

# Véletlenszám-generátorok

## 1. rész

– egy MATLAB<sup>®</sup> alapú megközelítés –

Róth Ágoston, Vas Orsolya

Matematika és Informatika Intézet, Babeş-Bolyai Tudományegyetem, Kolozsvár, Románia

(agoston.roth@gmail.com, vas.orsolya@yahoo.com)

1. labor / 2018. október 1–4.



- Tekintsük a természetes számok halmazán az  $m \geq 1$  osztót, az  $a \in \{0, 1, \dots, m-1\}$  szorzótényezőt,  $a, c \in \{0, 1, \dots, m-1\}$  növekedményt, valamint az  $X_1 \in \{0, 1, \dots, m-1\}$  kezdőértékkel ellátott

$$X_{i+1} = (aX_i + c) \pmod{m}, \quad i \geq 2 \quad (1)$$

rekurzív multiplikatív lineáris kongruenciát!

- Vegyük észre, hogy az (1)-es rekurzió az  $m, a, c$  és  $X_1$  paramétereknek bármely beállítására nem generál „véletlenszerű” számokat! Rossz paraméterezés esetén a generált sorozat túlságosan hamar periodikussá válhat, nyomon követhető mintát eredményezve. Például az  $m = 10$ ,  $X_1 = 2$  és az  $a = c = 3$  feltételek esetén a

$$2, 9, 0, 3, 2, 9, 0, 3, 2, 9, 0, 3, 2, \dots$$

periodikus számsorozatot kapjuk eredményül.

- Nyilván az (1)-es rekurzió bármely változata periodikus számsorozatot generál, célunk viszont az, hogy a periodikusan ismétlődő minta hosszát minél nagyobbra nyújtsuk.
- Az  $m$  számmal osztva összesen  $m$  darab különböző maradékot kaphatunk, de – ahogy a fenti példa is mutatja – nem megfelelő beállítás esetén a létrehozott sorozat nem tartalmazza az összes lehetséges maradékot.



- Ha  $c \neq 0$ , akkor az alábbi tétel [Hull, Dobell, 1962] segítségével elérhetjük, hogy az (1)-es rekurzió által felépített egész számsorozat periódusa maximális – azaz  $m$ -mel megegyező – legyen.

## Tétel (Maximális periódus biztosítása, ha $c > 0$ )

Az (1)-es rekurzió által generált egész számsorozat periódusa akkor és csak akkor egyezik meg az  $m$  osztó értékével, ha teljesül az alábbi három feltétel:

- 1  $a c \neq 0$  növekmény és az  $m$  osztó relatív prím;
- 2  $a \equiv 1 \pmod{p}$ , ha  $p$  az  $m$  szám prímosztója;
- 3  $a \equiv 1 \pmod{4}$ , ha az  $m$  osztó négynek többszöröse.

- A fenti tétel értelmében, ha az  $m$  osztó 2-nek hatványa (ami természetes egy bináris számrendszerre épülő architektúra esetén), akkor a maximális periódus biztosításához elégséges, ha a  $c$  növekmény páratlan és  $a \equiv 1 \pmod{4}$ .



## Értelmezés (Primitív gyök modulo $m$ )

Az  $m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_r^{\alpha_r} > 1$  prímtényezős felbontású természetes szám esetén az olyan  $g$  számot nevezzük „ $g$  primitív gyök modulo  $m$ ”-nek, amelyre a

$$g, g^2, \dots, g^{\varphi(m)}$$

hatványok különböző maradékot adnak  $m$ -mel osztva, azaz a  $g$  rendje modulo  $m$  pontosan  $\varphi(m)$ , ahol

$$\varphi(m) = m \prod_{i=1}^r \left(1 - \frac{1}{p_i}\right)$$

az Euler-féle  $\varphi$ -függvényt jelöli.



- A  $c = 0$  esetben a [Carmichael, 1910] cikkben közölt és alább kijelentett tételt alkalmazhatjuk a maximális periódus biztosítására.

## Tétel (Maximális periódus biztosítása, ha $c = 0$ )

Ha az (1)-es képletbeli  $c$  növekmény nulla, az  $X_1$  és  $m$  számok relatív prímek, valamint  $a$  primitív gyök modulo  $m$ , akkor a rekurzió által generált egész számsorozat periódusa maximális és ez a maximum megegyezik a

$$\lambda(m) = \begin{cases} 1, & m = 2, \\ 2, & m = 2^2 = 4, \\ 2^{e-2}, & m = 2^e, e \geq 3, \\ p^{e-1}(p-1), & m = p^e, p > 2, \\ \text{lkk} \{ \lambda(p_1^{\alpha_1}), \lambda(p_2^{\alpha_2}), \dots, \lambda(p_r^{\alpha_r}) \}, & m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_r^{\alpha_r} \end{cases}$$

számmal.

- Vegyük észre, hogy a fenti tétel értelmében, ha  $m$  prím, akkor  $m - 1$  periódusú számsorozatot generálhatunk az (1)-es rekurzióval!



## Példák

A továbbiakban az (1)-es rekurzió néhány olyan paraméterezését ismertetjük, amelyek által szült egyenletes eloszlású számgenerátorok „véletlenszerűséget” ellenőrző, szigorú statisztikai teszteken is átmentek:

- **URNG1** generátor:  $m = 2^{31} - 1$  (Mersenne-féle prím),  $a = 7^5$ ,  $c = 0$ , míg  $x_1 \geq 1$  tetszőlegesen rögzített természetes szám;
- **URNG2** generátor:  $m = 2^{31} - 1$ ,  $a = 2^{16} + 3$ ,  $c = 0$ , míg  $x_1 = 2k + 1$  alakú tetszőlegesen rögzített természetes szám.



## 1. Kódrészlet. Multiplikatív lineáris kongruencia módszere

```

1 % _____
2 % Description
3 % _____
4 % The function implements the linear congruential1 generator  $X_{i+1} = (aX_i + c) \bmod m, i \geq 2$ .
5 %
6 % _____
7 % Input
8 % _____
9 % m                – the modulus2  $m > 0$ 
10 % a                – the multiplier3  $a \in \{0, 1, \dots, m-1\}$ 
11 % c                – the increment4  $c \in \{0, 1, \dots, m-1\}$ 
12 % initial_value    – an integer that represents the first element of the
13 %                  generated sequence ( $0 < \text{initial\_value} < m$ )
14 % n                – the size of the output sequence
15 %
16 % _____
17 % Output
18 % _____
19 % X =  $[X_i]_{i=1}^n$     – an array of uniformly distributed integer random numbers5
20 % new_initial_value – an integer that can be used as an initial value in case
21 %                  of consecutive random sequence generations
22 function [X, new_initial_value] = LinearCongruentialGenerator(m, a, c, initial_value, n)
23
24 X = zeros(1, n);
25 X(1) = initial_value;
26
27 for i = 2:n
28     X(i) = mod(a * X(i-1) + c, m);
29 end
30

```



```
31 new_initial_value = mod(a * X(n) + c, m);
```

---

<sup>1</sup>congruence *kongruencia*; (combined) multiplicative linear  $\sim$  (összetett) multiplikatív lineáris kongruencia

<sup>2</sup>modulus *osztó*

<sup>3</sup>multiplier *szorzó*

<sup>4</sup>increment *növekedmény*

<sup>5</sup>number *szám*; (pseudo)random  $\sim$  (ál) véletlen szám; random  $\sim$  generator véletlenszám-generátor





## 2. Kódrészlet. Egyenletes eloszlású egész véletlenszám-generátor

```

1 % _____
2 % Description
3 % _____
4 % By means of parameters  $m = 2^{31} - 1$ ,  $a = 7^5$  and  $c = 0$ , the function implements a specialized
5 % version of the linear congruential generator  $X_{i+1} = (aX_i + c) \bmod m$ ,  $i \geq 2$ .
6 % _____
7 % Input
8 % _____
9 % initial_value      – an integer that must be  $\geq 1$  and  $< 2^{31} - 1$  and which represents
10 %                   the first element of the output sequence
11 % n                  – the size of the output sequence
12 % _____
13 % Output
14 % _____
15 %  $\mathbf{X} = [X_i]_{i=1}^n$       – a sequence of uniformly distributed integer random numbers
16 % new_initial_value – an integer that can be used as an initial value in
17 %                   case of consecutive random sequence generations
18 function [X, new_initial_value] = URNG1(initial_value , n)
19
20 m = 2^31-1;
21 a = 7^5;
22 c = 0;
23 [X, new_initial_value] = LinearCongruentialGenerator(m, a, c, initial_value , n);

```

## 1. feladat

Az URNG1 egyenletes eloszlású véletlenszám-generátor 2. kódrészletbeli implementálásához hasonlóan írjatok egy függvényt az URNG2 generátorra is! A **hist** parancsot használva készítsetek abszolút gyakoriságdiagramot (másképpen hisztogramot) az új függvény által visszatérített sorozatokról is!





- Egy  $[\alpha, \beta] \subset \mathbb{R}$  intervallumba eső egyenletes eloszlású véletlen valós számokból álló sorozatot egy alkalmas lineáris transzformációval állíthatunk elő, ahogy azt a 3. kódrészlet is mutatja.

### 3. Kódrészlet. Adott intervallumon egyenletes eloszlású valós véletlen számok generálása

```

1 % _____
2 % Description
3 % _____
4 % The function generates uniformly distributed real random numbers in the range  $[\alpha, \beta]$ .
5 % where  $\alpha < \beta$ .
6 %
7 % _____
8 % Input
9 % _____
10 % initial_value      – an integer which is required by the uniform integer
11 %                   random number generator ( $0 < \text{initial\_value} < m$ )
12 % generator_type     – specifies the type of the used uniform integer random
13 %                   number generator, that can be set as 'URNG1'/'URNG2' or 1/2
14 % [alpha, beta]     – all elements of the output sequence Y will fall in this range
15 % n                 – the size of the output sequence
16 %
17 % _____
18 % Output
19 % _____
20 %  $\mathbf{Y} = [Y_i]_{i=1}^n$       –  $\mathbf{Y} \sim \mathcal{U}([\alpha, \beta])$ 
21 % new_initial_value – an integer that can be used as an initial value in
22 %                   case of consecutive random sequence generations
23 %
24 function [Y, new_initial_value] = URealRNG(
25     initial_value, generator_type, alpha, beta, n)
26
27 switch (generator_type)
28     case {'URNG1', 1}

```



```
29     m = 2^31 - 1;  
30     [Y, new_initial_value] = URNG1(initial_value , n);  
31     Y = Y ./ (m - 1);  
32  
33     % handle another integer uniform random number generator  
34     ...  
35 end  
36  
37 Y = alpha + Y .* (beta - alpha);
```

## 2. feladat

Fejezzétek be és teszteljétek is a 3. kódrészletbeli **URealRNG** függvényt!



- Az (1)-es rekurzió különböző beállításából származó multiplikatív lineáris kongruenciákat ötvözhetjük is egyenletes eloszlású számgenerátorok kialakítására.
- Az idevágó elméleti háttérrel mellőzzük, viszont megemlítyük, hogy Pierre L'Ecuyer az

$$\begin{cases} X_{1,i} &= (40692 \cdot X_{1,i-1}) \pmod{2147483399}, \\ X_{2,i} &= (40014 \cdot X_{2,i-1}) \pmod{2147483563}, \\ X_i &= (X_{1,i} + X_{2,i} - 2) \pmod{2147483562}, \\ U_i &= \frac{X_i + 1}{2147483563} \end{cases} \quad (2)$$

összetett multiplikatív lineáris kongruenciákra épülő módszert ajánlotta a  $(0, 1)$  intervallumon egyenletes eloszlású valós számok generálására [L'Ecuyer, 1986], ahol  $i \geq 2$ , illetve  $X_{1,1}$  és  $X_{2,1}$  alkalmasan megválasztott kezdőértékek.

- A (2)-es képletekbeli lineáris kongruenciák periódusa rendre  $p_1 = 2147483398$ , illetve  $p_2 = 2147483562$ . Az összetett számgenerátor  $p$  periódusát a  $p_1$  és  $p_2$  számok legkisebb közös többszörőseként kaphatjuk meg, azaz

$$p = \frac{p_1 \cdot p_2}{2} = 2,305842648436451838 \cdot 10^{18},$$

ami tökéletesen megfelel a céljainknak.



## 4. Kódrészlet. L'Ecuyer egyenletes eloszlású számgenerátora

```

1 % _____
2 % Description
3 % _____
4 % The function generates continuous uniform pseudorandom numbers on the interval (0,1).
5 % Its implementation is based on the article [L'Ecuyer, 1986].
6 %
7 % _____
8 % Usage and syntax
9 % _____
10 % scalar      = ULEcuyerRNG
11 % row_matrix  = ULEcuyerRNG(column_count)
12 % matrix      = ULEcuyerRNG(row_count, column_count)
13 %
14 function result = ULEcuyerRNG(varargin)
15
16 % checking the possible input parameters
17 optional_input_argument_count = size(varargin,2);
18
19 if (optional_input_argument_count == 0)
20     row_count = 1;
21     column_count = 1;
22 else
23     if (optional_input_argument_count == 1)
24         row_count = 1;
25
26         if (isscalar(varargin{1}))
27             column_count = round(varargin{1});
28         else
29             error('Wrong_column_number!');
30         end
31     else
32         if (optional_input_argument_count == 2)

```



```

33         if ( isscalar(varargin{1}))
34             row_count = round(varargin{1});
35         else
36             error('Wrong_row_number! ');
37         end
38
39         if ( isscalar(varargin{2}))
40             column_count = round(varargin{2});
41         else
42             error('Wrong_column_number! ');
43         end
44     else
45         error('Too_many_input_arguments! ');
46     end
47 end
48
49 % Implementation of the combined multiplicative linear congruential generator [L'Ecuyer, 1986]
50 % for  $\mathcal{U}((0, 1))$  random sampling:
51 persistent seed1 seed2
52
53 if (isempty(seed1))
54     seed1 = 55555;
55 end
56
57 if (isempty(seed2))
58     seed2 = 99999;
59 end
60
61 persistent factor
62
63 if (isempty(factor))
64     factor = 1.0/2147483563.0;
65 end
66
67 result = zeros(row_count, column_count);

```



```
69
70 for i = 1:row_count
71     for j = 1:column_count
72
73         k = seed1 / 53668;
74         seed1 = 40014 * mod(seed1, 53668) - k * 12211;
75
76         if (seed1 < 0)
77             seed1 = seed1 + 2147483563;
78         end
79
80         k = seed2 / 52774;
81
82         seed2 = 40692 * mod(seed2, 52774) - k * 3791;
83
84         if (seed2 < 0)
85             seed2 = seed2 + 2147483399;
86         end
87
88         z = (seed1 - 2147483563) + seed2;
89
90         if (z < 1)
91             z = z + 2147483562;
92         end
93
94         result(i, j) = z * factor;
95     end
96 end
```





## Megjegyzés

- Vegyük észre a **persistent** kulcsszó használatát! A **persistent** típusú változók a globálisakhoz hasonlóan viselkednek (azaz értékeik permanens memóriaterületen tárolódnak el), azzal a különbséggel, hogy csak a változókat deklaráló függvény számára elérhetőek. Ezért értéküket nem változtathatjuk meg más függvényekben, vagy a `MATLAB®` parancssorából. Ezzel az apró kis trükkel megszabadulhatunk az **ULEcuyerRNG** egyenletes eloszlású számgenerátor által várt kezdőértékek folytonos megadásától, hiszen sorozatos függvényhívások esetén a már módosult kezdőértékek lépnek életbe.
- A **varargin**, **isscalar** kulcsszavak alkalmazásával az **ULEcuyerRNG** függvényt egyaránt használhatjuk  $(0, 1)$  intervallumon egyenletes eloszlású skalárok, sormátrixok, illetve tetszőleges méretű mátrixok generálására.



- Ha mégsem lennénk megelégedve a 4. kódrészletben ismertetett L'Ecuyer-féle egyenletes eloszlású számgenerátorral, akkor helyette – a [Matsumoto, Nishimura, 1998] cikkre alapozva – a Mersenne-twister számgenerátort ajánlhatjuk, amely szintén 32-bites architektúrát feltételez, és számos szigorú, véletlenszerűséget ellenőrző statisztikai (pl. Diehard-típusú [Marsaglia, 1985]) tesztet kiállt. A módszer a nevét a

$$2^{19937} - 1 = 4,3154247973881626480552355163379 \cdot 10^{6001}$$

kolosszális nagyságú periódusát meghatározó Mersenne-féle prímszámról kapta.

- Mejegyezzük, hogy a lent kódolt módszer a TestU01-típusú [L'Ecuyer, Simard, 2007] véletlenszerűséget tesztelő statisztikák zöme alapján is kiváló egyenletes eloszlású számgenerátornak bizonyult.
- Ha az 5. kódrészletbeli implementáció nem is teljesen optimális, tanítópédának mindenképpen alkalmas, mert hűen követi az [▶ algoritmus pszeudokódját](#). Az algoritmus helyességének és komplexitásának tanulmányozása nem tartozik a jelen kézirat célkitűzései közé.
- Az 5. kódrészletbeli **UMersenneTwisterRNG** függényt egyaránt használhatjuk (0, 1) intervallumon egyenletes eloszlású skalárok, sormátrixok, illetve tetszőleges méretű mátrixok generálására.



## 5. Kódrészlet. 32-bites Mersenne-twister egyenletes eloszlású számgenerátor

```

1 % _____
2 % Description
3 % _____
4 % The function generates continuous uniform pseudorandom numbers on the interval (0,1).
5 % Its implementation is based on the article [Matsumoto, Nishimura, 1998].
6 %
7 % _____
8 % Usage and syntax
9 % _____
10 % scalar = UMersenneTwisterRNG
11 % row_matrix = UMersenneTwisterRNG(column_count)
12 % matrix = UMersenneTwisterRNG(row_count, column_count)
13 %
14 function result = UMersenneTwisterRNG(varargin)
15
16     persistent MT
17     persistent index
18
19     % initialize the generator from a seed
20     function initialize_generator(seed)
21
22         index = 0;
23
24         MT = zeros(1, 624);
25         MT(1) = uint64(seed);
26
27         for k = 1:623
28             MT(k+1) = bitand(...
29                 uint64(1812433253 * bitxor(MT(k), bitshift(MT(k), -30)) + k), ...
30                 4294967295);
31         end
32

```



```

33     end
34
35 % generate an array of 624 untempered numbers
36 function generate_numbers()
37
38     for k = 0:623
39         y = bitand(MT(k + 1), 2147483648) + ...
40             bitand(2147483647, MT(mod(k + 1, 624) + 1));
41
42         MT(k + 1) = bitxor(MT(mod(k + 397, 624) + 1), bitshift(uint32(y), -1));
43
44         if (mod(y, 2) ~= 0)
45             MT(k + 1) = bitxor(MT(k + 1), 2567483615);
46         end
47     end
48
49 end
50
51 % extract a tempered pseudorandom number based on the index-th value,
52 % calling generate_numbers() every 624 numbers
53 function y = extract_number()
54
55     if (index == 0)
56         generate_numbers();
57     end
58
59     y = MT(index + 1);
60     y = bitxor(y, bitshift(y, -11));
61     y = bitxor(y, bitand(bitshift(y, 7), 2636928640));
62     y = bitxor(y, bitand(bitshift(y, 15), 4022730752));
63     y = bitxor(y, bitshift(y, -18));
64
65     index = mod(index + 1, 624);
66
67 end
68

```



```

69 % checking the possible input parameters
70 optional_input_argument_count = size(varargin, 2);
71
72 if (optional_input_argument_count == 0)
73     row_count = 1;
74     column_count = 1;
75 else
76     if (optional_input_argument_count == 1)
77         row_count = 1;
78         if (isscalar(varargin{1}))
79             column_count = round(varargin{1});
80         else
81             error('Wrong_column_number!');
82         end
83     else
84         if (optional_input_argument_count == 2)
85             if (isscalar(varargin{1}))
86                 row_count = round(varargin{1});
87             else
88                 error('Wrong_row_number!');
89             end
90
91             if (isscalar(varargin{2}))
92                 column_count = round(varargin{2});
93             else
94                 error('Wrong_column_number!');
95             end
96         else
97             error('Too_many_input_arguments!');
98         end
99     end
100 end
101
102 % we use a default initial value that is equal to the prime number 6199
103 if (isempty(MT) && isempty(index))
104     initialize_generator(6199);

```

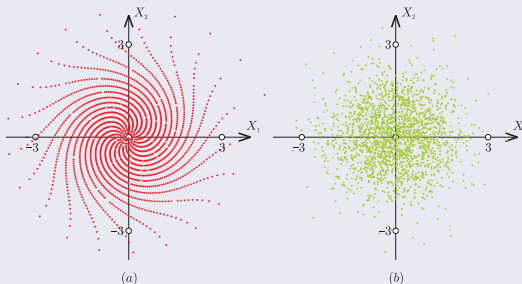


```
105     end
106
107     % allocate memory for the output sequence of uniform
108     result = zeros(row_count, column_count);
109
110     % generate random uniform integers and normalize them
111     for i = 1:row_count
112         for j = 1:column_count
113             result(i, j) = extract_number() / 4294967295;
114         end
115     end
116
117 end
```



## Megjegyzés

- Nemlineáris transzformációkra épülő és nemegyenletes eloszlású valószínűségi változókat mintavételező algoritmusok erősen függnak az általuk alkalmazott egyenletes eloszlású számgenerátorról is. Olyan egyenletes eloszlású valószínűségi változókat kell generálnunk, amelyek transzformálása során kapott valószínűségi változók is kellőképpen lefedik az eseményteret úgy, hogy semmilyen szabályosságot, ismétlődést, vagy mintát nem észlelünk az így nyert mintavételezésben.



**2. ábra.** Mindkét ábrán a Box–Muller-transzformáció alkalmazása során nyert 2000 darab, komponensekben független és standard normális eloszlású valószínűségi vektort láthatunk. Az (a) esetben a  $v_i = 97v_{i-1} \pmod{2^{17}}$ ,  $v_1 = 1$  ( $i \geq 2$ ) multiplikatív lineáris kongruencia által szült természetes számok normalizált értékeire, a (b) esetben pedig a 4. kódrészletben implementált L'Ecuyer-féle összetett multiplikatív lineáris kongruenciákra épülő számgenerátor által eredményezett  $(0, 1)$  intervallumbeli, egyenletes eloszlású, véletlen számokra alkalmaztuk a transzformációt. Látható, hogy az (a) esethez tartozó egyenletes eloszlású számgenerátor nem alkalmas véletlenszerű folyamatok modellezésére.

## Elfogadási feltételek

- Az alábbi feladatokat egyrészt elméleti úton, másrészt valamilyen programozási nyelvben, vagy matematikai szoftvercsomagban írt – a teljes eseményteret meghatározó elemek nagyszámú mintavételezésén alapuló – szimulációval is kötelező megoldani!
- Nyilván a szimuláción alapuló megoldások csak közelíteni fogják az elméleti úton meghatározott valószínűségi értékeket.
- Ahol csak lehetséges, ábrázoljátok grafikusan a vizsgált eseményhez tartozó teljes eseményteret, illetve a kedvező elemi események halmazát! Ugyanakkor animáljátok is a szimuláció egyes lépéseihez tartozó jelenségeket, valamint a közelített számértékeket! Ha egy feladat több paramétertől is függ, akkor ezeket soroljátok fel a szimulációt meghívó függvények paraméterlistájában!
- A feladatokat személyesen kell bemutatni a megfelelő gyakorlati órán! Ha az ellenőrzés során kiderül, hogy egy adott feladat nincs megoldva, vagy annak megoldása hibás, vagy esetlegesen másolt, akkor az adott feladat továbbra is házi feladat marad, továbbá egy újabb feladatot is meg kell oldani a mulasztás törlesztéséért.





### 3. feladat

Egy  $d > 0$  oldalhosszúságú, órajárással megegyező irányban irányított négyzet belsejében taláломra felvesszük az  $M$  pontot. Ha az  $M$  pontot összekötjük az alap, majd az azt követő oldalél végpontjaival, akkor két  $M$ -csúcspontú szöget alkothatunk. Milyen valószínűséggel lesz mindkét szög hegyesszög? (Azt is kérdezhetnénk, hogy milyen valószínűséggel nézünk az  $M$  pontból mindkét rögzített oldalra hegyesszög alatt.)

### 4. feladat

Egy 30 alkatrészt tartalmazó mintából 6 javítható, 4 selejtes, a többi jó. Ha 10 darabot taláломra kiválasztunk, milyen valószínűséggel kaptunk 7 jó, 2 javítható és 1 selejtes alkatrészt?

### 5. feladat

Egy kétszemélyes játék során a játékosok egymást váltogatva, szabályos dobókockával gurítanak. Az a játékos nyer, aki először páratlan prímszámot dob. Határozzátok meg mindkét játékos nyerési valószínűségét! Várhatóan hány lépésig tart egy-egy játszma? Mi történik akkor, ha a játszmát kezdő félnek a győzelméhez kétszer kell páratlan prímszámot dobnia, míg az őt követő játékosnak továbbra is elég egyetlen ilyen primérték a nyeréshez? (Megjegyzés: az 1-et nem tekintjük prímszámnak!)





R.D. Carmichael, **1910**.

*Note on a new number theory function*,  
Bulletin of the American Mathematical Society, **16**(5):232–238.



T.E. Hull, A.R. Dobell, **1962**.

*Random number generators*,  
SIAM Review, **4**(3):230–254.



P. L'Ecuyer, **1986**.

*Efficient and portable 32-bit random variate generators*.  
In Proceedings of the 1986 Winter Simulation Conference (J. Wilson, J. Henriksen, S. Roberts, eds.), 275–277.



P. L'Ecuyer, R. Simard, **2007**.

*TestU01: A C library for empirical testing of random number generators*,  
ACM Transactions on Mathematical Software, **33**(4):Article 22.



G. Marsaglia, **1985**.

*A current view of random numbers*.  
In Computer Science and Statistics, Proceedings of the Sixteenth Symposium on  
The Interface (L. Billard, ed.), North-Holland, Amsterdam, 3–10.





M. Matsumoto and T. Nishimura, **1998**.

*Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,*

ACM Transactions on Modeling and Computer Simulation, **8**(1):3–30.

