Mere om tilfældige tal

Denne artikel bygger oven på den forrige artikel om tilfældige tal.

Den forudsætter at man har læst den forrige artikel og har et vist kendskab til matematik.

Skrevet den **14. Feb 2010** af **arne_v** I kategorien **Programmering / Generelt** | ★★★☆☆

Historie:
V1.0 - 23/05/2005 - original
V1.1 - 29/12/2008 - tilføj links
V1.2 - 12/02/2010 - smårettelser

Tænke sig godt om

Forrige artikel http://www.eksperten.dk/guide/680 "Tilfældige tal" har forhåbentligt gjordt det klart at man skal tænke sig godt om i forbindelse med bruge af tilfældige tal generatorer.

Man skal:
1) vælge en god algoritme
2) vælge en fornuftig initialisering af algoritmen
3) undlade at ødelægge algoritmen med uheldige transformeringer

Med hensyn til #1 så er det bedste at finde en kendt algoritme som diverse matematikere ha sagt god for, det næstbedste er at basere sig på at den indbyggede tilfældige tal generator i ens sprog er god nok mens det dårligste er at forsøge at brygge noget sammen selv.

Med hensyn til #2 så skal man tænke sig godt om. Jeg kommer om lidt med yderligere et eksempel.

Med hensyn til #3 så skal man lade være med at transformere mere en allerhøjest nødvendigt. Punktum.

Når det drejer sig om initialisering så er det vigtigt at forstå at den rigtige initialisering afhænger af konteksten.

I alle de foregående eksempler har jeg brugt tiden. Det er også meget godt til mange formål.

Lad os tage et par eksempler fra Poker (det er jo meget moderne).

Hvor stor er sandsynligheden for at få en flush (5 kort af samme farve) ?

For nemheds skyld inkluderer vi straight flush og royal straight flush

i statistikken.

Poker1.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Poker1 {
    private static Random rng = new Random(System.currentTimeMillis());
    private List deck;
    public Poker1() {
        deck = new ArrayList();
        for(int i = 0; i < 52; i++) {
            deck.add(new Integer(i));
        }
    }
    public int takeCard() {
        Integer card = (Integer)deck.remove(rng.nextInt(deck.size()));
        return card.intValue();
    }
    public int[] takeHand() {
        int[] res = new int[5];
        for(int i = 0; i < 5; i++) {
            res[i] = takeCard();
        }
        return res;
    }
    public static int getColour(int card) {
        return (card / 13);
    }
    public static int getValue(int card) {
        return (card % 13);
    }
    public static boolean isFlush(int[] hand) {
        if((getColour(hand[0]) == getColour(hand[1])) &&
           (getColour(hand[1]) == getColour(hand[2])) &&
           (getColour(hand[2]) == getColour(hand[3])) &&
           (getColour(hand[3]) == getColour(hand[4]))) {
            return true;
        } else {
            return false;
        }
    }
    private static final int REP = 1000000;
    private static final int PLAYERS = 4;
    public static void main(String[] args) {
        int nflush = 0;
        int[] hand = new int[5];
        for(int i = 0; i < REP; i++) {
            Poker1 game = new Poker1();
            for(int j = 0; j < PLAYERS; j++) {
                if(Poker1.isFlush(game.takeHand())) {
```

```
                    nflush++;
                }
            }
        }
        System.out.println(PLAYERS*REP/(double)nflush);
    }
}
```

Programmet udskriver et tal som i de fleste tilfælde vil ligge mellem 500 og 510, hvilket passer meget fint da den ægte sandsynlighed er 1 ud af 505.

Med andre ord virker vores tilfældige tal generator glimrende.

Så er vi vel klar til at lave et poker spil ?

Poker2.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Poker2 {
    private static Random rng = new Random(System.currentTimeMillis());
    private List deck;
    public Poker2() {
        deck = new ArrayList();
        for(int i = 0; i < 52; i++) {
            deck.add(new Integer(i));
        }
    }
    public int takeCard() {
        Integer card = (Integer)deck.remove(rng.nextInt(deck.size()));
        return card.intValue();
    }
    public int[] takeHand() {
        int[] res = new int[5];
        for(int i = 0; i < 5; i++) {
            res[i] = takeCard();
        }
        return res;
    }
    public static int getColour(int card) {
        return (card / 13);
    }
    public static int getValue(int card) {
        return (card % 13);
    }
    private static final int PLAYERS = 4;
    public static void main(String[] args) {
        int[][] hand = new int[PLAYERS][5];
```

```
        Poker2 game = new Poker2();
        for(int i = 0; i < PLAYERS; i++) {
            hand[i] = game.takeHand();
            for(int j = 0; j < 5; j++) {
                System.out.print(" " + hand[i][j]);
            }
            System.out.println();
        }
    }
}
```

Eksempel på output:

```
8 36 4 18 16
50 35 29 27 39
28 42 48 31 1
24 9 5 43 2
```

Forskellige statistiske test på de hænder vil se OK ud.

Men prøv lige og check det her program.

Poker3.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

public class Poker3 {
    private Random rng;
    private List deck;
    public Poker3(long seed) {
        rng = new Random(seed);
        deck = new ArrayList();
        for(int i = 0; i < 52; i++) {
            deck.add(new Integer(i));
        }
    }
    public int takeCard() {
        Integer card = (Integer)deck.remove(rng.nextInt(deck.size()));
        return card.intValue();
    }
    public int[] takeHand() {
        int[] res = new int[5];
        for(int i = 0; i < 5; i++) {
            res[i] = takeCard();
        }
```

```
        return res;
    }
    public static int getColour(int card) {
        return (card / 13);
    }
    public static int getValue(int card) {
        return (card % 13);
    }
    public static boolean isSameHand(int[] h1, int[] h2) {
        int[] h1x = (int[])h1.clone();
        int[] h2x = (int[])h2.clone();
        Arrays.sort(h1x);
        Arrays.sort(h2x);
        if((h1x[0] == h2x[0]) &&
           (h1x[1] == h2x[1]) &&
           (h1x[2] == h2x[2]) &&
           (h1x[3] == h2x[3]) &&
           (h1x[4] == h2x[4])) {
            return true;
        } else {
            return false;
        }
    }
    private static final int PLAYERS = 4;
    public static void main(String[] args) {
        int[] myhand = { 8, 36, 4, 18, 16 };
        int[][] hand;
        long seed = System.currentTimeMillis();
        while(true) {
            hand = new int[PLAYERS][5];
            Poker3 game = new Poker3(seed);
            for(int i = 0; i < PLAYERS; i++) {
                hand[i] = game.takeHand();
            }
            if(Poker3.isSameHand(myhand, hand[0])) {
                break;
            }
            seed--;
        }
        for(int i = 0; i < PLAYERS; i++) {
            for(int j = 0; j < 5; j++) {
                System.out.print(" " + hand[i][j]);
            }
            System.out.println();
        }
    }
}
```

Output:

```
8 36 4 18 16
```

```
50 35 29 27 39
28 42 48 31 1
24 9 5 43 2
```

Udfra sin egen hånd og kendskab til koden i Poker2.java kan man på få sekunder regne ud hvad de andre sidder med af kort.

Ikke smart.

Problemet er at initialisering med tid er godt nok til at give en pæn statistisk fordeling. Men det er ikke godt nok til at forhindre andre i at genskabe sekvensen af tilfældige tal.

der er altså forskel på gode statistiske egenskaber og det at være svær/umulig at forudsige.

Hvis man f.eks. kørte på Linux kunne man lave en:

ps aux > md5.txt

på serveren og så læse md5.txt ind og beregne MD5 hash af den. Den vil være meget sværere at gætte end tiden.

De fleste større biblioteker (inkl. Java og .NET) har såkaldte secure random number generatorer, som er svære at forudsige. De findes normalt sammen med kryptografi funktionerne.

Lidt teori

Langt den mest udbredte algoritme til generering af tilfældige tal er LCG (Linear Congruential Generator):

x(i) = (a*x(i-1) + b) mod c

Den har så en specialisering MLCG (Multiplicative Linear Congruential Generator):

x(i) = (a*x(i-1)) mod c

Og en generalisering MRG (Multiple Recursive Generator):

x(i) = (a1*x(i-1) + a2*x(i-2) + ... + an*x(i-n) + b) mod c

Når man skal generere et tilfældigt heltal 0..N-1 så kan man bruge x(i) direkte hvor N=c.

Nogle gange vil man dog bruge x(i)>>m for at smide nogle af low bits væk. Det er en kendt svaghed ved LCG at low bits er mindre tilfældige end high bits.

Når man skal generere et tilfældigt decimal tal 0.0 ... 1.0 (hvor 0.0 er inklusive og 1.0 eksklusive) så kan man bruge x(i)/N.

Et nøgle begreb er cycle. Efter at have genereret et antal tilfældige

tal vil disse algoritmer altid starte forfra og gentage samme sekvens. Cycle angiver hvormange tilfældige tal de kan generere førend de begynder at gentage sig selv.

Det er absolut nødvendigt at en cycle er stor nok for en given problem stilling. Men en algoritme er ikke nødvendigvis bedre fordi den har større cycle.

Man kan betragte en algoritme som en serie af tal skrevet i en cirkel. Initialiseringen (seed) bestemmer hvor man starter i cirklen. Cycle fortæller hvor mange tal der er i cirklen.

Det er indlysende at for en LCG kan cycle aldrig være større end c.

Det er bevist at hvis c er en potens af 2, så vil cycle være c, hvis:
- a mod 4 er 1
- b er ulige

Nogle af de kendte navne indenfor tilfældige tal algoritmer er: Knuth, Marsaglia og L'Ecuyer.

Inden vi kaster os over algoritmerne skal vi lige se en lille fix omskrivning som tit er nødvendig for LCG for at undgå integer overflow som enten kan give fejl eller negative tal.

rewrite2.c

```
#include <stdio.h>

static long int seed1;

void mysrand1(long int ss)
{
    seed1 = ss;
    return;
}

long int myrand1()
{
    long long int tmp = seed1;
    tmp = (3125LL * tmp) % 67108864LL;
    seed1 = tmp;
    return seed1;
}

static long int seed2;

void mysrand2(long int ss)
{
    seed2 = ss;
    return;
}
```

```
long int myrand2()
{
    seed2 = (3125L * seed2) % 67108864L;
    return seed2;
}

static long int seed3;

void mysrand3(long int ss)
{
    seed3 = ss;
    return;
}

long int myrand3()
{
    long int help1 = 67108864L % 3125L;
    long int help2 = 67108864L / 3125L;
    long int tmp = 3125L * (seed3 % help2) - help1 * (seed3 / help2);
    if(tmp >= 0)
        seed3 = tmp;
    else
        seed3 = tmp + 67108864L;
    return seed3;
}

int main()
{
    int i;
    mysrand1(1234567);
    mysrand2(1234567);
    mysrand3(1234567);
    for(i=0;i<20;i++)
    {
        printf("%d %d %d\n",myrand1(),myrand2(),myrand3());
    }
    return 0;
}
```

Output:

```
32816627 -34292237 32816627
9615183 9615183 9615183
49784667 -17324197 49784667
18737623 18737623 18737623
36142467 -30966397 36142467
991263 991263 991263
10689131 -56419733 10689131
50428967 -16679897 50428967
18909203 -48199661 18909203
35459055 -31649809 35459055
```

```
12812411 -54296453 12812411
41901431 41901431 41901431
12578211 12578211 12578211
48223935 48223935 48223935
40397195 40397195 40397195
9461191 9461191 9461191
38321715 -28787149 38321715
33145999 33145999 33145999
32269723 32269723 32269723
45370647 45370647 45370647
```

Nogle anerkendte algoritmer

Nu vil jeg vise 4 af de rimeligt anerkendte tilfældige tal generatorer i objektorienteret framework i både Java og C#.

RNG.java

```
abstract public class RNG {
    abstract public int getInt();
    abstract public int getMaxInt();
    public double getDouble() {
        return (getInt() / (double)getMaxInt());
    }
}
```

LCG.java

```
public class LCG extends RNG {
    private long a;
    private long b;
    private long c;
    private long seed;
    public LCG(int a, int b, int c, int seed) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.seed = seed;
    }
    public int getInt() {
        seed = (a * seed + b) % c;
        return (int)seed;
    }
    public int getMaxInt() {
        return (int)c;
    }
}
```

MRG.java

```
public class MRG extends RNG {
    private long[] a;
    private long b;
    private long c;
    private long[] seed;
    public MRG(int[] a, int b, int c, int[] seed) {
        this.a = new long[a.length];
        for(int i = 0; i < a.length; i++) this.a[i] = a[i];
        this.b = b;
        this.c = c;
        this.seed = new long[seed.length];
        for(int i = 0; i < seed.length; i++) this.seed[i] = seed[i];
    }
    public int getInt() {
        long tmp = 0;
        for(int i = 0; i < a.length; i++) {
            tmp = tmp + a[i] * seed[i];
        }
        tmp = tmp + b;
        tmp = tmp % c;
        for(int i = 1; i < seed.length; i++) {
            seed[i] = seed[i-1];
        }
        seed[0] = tmp;
        return (int)seed[0];
    }
    public int getMaxInt() {
        return (int)c;
    }
}
```

MLCG.java

```
public class MLCG extends LCG {
    public MLCG(int a, int c, int seed) {
        super(a, 0, c, seed);
    }
}
```

MinStd.java

```
// Minimal Standard algoritme
//
// Oprindeligt foreslået af Lewis, Goodman & Miller 1969, men populariseret
// af Park & Miller CACM Oct 1988.
```

```
//
// Navnet har den fordi man aldrig bør vælge en algoritme som er dårligere
// end denne.
//
// cycle = 2 147 483 647 = 2^31-1

public class MinStd extends MLCG {
    public MinStd(int seed) {
        super(16807, 2147483647, seed);
    }
}
```

LEcuyer.java

```
// L'Ecuyer algoritme
//
// Ecuyer & Blouin & Coutre ACM Simulations 1993.
//
// cycle = ca. 1E46

public class LEcuyer extends MRG {
    public LEcuyer(int[] seed) {
        super(new int[] { 107374182, 0, 0, 0, 104480 }, 0, 2147483647, seed);
    }
}
```

Combiner.java

```
public interface Combiner {
    public int combine(int[] num);
    public int getMaxInt();
}
```

Combined.java

```
public class Combined extends RNG {
    private RNG[] g;
    private Combiner cmbn;
    public Combined(RNG[] g, Combiner cmbn) {
        this.g = g;
        this.cmbn = cmbn;
    }
    public int getInt() {
        int[] num = new int[g.length];
```

```
        for(int i = 0; i < num.length; i++) {
            num[i] = g[i].getInt();
        }
        return cmbn.combine(num);
    }
    public int getMaxInt() {
        return cmbn.getMaxInt();
    }
}
```

Uniform32.java

```

// Uniform32 algoritme
//
// L'Ecuyer CACM Jun 1988.
//
// cycle = ca. 2.31E18

public class Uniform32 extends Combined {
    public Uniform32(int seed1, int seed2) {
        super(new RNG[] { new MLCG(40014, 2147483563, seed1), new MLCG(40692,
2147483399, seed2) },
              new Uniform32Combiner());
    }
}

class Uniform32Combiner implements Combiner {
    public int combine(int[] num) {
        long tmp = num[0] - num[1];
        tmp = (tmp + 2147483562) % 2147483562;
        return (int)tmp;
    }
    public int getMaxInt() {
        return 2147483562;
    }
}
```

CMRG.java

```

// CMRG algoritme
//
// L'Ecuyer Operations Research 1996.
//
// cycle = 2^205 = 1E61

public class CMRG extends Combined {
    public CMRG(int[] seed1, int[] seed2) {
        super(new RNG[] { new MRG(new int[] { 0, 63308, -183326 }, 0,
```

```
2147483647, seed1),
                          new MRG(new int[] { 86098, 0, -539608 }, 0,
2145483479, seed2) },
              new CMRGCombiner());
    }
}

class CMRGCombiner implements Combiner {
    public int combine(int[] num) {
        long tmp = num[0] - num[1];
        tmp = (tmp + 2147483647) % 2147483647;
        return (int)tmp;
    }
    public int getMaxInt() {
        return 2147483647;
    }
}
```

Test.java

```
public class Test {
    public static void main(String[] args) {
        testMinStd();
        testUniform32();
        testLEcuyer();
        testCMRG();
    }
    private static void testMinStd() {
        MinStd rng = new MinStd(1234567);
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getInt());
        }
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getDouble());
        }
    }
    private static void testUniform32() {
        Uniform32 rng = new Uniform32(1234567, 1234567);
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getInt());
        }
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getDouble());
        }
    }
    private static void testLEcuyer() {
        LEcuyer rng = new LEcuyer(new int[] { 1234567, 1234567, 1234567,
1234567, 1234567 });
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getInt());
        }
```

```
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getDouble());
        }
    }
    private static void testCMRG() {
        CMRG rng = new CMRG(new int[] { 1234567, 1234567, 1234567 },
                            new int[] { 1234567, 1234567, 1234567 });
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getInt());
        }
        for(int i = 0; i < 10; i++) {
            System.out.println(rng.getDouble());
        }
    }
}
```

RNG.cs

```
abstract public class RNG
{
    abstract public int GetInt();
    abstract public int MaxInt
    {
        get;
    }
    public double GetDouble()
    {
        return (GetInt() / (double)MaxInt);
    }
}

public class LCG : RNG
{
    private long a;
    private long b;
    private long c;
    private long seed;
    public LCG(int a, int b, int c, int seed)
    {
        this.a = a;
        this.b = b;
        this.c = c;
        this.seed = seed;
    }
    public override int GetInt()
    {
        seed = (a * seed + b) % c;
        return (int)seed;
    }
    public override int MaxInt
    {
```

```
        get
        {
            return (int)c;
        }
    }
}

public class MRG : RNG
{
    private long[] a;
    private long b;
    private long c;
    private long[] seed;
    public MRG(int[] a, int b, int c, int[] seed)
    {
        this.a = new long[a.Length];
        for(int i = 0; i < a.Length; i++) this.a[i] = a[i];
        this.b = b;
        this.c = c;
        this.seed = new long[seed.Length];
        for(int i = 0; i < seed.Length; i++) this.seed[i] = seed[i];
    }
    public override int GetInt()
    {
        long tmp = 0;
        for(int i = 0; i < a.Length; i++)
        {
            tmp = tmp + a[i] * seed[i];
        }
        tmp = tmp + b;
        tmp = tmp % c;
        for(int i = 1; i < seed.Length; i++)
        {
            seed[i] = seed[i-1];
        }
        seed[0] = tmp;
        return (int)seed[0];
    }
    public override int MaxInt
    {
        get
        {
            return (int)c;
        }
    }
}

public class MLCG : LCG
{
    public MLCG(int a, int c, int seed) : base(a, 0, c, seed)
    {
    }
}

// Minimal Standard algoritme
```

```
//
// Oprindeligt foreslået af Lewis, Goodman & Miller 1969, men populariseret
// af Park & Miller CACM Oct 1988.
//
// Navnet har den fordi man aldrig bør vælge en algoritme som er dårligere
// end denne.
//
// cycle = 2 147 483 647 = 2^31-1

public class MinStd : MLCG
{
    public MinStd(int seed) : base(16807, 2147483647, seed)
    {
    }
}

// L'Ecuyer algoritme
//
// Ecuyer & Blouin & Coutre ACM Simulations 1993.
//
// cycle = ca. 1E46

public class LEcuyer : MRG
{
    public LEcuyer(int[] seed) : base(new int[] { 107374182, 0, 0, 0, 104480
}, 0, 2147483647, seed)
    {
    }
}

public interface Combiner
{
    int Combine(int[] num);
    int MaxInt
    {
        get;
    }
}

public class Combined : RNG
{
    private RNG[] g;
    private Combiner cmbn;
    public Combined(RNG[] g, Combiner cmbn)
    {
        this.g = g;
        this.cmbn = cmbn;
    }
    public override int GetInt()
    {
        int[] num = new int[g.Length];
        for(int i = 0; i < num.Length; i++)
        {
            num[i] = g[i].GetInt();
        }
```

```
        return cmbn.Combine(num);
    }
    public override int MaxInt
    {
        get
        {
            return cmbn.MaxInt;
        }
    }
}

// Uniform32 algoritme
//
// L'Ecuyer CACM Jun 1988.
//
// cycle = ca. 2.31E18

internal class Uniform32Combiner : Combiner
{
    public int Combine(int[] num)
    {
        long tmp = num[0] - num[1];
        tmp = (tmp + 2147483562) % 2147483562;
        return (int)tmp;
    }
    public int MaxInt
    {
        get
        {
            return 2147483562;
        }
    }
}

public class Uniform32 : Combined
{
    public Uniform32(int seed1, int seed2) : base(new RNG[] { new MLCG(40014,
2147483563, seed1),
                                                              new MLCG(40692,
2147483399, seed2) },
                                                  new Uniform32Combiner())
    {
    }
}

// CMRG algoritme
//
// L'Ecuyer Operations Research 1996.
//
// cycle = 2^205 = 1E61

internal class CMRGCombiner : Combiner
{
    public int Combine(int[] num)
    {
```

```
        long tmp = num[0] - num[1];
        tmp = (tmp + 2147483647) % 2147483647;
        return (int)tmp;
    }
    public int MaxInt
    {
        get
        {
            return 2147483647;
        }
    }
}

public class CMRG : Combined {
    public CMRG(int[] seed1, int[] seed2) : base(new RNG[] { new MRG(new int[]
{ 0, 63308, -183326 }, 0, 2147483647, seed1),
                                                             new MRG(new int[]
{ 86098, 0, -539608 }, 0, 2145483479, seed2) },
                                                    new CMRGCombiner())
    {
    }
}
```

Test.cs

```
using System;

public class Test
{
    public static void Main(string[] args)
    {
        TestMinStd();
        TestUniform32();
        TestLEcuyer();
        TestCMRG();
    }
    private static void TestMinStd()
    {
        MinStd rng = new MinStd(1234567);
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetInt());
        }
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetDouble());
        }
    }
    private static void TestUniform32()
    {
        Uniform32 rng = new Uniform32(1234567, 1234567);
```

```
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetInt());
        }
        for(int i = 0; i < 10; i++
        ) {
            Console.WriteLine(rng.GetDouble());
        }
    }
    private static void TestLEcuyer()
    {
        LEcuyer rng = new LEcuyer(new int[] { 1234567, 1234567, 1234567,
1234567, 1234567 });
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetInt());
        }
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetDouble());
        }
    }
    private static void TestCMRG()
    {
        CMRG rng = new CMRG(new int[] { 1234567, 1234567, 1234567 },
                            new int[] { 1234567, 1234567, 1234567 });
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetInt());
        }
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine(rng.GetDouble());
        }
    }
}
```

Ikke uniformt fordelte

Alt hidtil har drejet sig om uniformt fordelte tal, men der er
jo også sommetider brug for andre fordelinger.

Imidlertid kan de som oftest laves udfra uniform fordelte tal.

Hvis man f.eks. skal bruge et normal fordelt tal, så
kan man bruge den såkaldte Box Mueller transformation.

u1 og u2 er uniform fordelte 0.0 .. 1.0

h1 = sqrt(-2*log(u1));
h2 = 2*PI*u2

n1 = e + s*h1*sin(h2)
n2 = e + s*h1*cos(h2)

så vil n1 og n2 være normal fordelte N(e,s)

Normal.cs

```
using System;

public class Normal
{
    public const int N = 100000;
    public static void Test(double e, double s)
    {
        Random rng = new Random();
        int[] one1 = new int[4];
        int[] one2 = new int[4];
        for(int i=0; i<N; i++)
        {
            double u1 = rng.NextDouble();
            double u2 = rng.NextDouble();
            double h1 = Math.Sqrt(-2*Math.Log(u1));
            double h2 = 2*Math.PI*u2;
            double n1 = e + s*h1*Math.Sin(h2);
            double n2 = e + s*h1*Math.Cos(h2);
            if(n1<e-1.96*s)
            {
                one1[0]++;
            }
            else if(n1<e)
            {
                one1[1]++;
            }
            else if(n1<e+1.96*s)
            {
                one1[2]++;
            }
            else
            {
                one1[3]++;
            }
            if(n2<e-1.96*s)
            {
                one2[0]++;
            }
            else if(n2<e)
            {
                one2[1]++;
            }
            else if(n2<e+1.96*s)
            {
                one2[2]++;
            }
```

```
            else
            {
                one2[3]++;
            }
        }
        for(int i=0; i<4; i++)
        {
            Console.WriteLine(one1[i] + " " + one2[i]);
        }
    }
    public static void Main(string[] args)
    {
        Test(0, 1);
        Test(7, 3);
    }
}
```

Output:

```
2544 2471
47544 47787
47360 47319
2552 2423
2474 2532
47367 47736
47694 47248
2465 2484
```

Hvilket jo ser meget fornuftigt ud (det skal være 2.5% 47.5% 47.5% 2.5%).

Videre

Se artiklen:
* http://www.eksperten.dk/guide/951 "Endnu mere om tilfældige tal" som forklarer lidt mere teori og har nogle
  eksempler i PHP og ASP

Kommentar af simonvalter d. 25. Apr 2005 | 1

meget interessant

Kommentar af mikze d. 28. Apr 2005 | 2

interessant

Kommentar af gaflen89 d. 03. May 2005 | 3

Selvom jeg ikke har forstand på java, så skal den da ikke have den værste karakter, da jeg kan se den beskriver det titlen.

Kommentar af hyberpreprocessor d. 25. Apr 2005 | 4

Kommentar af tobiasahlmo d. 10. Oct 2007 | 5

super