden gerne simuliert.

Wartesysteme

FAKULTÄT INFORMATIK

► *Rechnen Sie folgendes Beispiel durch:*

Bei einer Kassiererin kommen 15 Kunden pro Stunde vorbei. Die durchschnittliche Abfertigungszeit beträgt 3 min.

Wie groß ist die durchschnittliche Warteschlangenlänge $E[N_q]$ *und die erwartete Wartezeit* $E[T_W]$*?*

Wie ändern sich diese Kennzahlen, wenn 18 Kunden pro Stunde kommen?

Wartesysteme

FAKULTÄT INFORMATIK

Beispiel zur Simulation eines **M/M/1** – Systems mit n = 6 Konsumenten:

i	1	2	3	4	5	6
$T_A(i)$	0.0	1.0	0.5	4.0	1.5	8.0
$T_B(i)$	2.0	4.0	4.0	1.0	1.0	3.0
$T_{Ank}(i)$	0.0	1.0	1.5	5.5	7.0	15.0
$T_{Anf}(i)$	0.0	2.0	6.0	10.0	11.0	15.0
$T_{End}(i)$	2.0	6.0	10.0	11.0	12.0	18.0
$T_W(i)$	0.0	1.0	4.5	4.5	4.0	0.0

$\Rightarrow E[T_W] = \Sigma_i T_W(i)/n =$
$(0 + 1 + 4.5 + 4.5 + 4 + 0)/6 = 14/6 = 2.333$

T_A und T_B werden exponentialverteilt von einem Zufalls-zahlengenerator generiert.

Man startet zum Zeitpunkt T = 0.

Alle weiteren Größen, d.h. Ankunftszeit T_{Ank}, Anfangszeit T_{Anf}, Endzeit T_{End} und Wartezeit T_W lassen sich direkt berechnen.

Wartesysteme

FAKULTÄT INFORMATIK

Berechnung der Kenngrößen (Abfertigungsstrategie FCFS, d.h. Abfertigung in der Reihenfolge des Eintreffens der Konsumenten):

- $T_{Ank}(1) = 0$
- $T_{Ank}(i) = T_{Ank}(i-1) + T_A(i)$
- $T_{Anf}(1) = 0$
- $T_{Anf}(i) = \max(T_{Ank}(i), T_{Anf}(i-1) + T_B(i-1))$
- $T_{End}(i) = T_{Anf}(i) + T_B(i)$
- $T_W(i) = T_{Anf}(i) - T_{Ank}(i)$
- $E[T_W] = 1/n \cdot \Sigma_i T_W(i)$

Die Gesamtlänge des Simulationszeitraums ist

- $T_{gesamt} = T_{End}(n) - T_{Ank}(1)$

Wartesysteme

FAKULTÄT INFORMATIK

D. Maister hat in *The Psychology of Waiting Lines* (1985) folgende Leitsätze über das subjektive Empfinden von Warteschlangen aufgestellt:

- *Occupied Time Feels Shorter Than Unoccupied Time.*
- *Anxiety Makes Waits Seem Longer.*
- *Uncertain Waits Are Longer than Known, Finite Waits.*
- *Unexplained Waits Are Longer than Explained Waits.*
- *Unfair Waits Are Longer than Equitable Waits.*
- *The More Valuable the Service, the Longer the Customer Will Wait.*
- *Solo Waits Feel Longer than Group Waits.*

► *Wie können Oberflächen und Webseiten die „gefühlten Wartezeiten" für den Nutzer zu verkürzen?*