Diszkrét logaritmus

Diszkrét modellek alkalmazásai

1. Moduláris hatványozás

Az előző jegyzetben (Kongruenciák) szó volt arról, hogy a kongruenciákat lehet összeadni, kivonni, szorozni (osztani nem mindig), és hatványozni. Ebben a jegyzetben az utóbbi műveletről lesz szó. Emeljük ki még egyszer:

1.1. Tétel. *Ha* $a \equiv b \pmod{m}$*, akkor* $\forall n \in \mathbb{N} : a^n \equiv b^n \pmod{m}$.

Példa. $5 \equiv 12 \pmod{7} \implies 5^3 \equiv 12^3 \pmod{7}$, azaz mindegy, hogy a kongruens alapok (5 és 12) közül melyiket választjuk, amikor hatványozunk.

(De vigyázat: ez csak a hatványalapra igaz, a kitevőre nem! Tehát például $5^{10} \not\equiv 5^3 \pmod{7}$. Erről a következő jegyzetben még lesz szó.)

Ebben a jegyzetben elsősorban azzal a speciális esettel foglalkozunk, amikor a modulus prímszám. Ilyenkor bizonyos műveletek leegyszerűsödnek, például az előző jegyzetben láttuk, hogy ekkor minden nemnulla maradék invertálható:

1.2. Tétel. *Ha* p *prímszám, akkor minden* $a \not\equiv 0 \pmod{p}$ *szám invertálható modulo* p.

De nemcsak az inverz, hanem a szorzás is jól viselkedik prímszám modulusokra:

1.3. Tétel. *Ha* p *prímszám,* $a \not\equiv 0 \ (p)$ *és* $b \not\equiv 0 \ (p)$*, akkor* $a \cdot b \not\equiv 0 \ (p)$.

Vagyis nemnulla maradékok szorzata sosem ad 0 maradékot. Ez megfelel annak, hogy ha két nemnulla valós számot összeszorzunk, akkor a szorzat sem lesz nulla. Ez nagyon természetesnek tűnik, és nagyon hasznos is lesz, viszont csak prímszám modulus esetén igaz! Például ha a modulus $m = 10$, akkor $4 \cdot 5 \equiv 20 \equiv 0 \pmod{10}$. (Az ilyeneket fogjuk később *nullosztók*nak nevezni.)

És mivel a hatványozás nem más, mint ismételt szorzás, ezért nemnulla maradék hatványai sem adhatnak 0 maradékot:

1.4. Tétel. *Ha* p *prímszám és* $a \not\equiv 0 \ (p)$*, akkor* $\forall n \in \mathbb{N} : a^n \not\equiv 0 \ (p)$.

(De itt is fontos, hogy a modulus prímszám legyen, például $6^2 \equiv 0 \pmod{12}$.)

Sage-ben többféleképpen lehet kiszámítani az $a^n \bmod m$ moduláris hatványt. A naiv módszer, `a^n % m`, nagy kitevőkre nem hatékony, mert először kiszámítja az a^n hatványt, amely nagyon nagy is lehet, és aztán veszi a maradékát, azaz eldobja a nehezen kiszámított szám nagy részét. Ennél sokkal hatékonyabb a `power_mod()`:

```
sage: 5^99999999 % 7              # LASSÚ!
6
sage: power_mod(5, 99999999, 7)   # gyors
6
```

2. Gyorshatványozás

Most arra keressük a választ, hogy hogyan lehet hatékonyan kiszámítani – a Sage függvényei nélkül – az $a^n \bmod m$ hatványt. Erre való a **gyorshatványozás** algoritmusa, amely a hatványok hatékony formába való átrendezésén alapul. (Ezt használja a Sage `power_mod()` függvénye is.)

Például hogyan számítjuk ki a^8-t? A naiv módszerrel $a^8 = a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$, ami hét szorzás. Lehet ennél kevesebb szorzást használni? Igen: $a^8 = \left(a^4\right)^2 = \left(\left(a^2\right)^2\right)^2$, ami mindössze három szorzás (három négyzetreemelés). Ezt általánosíthatjuk a következőképpen: ha a kitevő páros, akkor az első lépés ugyanaz: $a^{2n} = (a^n)^2$, és ezt addig folytatjuk, amíg a belső n is páros marad; ha pedig páratlan, akkor elkülönítünk egy a-t: $a^{2n+1} = (a^n)^2 \cdot a$, és innen folytatjuk tovább. Például:

$$a^{22} = \left(a^{11}\right)^2 = \left(\left(a^5\right)^2 \cdot a\right)^2 = \left(\left(\left(a^2\right)^2 \cdot a\right)^2 \cdot a\right)^2.$$

Így tehát a^{22}-t nem 21, hanem mindössze 6 szorzással ki tudjuk számítani.

A fenti felírásnak szoros kapcsolata van a kitevő kettes számrendszerbeli (*bináris*) alakjához. Minden lépésben van egy négyzetreemelés, ami után néha megszorozzuk az eredményt a-val, néha nem. Ha minden a-val való szorzáskor írunk egy 1-est, máskor pedig egy 0-t, majd az egész elejére írunk még egy 1-est (a legbelső a-hoz), akkor éppen a kitevő bináris felírását kapjuk, pl. $22 = 10110_2$.

Mivel modulárisan számolunk, ezért vennünk kell az eredmény maradékát – de ne csak a legvégén, hanem minden lépés után, különben a szám nagyon nagyra nőhet, és a "gyorshatványozás" lassú lesz.

Gyorshatványozás(a, n, m)**:**

- **Bemenet:** $a \in \mathbb{Z}$ alap, $n \in \mathbb{N}^+$ kitevő, $m \in \mathbb{N}^+$ modulus
- $B :=$ Kettes számrendszerbe(n) (lásd lent)
- $r := a := a \bmod m$
- **ciklus** $b := B$ *elemei visszafelé, kivéve az utolsó* 1*-est*
  - $r := r \cdot r \bmod m$
  - **ha** $b = 1$
    - $r := r \cdot a \bmod m$
- **Kimenet:** r

Kettes számrendszerbe(n)**:**

- **Bemenet:** $n \in \mathbb{N}^+$
- **Kimenet:** B, a számjegyek listája fordított sorrendben: $n = (B[l] \ldots B[1]B[0])_2$, azaz $n = B[0] + B[1] \cdot 2 + B[2] \cdot 2^2 + \ldots + B[l] \cdot 2^l$ (ahol $B[l]$ mindig 1)
- $B :=$ *(üres lista)*
- **ciklus amíg** $n \neq 0$
  - B végére szúrjuk be $n \bmod 2$-t
  - $n := n \operatorname{div} 2$

Az algoritmus nagyon hatékony, mert legfeljebb $2L$ szorzást végez, ahol L a kitevő hossza kettes számrendszerben, azaz $L \approx \log_2 n$. Például ha $n = 1000000$, akkor ez legfeljebb 40 szorzást jelent.

Megjegyzés: a gyorshatványozás nemcsak moduláris hatványozásra működik, hanem bármely olyan matematikai objektum hatványozására, amelyet lehet szorozni (pl. egész szám, valós szám, mátrix stb.). Ekkor a maradékképzéseket $(\bmod m)$ értelemszerűen elhagyjuk.

3. A hatványok struktúrája

Most vizsgáljuk meg különböző számok moduláris hatványait a $p = 7$ modulusra:

```
sage: matrix([[power_mod(a, n, 7) for n in range(30)] for a in range(1, 7)])
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4]
[1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5]
[1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2]
[1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3]
[1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6]
```

Először is vegyük észre, hogy a hatványok periodikusan ismétlődnek. Az ismétlődés nem meglepő, hiszen csak véges sok maradék lehetséges, tehát nem lehet minden hatvány különböző. Az is látható, hogy legelőször mindig az $1 (= a^0)$ ismétlődik.

Miért van ez így? Mert ha van bármilyen más ismétlés, pl. $a^8 \equiv a^2$ (7), akkor ezt megszorozva a^2 moduláris inverzével, kapunk egy korábbi ismétlést: $a^8(a^2)^{-1} \equiv a^6 \equiv 1$ (7).

Most tegyük fel azt a kérdést, hogy *mikor* lesz a legelső ismétlés.

3.1. Definíció. $a \in \mathbb{Z}$ **rendje** modulo m az a legkisebb $n \in \mathbb{N}^+$ kitevő, amelyre $a^n \equiv 1 \pmod{m}$. [Angolul: *order*.] Jelölés: $\mathrm{ord}_m(a)$.

Példa. $\mathrm{ord}_7(4) = 3$, azaz 4 rendje 3 modulo 7, mert $4^3 \equiv 1$ (7), de semelyik kisebb hatványa sem 1: $4^1 \equiv 4$ (7) és $4^2 \equiv 2$ (7). Vagy például a táblázatból azt is leolvashatjuk, $\mathrm{ord}_7(5) = 6$.

Egy szám rendje tehát az a legkisebb pozitív kitevő, amelyre emelve 1 maradékot ad. Mivel láttuk, hogy az 1 az első ismétlődő hatvány, ezért a rend megegyezik az ismétlés periódusának hosszával: $a^{k+\mathrm{ord}_m(a)} \equiv a^k a^{\mathrm{ord}_m(a)} \equiv a^k \cdot 1 \equiv a^k \pmod{m}$.

A táblázatból az is látszik, hogy minden nemnulla maradék hatodik hatványa 1 (de nem mindenütt ez az első 1-es). Ez általában is igaz:

3.2. Tétel (Kis Fermat-tétel)**.** *Ha p prímszám és $a \not\equiv 0 \pmod{p}$, akkor $a^{p-1} \equiv 1 \pmod{p}$.*

Ebből tehát látható, hogy minden szám rendje legfeljebb $p-1$ lehet, azaz $\mathrm{ord}_p(a) \leq p-1$. De ennél többet is tudunk mondani a rendről:

3.3. Tétel (Lagrange-tétel)**.** *Ha p prímszám és $a \not\equiv 0 \pmod{p}$, akkor $\mathrm{ord}_p(a) \mid p-1$.*

És valóban, leolvasható, hogy a számok rendje 1, 2, 3 vagy 6, amelyek éppen a $7-1 = 6$ osztói.

Külön felhívjuk a figyelmet két speciális esetre. Egyrészt természetesen bármely modulusra $\mathrm{ord}_m(1) = 1$, hiszen 1 bármely hatványa 1. Másrészt $\mathrm{ord}_7(6) = 2$, és ez is igaz általában is: $\mathrm{ord}_m(m-1) = 2$, mert $m-1$ kongruens -1-gyel, és $(-1)^2 = 1$, azaz $(m-1)^2 \equiv (-1)^2 \equiv 1 \pmod{m}$ (kivéve $m = 2$ esetén, ahol nem ez az első 1-es, mert $-1 \equiv 1 \pmod{2}$).

Megfigyelhető az is, hogy bármely szám moduláris hatványai között szerepel a moduláris inverze. Miért? Mert a kis Fermat-tétel szerint $a^{p-1} \equiv 1$ (p), és ezt megszorozva a-nak az inverzével azt kapjuk, hogy $a^{p-2} \equiv a^{-1}$ (p). A legelső előfordulása a^{-1}-nek $a^{-1} \equiv a^{\mathrm{ord}_p(a)-1}$ (p). (Például $p = 7$ esetén 4 inverze 2, rendje 3, és valóban: $4^{3-1} \equiv 2$ (7).) Ezzel tehát kaptunk egy újabb módszert a moduláris inverz kiszámítására (az előző jegyzetben tárgyaltakon felül).

Különösen érdekesek azok a számok, amelyek rendje a lehető legmagasabb érték:

3.4. Definíció. Egy p prímszám esetén $g \in \mathbb{Z}$ **primitív gyök** modulo p, ha $\mathrm{ord}_p(g) = p-1$.
(Másik elnevezése: **generátor**.)

Példa. $p = 7$ esetén primitív gyök a 3 és az 5, mert azok rendje 6 (lásd a táblázatot a jegyzet elején).

Miért érdekesek a primitív gyökök? Nézzük például 5 hatványait modulo 7: $1, 5, 4, 6, 2, 3, 1, 5, \ldots$. Vegyük észre, hogy minden nemnulla maradék szerepel közöttük. Ez igaz általában is:

3.5. Tétel. *Ha p prímszám és g primitív gyök modulo p, akkor $\forall a \not\equiv 0 \ (p) : \exists n \in \mathbb{N} : g^n \equiv a \ (p)$.*

Más szóval: egy primitív gyök a hatványaival előállítja ("generálja") az összes létező maradékot.

(A bizonyítás egyszerű: a primitív gyök definíciója miatt az első ismétlés a $p-1$-edik hatványban lesz, tehát az első $p-1$ hatványnak különbözőnek kell lennie, viszont mivel éppen $p-1$-féle maradék van, ezért mindegyiknek szerepelnie kell az első $p-1$ hatvány között.)

Felmerül a kérdés, hogy vajon minden p prímszámra találunk-e primitív gyököt. A válasz igenlő (de nehéz bizonyítani, ezért azt mellőzzük):

3.6. Tétel. *Ha p prímszám, akkor létezik primitív gyök modulo p.*

4. Diszkrét logaritmus

Egy primitív gyök tehát előállít minden nemnulla maradékot, és periódusonként mindegyiket pontosan egyszer. Adja magát a kérdés: egy adott maradék hányadik hatványként áll elő? Vegyük észre, hogy ez a kérdés éppen a hatványozás inverze: most nem a hatvány értékére vagyunk kíváncsiak az alap és a kitevő ismeretében, hanem a kitevőt szeretnénk megkapni az alapból és a hatványból. Ez nagyon hasonlít a valós számoknál ismert logaritmusra: az $b^x = a$ valós egyenlet megoldása $x = \log_b a$, például $2^x = 4\sqrt{2}$ megoldása $x = \log_2 4\sqrt{2} = 2{,}5$. Vezessük be ennek a moduláris analógját:

4.1. Definíció. $a, b \in \mathbb{Z}$ esetén a-nak a b-alapú **diszkrét logaritmusa** (vagy **indexe**) modulo m az a legkisebb $n \in \mathbb{N}$ kitevő, amelyre $b^n \equiv a \pmod{m}$. Jelölés: $n = \operatorname{ind}_b a \pmod{m}$.

Példa. 6-nak az 5-ös alapú diszkrét logaritmusa 3 modulo 7, azaz $\operatorname{ind}_5 6 = 3 \ (7)$, mert $5^3 \equiv 6 \ (7)$.

A fentiek alapján ha g egy primitív gyök modulo p, akkor bármely $a \not\equiv 0 \ (p)$ számra létezik $\operatorname{ind}_g a$.

De hogyan tudjuk kiszámítani a diszkrét logaritmust? A legegyszerűbben lineáris kereséssel, azaz hogy kiszámítjuk g moduláris hatványait g^{p-2}-ig – ehhez minden lépésben egy szorzás és egy maradékos osztás kell –, és amelyik megegyezik a-val, annak a kitevője lesz a diszkrét logaritmus. Ez azonban nagy p-kre nagyon lassú, pl. $p \approx 1000000$ esetén kb. egymillió szorzást igényel.

Igazán hatékony módszert azonban nem ismerünk; a diszkrét logaritmus probléma mind a mai napig a számításelmélet nyitott problémái közé tartozik. Bár vannak a fenti lineáris keresésnél *valamelyest* hatékonyabb módszerek (egy ilyet bemutatunk a következő szakaszban), de egyik sem működik olyan nagy számokra, mint a gyorshatványozás. Vagyis a moduláris hatványozás műveletét el tudjuk végezni gyorsan, de az inverzét, a diszkrét logaritmust nem.

De ez nem baj. Sőt, mint majd látni fogjuk, kifejezetten hasznos!

A Sage a diszkrét logaritmust a `discrete_log()` függvénnyel tudja kiszámítani (ami sokkal általánosabb, mint amire szükségünk van, ezért kissé körülményesen kell meghívni):

```
sage: discrete_log(mod(6, 7), mod(5, 7))   # 5-ös alapú log 6 (mod 7)
3
sage: power_mod(5, 3, 7)    # ellenőrzés
6
```

De nagyon nagy modulusokra persze a `discrete_log()`-ot is hiába próbáljuk:

```
sage: p = next_prime(2^1000)
sage: a = power_mod(7, randrange(p), p)
sage: discrete_log(mod(a, p), mod(7, p))    # nem fog menni
...
```

4.1. Baby-step giant-step

A diszkrét logaritmusra egy – viszonylag – hatékony módszer a **baby-step giant-step** algoritmus. Az ötlet az, hogy vegyünk egy közbülső M számot, amelyre kb. $M^2 \approx p-1$, és keressük az $a \equiv g^n \ (p)$ hatvány ismeretlen n kitevőjét $n = iM+j$ alakban, ahol $0 \le i, j < M$ (például ha $M = 10$ és $n = 76$, akkor $n = 7M+6$, azaz $i = 7$ és $j = 6$). Először számítsuk ki az első néhány g^j hatványt (modulo p): $1,\ g,\ g^2,\ \ldots,\ g^{M-1}$ ("baby-step"), és tároljuk el őket. Ezután vegyük a kérdéses hatványt (a), és kezdjük el szorozgatni g^{-M}-nel: $a,\ g^{-M}a,\ g^{-2M}a,\ \ldots$ ("giant-step"). Ez azért jó, mert a két sorozat (a "baby-step" és a "giant-step") előbb-utóbb összeér: amikor a-t éppen g^{-iM}-nel szoroztunk meg (ahol i az $n = iM + j$ felírásból van, tehát nem ismerjük), akkor $g^{-iM}a = g^{-iM+n} = g^j$, ami szerepel az eltárolt g^j értékek között. Ha ezt megtaláltuk, akkor megvan i és j, és abból összerakhatjuk n-et.

Baby-step giant-step algoritmus:

Bemenet: p prímszám, $a \in \mathbb{Z}$ (a hatványérték), g primitív gyök modulo p
$M := \left\lceil \sqrt{p-1} \right\rceil$
$b := 1$
ciklus $j = 0, 1, \ldots, M-1$
  Tároljuk el (b, j)-t egy táblában. (Ahol $b \equiv g^j \ (p)$.)
  $b := b \cdot g \bmod p$
$c := b^{-1} \bmod p \ \ (= g^{-M} \bmod p)$
$b := a$
ciklus $i = 0, 1, \ldots, M-1$
  ha *b benne van a táblában valamely j-vel*
    megtaláltuk: $\rightarrow n := iM + j$
  $b := b \cdot c \bmod p$

Ahhoz, hogy ez hatékony legyen, az kell, hogy a "táblában" gyorsan tudjunk értékeket tárolni és visszakeresni, erre például egy hash-tábla alkalmas. A tábla kulcsa b, az értéke pedig a j index. Például Sage-ben használhatjuk a Python-féle szótár (`dict`) típust (üres szótár: `T = {}`, elem hozzáadása: `T[b] = j`, elem keresése: `b in T`, majd lekérdezése: `T[b]`).

Az algoritmus kb. $2\sqrt{p}$ szorzást igényel, amely a lineáris keresésnél sokkal jobb (pl. $p \approx 1000000$ esetén csak $\sim$2000 szorzás kell hozzá). De ha összehasonlítjuk a gyorshatványozással, akkor ez nagy p-kre még mindig lassú. Ráadásul a memóriaigénye is sokkal nagyobb (kb. $2\sqrt{p}$ számot kell tárolni).

5. Diffie–Hellman-kulcscsere

Tegyük fel, hogy két ember titkosítva szeretne kommunikálni egy olyan csatornán keresztül, amit bárki lehallgathat (pl. internet). Ehhez szükségük van egy *titkos kulcs*ra, amelynek ismeretében ők ketten el tudják olvasni egymás üzeneteit, mások viszont nem. A probléma az, hogy még sosem találkoztak, így a kulcsot is csak ezen a nyilvános csatornán keresztül tudják megbeszélni, ahol bármit mondanak, azt más is hallhatja. Hogyan tudják mégis egyeztetni a közös titkos kulcsot?

Erre az első ránézésre lehetetlennek tűnő kérdésre ad választ a **Diffie–Hellman-kulcscsere**. A módszer azon alapul, hogy a diszkrét logaritmus probléma nehéz, azaz ha van egy titkos $n \in \mathbb{N}$ kitevőnk, akkor a $g^n \bmod p$ hatványt nyugodtan nyilvánosságra hozhatjuk (g-vel és p-vel együtt), hiszen senki nem fogja tudni kiszámítani belőle a titkos n kitevőt.

A Diffie–Hellman-kulcscsere a következőképpen működik (minden hatványt modulo p értünk):

1. A két résztvevő, Alíz és Bob megegyeznek egy p prímszámban és egy g primitív gyökben modulo p (ezek nyilvános adatok, és akár előre rögzített konstansok is lehetnek).
2. Mindketten generálnak egy-egy véletlen, titkos kitevőt 1 és $p-1$ között: a és b, ahol a-t csak Alíz ismeri, és b-t csak Bob.
3. Alíz kiszámítja g^a-t, és elküldi Bobnak.
4. Bob kiszámítja g^b-t, és elküldi Alíznak.
5. Ezután mind a ketten kiszámítják g^{ab}-t az általuk ismert adatokból:
   - Alíz ismeri a-t és g^b-t, ebből $g^{ab} = \left(g^b\right)^a$.
   - Bob ismeri b-t és g^a-t, ebből $g^{ab} = \left(g^a\right)^b$.

A közös titok tehát g^{ab}, amelyet a fenti műveletsor után mind Alíz, mind Bob ki tud számítani. Viszont bárki más, aki hallgatózik a csatornán, csak g-t, g^a-t és g^b-t láthatja, ezekből pedig nem fogja tudni kiszámítani g^{ab}-t, mert ahhoz ismernie kéne valamelyik titkos kitevőt (a-t vagy b-t), ahhoz pedig meg kéne oldania a diszkrét logaritmus problémát.

Példa.

1. Legyen a modulus $p = 23$ és a primitív gyök $g = 5$.
2. Alíz véletlen titkos kitevője $a = \mathbf{7}$, Bobé pedig $b = \mathbf{21}$.
3. Alíz számítása: $g^a \equiv 5^7 \equiv \mathbf{17} \pmod{23}$, amit elküld Bobnak.
4. Bob számítása: $g^b \equiv 5^{21} \equiv \mathbf{14} \pmod{23}$, amit elküld Alíznak.
5. Ebből mindketten ki tudják számítani g^{ab}-t:
   - Alíz: $g^{ab} \equiv \left(g^b\right)^a \equiv 14^7 \equiv \mathbf{19} \pmod{23}$.
   - Bob: $g^{ab} \equiv \left(g^a\right)^b \equiv 17^{21} \equiv \mathbf{19} \pmod{23}$.

A Diffie–Hellman-kulcscsere biztonságához többek között az kell, hogy a p prímszám elég nagy legyen (és bizonyos egyéb tulajdonságoknak is megfeleljen, amelyeket itt nem részletezünk). A gyakorlatban néhány ezer bites prímszámokat szoktak használni (pl. 2048-bitest, azaz p felírása kettes számrendszerben 2048 számjegy, ami tízes számrendszerben 617 számjegy), lásd pl.: [5].

A Diffie–Hellman-kulcscserét számos helyen használják, különösen internetes rendszerekben, mint HTTPS, SSH, VPN-ek stb. – általában más kriptográfiai algoritmusokkal együtt, pl. RSA és AES (amelyekről későbbi jegyzetekben lesz szó).

6. Véletlenszám-generátorok

A legtöbb programozási nyelvben van *véletlenszám-generátor* [*random number generator (RNG)*]. Ezek azonban legtöbbször nem "valódi" véletlen számokat állítanak elő (mint amit például kockadobással vagy egyéb fizikai eljárással kaphatnánk), hanem egy egyszerű algoritmussal számítják ki a – kellően véletlennek tűnő – értékeket. Az ilyeneket *pszeudovéletlen számok*nak nevezzük.

Egy nagyon egyszerű pszeudovéletlenszám-generátort kaphatunk moduláris hatványozással. Bár a hatványok periodikusan ismétlődnek, de egy perióduson belül kellően véletlenszerűnek tűnnek (például 5 hatványai modulo 7 a következők: $1, 5, 4, 6, 2, 3$). Például C++-ban az `std::minstd_rand` [6] generátor a következő paramétereket használja:

$$x_n := 48271^n x_0 \bmod 2147483647,$$

ahol x_0 tetszőleges kezdőérték [angolul *seed*] (ezt gyakran az aktuális időből hozzák létre, hogy mindig más legyen); a modulus $p = 2147483647 = 2^{31} - 1$ egy olyan prímszám, hogy a maradékok éppen beleférjenek egy előjeles 32-bites számba (és hogy hatékonyan lehessen vele maradékosan osztani); a szorzó, 48271 pedig egy primitív gyök modulo p.

Ez a véletlenszám-generátor nagyon egyszerű és nagyon hatékony, ezért jól használható például szimulációkban és játékokban. Nem alkalmas viszont a Diffie–Hellman-kulcscsere kitevőinek (a és b) generálására (sem bármely más olyan algoritmusban, ahol a biztonság fontos), mert az egyszerűsége miatt nagyon könnyű megjósolni a következő számokat, és ezáltal az egész eljárás értelmét veszti.

Ehelyett ún. *biztonságos véletlenszám-generátorokat* használnak [*cryptographically secure pseudo-random number generators (CSPRNG)*], amelyek bonyolultabbak, de kellő matematikai garanciával rendelkeznek arra, hogy ne lehessen megjósolni a következő számokat (még egyes részeiket sem).

Számos programozási nyelvben, így Sage-ben (és Python-ban) is vannak külön biztonságos és nem biztonságos véletlenszám-generátorok:

```
sage: randrange(100)           # NEM biztonságos
94
sage: import secrets
sage: secrets.randbelow(100)   # biztonságos
67
```

7. Szerző

Ezt a jegyzetet Uray M. János írta, részben felhasználva az alábbi forrásokat:

[1] https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa

[2] https://en.wikipedia.org/wiki/Modular_exponentiation

[3] https://en.wikipedia.org/wiki/Discrete_logarithm

[4] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

[5] Prímek a Diffie–Hellman-kulcscseréhez: https://www.rfc-editor.org/rfc/rfc3526

[6] https://en.cppreference.com/w/cpp/numeric/random/linear_congruential_engine.html

Hibákat, észrevételeket itt lehet jelezni: uray.janos@inf.elte.hu.