Diszkrét matematika
12. előadás

MÁRTON Gyöngyvér
mgyongyi@ms.sapientia.ro

Sapientia Egyetem,
Matematika-Informatika Tanszék
Marosvásárhely, Románia

2022, őszi félév

Miről volt szó az elmúlt előadáson?

- a kínai maradéktétel
- az RSA és a kínai maradéktétel
- másodfokú kongruenciák, kvadratikus maradékok
- a Legendre és Jacobi szimbólumok,
- a Rabin rendszer: titkosító, digitális aláírás rendszer

Miről lesz szó?

- összetett számok faktorizációja:
  - a Fermat féle faktorizáció
  - a Pollard ρ féle faktorizáció
- a diszkrét logaritmus, algoritmusok
  - a brute-force algoritmus
  - Baby-step Giant-step, Shanks algoritmusa
- ál-véletlenszám generátorok (pseudo-random number generators)
  - a lineáris kongruencián alapuló generátor
  - a közép-négyzet módszer (middle-square)
  - a Blum-Blum-Shub generátor

Összetett számok faktorizációja

- az összetett számok faktorizációjával, azaz prímtényezők szotzatára való bontással az ókori görögök is intenzíven foglalkoztak,
- nagy összetett számok faktorizációjára nem ismert hatékony algoritmus,
- nagyobb számok faktorizációját a számítógépek megjelenése tette lehetővé,
- Peter Shor, 1997-ben bemutatott egy hatékony algoritmust, amely elméletben, kvantum számítógépek számításaira alapulva képes nagy számokat is faktorizálni, de 2001-ben még csak olyan kvantum gép létezett, amely a 15-ös számot volt képes faktorizálni,
- a mai kutatások olyan n összetett számokat próbálnak faktorizálni, ahol $n = p \cdot q$, és p, q prímszámok,
- RSA factoring Challenge: 1991-ben induló verseny, amely komoly pénzösszeggel jutalmazza azokat, akik bizonyos nagy számokat faktorizálnak:
  - 50,000$ kapott 2009-ben Thorsten Kleinjung és csapata a 768 bites RSA768 szám faktorizálásáért,
  - 100,000$ pénzjutalomat kap az, aki az 1024 bites RSA1024 számot faktorizálja, stb.

Összetett számok faktorizációja

A faktorizációs algoritmusok két csoportra oszthatók, a már bemutatott osztási próba és Eratosztenész szitája algoritmusok mellett:

- a speciális célú algoritmusok csak speciális alakú számok esetében alkalmazhatóak sikeresen, általános esetben nem működnek:
  - Fermat faktorizációs módszere: olyan összetett számok, ahol a p és q között kicsi a különbség
  - Pollard rho faktorizációs módszere: olyan összetett számok, ahol az osztók kis számok,
  - Pollard $p-1$ módszere: olyan összetett számok, ahol az osztók szomszédai kis számok,
  - Lenstra ECM módszere: elliptikus görbéken alapuló faktorizációs eljárás, leggyorsabb a speciális célú algoritmusok között, stb.
- az általános célú algoritmusok nagy memória és tár kapacitást igényelnek, ha a speciális célú algoritmusok nem adnak eredményt, akkor szokták őket használni:
  - faktorizálás lánctörtekkel,
  - a kvadratikus szita módszer,
  - az általánosított szám test szita (general number field sieve) módszer a leggyorsabb

Fermat faktorizációs módszere

Legyen $n = p \cdot q$, ahol feltételezzük, hogy a p és a q prímek közötti különbség kicsi:

- megpróbáljuk felírni n-et $n = a^2 - b^2$, alakba, ahol a, b tetszőleges egész számok, ekkor azonban $p = a - b$, $q = a + b$,
- meghatározzuk n négyzetgyökének felső egészrészét, legyen ez a
- $a, a+1, a+2, \ldots$ értékekre meghatározzuk a $b1 = a^2 - n$ értékeket, mindaddig amíg $b1$ nem lesz négyzetszám, ekkor megadható n két osztója: $a - \sqrt{b1}$ és $a + \sqrt{b1}$.

Példa. Határozzuk meg $n = 6283$ két prímosztóját, a Fermat faktorizációs módszerével:

$\lceil \sqrt{n} \rceil$	a	$b1 = a^2 - n$	négyzetszám-e?
80	80	117	nem
	81	278	nem
	82	441	igen

$b = \sqrt{b1} = \sqrt{441} = 21 \Rightarrow$

$$\begin{aligned} p &= 82 - 21 = 61, \\ q &= 82 + 21 = 103 \end{aligned}$$

Fermat faktorizációs módszere

1. feladat

Írjunk egy Python függvényt, amely Fermat faktorizációs módszerével meghatározza egy összetett szám prímtényezős felbontását.

```
from decimal import Decimal, getcontext
def fermatFaktorizacio(n):
    getcontext().prec = 400
    a = int(Decimal(n).sqrt()) + 1
    while True:
        b1 = a * a - n
        b = negyzetTeszt(b1)
        #print("%6i%6i%6i" % (a, b1, b))
        if b != -1:
            return (a - b, a + b)
        a += 1

>>> fermatFaktorizacio(4668999961)
    (29033, 160817)
```

```
def negyzetTeszt(x):
    i = int(Decimal(x).sqrt())
    if i*i == x: return i
    else: return -1
```

Fermat faktorizációs módszere

2. feladat

Írjunk egy Python függvényt, amely a compNr.txt állományban található számokat megpróbálja faktorizálni Fermat faktorizációs módszerével. Módosítsuk úgy a `fermatFaktorizacio` függvényt, hogy TIME LIMIT hibaüzenetet adjon, ha 10 másodperc alatt sem sikerül faktorizálni egy adott számot.

```
from time import time
from decimal import Decimal, getcontext
def fermatFaktorizacioTime(n):
    st = time()
    getcontext().prec = 400
    a = int(Decimal(n).sqrt()) + 1
    while True:
        b1 = a * a - n
        b = negyzetTeszt(b1)
        fs = time()
        if fs - st > 10: return -1
        if b != -1:
            return (a - b, a + b)
        a += 1
```

```
def fermatTime(nev = 'compNr.txt'):
    inf = open(nev, 'rt')
    temp = inf.read()
    L = temp.split('\n')
    inf.close()
    for elem in L:
        elem = int(elem)
        print(elem)
        res = fermatFaktorizacioTime(elem)
        if res == -1: print('TIME LIMIT')
        else: print(res)
        print()
```

Pollard ρ faktorizációs módszere

- John Pollard publikálta 1975-ben,
- az algoritmus keresi, azt a p számot, amely az n osztója lehet,
- ha a, b, két egész melyre: $0 < a, b < n$ és $a \neq b$, és $a = b \pmod{p}$, akkor $a - b$ a p egy többszöröse lesz,
- ekkor $p \leq gcd(a - b, n) < n$, azaz $a - b$ és n legnagyobb közös osztója egy nem triviális osztója lesz n-nek,
- az $f(x_i) = (x_{i-1}^2 + 1) \pmod{n}$ függvénnyel egy-egy számsorozatot generálunk, amelybe tulajdonképpen álvéletlen módon előállított számok kerülnek, ahol $x_0 = 1$,
- az előállított számsorozatban $a = x_i$ és $b = x_j$-re vizsgáljuk, hogy mikor lesz $gcd(x_i - x_j, n) \neq 1$,
- a hatékonyság miatt csak, ha $i = 2 \cdot j$, akkor számolunk legnagyobb közös osztót,
- a legnagyobb közös osztó meghatározását az eukleidészi algoritmussal végezzük.

Pollard ρ faktorizációs módszere

Határozzuk meg $n = 221$ két prímosztóját, a Pollard ρ féle faktorizációs módszerrel, ahol $x_0 = 1$:

a	b	$a - b$	$gcd(a - b, n)$
2	26	24	1
5	197	192	1
26	104	78	13
$\Rightarrow$			

$$p = 13,$$
$$q = n/13 = 17$$

Pollard ρ faktorizációs módszere

3. feladat

Határozzuk meg egy összetett szám prímtényezős felbontását a Pollard ρ faktorizációs algoritmussal.

```
from math import gcd
def pollard_rho(n):
    a = 1
    b = 2   #b = a * a + 1
    while True:
        a = (a * a + 1) % n
        b = (b * b + 1) % n
        b = (b * b + 1) % n
        d = gcd (a-b, n)
        print("%7i%7i%7i%7i" % (a, b, a-b, d))
        if 1 < d and d < n: return d
        if d == n: return n
```

Pollard ρ faktorizációs módszere

```
>>> pollard_rho(38989)
         2    26    -24     1
         5 29451 -29446     1
        26  5025  -4999     1
       677 10772 -10095     1
     29451 25123   4328     1
     12108  6581   5527     1
      5025 34775 -29750     1
     24743  8613  16130     1
     10772 16868  -6096   127
     (127, 307)

>>> pollard_rho(212612371982541690127801)
    (145812335489, 145812335609)

>>> pollard_rho(14906395069378547320638764 3)
    (10223, 14581233560968939959541)
```

A diszkrét logaritmus, algoritmusok

1. értelmezés

Az egész számok $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ *véges halmaza esetében, ahol adott egy* p *prímszám,* p*-nek egy* g *generátor eleme, és egy* $A \in \mathbb{Z}_p^*$ *egész szám a diszkrét logaritmusa (DL) probléma azt jelenti, hogy megadott megkeressük azt az* $a \in \mathbb{Z}_p^*$ *pozitív egész számot, melyre fennáll:*

$$g^a \equiv A \pmod{p}.$$

- nagy számok esetében a DL probléma meghatározására nem ismert hatékony algoritmus,
- nem triviális algoritmusok a DL problémára:
  - a baby-step giant-step (Shanks) algoritmus,
  - a Pohlig-Hellman algoritmus,
  - az indexkalkulus algoritmus.
- a baby-step giant-step (BsGs) algoritmus alapötlete Daniel Shanks-től származik 1971-ből, és a korábban tárgyalt a brute force algoritmus egy módosított változata.

Baby-step Giant-step, Shanks algoritmusa

- a $g^a \equiv A \pmod{p}$ esetében, g, A, p bemenetre keressük az a kitevő értékét
- a kitevő értéke felírható a következő formában:

$$a = m \cdot i + j, \text{ ahol } 0 \leq i,\ j < m \text{ és } m = \lceil \sqrt{p-1} \rceil, \text{ akkor}$$

$$\begin{aligned} g^a &= g^{m \cdot i + j} = g^{m \cdot i} \cdot g^j \Rightarrow \\ g^j &= g^a \cdot g^{-m \cdot i} = A \cdot (g^{-1})^{m \cdot i}. \end{aligned}$$

- Az algoritmus:
  - meghatározzuk a g^j értékeket, $j = 0,\ 1,\ \ldots,\ m-1$
  - meghatározzuk g inverzét $\pmod{p}$ szerint: $gInv = g^{-1}$-t
  - meghatározzuk $gInvM = gInv^m = (g^{-1})^m$ értékét
  - megvizsgáljuk, hogy milyen $i = 0,\ 1,\ \ldots,\ m-1$ -re egyezik meg $A \cdot ((g^{-1})^m)^i$ értéke valamely g^j-vel.

Baby-step Giant-step, Shanks algoritmusa

Példa: Határozzuk meg a diszkrét logaritmus értékét $p = 19, g = 3, A = 11$ esetében.

- a kitevő értékét felírjuk: $a = m \cdot i + j$, ahol $m = \lceil \sqrt{p-1} = \lceil \sqrt{18} \rceil = 5$
- meghatározzuk a g^j értékeket, j = 0, 1, 2, 3, 4-re:
$$g^0 = 1, g^1 = 3, g^2 = 9, g^3 = 8, g^4 = 5,$$
- meghatározzuk $g = 3$ inverzét: $g^{-1} = 13$, fennáll: $3 \cdot 13 = 1 \pmod{19}$
- meghatározzuk $(g^{-1})^m = 13^5 = 14 \pmod{19}$ értéket
- megvizsgáljuk, hogy melyik $i = 0, 1, 2, 3, 4$ -re egyezik meg $A \cdot (gInvM^i)$ értéke valamely g^j-vel:
$$11 \cdot 14^0 = 11, 11 \cdot 14^1 = 2, 11 \cdot 14^2 = 9$$
- $i = 2, j = 2$ esetén van találat, azaz $g^2 = 11 \cdot (g^{-1})^{5 \cdot 2}$, azaz $g^{2+5 \cdot 2} = 11$
- tehát a kitevő $a = 2 + 5 \cdot 2 = 12$

Baby-step Giant-step, Shanks algoritmusa

A dLogAux1 függvényben meghatározzuk a $g^0, g^1, \ldots, g^{\lceil\sqrt{p-1}\rceil-1}$ hatványértékeket:

```
def dLogAux1(g = 3, A = 11, p = 19):
    m = int((p-1) ** 0.5) + 1
    L, gPow = [], 1
    for j in range(m):
        L = [ pow(g, j, p) ] + L
    print('m:', m, ', L: ', L)

>>> dLogAux1()
    m: 5 , L:  [5, 8, 9, 3, 1]
```

A dLogAux2 a hatványértékeket úgy határozza meg, hogy nem a pow függvényt alkalmazza, hanem az előzőleg már kiszámított hatványértéket szorozza g-vel:

```
def dLogAux2(g = 3, A = 11, p = 19):
    m = int((p-1) ** 0.5) + 1
    L, gPow = [], 1
    for j in range(m):
        L = [gPow] + L
        gPow = (gPow * g) % p
    print('m:', m, ', L: ', L)
```

Baby-step Giant-step, Shanks algoritmusa

4. feladat

Írjunk egy Python függvényt, amely a baby-step giant-step algoritmussal a g, A, p bemenetre meghatározza a-t, úgy hogy teljesüljön: $A = g^a \pmod{p}$.

```
from eload10 import inverz
def dLog(g, A, p):
    m = int((p-1) ** 0.5) + 1
    L, gPow = [], 1
    for j in range(m):
        L += [gPow]    #L = [gPow] + L
        gPow = (gPow * g) % p
    #print('m:', m, ', L: ', L)
    gInv = inverz(g, p)
    gInvM = pow(gInv, m, p)
    #print('gInv: ', gInv, ', gInvM:', gInvM)
    gPow = 1
    for i in range(m):
        c = (A * gPow) % p
        #print(c, end = ' ')
        try:
            j = L.index(c)  #j = (m-1) - L.index(c)
            #print()
            return i * m + j
        except:
            gPow = (gPow * gInvM) % p
            continue
    return -1
```

```
>>> dLog(3, 11, 19)
    m: 5 , L:  [5, 8, 9, 3, 1]
  gInv:  13 , gInvM: 14
  11 2 9
  12

>>> L = [5, 8, 9, 3, 1]
>>> L.index(11)
    ...ValueError: 11 is not in list

>>> L.index(2)
    ...ValueError: 2 is not in list

>>> L.index(9)
    2


>>> dLog(1234, 10000, 530063)
    73132

>>> dLog(2, 1982257490, 2392893431)
    1234567
```

Véletlenszám generátorok

- véletlen, illetve álvéletlen módon generált számokra számos algoritmusban szükség van,
- véletlen módon generált számok (true random numbers): hardver eszközökkel,
- álvéletlen módon generált számok (pseudo random numbers): szoftver eszközökkel,
- a kriptográfiában komolyabb biztonsági kritériumnak eleget tevő álvéletlen számokat generáló algoritmusokat is használnak, ezek *erős* algoritmusok, ezek kulcs szerepet játszanak a rendszerek biztonságában,
- a gyakorlatban a Diffi-Hellman, az RSA, a Rabin rendszerek által kezelt üzeneteket, titkos információkat kiegészítik pszeudo random módon genereált bittekkel, ezek során *erős* algoritmusokat használnak,
- a publikus, a privát kulcsok generálása során nem szükséges *erős* algoritmusokat használni,
- álvéletlen számokat generáló algoritmusok:
  - a lineáris kongruencián alapuló generátor
  - a közép-négyzet módszer (middle-square)
  - a Blum-Blum-Shub generátor: *erős* álvéletlen számokat generál
- minden programozási nyelv rendelkezik álvéletlenszámokat generáló algoritmusokkal, amelyek nem alkalmasak kriptográfiai rendszerekben.

A lineáris kongruencián alapuló generátor

- álvéletlen számokat generál, nagyon gyors, kis memória igényű,
- jó statisztikai viselkedés jellemzi $\Rightarrow$ olyankor alkalmazzák, amikor nem követelmény a rendszerek biztonsága
- az alkalmazott matematikai összefüggés:

  $$z_{i+1} = (a \cdot z_i + b) \pmod{m}, \text{ ahol } 1 \leq a, b \leq m-1, \text{ és } m \geq 2, \text{ ahol}$$

  az m -modulus, az a -szorzó, a b -növekedési-érték, és z_0 -kezdeti érték (seed) konstans értékek, kiválasztásuk a generátor használhatóságát is meghatározza.
- Példa, legyen $a = 1, b = 7, m = 10, z_0 = 3$, ekkor
  - az alkalmazott képlet: $z_{i+1} = (z_i + 7) \pmod{10}, z_0 = 3$
  - a generált számsorozat: 0 7 4 1 8 5 2 9 6 3 0 7 4 1...

A lineáris kongruencián alapuló generátor

5. feladat

Írjunk egy Python programot, amely a $z_0 = 3, m = 10, a = 1, b = 7$ bemenetek esetén meghatározza a lineáris kongruencián alapuló generátor által generált első 15 értéket.

```
from time import time
def pLinGen(z0, m = 10, a = 1, b = 7):
    if z0 != None:
        z1 = z0
    else: z1 = int(time())
    z1 = (z1 * a + b ) % m
    return z1

def main(k, seed = None):
    for i in range(k):
        seed = pLinGen(seed)
        print(seed, end = ' ')

>>> main(15)
    5 2 9 6 3 0 7 4 1 8 5 2 9 6 3
>>> main(15, 3)
    0 7 4 1 8 5 2 9 6 3 0 7 4 1 8
```

A lineáris kongruencián alapuló generátor

- a generátor periódusa akkor és csakis akkor lesz maximális, ha:
  - a és b relatív prímek,
  - $a - 1$ osztható m minden prímosztójával,
  - $a - 1$ és m is 4 valamely többszöröse.
- tipikus értékek:

	m	a	b
`ANSI C`	2^{31}	1103515245	12345
`C/C++`	2^{31}	214013	2531011
`Java`	2^{48}	25214903917	11

6. feladat

Írjunk egy Python programot, amely az AnsiC-nél használt bemeneti paraméterek esetében meghatározza a generátor által generált első értéket.

```
def randAnsiC(z0 = None):
    m = 1 << 31
    a = 1103515245
    b = 12345
    return pLinGen(z0, m, a, b)
```

```
>>> randAnsiC()
    ...
```

A Python véletlenszám generátora

- Mersenne Twister-nek (megbízhatatlan Mersenne) hívják. A periódusa $2^{19937} - 1$ és 53-bites valós számot állít elő. A háttérben levő matematika: lineáris algebra, véges testek, bináris testek elmélete.
- a Mersenne Twister algoritmust 1997-ben publikálta, Takuji Nishimura, Makoto Matsumoto két japán kutató,
- a generátor periódusa egy Mersenne prím,
- Mersenne prímek: azok a $2^n - 1$ alakú prímszámok, ahol n is prímszám,
- Mersenne számok: azok a $2^n - 1$ alakú egész számok, ahol n prímszám, de $2^n - 1$ lehet prím, vagy összetett szám
  - Mersenne prímek: $3 = 2^2 - 1, 7 = 2^3 - 1, 31 = 2^5 - 1, 127 = 2^7 - 1,$
  - Mersenne számok, de nem Mersenne prímek:

$$\begin{array}{lcccl} 2^{11} - 1 & = & 2047 & = & 23 \cdot 89 \\ 2^{23} - 1 & = & 8388607 & = & 47 \cdot 178481 \end{array}$$

  - ha $2^n - 1$ prímszám akkor n prímszám,
  - ha n prímszám, akkor $2^n - 1$ lehet összetett is, és prímszám is,
  - ha n összetett, akkor $2^n - 1$ is összetett,
  - 2018 óta a legnagyobb ismert prímszám a $2^{82,589,933} - 1$ Mersenne prím.

A közép-négyzet módszer (middle-square)

Neumann János módszere, nem alkalmas kriptográfiában:

- a kezdeti seed egy n számjegyű szám
- ezt a számot négyzetre emeljük, szükség esetén kiegészítjük balról nullásokkal
- a véletlenszerűen generált számok a középső számjegyekből képezett n számjegyű számok lesznek

Legyen a seed = 980

$$
\begin{array}{lll}
980^2 = & 0960400 & 604 \\
604^2 = & 0364816 & 648 \\
648^2 = & 0419904 & 199 \\
199^2 = & 39601 & 960 \\
960^2 = & 0921600 & 216 \\
\ldots & &
\end{array}
$$

A közép-négyzet módszer (middle-square)

7. feladat

Írjunk egy-egy Python függvényt, amely megadott r kezdőérték esetén meghatározza a közép-négyzet módszer által generált n-ik értéket, illetve megadja a generátor periódusát.

```
def midSqr(r, n):
    k = len(str(r))
    for i in range(n):
        rL = str(r * r)
        x = (len(rL) - k) // 2
        r = int(rL[x: x + k])
    print(r, end = ' ')
```

```
>>> midSqr(980, 10)
    705
```

```
def midSqrPeriodus(r):
    k = len(str(r))
    L = []
    while len(L) == len(set(L)):
        rL = str(r * r)
        x = (len(rL) - k) // 2
        r = int(rL[x: x + k])
        L += [r]
    return L
```

```
>>> midSqrPeriodus(8193)
    [1252, 5675, 2056, 2271, 1574, 4774, 7910, 5681, 2737, 4911, 1179,
    3900, 2100, 4100, 8100, 6100, 2100]
```

A Blum-Blum-Shub generátor

- alkalmas kriptográfiai rendszerekben, biztonságát a faktorizációs és kvadratikus maradék problémák nehézsége adja
- nagy számításigényű, ezért csak nagy biztonságot követelő rendszereknél alkalmazzák
- a moduláris négyzetreemelés az alkalmazott függvény, ahol az algoritmus a $(b_1, b_2, \ldots b_n)$ bitsorozatot a következőképpen hozza létre:

$$b_i = u_i \pmod 2 \text{ és } u_i = u_{i-1}^2 \pmod N, \text{ ahol}$$

  - u_0 a generátor egy kezdeti értéke, amelyet a következőképpen választunk ki: $u_0 = r^2 \pmod N$, ahol $r \overset{R}{\leftarrow} \{2, 3, \ldots, N-1\}$ és $lnko(r, N) = 1$
  - $N = P \cdot Q$ és P, Q prímek, úgy hogy $P = 2 \cdot p + 1$, $Q = 2 \cdot q + 1$ és p, q is prímek
- a generált P, Q értékek tehát biztonságos prímek, amelyeket Blum egészeknek is hívnak, fennáll: $P \equiv Q \equiv 3 \pmod 4$.

A Blum-Blum-Shub generátor

Példa

- legyen $P = 23, Q = 59$ két biztonságos prím, $N = 1357$
- legyen $u_0 = 9$
- $u_i = u_{i-1}^2 \pmod N, b_i = u_i \% 2$

i	u_i	b_i
1	81	1
2	1133	1
3	1324	0
4	1089	1
5	1260	0
6	1267	1
7	1315	1
8	407	1

A generált bitsorozat: $1, 1, 0, 1, 0, 1, 1, 1$

A Blum-Blum-Shub generátor

8. feladat

Generáljunk l darab bitet a Blum-Blum-Shub generátorral, ahol az alkalmazott N legyen egy k bites szám.

```
from random import getrandbits, randint
def bbs(k, l):
    P = safePrime(k//2)
    Q = safePrime(k//2)
    N = P * Q
    r = randint(2, N)
    u = (r * r) % N
    for i in range(0, l):
        u = (u * u) % N
        print(u & 1, end = ' ')
```

```
def safePrime(k):
    q = getrandbits(k)
    if not (q & 1): q += 1
    while True:
        p = 2 * q + 1
        if miller_rabinT(q, 10) and miller_rabinT(p, 10):
            return p
        q = getrandbits(k)
        if not (q & 1): q += 1
```

```
>>> bbs(64, 16)
    0 1 1 0 1 0 0 1 1 1 0 1 1 1 0 1
```

A Blum-Blum-Shub generátor

9. feladat

Írjunk egy Python programot, amely menüpontokból, a következőket teszi lehetővé:

- *a BBS generátorhoz szükséges P, Q prímszámok generálása, illetve base64-es alakjuknak, állományba való, mentése*
- *a BBS generátorhoz szükséges P, Q prímszámok állományból való kiolvasása*
- *tetszőleges hosszúságú számsorozat generálása, BBS módszerrel*

A BBS generátor biztonsága, akkor megfelelő, ha N legalább 1024 bites. A hatékonyság növelése érdekében, ha $N = 1024$, akkor egyszerre 8 bitet, azaz 1 bájtot is lekérhetünk, a `u % 256` vagy `u & 255` műveletet alkalmazva:

```
def bbs256(P, Q, l):
    N = P * Q
    r = randint(2, N)
    u = (r * r) % N
    for i in range(0, l):
        u = (u * u) % N
        print(u & 255, end = ' ')
```

A Blum-Blum-Shub generátor

```
def primeRead(nev):
    f = open(nev, 'rt')
    temp = f.read()
    f.close()
    b64P, b64Q = temp.split('\n')
    P = nrFromBase256(b64decode(b64P.encode()))
    Q = nrFromBase256(b64decode(b64Q.encode()))
    return P, Q
```

```
def primeWrite(k, nev):
    P = safePrime(k//2)
    Q = safePrime(k//2)
    b64P = b64encode(bytes(nrToBase256(P)))
    b64Q = b64encode(bytes(nrToBase256(Q)))
    f = open(nev, 'wb')
    f.write(b64P + b'\n' + b64Q)
    f.close()
    print('P: ', P, '\nQ: ', Q)
    return P, Q
```

```
def nrFromBase256(L):
    nr = 0
    for elem in L:
        nr = (nr << 8) + elem
    return nr

def nrToBase256(nr):
    L = []
    while nr > 0:
        L = [nr & 255] + L
        nr = nr >> 8
    return L
```

A Blum-Blum-Shub generátor

A `primeRead1`, illetve `primeWrite1` függvényekben a 256-os számrendszerbe való átalakításhoz könyvtárfüggvényeket/metódusokat használtunk, pirossal ki vannak emelve ezek a műveletsorok. Az előző oldalon megadott változatok helyett ezeket is lehet használni.

```
def primeRead1(nev):
    f = open(nev, 'rt')
    temp = f.read()
    f.close()
    b64P, b64Q = temp.split('\n')
    L = b64decode(b64P.encode())
    P = int.from_bytes(L, byteorder = 'big')
    L = b64decode(b64Q.encode())
    Q = int.from_bytes(L, byteorder = 'big')
    return P, Q

def primeWrite1(k, nev):
    P = safePrime(k//2)
    Q = safePrime(k//2)
    bl = (k // 2) // 8 + 1
    b64P = b64encode(P.to_bytes(bl, byteorder = 'big'))
    b64Q = b64encode(Q.to_bytes(bl, byteorder = 'big'))
    f = open(nev, 'wb')
    f.write(b64P + b'\n' + b64Q)
    f.close()
    print('P: ', P, '\nQ: ', Q)
    return P, Q
```

A Blum-Blum-Shub generátor

```
from base64 import b64decode, b64encode
def mainBBS():
   P, Q = None, None
   while True:
      x = input('''
            0- Kilépés
            1- prímszám generálás/fileba írás
            2- prímszám beolvasás
            3- BBS generátor
            ''')
      if x == '0': break
      if x == '1':
         nev = input('az állomány neve: ')
         k = int(input('a primek bitmérete: '))
         P, Q = primeWrite(k, nev)
      if x == '2':
         nev = input ('az állomány neve: ')
         P, Q = primeRead(nev)
      if x == '3':
         if P == None or Q == None:
            print('nincs prim')
            continue
         print('P: ', P, '\nQ: ', Q)
         l = int(input('a számsorozat hossza: '))
         print('a veletlen számsorozat: ')
         bbs256(P, Q, l)
```