Diszkrét matematika
8. előadás

MÁRTON Gyöngyvér
mgyongyi@ms.sapientia.ro

Sapientia Egyetem,
Matematika-Informatika Tanszék
Marosvásárhely, Románia

2022, őszi félév

Miről volt szó az elmúlt előadáson?

- Python: , az `index` metódus, az `str, bytes` típusok, átalakítások, a `random` modul, bináris állományok,
- kódolási technikák: ASCII, UTF-8, Unicode,
- algoritmusok: a Hamming súly, titkosítás (az XOR egy alkalmazása),

Miről lesz szó?

- kódolási technikák: base64
- titkosítás és base64 (az XOR egy alkalmazása mégegyszer)
- számelmélet:
  - prímszámok
  - prímtesztelés: osztási próba / brute force,
  - prímszámok listája: Eratosztenész szitája,
  - a számelmélet alaptétele,
  - prímfaktorizáció: osztási próba / brute force

A base64 kódolás

- base64 kódolás egy tetszőleges bájtsorozatot alakít át olyan bájtsorozattá, amelyben csak a következő 64 fajta szimbólumnak megfelelő bájtérték szerepel:

`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=`

- az algoritmus a bementi bájtsorozat minden 3 bájtjából 4 bájtot hoz létre
- = karakternek speciális szerepe van, a kódolt adatsor végére kerül és azt jelzi, hogy a bemeneti bájtsorozat 3-al való osztási maradéka 1 vagy 2.
- Pythonban a base64 könyvtárcsomagot kell importálni: `b64encode` lesz a kódoló függvény, `b64decode` lesz a dekódoló függvény
- a `b64encode, b64decode` függvények bemenetként `bytes` típusú értéket várnak és a kimenetük is `bytes` típusú érték lesz

```
>>> b64encode('Saientia'.encode())
    b'U2FpZW50aWE='

>>> type(b64encode('Saientia'.encode()))
    <class 'bytes'>

>>> b64encode('Saientia')
    ...TypeError: a bytes-like object is required, not 'str'
```

A base64 kódolás

Python példák:

```
>>> from base64 import b64encode, b64decode

>>> b64encode('Sapientia'.encode())
    b'U2FwaWVudGlh'
>>> b64encode('Sap'.encode())
    b'U2Fw'
>>> b64encode('Sapi'.encode())
    b'U2FwaQ=='
>>> b64encode('Sapie'.encode())
    b'U2FwaWU='
>>> b64encode('Sapien'.encode())
    b'U2FwaWVu'
>>> b64encode('Sapientia'.encode()).decode()
    'U2FwaWVudGlh'

>>> b64decode(b'U2FwaWVudGlh')
    b'Sapientia'
>>> b64decode(b'U2FwaWVudGlh').decode()
    'Sapientia'
```

A Sapi szó base64 kódolása

szöveg	S	a	p	i		
karakter kód	83	97	112	105	0	0
bitminta	01010011	01100001	01110000	01101001	00000000	00000000
bitminta	010100	110110	000101	110000	011010	010000	000000	000000
kódindex	20	54	5	48	26	16	0	0
base64 kód	U	2	F	w	a	Q	=	=

```
>>> ABC64='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/='
>>> print(ord('S'), ord('a'), ord('p'), ord('i'))
    83 97 112 105
>>> print(ABC64[20], ABC64[54], ABC64[5], ABC64[48])
    U 2 F w
>>> print(ABC64[26], ABC64[16])
    a Q
```

A Sapientia szó base64 kódolása

A 'Sapientia' szó lépésenkénti base64 kódolását a következő Python kód végrehajtásával tudjuk nyomon követni:

```
def nyomonkovetesBase64(myStr):
    lStr = len(myStr)
    for i in range(0, lStr):
        codedStr = b64encode(myStr[:i+1].encode())
        print('%10s\t%s' % (myStr[:i+1], codedStr.decode()))

>>> nyomonkovetesBase64('Sapientia')
             S        Uw==
            Sa        U2E=
           Sap        U2Fw
          Sapi        U2FwaQ==
          ...
```

A base64 kódolás

1. feladat

Az alábbi Python függvény meghatározza egy bytes típusú bemenet base64-es alakját.

```
ABC64='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/='
def myB64Encode(mB):
    E = b''
    n = len(mB)
    for i in range (0, n, 3):
        b = (mB[i] & 0xFC) >> 2
        E += ABC64[b].encode()
        b = (mB[i] & 0x03) << 4
        if i + 1 < n:
            b = b | ( (mB[i + 1] & 0xF0) >> 4 )
            E += ABC64[b].encode()
            b = ( mB[i + 1] & 0x0F) << 2
            if i + 2 < n:
                b = b | ( ( mB[i + 2] & 0xC0) >> 6 )
                E += ABC64[b].encode()
                b = mB[i + 2] & 0x3F
                E += ABC64[b].encode()
            else:
                E += ABC64[b].encode()
                E += b'='
        else:
            E += ABC64[b].encode()
            E += b'=='
    return E
```

A base64 kódolás

2. feladat

Az alábbi Python függvény meghatározza egy base64-es formában megadott bemeneti paraméter eredeti alakját.

```
def myB64Decode(eB):
    D = bytes([])
    n = len(eB)
    for i in range(0, n, 4):
        b0 = ABC64.index(chr(eB[i]))
        b1 = ABC64.index(chr(eB[i + 1]))
        b2 = ABC64.index(chr(eB[i + 2]))
        b3 = ABC64.index(chr(eB[i + 3]))
        temp = ((b0 << 2) | (b1 >> 4)) & 255
        D += bytes([temp])
        if b2 < 64:
            temp = ((b1 << 4) | (b2 >> 2)) & 255
            D += bytes([temp])
            if b3 < 64:
                temp = ((b2 << 6) | b3) & 255
                D += bytes([temp])
    return D

>>> myB64Encode('műszaki'.encode())
    b'bcWxc3pha2k='
>>> myB64Decode(b'bcWxc3pha2k=').decode()
    'műszaki'
```

Az XOR egy alkalmazása: titkosítás, base64 kódolás

Írjunk egy Python programot, amely menüpontok segítségével a következőket végzi:

- véletlenszerűen generál egy `k` bájtos kulcsot és kiírja egy bináris állományba a kulcs base64-es alakját,
- beolvassa a megfelelő kulcs értékét egy bináris állományból, és meghatározza egy billentyűzetről beolvasott karakterlánc rejtjelezett értékét, amelynek base64-es alakját kiírja egy bináris állományba,
- beolvassa a megfelelő kulcs értékét és a rejtjelezett szöveg értékét egy-egy bináris állományból, és visszafejti a rejtjelezett szöveget,
- a titkosítást és a visszafejtést az `XOR` művelettel végzi, ahogyan korábban is leírtuk.

A szükséges importok:

```
from base64 import b64encode, b64decode
import random
```

Az XOR egy alkalmazása: titkosítás, base64 kódolás

```
def mainCrypt():
    while True:
        x = input('''
            0- kilépés
            1- titkosítás
            2- visszafejtés
            3- kulcs generálás/mentés
            ''')
        if x == '0': break
        if x == '1':
            bKey = keyRead()
            if bKey != -1:
                myStr = input('titkositasra szant szoveg: ')
                cStr = cryptFunc(myStr.encode(), bKey)
                cryptWrite(cStr)
        if x == '2':
            bKey = keyRead()
            if bKey != -1:
                cStr = cryptRead()
                if cStr != -1:
                    try:
                        myStr = cryptFunc(cStr, bKey)
                        print('visszafejtett szoveg: ')
                        print(myStr.decode())
                    except:
                        print('nem lehet visszafejteni!')
        if x == '3':
            k = int(input('a kulcs bajt merete: '))
            keyWrite(k)
```

Az XOR egy alkalmazása: titkosítás, base64 kódolás

```
def keyGen(k):
    key = bytes([])
    for i in range(k):
        key += bytes([random.randint(0,255)])
    return key

def keyWrite(k):
    bKey = keyGen(k)
    temp = b64encode(bKey)
    keyF = input('kulcsallomany neve: ')
    outfK = open(keyF, 'wb')
    outfK.write(temp)
    outfK.close()

def keyRead():
    try:
        keyF = input('kulcsallomany neve: ')
        infK = open(keyF, 'rb')
        temp = infK.read()
        infK.close()
        bKey = b64decode(temp)
        return bKey
    except:
        print('Allomany megnyitasi problema!')
        return -1
```

Az XOR egy alkalmazása: titkosítás, base64 kódolás

```
def cryptFunc(bStr, bKey):
    C, i, n = bytes([]), 0, len(bKey)
    for elem in bStr:
        C += bytes([elem ^ bKey[i]])
        if i ==  n - 1: i = -1
        i += 1
    return C

>>> cryptFunc('my first plaintext for Crypt, 2022'.encode(), 'myKey2022'.encode())
    b'\x00\x00k\x03\x10@CF\x12\x1d\x15*\x0c\x17FUJFM\x1f$\x17YqBKB\x19UkWI\x00\x02'
>>> b64encode(cryptFunc('my first plaintext for Crypt, 2022'.encode(), 'myKey2022'.encode()))
    b'AABrAxBAQOYSHRUqDBdGVUpGTR8kF1lxQktCGVVrV0kAAg=='

def cryptWrite(cStr):
    cryptF = input('titkositott szoveg, allomanynev: ')
    outfC = open(cryptF, 'wb')
    outfC.write(b64encode(cStr))
    outfC.close()

def cryptRead():
    try:
        cryptF = input('titkositott szoveg, allomanynev:')
        infC = open(cryptF, 'rb')
        cStr = b64decode(infC.read())
        infC.close()
        return cStr
    except:
        print('Allomany megnyitasi problema!')
        return -1
```

Prímszámok

- Prímszámok, azok az 1-nél nagyobb egész számok, amelyek nem oszthatóak, csak 1-el és önmagukkal: 2, 3, 5, 7, ...
- Összetett számok, azok az 1-nél nagyobb egész számok, amelyek nem prímszámok: 4, 6, 8, 9, 10, ...
- az 1 sem nem prím se nem összetett.

1. tétel

Minden 1-nél nagyobb pozitív egész számnak van egy prímosztója.

- A bizonyítás indirekt bizonyítási módszerrel történik: tegyük fel az ellenkezőjét, azaz létezik egy olyan szám amelyiknek nincs prímosztója; ezek a számok meghatároznak egy nem üres halmazt.
- A természetes számok jólrendzettség tulajdonsága alapján ebben a halmazban van egy legkisebb ilyen elem, legyen ez n.
- Mivel n-nek nincs primosztója, de n osztja n-t, következik, hogy n nem prímszám. Azaz felírható: $n = a \cdot b$, ahol $1 < a < n, 1 < b < n$. Mivel $a < n$ következik, hogy a-nak van primosztója.
- De a bármely osztója osztja n-t, ez pedig ellentmondás.

Prímszámok

2. tétel

Végtelen sok prímszám létezik.

- Egyik bizonyítási módszer Eukleidész, Elemek című könyvében található: feltételezzük, hogy a prímszámok halmaza véges.
- Jelöljük ennek a halmaznak az elemeit $p_1, p_2, \ldots, p_n$-nel. Jelöljük $Q = p_1 \cdot p_2 \ldots p_n + 1$. Bebizonyítható, hogy Q-nak viszont van egy olyan prímosztója amelyik nem szerepel a fenti halmazban.
- Feltételezzük az ellenkezőjét: azaz legyen q, Q egy osztója, és feltételezzük, hogy ez megegyezik valamelyik p_j-vel, ahol $j = 1, \ldots, n$ (az 1. tétel alapján ezt feltételezhetjük).
- Ez azonban azt jelenti hogy q osztja $Q - p_1 \cdot p_2 \ldots p_n = 1$, azaz 1-t. Ez ellentmondás, mert egy prímszám nem oszthatja az 1-et.
- Pl. Ha összeszorozzuk az első 6 prímszámot, akkor kapunk egy olyan számot, amelynek biztosan lesz egy új prímszám osztója:

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30031 = 59 \cdot 509$$

- Hendrik Lenstra: Végtelen sok összetett szám létezik. Ahhoz, hogy egy új összetett számot kapjunk szorozzuk össze az első n összetett számot és ne adjunk hozzá 1-et. 

Prímszámok

3. tétel

Ha n egy összetett szám, akkor n-nek van egy olyan prímosztója, amelyik kisebb vagy egyenlő mint $\sqrt{n}$.

- Bizonyítása a 1. tétel alapján történik.
  - Ha n összetett, akkor $n = a \cdot b$, ahol $1 < a \leq b < n$.
  - Fenn kell álljon, hogy $a \leq \sqrt{n}$, mert másképp $\sqrt{n} < a \leq b \Rightarrow n = \sqrt{n} \cdot \sqrt{n} < a \cdot b$.
  - Az 1. tétel alapján a-nak van egy prímosztója, amelyik n-nek is prímosztója, amelyik tehát $\leq \sqrt{n}$.
- A tétel alapján algoritmusok adhatóak meg a prímszámok vizsgálatára:
  - osztási próba módszere (trial division): egy adott számról vizsgáljuk, hogy prím-e,
  - Eratosztenész szitája: meghatározza egy adott n-ig az összes prímszámot,
  - hatékonyabb algoritmusok: Miller-Rabin prímteszt, Solovay-Strassen prímteszt, AKS prímteszt, stb.

Prímtesztelő algoritmusok

Rossz hatékonyságú algoritmus: meghatározzuk egy szám osztóinak számát, ha az egyenlő kettővel akkor prímszám, másképp nem prímszám.

Osztási próba módszere, brute-force (trial division): sorra próbáljuk a páratlan számokkal való oszthatóságot. Az első osztónál leáll az algoritmus, azzal a kimenettel, hogy a szám nem prímszám. Ha a tesztelő szám négyzetgyökéig nem találunk osztót, akkor a szám prímszám. 10^6-ig ezt az algoritmust használják

Eratosztenész szitája: kizárásos módszerrel adott n-ig meghatározzuk a prímszámok listáját

Miller-Rabin prímteszt: egy adott páratlan számról biztosan megállapítja hogy az összetett, az azonban csak nagyon nagy valószínűséggel állítja, hogy a szám prím. Egy későbbi előadásban kerül bemutatásra.

A gyakorlatban nagyon fontos feladat eldönteni egy nagy (több mint 300 számjegyű) számról hogy prímszám-e vagy sem.

Prímtesztelés: osztási próba / brute force

Vizsgáljuk meg az osztási próba módszerével, hogy 101 prímszám-e:

- a cél: minél kevesebb osztást végezzünk az oszthatóság eldöntéséhez,
- milyen számokkal vizsgáljuk az oszthatóságot? $\rightarrow$ csak a páratlan számokkal vizsgáljuk az oszthatóságot, mert a 2-nél nagyobb prímszámoknak nem lehet páros osztójuk,
- 101 nem osztható $3, 5, 7, 9 \Rightarrow$ 101 prímszám,
- a 11-el már nem kell megvizsgálni az oszthatóságot, mert $11 \cdot 11 = 121 > 101$, vagy $\lfloor \sqrt{101} \rfloor < 11$.

Prímtesztelés - osztási próba / brute force

3. feladat

Írjunk Python függvényt, amely megvizsgálja az osztási próba módszerével, hogy n prímszám-e.

```
def tDPrimeTest1(nr):
    if nr == 2: return True
    if nr & 1 == 0: return False
    i = 3
    while i * i <= nr:
        if nr % i == 0:
            return False
        i += 2
    return True

>>> tDPrimeTest1(304107229823269)
    True
```

```
def tDPrimeTest2(nr):
    i = 3
    while i*i <= nr:
        if nr % i == 0:
            return False
        i += 2
    return True

>>> tDPrimeTest2(19267)
    True
>>> tDPrimeTest2(19269)
    False
```

Ha bemenetnek csak páratlan számot adhatunk, akkor a `tDPrimeTest2` függvénnyel dolgozunk.

Prímtesztelés - osztási próba / brute force

Az osztási próba módszerét optimalizálhatjuk, ha a 3-al való oszthatóságot is külön teszteljük. Ekkor a 3-nál nagyobb prímszámok csak $6 \cdot i + 5$, vagy $6 \cdot i + 1$ alakúak lehetnek. Az oszthatóságot tehát csak a következő számokkal kell vizsgálni:

- $5, 11, 17, 23, 29, 35, 41, \ldots$
- $7, 13, 19, 25, 31, 37, 43, 49, \ldots$

A módosított a `tDPrimeTest` függvény a következő:

```
def tDPrimeTest(nr):
#bemeneti paraméterként egy páratlan számot kell megadni
    if nr == 3: return True
    if nr % 3 == 0: return False
    i = 5
    while i * i <= nr:
        if nr % i == 0:
            return False
        if nr % (i + 2) == 0:
            return False
        i += 6
    return True
```

Prímtesztelés - osztási próba / brute force

4. feladat

Írjunk egy Python függvényt, amely a `tDPrimeTest` függvényt alkalmazva véletlenszerűen generál egy `k`-nál kisebb páratlan prímszámot.

```
import random
def primeGen(k):
    nr = random.randint(0, k)
    if nr & 1 == 0: nr += 1
    while True:
        if tDPrimeTest(nr): return nr
        nr += 2

>>> primeGen(10**15)
    ...
```

- Ha a generált `nr` szám páros, akkor a `nr` értékét egyel növeljük, mert a `tDPrimeTest` páratlan számot vár bemenetként.
- A `while` keretén belül addig növeljük kettesével a `nr` értékét, amíg egy prímszámra nem találunk.
- Az algoritmus időigénye hogyan nő, aszerint hogy a bemenet értékét növeljük.

Prímszámok listája - Eratosztenész szitája

Eratosztenész szitájával határozzuk meg 100-ig a prímszámokat: A 3-al kezdődő páratlan számokat tartalmazó listában az első szám a 3-as prímszám, az összes többszöröse összetett szám lesz, ezen sorszámú elemek értékét az algoritmus szerint `False`-ra fogjuk állítani. A következő táblázatban ezeket az értékeket áthúztuk. 3 első többszöröse, ami a listában szerepel az a $3 \cdot 3 = 9$-es érték:

3	5	7	~~9~~	11	13	~~15~~	17	19	~~21~~	23	25	~~27~~
29	31	~~33~~	35	37	~~39~~	41	43	~~45~~	47	49	~~51~~	53
55	~~57~~	59	61	~~63~~	65	67	~~69~~	71	73	~~75~~	77	79
~~81~~	83	85	~~87~~	89	91	~~93~~	95	97	~~99~~			

A maradék 5-el kezdődő számokat tartalmazó listában az 5 első többszöröse, ami a listában szerepel az a $5 \cdot 5 = 25$-ös érték:

5	7	11	13	17	19	23	~~25~~	29	31	~~35~~	37	41
43	47	49	53	~~55~~	59	61	~~65~~	67	71	73	77	79
83	~~85~~	89	91	~~95~~	97							

Prímszámok listája - Eratosztenész szitája

A maradék 7-el kezdődő számokat tartalmazó listában 7 első többszöröse, ami a listában szerepel az a $7 \cdot 7 = 49$-es érték.

7	11	13	17	19	23	29	31	37	41	43	47	~~49~~
53	59	61	67	71	73	~~77~~	79	83	89	~~91~~	97	

A maradék 11-el kezdődő számokat tartalmazó listában a megmaradt páratlan sorszámú elemek prímszámok lesznek, mert 11 első többszöröse, ami a listában szerepel az a $11 \cdot 11 = 121$ nagyobb mint 100:

11	13	17	19	23	29	31	37	41	43	47	53	59
61	67	71	73	79	83	89	97					

100-ig a prímszámok, tehát:

2	3	5	7	11	13	17	19	23	29	31	37	41
43	47	53	59	61	67	71	73	79	83	89	97	

Prímszámok listája - Eratosztenész szitája

5. feladat

Írjunk egy Python függvényt, amely Eratosztenész szitájával kigenerálja n-ig a prímszámokat.

```
def eratL1(n):
    L = [True] * (n + 1)
    L[0] = L[1] = False
    i = 3
    while i * i <= n + 1:
        if L[i] == True:
            for j in range(i * i, n + 1, i): L[j] = False
        i += 2
    Prim = [2]
    for i in range(3, len(L), 2):
        if L[i]: Prim += [i]
    return Prim

>>> eratL1(50)
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> len(eratL1(1000000))
    78498
```

Prímszámok listája - Eratosztenész szitája

Az eratL1 függvényt is optimalizálhatjuk, ha a tesztelést csak $6 \cdot i + 5$, vagy $6 \cdot i + 1$ alakú számokkal végezzük:

```
def eratL(n):
    L = [True] * (n + 1)
    L[0] = L[1] = False
    i = 5
    while i * i <= n + 1:
        if L[i] == True:
            for j in range(i * i, n + 1, i): L[j] = False
        i += 2
        if L[i] == True:
            for j in range(i * i, n + 1, i): L[j] = False
        i += 4
    Prim = [2, 3]
    for i in range(5, len(L) - 2, 6):
        if L[i]: Prim += [i]
        if L[i + 2]: Prim += [i + 2]
    return Prim

>>> len(eratL(10000000))
    664579
```

A számelmélet alaptétele

4. tétel

Bármely 1-nél nagyobb, pozitív egész szám a szorzótényezők sorrendjétől eltekintve, egyértelműen felírható prímszámok szorzataként. A kapott szorzatot ***prímtényezős*** *vagy törzstényezős felbontásnak hívjuk.*

- egy *nr* szám prímtényezős felbontásában egy prímszám többször is előfordulhat,
- azt az alakot, amelyben minden prímszámot csak egyszer, a megfelelő hatványértéken írunk kanonikus alaknak hívjuk, jelölése:

$$nr = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_n^{a_n} = \prod_{i=1}^{n} p_i^{a_i}$$

- Példák:
  $2200 = 2 \cdot 2 \cdot 2 \cdot 5 \cdot 5 \cdot 11 = 2^3 \cdot 5^2 \cdot 11$
  $361 = 19 \cdot 19 = 19^2$
  $10127 = 13 \cdot 19 \cdot 41$
- a prímtényezős felbontás folyamatát prímfelbontásnak, prímfaktorizációnak, vagy csak egyszerűen faktorizációnak nevezzük,
- fontosak azok az algoritmusok, amelyek képesek hatékonyan megadni egy nagy szám prímtényezős felbontását,
- nagy számok esetében a prímtényezős felbontás meghatározására, ha a számításokat nem kvantumszámítógépen végezzük nem ismert hatékony eljárás.

Prímfaktorizációs módszerek

- az egyik legegyszerűbb prímfaktorizációs módszer a prímtesztelésnél bemutatott osztási próba módszeréhez hasonlít,
- ez is brute-force típusú algoritmus, és a szakirodalomban erre is trial division néven hivatkoznak,
- léteznek sokkal hatékonyabb prímfaktorizációs algoritmusok, egy későbbi előadásban szó lesz róluk,
- a következő oldalon megadott `tDPrimeFact` függvényben:
  - az első `while` ciklusban megszámoljuk hogy hányszor osztható a szám 2-vel, ha osztható, akkor el is végezzük az osztást,
  - következő `while` keretén belül a páratlan számokkal való oszthatóságot vizsgáljuk, ha lehetséges akkor elvégezzük a megfelelő számmal való osztást,
  - az eredményt egy tuple elemtípusú listában határozzuk meg: az elem első értéke a prímszámot, a második érték a hatványkitevő értéket jelöli,
  - a futási idő meghatározásához a `time` függvényt alkalmaztuk: a számítási sorozat előtt, majd utána eltároltuk a számítógép által mutatott időértéket, a mért idők közti különbség mutatja a futási időt, amelyet kiíratunk a képernyőre.

Prímfaktorizáció - osztási próba / brute force

6. feladat

Írjunk egy Python függvényt, amely meghatározza az osztási próba módszerével egy szám prímtényezős felbontását.

```
def tDPrimeFact1(nr):
    Prim, db = [], 0
    while nr & 1 == 0:
        nr, db = nr >> 1, db + 1
    if db != 0: Prim += [(2, db)]
    x = 3
    while x * x <= nr:
        db = 0
        while nr % x ==0:
            nr, db = nr // x, db + 1
        if db != 0: Prim += [(x, db)]
        x += 2
    if  nr > 2: Prim += [(nr, 1)]
    return Prim

>>> tDPrimeFact1(14694400978298424544)
    [(2, 5), (7, 3), (41, 3), (1789, 1), (10857901, 1)]
```

Prímfaktorizáció - osztási próba / brute force

Az tDPrimeFact1 függvényt is optimalizálhatjuk, ha a tesztelést csak $6 \cdot i + 5$, vagy $6 \cdot i + 1$ alakú számokkal végezzük:

```
def tDPrimeFact(nr):
    Prim, db = [], 0
    nr, Prim = szamolOszthatosag(2, nr, Prim)
    nr, Prim = szamolOszthatosag(3, nr, Prim)
    x = 5
    while x * x <= nr:
        nr, Prim = szamolOszthatosag(x, nr, Prim)
        nr, Prim = szamolOszthatosag(x + 2, nr, Prim)
        x += 6
    if  nr > 4: Prim += [(nr, 1)]
    return Prim

def szamolOszthatosag(x, nr, Prim):
    db = 0
    while nr % x == 0:
        nr = nr // x
        db += 1
    if db != 0: Prim += [(x, db)]
    return nr, Prim
```

Prímfaktorizáció - osztási próba / brute force

7. feladat

Írjunk egy Python függvényt, amely különböző páratlan számok esetében meghatározza a szám prímtényezős felbontását és a futási időt is leméri:

```
from time import time
def idomeres():
    st = time()
    print(tDPrimeFact(833005931221827589280560136312 32))
    fs = time()
    print('idoigeny: ', fs - st)

    st = time()
    print(tDPrimeFact(10838496441610139787933089695749 83))
    fs = time()
    print('idoigeny: ', fs - st)

    st = time()
    print(tDPrimeFact(19971526367099 4809))
    fs = time()
    print('idoigeny: ', fs - st)
```

Prímfaktorizáció - osztási próba / brute force

```
>>> idomeres()
    [(2, 8), (101, 7), (1789, 3), (530063, 1)]
    idoigeny:  0.050162553787231445
    [(17, 8), (83, 7), (1789, 3)]
    idoigeny:  0.010589599609375
    [(206864963, 1), (965437843, 1)]
    idoigeny:  19.388545989990234
```

- a legrosszabb futási időt akkor kaptuk, amikor a legkisebb számot próbáltuk faktorizálni: a bemenet két nagyobb prímszám szorzatából áll,
- az első két meghívás esetében a prímtényezők legtöbbje kicsi szám,
- az igazi kihívást a prímfaktorizációs algoritmusok terén az jelenti, hogy nagy prímszámok szorzatából álló számnak határozuk meg a prímtényezőit.