MÁRTON GYÖNGYVÉR

SZEMELVÉNYEK A DISZKRÉT MATEMATIKA FEJEZETEIBŐL
PYTHON ALGORITMUSOK

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
MATEMATIKA–INFORMATIKA TANSZÉK

MÁRTON GYÖNGYVÉR

SZEMELVÉNYEK A DISZKRÉT MATEMATIKA FEJEZETEIBŐL

PYTHON ALGORITMUSOK

Scientia Kiadó
Kolozsvár · 2024

sapientia tankönyvek

informatika

Készült a Magyar Kormány támogatásával

Miniszterelnökség Nemzetpolitikai Államtitkárság

Bethlen Gábor Alap

Felelős kiadó:
dr. Sorbán Angella

Lektorok:
dr. Hannusch Carolin (Debrecen)

Borítóterv:
Tipotéka Kft.

Kiadói koordinátor:
Szabó Beáta

A szakmai felelősséget teljes mértékben a szerző vállalja.

Első magyar nyelvű kiadás: 2024

Descrierea CIP a Bibliotecii Naţionale a României
MÁRTON, GYÖNGYVÉR
Szemelvények a diszkrét matematika fejezeteiből : Python algoritmusok /
Márton Gyöngyvér. - Cluj-Napoca : Scientia, 2024
Conţine bibliografie
ISBN 978-606-975-087-2

004

TARTALOMJEGYZÉK

CUPRINS

CONTENTS

1. fejezet

Bevezető

Jelen egyetemi tankönyv célja, hogy azon alapismeretekkel, alapalgoritmusokkal ismertesse meg az olvasót, amelyekre elengedhetetlenül szükség van az informatika, számítástechnika világában. A tankönyv kifejezetten főiskolás diákoknak készült, és azokat a kérdésköröket ismerteti, amelyeket a hallgatóknak a Sapientia EMTE-n a diszkrét matematikához kapcsolódó elméleti és gyakorlati órákon el kell sajátítaniuk.

A jegyzet nem annyira elméleti, mint inkább gyakorlati szempontból vezeti be az olvasót a diszkrét matematika világába. Ez pontosabban azt jelenti, hogy a jegyzet a fontosabb elemi algoritmusok bemutatására koncentrál, és ezeket nem a matematika szemszögéből, hanem az informatika oldaláról közelíti meg. Legtöbb esetben a bemutatásra kerülő algoritmusoknak megadjuk a **Python** programozási nyelvben való implementációját. Éppen ezért a tankönyv kitér a Python programozási nyelv szintaktikájára is, de csak olyan mértékben, amennyire szükséges, hogy a tankönyvben tárgyalt algoritmusok programkódját meg tudjuk érteni, illetve hogy meg tudjuk oldani a fejezetek végén kitűzött feladatokat, ahol a kitűzött feladatokhoz szükséges bemeneti állományok a következő mappából tölthetők le:

https://ms.sapientia.ro/ mgyongyi/DiszkretMat/Jegyzet

A tankönyvnek nem célja, hogy megtanítsa a Python programozási nyelvet, a Pythont csupán mint segédeszközt használja a tantárgy keretén belül tárgyalt algoritmusok megértéséhez.

A választás azért esett a Python nyelvre, mert alapszinten könnyen megtanulható, szintakszisa nem rejti el az algoritmusok lényegét, szabadon

elérhető, könnyen telepíthető, számos digitális tananyag található hozzá, ugyanakkor számos valós fejlesztés Python nyelven történik.

A diszkrét matematika a számítástechnikai adatkezelés alapja, hiszen a digitális gépek számításai a `0` és `1` diszkrét értékek feldolgozásával valósulnak meg. Ezeknek az értékeknek a feldolgozási módja számos egyszerű és komplex algoritmus megjelenését eredményezte, amelyek megértéséhez, megírásához, továbbfejlesztéséhez elengedhetetlenül szükség van mind matematikai, mind informatikai ismeretekre. A diszkrét matematika számos szakterületet foglal magába, mint például gráfelmélet, halmazelmélet, diszkrét valószínűségszámítás, lineáris algebra stb. Jelen tankönyv azokra a területekre koncentrál, amelyeket a diákok nem tanulnak a Sapientia EMTE-n külön tantárgy keretén belül.

A tankönyv első fejezete ismerteti azokat a Python programozási nyelvi elemeket, amelyek szükségesek a bemutatásra kerülő algoritmusok megírásához. Elsősorban a procedurális programozás alapjaira térünk ki. A további fejezetekben csak akkor kerülnek bevezetésre az újabb és újabb Python programozási struktúrák, ha azok szükségesek. Ha adott helyen olyan fogalmat, kijelentést használunk, amelyet korábban adtunk meg, akkor a jegyzet végén található tárgymutató segítségével kereshetünk rá arra a részre, ahol a keresett kifejezés, nyelvi elem, értelmezés található. Hasonlóképpen járunk el a diszkrét matematikához kapcsolódó algoritmusok tárgyalásakor, csak akkor és annyi matematikai ismeret kerül bevezetésre, amennyi szükséges az algoritmusok megértéséhez, implementálásához. A bemutatásra kerülő tételek nem mindegyikét bizonyítjuk, ezeknek utána lehet nézni a szakirodalomban feltüntetett könyvekben.

A Python alapszintű bemutatása után kerül sor a kombinatorikai, majd a különböző számhalmazokhoz kapcsolódó alapalgoritmusok ismertetésére. A következő fejezet a számrendszereket és azok alkalmazásait tárgyalja, utána pedig kódolási technikákra kerül sor. Az utolsó fejezetben számelmélethez és kriptográfiához kapcsolódó matematikai tételek, algoritmusok kerülnek bemutatásra. A tankönyvben bemutatásra kerülő Python kódoknak megadjuk fejezetenként a forráskódját. Felhívjuk itt a figyelmet arra, hogy a jegyzetben található sorrendhez képest fordított sorrendbe írtuk ezekbe az állományokba a forráskódokat. Ezek letölthetők a következő linkről:

https://ms.sapientia.ro/ mgyongyi/DiszkretMat/Jegyzet/Jegyzet.zip

A tankönyv az elmúlt öt év előadásjegyzetei alapján készült. Bármilyen észrevételt, hibajelzést, kérdést az `mgyongyi@ms.sapientia.ro` e-mail-címen lehet jelezni.

2. fejezet

Python-alapfogalmak

2.1. Az első lépések

A Python 0.9.0 programozási nyelv 1991-ben jelent meg Guido von Rossum fejlesztésével, egy korábbi programozási nyelv továbbfejlesztett változataként. A Python 3.0 2008-ban jelent meg, amely a korábbi Python 2.0 verziókhoz képest számos új funkciót vezetett be. Fontos tudni, hogy a Python 3 verziók nem teljesen kompatibilisek a korábbi Python 2 verziókkal. A jegyzet keretén belül a **Python 3** verzióban írjuk és mutatjuk be az algoritmusokat, a programozási nyelvre való hivatkozáskor pedig csak Pythont fogunk írni.

A Python tömör és könnyen olvasható programok írását teszi lehetővé. Ellentétben a fordító (kompilátor) típusú programozási nyelvekkel értelmezője a saját utasításait bemenő adatként kezeli, ezeket átalakítja a futtató gép utasításává, majd rögtön kiértékeli őket. Lehetővé teszi a nagy programok kisebb részekre, modulokra való felosztását, amelyek aztán más Python-projektekben is felhasználhatók. Mások által megírt modulok is könnyen hozzácsatolhatók Python-alkalmazásainkhoz. A nyelvhez tartozó számos alapkönyvtár, modul biztosítja a komplexebb műveletek, a matematikai algoritmusok egyszerű implementációját [13, 38].

A Python telepítése egyszerű feladat, a szükséges állományt a Python hivatalos weboldaláról lehet letölteni:

https://www.python.org/downloads/

Itt a Python 3 verziók közül az alapján választjuk ki az állományt, amelyet szeretnénk telepíteni, hogy milyen operációs rendszer fut a számítógépünkön. A telepítés során a Python grafikus felhasználói felülete, az IDLE is felkerül a számítógépünkre, amelyet, ha megnyitunk, akkor egy kétsoros standard szöveggel (Python verziószám stb.) egy interaktív ablak, a **shell** jelenik meg a képernyőn és abban a parancsjel (prompt): **>>>**. A prompt után kifejezések, utasítások írhatók, amelyeket a Python értelmezője (interpretere) rögtön kiértékel, ezért akár asztali számológépnek is használható a nyelv.

Próbáljuk ki a következőket, azaz írjuk be a prompt után az alábbi utasításokat, ahol az utasítások után feltüntettük a kiértékelések eredményét is:

```
>>> print('Hello Sapientia!')
    Hello Sapientia!
>>> 251 + 965
    1216
>>> "Diszkrét" + " matematika"
    'Diszkrét matematika'
>>> 3 * 11
    33
>>> 3 * 'Diszkrét '
    'Diszkrét Diszkrét Diszkrét'
>>> 50 % 13, 16 // 13, 16 / 13
    (11, 1, 1.2307692307692308)
```

Az első műveletsorban a `print` függvény kiértékeli a zárójelben megadott kifejezést, és az eredményt kiírja a standard kimenetre. A `print`-nek megadott kifejezést, jelen esetben a `'Hello Sapientia!'` karakterláncot a függvény bemenetének vagy bemeneti paraméterének hívjuk. A karakterláncokat idézőjelbe (`'`), vagy macskaköröm (`"`) közé kell írni, ahol a két szimbólum közül bármelyiket lehet használni.

A következő két műveletsorban a `+` operátor került alkalmazásra. Az első esetben a művelet kiértékelésének eredménye természetesen a két szám összege lesz, a második kiértékelés eredménye a bemeneti két karakterlánc összefűzött értéke lesz. Vegyük észre, hogy ugyanaz az operátor különböző fajta, azaz különböző típusú adatokon is alkalmazható, erre azt mondjuk, hogy a `+` operátor felüldefiniált (overloaded).

A következő két műveletsorban, hasonlóan a `+` operátorhoz, kétfajta adaton alkalmaztuk az `*` operátort. Első használatakor két szám szorzata, másodszor pedig karakterláncok összefűzése az eredmény.

A Pythonban három osztással kapcsolatos operátor is használható, az osztási maradékot a `%` operátorral, az osztási egészrészt a `//` operátorral, míg a valós osztás eredményét a `/` operátorral tudjuk meghatározni. Figyeljük meg, hogy a három műveletet egy sorba, vesszőkkel elválasztva adtuk meg, az eredmény pedig egy `tuple` típusú adatszerkezet lesz, amely jelen esetben a megfelelő osztási eredményekhez tartozó három értéket fogja tárolni. Egyelőre azt figyeljük meg, hogy az eredményt kerek zárójelbe teszi a Python, a `tuple` típusra pedig hamarosan visszatérünk.

A Python tetszőlegesen nagy egész számokkal is tud műveleteket végezni, például a 2^{100} meghatározásához nem kell nagy számokat kezelő könyvtárcsomagot telepíteni, a `**` operátort vagy a `pow` függvényt lehet használni:

```
>>> 2**100
    1267650600228229401496703205376
>>> pow(2, 100)
    1267650600228229401496703205376
```

A fenti kódsorban a `pow` függvénynek két bemeneti paramétere van, az első az alap, a második a kitevő értékét jelöli. A `pow` függvény által meghatározott hatványértékről azt mondjuk, hogy ez a függvény visszatérési értéke vagy a függvény kimenete. Pythonban a függvényeknek nem kell visszatérési értékkel rendelkezniük, ekkor alapértelmezetten `None` lesz a visszatérési értékük. A korábban használt `print` függvénynek is `None` a visszatérési értéke. Ezt úgy tudjuk ellenőrizni, ha az értékadó utasítást alkalmazva eltároljuk a visszatérési értéket, majd az eltárolt értéket kiíratjuk. Értékadó utasítás az egyenlőség (**=**) jelet használva adható meg:

```
>>> r = pow(2, 100)
>>> print(r)
    1267650600228229401496703205376
>>> r = print('helo')
    helo
>>> print(r)
    None
```

Valamely szám négyzetgyökének a meghatározásához, azaz valós kitevő esetében is használható a `**` operátor, de a `pow` függvény is. Aszerint, hogy egész vagy valós típusú kitevőértéket adunk meg, az eredmény is egész vagy valós szám lesz.

```
>>> 2**0.5
    1.4142135623730951
>>> pow(2, 100.0)
    1.2676506002282294e+30
```

A négyzetgyök értékét ki tudjuk számolni úgy is, hogy a `math` modul `sqrt` függvényét alkalmazzuk:

```
>>> import math
>>> math.sqrt(2)
    1.4142135623730951
```

A következő kódsorban adott osztási maradék, azaz modulus szerinti hatványértéket határozunk meg, kétféleképpen is. Figyeljük meg, hogy az alkalmazásra kerülő `pow` függvény most nem két, hanem három bemeneti paraméterrel kerül meghívásra, a harmadik a modulus értékét jelöli. Mindez azért lehetséges, mert a Python könnyedén kezeli a változó számú paraméterrel rendelkező függvényeket.

```
>>> (17**5) % 11
    10
>>> pow(17, 5, 11)
    10
```

A prompt után beírt korábbi műveletek újrahívhatók, anélkül, hogy újra be kellene őket írni. Az `Alt+N`, illetve `Alt+P` billentyűkombinációkkal korábbi műveletek érhetőek el, az első *előre*, a második *hátrefele* sorrendben teszi elérhetővé a beírt műveleteket.

2.2. Változók és adattípusok

Programírás során az adatok tárolása, módosítása elemi programozói feladat. A következő kódrészletben az első két utasítás az egyenlőség operátor segítségével értékadást végez. Hatásukra `x` a 251-es, `y` a 965-ös értéket fogja jelölni. Az utolsó utasításban először a zárójelben megadott kifejezés kerül kiértékelésre, amelynek eredményét a `print` kiírja a standard kimenetre:

```
>>> x = 251
>>> y = 965
>>> print(x + y)
    1216
```

A következő kódrészlet első sora két értékadó műveletet hajt végre. A második sorban a `print` segítségével három aritmetikai művelet eredményét írjuk ki:

```
>>> a, b = 17, 5
>>> print(a / b, a // b , a % b)
    3.4 3 2
```

A fenti példákban az `x, y, a` és `b` szimbólumokat **objektumreferenciáknak** hívjuk. Más programozási nyelvekben ezeket változóknak mondják, de a Python nem rendelkezik a hagyományos értelemben vett változókkal. Hogy a változó, az objektumreferencia vagy az objektum pontosan mit jelent, az programozási nyelvektől függően változik. A tankönyvben tárgyalásra kerülő algoritmusok esetében ennek nincs különösebb jelentősége, ezért az egyszerűség kedvéért a továbbiakban a változó megnevezést fogjuk használni.

A változókkal végzett műveletsorok képezik a programírás alapjait. Hogy milyen szimbólumot, szimbólumsorozatot, azaz milyen nevet választunk egy változónak, azt mi magunk döntjük el. A változók tehát egy adott értéket jelölnek, amelyeket értékadó, értékmódosító utasításokkal határozunk meg, és mindig az utolsónak megadott vagy kiszámított érték lesz a változó értéke:

```
>>> x = 51
>>> x = 96
>>> print(x)
    96
```

A változónak mindig van egy **típusa**, ami meghatározza, hogy milyen fajta adatot tárol, illetve kezel. Egy adott változó típusa pont olyan fontos informatikai fogalom, mint az értéke.

Pythonban egy változó használata előtt nem kell meghatározni, hogy a változó milyen fajta értéket fog kezelni, mi lesz a típusa, azaz nincs explicit változódeklarálás. A változódeklarálás **automatikus**, a kezdeti érték alapján, illetve a változón alkalmazott művelet alapján a Python ki tudja következtetni a változó típusát.

Pythonban a típus fogalma szorosan kapcsolódik az **osztály**, a `class` fogalmához. A nyelvbe beépített alaptípusok alkotják tulajdonképpen a típusokat, de a nyelv ezeket is `class`-nak címkézi. A programozó által létrehozott új szerkezetek, objektumok lesznek a tulajdonképpeni `class`-ok, amelyek a létrehozott objektumok struktúráját, viselkedését is meghatározzák.

A Pythonban használt alaptípusok az `int, float, str, bool`, amelyek rendre egész számokat, valós számokat, karakterláncokat, logikai értéket definiálnak.

A következő kódsorok azt mutatják be, hogy a `type` függvénnyel egy adott értéknek hogyan ellenőrizhetjük le a típusát.

```
>>> type(10)
    <class 'int'>
>>> type(891099511627776987983)
    <class 'int'>
>>> type('helo vilag')
    <class 'str'>
>>> type('103373189')
    <class 'str'>
>>> type(106.909)
    <class 'float'>
>>> type(True)
    <class 'bool'>
```

Sokszor szükséges, hogy egy adott értéket egyik típusból egy más típusú értékké alakítsunk. Ezt a folyamatot **típuskonverziónak** hívjuk. Például ha valamely értéket egész, azaz `int` típusú értékké szeretnénk átalakítani, akkor az `int` függvényt használjuk:

```
>>> int(23.11)
    23
>>> int(-23.900)
    -23
>>> int('23 szo')
    ...
    ValueError: invalid literal for int()...
>>> int(10/3)
    3
>>> int('23')
    23
```

Ha valós számokat alakítunk egész számmá, eredményként a számok egészrészét kapjuk. Karakterlánc típusú értékeket is megpróbálhatunk átalakítani, de amikor az átalakítás nem lehetséges, hibaüzenetet kapunk, ahogyan az utolsó műveletsor végrehajtása után látható.

Ha egy változót valós típusú értékké szeretnénk átalakítani, akkor a `float` függvényt, ha pedig karakterlánc típusúvá akarunk alakítani, akkor az `str` függvényt kell használni:

```
>>> float(15)
    15.0
>>> float('23.67')
    23.67
>>> str(12)
    '12'
>>> str(12.67)
    '12.67'
```

Az alaptípusok mellett a nyelv magas szintű adatszerkezetek használatát is biztosítja. A tananyag során ezek közül a `list` és `tuple` adattípusokkal fogunk foglalkozni. Használatuk számos előnnyel jár, például tetszőleges számú adat, illetve adatszerkezet kezelhető velük.

A **`tuple`** adattípust a magyar terminológiában `n`-esnek is mondják, de a jegyzetben a `tuple` megnevezés mellett maradunk. Egy `tuple` típusú adat elemeit vesszővel kell elválasztani, és kerek zárójelet csak akkor szükséges használni, ha egy függvény paramétereként használjuk. Létrehozása után egy `tuple` típusú adat nem változtatható meg (immutable), szemben egy `list` típusú adattal, amely megváltoztatható (mutable).

A következő kódsorban egy kételemű `tuple` típusát ellenőrizzük le, majd létrehozunk egy három elemet tartalmazó `tuple` típusú változót, amit `T`-vel jelölünk. Mind a `tuple`, mind a `list` típusú elemek sorszámozása nullától kezdődik, ezért az első elem sorszáma 0, a másodiké 1, míg a harmadik elem sorszáma 2 lesz. Ennek szemléltetése végett a jobb oldali oszlopban az indexértékek segítségével a megfelelő sorszámú elemekre hivatkozunk:

```
>>> type(('samsung', 1500))
    <class 'tuple'>
>>> T = 1, 'Sára', 8.5
>>> T
    (1, 'Sára', 8.5)
```

```
>>> T[1]
    'Sára'
>>> T[0], T[2]
    1, 8.5
```

Figyeljük meg, hogy a `T` lekérdezésekor a Python kerek zárójelbe tette az elemeket, és kerek zárójeleket használunk egy üres `tuple` létrehozásakor is:

```
>>> uresT = ()
>>> uresT
    ()
```

Az alábbi kódsor a `T` változóval végez további műveleteket. Ha megpróbáljuk megváltoztatni valamelyik elem értékét, legyen ez most a `T[1]`, akkor hibaüzenetet kapunk:

```
>>> T[1] = 'Zsuzsi'
    TypeError: 'tuple' object does not support assignment
```

Új elemet csak kisebb trükkel tudunk *hozzáadni* a `T`-hez, hiszen a `tuple` immutable. Az `=` operátorral egy ugyanolyan nevű `tuple` típusú adatot hozunk létre, amelyhez `+` operátor alkalmazásával, a már meglévők után hozzáteszünk még egy elemet. Természetesen több elemet is meg lehet adni,

a kerek zárójel és vessző használata azonban kötelező, ellenkező esetben hibaüzenetet kapunk:

```
>>> T = T + (9.25, )
    (1, 'Sára', 8.5, 9.25)
```

A következő kódsor az elemek egy másik elérési módját mutatja. Létrehozunk egy négyelemű `tuple` típusú adatot, ahol az elemek jelölésére négy változót használunk:

```
>>> T0, T1, T2, T3 = (2018, 'samsung', 25, 800)
>>> T1
    'samsung'
```

Tuple típusú adatok kezelésére számos beépített függvény létezik, ezek közül most a `len` függvényt alkalmazzuk, amelynek segítségével a korábban definiált `T` változó elemszámát határozzuk meg:

```
>>> len(T)
    4
```

A `len` különböző típusú adatszerkezeten is használható, ahogyan azt a következő kódsorok mutatják. Figyeljük meg, hogy a második példában tulajdonképpen azt határozzuk meg, hogy hány számjegyből áll 5^{100}:

```
>>> len('sapientia egyetem')
    17
>>> t = pow(5, 100)
>>> len(str(t))
    70
```

A **`list`**, azaz lista típusú adatok létrehozásakor szögletes zárójelet használunk. Egy `list` típusú adat típusának a lekérdezéséhez, hasonlóan a korábbi példákhoz, a `type` függvényt lehet használni. A következő kódsorok azt is szemléltetik, hogy egy `list` típusú változó különböző típusú elemeket is tárolhat, illetve hogy a lista egy eleme miként változtatható meg:

```
>>> type([1, 2, 4, 8, 16])
    <class 'list'>
>>> L = [1789, 'Sara', 8.5]
>>> L[0]
    '1789'
>>> L[1] = 'Zsuzsi'
>>> L
    [1789, 'Zsuzsi', 8.5]
```

Listákat generálhatunk is, ehhez alkalmazni fogjuk a * operátort. A következő kódsorokban `bool` típusú listák kerülnek meghatározásra, ahol minden listaművelet után a `print` függvénnyel íratjuk ki az eredményt. Természetesen a listaelemek lehettek volna más típusúak is.

```
>>> L = 5 * [True]
>>> print(L)
    [True, True, True, True, True]
>>> L = 3 * [True, False]
>>> print(L)
    [True, False, True, False, True, False]
```

Listaelemek kezdeti értékadásakor óvatosan használhatjuk az = operátort, mert például hibaüzenetet kapunk, ha a következőképpen próbálkozunk:

```
>>> ujL[0] = 11
    ...NameError: name 'ujL' is not defined
```

A helyes megoldáshoz két műveletsort szükséges megadni. A következő kódsorban az első létrehoz egy üres listát, a második pedig egy új elemet fűz a lista végéhez, így egy egyelemű listát kapunk, amelynek a nulladik mezőjére már, de első mezőjére még nem lehet hivatkozni:

```
>>> ujL = []
>>> ujL += [11]
>>> ujL[0]
    11
>>> ujL[1]
    ...
    IndexError: list index out of range
```

A += operátor alkalmazható több listaelem befűzésére is:

```
>>> R += [10000, 625, 1369]
>>> R
    [169, 144, 10000, 625, 1369]
```

Fontos megjegyeznünk, hogy különbség van egy karakterekből felépülő `list` típusú adathalmaz és egy `str` típusú adathalmaz között. Korábban láttuk, hogy egy `list` típusú adathalmaz elemei megváltoztathatók, most jelezzük, hogy egy `str` típusú adathalmaz elemei viszont **nem**:

```
>>> strS = 'sapi'
>>> type(strS)
    <class 'str'>
>>> listS = ['s', 'a', 'p', 'i']
>>> type(listS)
    <class 'list'>
>>> strS[0] = 'S'
    ...TypeError: 'str' object does not support  assignment
>>> listS[0] = 'S'
>>> listS
    ['S', 'a', 'p', 'i']
```

A következő kódsorban a `range` és `list` függvényeket mutatjuk be, létrehozunk egy `list` típusú adatszerkezetet:

```
>>> L = list(range(10))
>>> print(L)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A `range` segítségével egy számsorozat elemei adhatók meg. Jelen esetben a számsorozat első eleme nulla lesz, és a sorozatban egyesével fogják követni egymást az elemek. Az utolsónak generált elem értékét a `range` bemenete határozza meg, pontosabban ez egyenlő lesz a bemeneti paraméter értéke mínusz egy. A fenti kódsorban az utolsónak generált érték tehát a 9. A `list` függvény segítségével a `range` által meghatározott tartomány `list` típusú adatszerkezetté alakul.

A `range` függvény használata többféleképpen is lehetséges. Ha nem nullás kezdőértékkel szeretnénk a számsorozat elemeit kezdeni, akkor ezt mint első bemeneti paraméter lehet megadni:

```
>>> L = list(range(10, 15))
>>> print(L)
    [10, 11, 12, 13, 14]
```

Az elemek közötti lépésköz értékét a harmadik bemeneti értékében adhatjuk meg:

```
>>> L = list(range(10, 100, 10))
>>> print(L)
    [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Egy már létrehozott listához mindig hozzáfűzhetünk egy új elemet, amely kerülhet akár a lista végére, akár a lista elejére:

```
>>> L += [100]
>>> print(L)
    [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> L = [0] + L
>>> print(L)
    [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Listák esetében a `:` illetve `::` operátoroknak különös szerepe van, szeleteléseket (slicing) lehet velük végezni. Például megfordíthatjuk a lista elemeit, vagy kiválaszthatunk egy részlistát:

```
>>> L = list(range(10, 100, 10))
>>> print('a fordított sorrend:', L[::-1])
    a fordított sorrend: [90,80,70,60,50,40,30,20,10]
>>> print('a lista első 3 eleme:', L[:3])
    a lista első 3 eleme: [10, 20, 30]
>>> print('a lista elemei a 5. pozíciótól: ', L[5:])
    a lista elemei a 3. pozíciótól: [60, 70, 80, 90]
```

Szeleteléseket `str` típusú adatokon is végezhetünk. A következő kódsorban a `wStr` változó `str` típusú. A `:` operátorral rendre kiválasztjuk a karakterlánc különböző részeit, először a kilencedik pozíciótól hátrafelé az összest, aztán a kilenc és harmincötödik közöttieket, majd az elejétől az `(n-4)`-dik pozícióig, végül a kilencedik és `(n-4)` közöttieket, ahol `n` a karakterlánc hosszát jelöli. Az értékadás során elválasztójelnek a `\` jelet azért használjuk, hogy több sorba tudjuk megadni a feldolgozásra kerülő hosszú karakterláncot:

```
>>> wStr = '<a href="http://www.ms.sapientia.ro">\
    Sapientia EMTE</a>'
>>> wStr[8:]
    '"http://www.ms.sapientia.ro">Sapientia EMTE</a>'
>>> wStr[9:35]
    'http://www.ms.sapientia.ro'
>>> n = len(wStr)
>>> wStr[:n-4]
    '<a href="http://www.ms.sapientia.ro">Sapientia EMTE'
>>> m = len(wStr[:37])
>>> wStr[m:n-4]
    'Sapientia EMTE'
```

A Python `in` operátorával eldönthető egy adott értékről, hogy hozzátartozik egy adathalmazhoz vagy sem:

```
>>> 4 in range(10)
    True
>>> "A" in "sapientia"
    False
```

Ha azt szeretnénk megtudni, hogy nem tartozik hozzá egy adott érték az adathalmazhoz, akkor a `not in` operátort kell használni:

```
>>> -1 not in range(10)
    True
>>> "int" not in ["float", "bool", "int", "str"]
    False
```

Összegzésképpen elmondhatjuk, hogy egy `tuple` tulajdonképpen egy rögzített tömb, ahol az elemek mutatók, ami lehetővé teszi, hogy az elemek mérete változó és tetszőleges típusú legyen. Éppen ezért egy `tuple` létrehozás után az már nem változtatható meg. Akkor érdemes használni, amikor olyan adatokat akarunk feldolgozni, amelyeknek mérete a program futása során nem fog megváltozni. A `list` mögött azonban dinamikus tömb van, ami azt jelenti, hogy az indexelést, a hozzáadást, a törlést stb. konstans időben lehet megvalósítani, és a lista méretét is dinamikusan lehet változtatni. Az `str` tulajdonképpen karakterekből álló fix tömböt jelent, ahol minden karakter egy bájton van eltárolva. Minden módosítás, amit egy `str`-en végzünk, egy új karakterlánc létrehozását fogja eredményezni.

2.3. Vezérlési szerkezetek

Programkódok műveletsorainak elvégzési sorrendjét vezérlési szerkezetek alkalmazásával lehet irányítani. A Pythonban három vezérlési szerkezetet, utasítást lehet használni: az `if`, a `for` és a `while` utasításokat.

A következő kódsorban az **`if`** feltételes utasítás használatát mutatjuk be. Segítségével eldönthetjük, hogy az `x` vagy az `y` jelöli a nagyobbik számot. Írjuk be a prompt után az alábbi műveletsorokat, ahol minden sort az `Enter` lenyomásával fejezzünk be, kivéve az utolsót, amelyet két `Enter`-rel zárjunk. A shellben mindig két `Enter`-rel zárjuk le az összetett utasításokat.

```
>>> x, y = 10, 13
>>> if x > y: print(x,' nagyobb, mint ', y)
    else: print(y, ' nagyobb vagy egyenlő, mint ', x)
    13  nagyobb vagy egyenlő, mint  10
```

Az `if` általános alakja a következő:

```
if <logikaiKifejezes>:
    <utasitasTorzs>
elif <logikaiKifejezes>:
    <utasitasTorzs>
...
elif <logikaiKifejezes>:
    <utasitasTorzs>
else:
    <utasitasTorzs>
```

Jelentése a következő: ha az `if` utáni `logikaiKifejezes` logikai értéke igaz, akkor végrehajtásra kerül az `utasitasTorzs`, ellenkező esetben a soron következő `elif` utáni `logikaiKifejezes` kerül kiértékelésre, majd a következő és így tovább. Mindig az az `utasitasTorzs` kerül végrehajtásra, amely előtt igaznak bizonyul a `logikaiKifejezes`. Az `elif`, illetve `else` elágazások nem kötelezőek. Az `utasitasTorzs`-ben helyet foglaló műveletsorokat négy szóközzel, mindegyiket azonos tördelési mérettel kell bennebb írni. A szóközök helyett elfogadott a tabulátor használata is. Más vezérlési szerkezetek esetében is tördeléssel kell jelezni, hogy mely műveletsorok alkotnak egy blokkot.

Egy `logikaiKifejezes` logikai értéke `False`, ha
- a kifejezés egyenlő a `False` vagy a `None` konstanssal,
- ha egyenlő üres `list`, üres `tuple` vagy üres `str`-rel,
- ha a kifejezés numerikus és értéke `0`.

Minden más esetben `True` a `logikaiKifejezes` logikai értéke.

Logikai kifejezések **összehasonlító operátorok** segítségével adhatók meg. Pythonban ezek az operátorok a következők: `==`, `!=`, `>`, `>-`, `<`, `<=`, a kifejezések logikai értéke pedig:

a == b, ha `a` egyenlő b-vel, akkor `True`,
a != b, ha `a` nem egyenlő b-vel, akkor `True`,
a > b, ha `a` nagyobb, mint b, akkor `True`,
a >= b, ha `a` nagyobb vagy egyenlő, mint b, akkor `True`,
a < b, ha `a` kisebb, mint b, akkor `True`,
a <= b, ha `a` kisebb vagy egyenlő, mint b, akkor `True`.

Programírás során elengedhetetlen a **megjegyzések**, kommentek használata. A programkódok közé elhelyezett megjegyzéseket a kiértékelés során mellőzi a Python, ezeket elsősorban magyarázat céljából írják a programozók a kódsorok közé. A Pythonban megjegyzések használatára két lehetőség

is van aszerint, hogy egysoros vagy többsoros megjegyzést, magyarázatot szeretnénk hozzáfűzni egy utasításhoz vagy programrészhez.

```
#ez egy egysoros megjegyzés

"""ez egy
többsoros megjegyzés
"""
```

A `for` és a `while` segítségével műveletek ismételt végrehajtását lehet megvalósítani, ezért ezeket ciklusnak vagy iterációnak is hívjuk.

A következő kódsorokban először a **`for`** ciklust mutatjuk be. Kiíratjuk a természetes számokat, szóközökkel elválasztva nullától tízig, másodszor pedig szóközök nélkül. A kiíratás formáját a `print` utasítás megadásakor szabályozzuk. Az `end` paraméter használatával tetszőleges elválasztójelet vagy elválasztójel-sorozatot adhatunk meg. A harmadik `for` utasítás 2 hatványait írja ki, szóközökkel és vesszőkkel elválasztva egymástól az értékeket.

```
>>> for i in range(10):
        print(i, end = ' ') #ügyeljünk a tördelésre!!
    0 1 2 3 4 5 6 7 8 9
>>> for i in range(10):
        print(i, end = '')
    0123456789  #nem lesz a számok között elválasztójel
>>> for i in range(10):
        print(2**i, end = ', ')
    1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
```

Figyeljük meg, hogy mindhárom `for` keretén belül meghívásra kerül egy-egy `print`, amelyet minden esetben egy tabulátornyi hellyel bennebb írtunk. Ahogy korábban jeleztük, a tördelés a Python esetében szintaktikai elem, ezért a `for`-hoz tartozó utasítást megfelelő tördelést alkalmazva adtuk meg.

Mindhárom esetben alkalmazásra került egy `i` ciklusváltozó, amely a `for` működése alapján a `range` által specifikált tartomány értékeit fogja rendre jelölni, ezért mindhárom esetben tíz kiíratásra kerül sor. Az első két `for` esetében a megfelelő `i` értéke kerül kiíratásra, az utolsó esetben az `i`-nek kétféle szerep is jut, amellett, hogy számolja, hogy hányadik `print` utasításnál tartunk, a hatványkitevő szerepét is betölti.

A `for` ciklus általános alakja a következő:

```
for <valtozo> in <iteralhatoHalmaz>:
    <utasitasTorzs>
```

A megadott műveletsor azt jelenti, hogy a `valtozo` az `iteralhatoHalmaz` minden elemének értékét sorban felveszi, és ezekre az értékekre végrehajtja az `utasitasTorzs`-ben szereplő utasításokat. Az `utasitasTorzs` annyiszor kerül végrehajtásra, amennyi az `iteralhatoHalmaz` elemszáma. Az `iteralhatoHalmaz` bármilyen olyan típusú adathalmaz lehet, amelynek az elemeire sorban lehet hivatkozni. A jegyzet keretén belül leggyakrabban `list, str, tuple` típusú adathalmazokat fogunk `iteralhatoHalmaz`-ként alkalmazni. A gyakorlatban a `for` általános alakja valamivel bonyolultabb, megengedett például a `break, continue, else` ágak használata is, ezekre a későbbiekben látunk majd példát.

A következő kódsorban a `honap` változó rendre a listában megadott értékeket fogja jelölni, eredményként pedig kikerülnek a képernyőre, szóközzel elválasztva a megfelelő listaelemek:

```
>>> for honap in ['Szeptember', 'Október', 'November']:
        print(honap, end = ' ')
    Szeptember Október November
```

Az `if` feltételes utasítást alkalmazva a következő kódsor megszámolja, hogy a latin nagybetűs ábécében hány mássalhangzó van, pontosabban fogalmazva azt vizsgáljuk, hogy hány olyan eset van, amikor egy betű nem magánhangzó:

```
>>> db = 0
>>> ABC = 'ABCDEEFGHIJKLMNOPQRSTUVWXYZ'
>>> for betu in ABC:
        if betu not in 'AEIOU':
            db += 1
>>> print('Mássalhangzók száma: ', db)
    Mássalhangzók száma:  21
```

Az `i` ciklusváltozó egy `tuple` elemeinek az értékeit is jelölheti. A következő kódsorban az `if` feltételben az `i` által jelölt `tuple` elemet átalakítjuk `str` típusú értékké, és megvizsgáljuk, hogy első, azaz nulladik karaktere benne van-e az előző példánál definiált `ABC`-ben:

```
>>> T = 1, 'Termeszettudomanyi Muzeum', 2015, 30000, 'Mvh'
>>> for i in T:
        if str(i)[0] in ABC:
            print('Betuvel kezdodo elem: ', i)
    Betuvel kezdodo elem:  Termeszettudomanyi Muzeum
    Betuvel kezdodo elem:  Mvh
```

Ha nem tudjuk előre, hogy hányszor szeretnénk egy utasítássort elvégezni, akkor a **while** ciklus használatára van szükségünk. Ennek általános alakja a következő:

```
while <logikaiKifejezes>:
        <utasitasTorzs>
```

Az utasitasTorzs addig kerül ismételten végrehajtásra, ameddig a logikaiKifejezes logikai értéke igaz. Megfelelő tördeléssel itt is kell jelezni, hogy melyek azok az utasítások, amelyek az utasitasTorzs-höz tartoznak. Ha a logikaiKifejezes értéke indulásból hamis, akkor egyszer sem hajtódik végre az utasitasTorzs. Az utasitasTorzs utasításai között most is szerepelhet a break, continue és else ágak valamelyike.

A következő kódsorban meghatározzuk az első n szám szorzatát, azaz n faktoriális értékét. A pontos matematikai definíció szerint n faktoriális, azaz $n!$ a természetes számok halmazán értelmezett és fennáll:

$$n! = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1, \text{ és } 0! = 1.$$

```
>>> n, res = 10, 1
>>> while n >= 1:
        res = res * n
        n = n - 1
>>> print(res)
    3628800
```

Az utasítássor elején az n értékét 10-re inicializáltuk, azaz az n kezdőértékét 10-re állítottuk, ezért az eredmény 10!, azaz 3628800 lesz. Az eredményt a res változóba számoljuk, amelynek kezdeti értéke 1. A while keretén belül két műveletsort ismétlünk, az első utasítás megszorozza a res értékét n-nel. Az így módosított értéket az egyenlőség operátor használata miatt ugyancsak a res fogja jelölni. A második utasítás az n értékét csökkenti 1-gyel. A két utasítás egymás után való ismétlésével érjük el, hogy először n-et, majd (n-1)-et és így tovább, szorozzuk a res értékével. Az utasításokat addig végezzük, amíg az n értéke nagyobb vagy egyenlő, mint 1.

Ha nyomon szeretnénk követni az n és a res által jelölt értékek változását, akkor ki lehet egészíteni egy print utasítással a while-hoz tartozó műveletsort. A későbbiek során a részszámításások megjelenítéséhez szükséges print műveletek elé # jelet fogunk tenni, amelyet az olvasó a kód végrehajtásakor tetszés szerint eltávolíthat aszerint, hogy szeretné vagy sem megjeleníteni a részszámításokat.

```
>>> n, res = 10, 1
>>> while n >= 1:
        #print(n, res)
        res = res * n
        n = n - 1
    10 1
    9 10
    ...
    2 1814400
    1 3628800
```

A logikai kifejezések a korábban látott egyszerű kifejezések mellett lehetnek összetettek is. Összetett logikai kifejezések **logikai operátorok** segítségével adhatók meg, ezek a Pythonban a következők: `and`, `or`, `not`, amelyek rendre a logikai *és*, a *vagy*, és a *tagadás*-nak felelnek meg.

A következő kódsorban azt vizsgáljuk, hogy `x` kisbetűt vagy más karaktert jelöl. Ehhez két feltételt adunk meg, amelyek közé `and` kerül, hiszen egyszerre kell fennálljon mindkettő.

```
>>> x = 'a'
>>> if 'a' <= x and x <= 'z': print('kisbetu')
    else: print('nem kisbetu')
```

A fenti feltétel megadható egyszerűbben, a logikai kifejezések összeláncolásával:

```
>>> if 'a' <= x <= 'z': print('kisbetu')
    else: print('nem kisbetu')
```

Az alábbi kódsorban azt vizsgáljuk, hogy az `mStr` karakterlánc utolsó karaktere milyen. Ha felkiáltójel vagy kérdőjel, akkor nem lehet kijelentő mondat, a kifejezések között most `or` kapcsolat van. Figyeljük meg, hogy valamely karakterlánc utolsó karakterére való hivatkozásnál a -1 index értéket lehet használni.

```
>>> mStr = 'Helo világ!'
>>> if mStr[-1] == '!' or mStr[-1] == '?':
           print('nem kijelento mondat')
    else: print('kijelento mondat')
```

2.4. Programfuttatás

A prompt után beírt kifejezések, utasítások elvesznek, ha kilépünk a Pythonból. Az IDLE a Python standard fejlesztői környezete azonban lehetővé teszi, hogy ezeket a kifejezéseket elmentsük, hogy ezekből az utasításokból programokat állítsunk össze, hogy ezeket futtassuk. A Python-utasításokat tehát állományba írhatjuk, amelyeket szkriptnek is hívunk. A szkriptekben levő műveletsorok a beírási sorrend szerint kerülnek sorról sorra végrehajtásra. A végrehajtási sorrend természetesen többféleképpen is befolyásolható, elsősorban a korábban bemutatott vezérlési szerkezetek segítségével.

A programírás során természetesen az IDLE mellett más programszerkesztőt is lehet használni, a jegyzet keretén belül ezekre azonban nem térünk ki.

Az IDLE programszerkesztőt, amely egy új ablak megjelenését is jelenti, a `File/NewFile` menüpontból lehet elindítani, melynek főbb menüpontjai a következők:

- `File/New File` – új állomány létrehozása,
- `File/Save` – állomány mentése,
- `Run/Run Module` vagy `Ctrl + F5` -a program futtatása,
- stb.

Mentsük el a következő kódsorokat egy `PythonAlap1.py` nevű állományba:

```
n = 10
for i in range(n):
    print(i/2, end = ', ')
```

A futtatást a `Run/Run Module` menüponttal vagy az `F5` vagy a `Ctrl + F5` gyorsgomb lenyomásával érhetjük el. Ha helyesen jártunk el, akkor a képernyőn megjelenik az első tíz természetes szám `2`-vel való osztási eredménye:

```
0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,
```

A fenti kódsort parancssorból is lehet futtatni, ez Windows esetében a következőképpen lehetséges:

- szükségünk lesz a `python.exe` elérési útvonalára, ehhez a shellbe írjuk be a következőket:

```
>>> import os
>>> import sys
>>> os.path.dirname(sys.executable)
```

- adjuk meg `python.exe` elérési útvonalát a `PATH`-ben: az `Edit the system environment variables` menüpontban a `System Variables`-nél adhatjuk ezt meg,
- nyissunk meg egy terminált a `cmd` paranccsal,
- a parancssorban a `cd` paranccsal válasszuk ki azt a mappát, ahol a `PythonAlap1.py` állomány van. Feltételezve, hogy ez a `C:\PythonProgramok`, akkor a következők lesznek az utasítássorok:

```
> cd C:\PythonProgramok
> python PythonAlap1.py
```

További műveletsorokat is írhatunk az állományba, de ha nem akarjuk, hogy mindegyik végrehajtásra kerüljön, akkor kommenteljük ki azokat, amelyeket nem szeretnénk elvégezni. Használjunk `#` -t a sorok elején, vagy a megfelelő utasításblokkot tegyük `"""` közé.

Írjuk most a következőket a `PythonAlap1.py` állományba, majd futtassuk a szkriptet:

```
n = 6
for i in range(1, n + 1):
    print('az osztasi egészrész', end = ' ')
    print(i, '-el: ', end = '')
    print(n // i)
```

Ha helyesen írtuk be a kódsorokat, futtatás után a következő kell megjelenjen a képernyőn:

```
az osztasi egeszresz 1 -el: 6
az osztasi egeszresz 2 -el: 3
az osztasi egeszresz 3 -el: 2
...
```

A kódsorban a második `print` függvénynek három bemenete van: az első paraméter az `i` értéke, amivel azt érjük el, hogy kiíratjuk az `i` által aktuálisan jelölt értéket, a második paraméter egy idézőjelek között megadott karakterlánc, ami kiíratódik a képernyőre, végül harmadik paraméterként az értékek közötti elválasztójelet adtuk meg.

2.5. Függvények

Programírás során gyakran szükségünk van arra, hogy egy utasítássorozatot egy program különböző pontjain, különböző kezdeti értékeket megadva

tudjunk elvégezni. Ezeket az utasítássorozatokat a végső program részfeladatának szokás tekinteni. Egy adott részfeladatot megoldó utasítássorozatot érdemes különálló egységként kezelni. Pythonban, mint számos más programozási nyelvben, egy elkülönített részfeladatot függvénynek hívunk, amely megírásakor, alkalmazásakor jól meghatározott szabályrendszert kell betartani. Függvények írásával, alkalmazásával könnyedén megoldható a programkód strukturáltsága is. Korábban már láttunk példát beépített függvények használatára (`len`, `print`), most azt mutatjuk meg, hogy hogyan tudunk mi is függvényeket írni, illetve hogyan tudjuk az általunk megírt függvényeket alkalmazni, pontosabban meghívni.

A Pythonban egy függvény megadása a következőképpen történik:

```
def <fuggvenyNev>(<parameterLista>):
        <fuggvenyTorzs>
```

A fenti definíció alapján azt látjuk, hogy egy függvény megírásakor a `def` kulcsszót szükséges használni, utána pedig egy tetszőleges függvénynevet, majd zárójelben a bemeneti paraméterekeket kell megadni. A `fuggvenyTorzs` a tulajdonképpeni utasítássorozatot jelenti. A függvénytörzsben szereplő utasításokat egymás alá egy tabulátornyi hellyel szokás bennebb írni, a tabulátor helyett használhatunk szóközöket is. Fontos, hogy mindegyik utasítás azonos pozícióval kerüljön bennebb, mert ahogyan korábban említettük, a tördeléssel jelezzük, hogy mely műveletsorok tartoznak a függvényhez.

A megírt függvényt meghívhatjuk a Python shellből, egy végső programból, de egy másik függvényből is. A függvény meghívása azt fogja jelenteni, hogy azon a ponton, ahol meghívjuk, a paramétereknek megadott értékekkel végrehajtásra kerülnek a függvényben megadott utasítások. Egy függvény megadható bemeneti paraméterek nélkül is, ekkor a meghíváskor a kerek zárójelek közé nem kell semmit írni. Ilyenkor a függvényben szereplő utasítások nem függenek bemeneti értékektől.

Egy Python függvénynek mindig van egy visszatérési értéke, ezért valamely függvényhívás egy értékadó utasítás jobb oldalán is állhat. Az alapértelmezett visszatérési érték a `None`, és a függvényből való kilépés a függvénytörzsben megadott utolsó utasítás után következik be. A függvénytörzs valamely sora tartalmazhat egy vagy több `return` utasítást, amivel további kilépési pontokat lehet megadni, illetve itt adható meg a függvény egy `None`-tól különböző visszatérési értéke is. Meghíváskor a kimeneti értéket nem kötelező felhasználni. A `return` használatakor pedig nem kötelező visszatérési értéket megadni.

2.1. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy pozitív, természetes szám, az `n` osztóinak számát.

```
def osztokSzama(n):
    if n <= 0:
        print('helytelen bemenet')
        return
    db, i = 0, 2
    while i <= n // 2:
        if n % i == 0: db += 1
        i += 1
    return db
```

A függvénytörzs első műveletsorában az `if` utasítás a helytelen bemenetet kezeli, hibaüzenettel ér véget a függvénykiértékelés, ha `n` értéke negatív, vagy ha nulla. Ezután két értékadó műveletet adtunk meg. A `db` változóban fogjuk meghatározni az `n` osztóinak számát, ez kezdetben `0`. Az `i` változó a lehetséges osztók értékeit fogja rendre jelölni, kezdeti értéke `2` lesz, mert minden szám osztható 1-gyel és önmagával.

A `while` utasításhoz két művelet tartozik, amelyeket addig ismétlünk, amíg az `i` az összes lehetséges osztó értékét fel nem veszi. A `2`-vel kezdődő természetes számoktól egészen az `n` szám feléig tartó intervallumban keressük a lehetséges osztókat. Az oszthatóságot az `if` alkalmazásával vizsgáljuk, ha fennáll az `if`-ben megadott feltétel, akkor a `db` változó értékét `1`-gyel növeljük. A függvény utolsó sora a visszatérési értéket határozza meg. Vegyük észre, hogy két ponton történhet meg a függvényből való kilépés.

A megírt függvényt mentsük el az `PythonAlap1.py` állományba, majd fordítsuk le. Ezek után a függvényt meghívhatjuk a shellből. Többféleképpen is megtehetjük ezt:

```
>>> osztokSzama(60)
    10
>>> osztokSzama(-60)
    helytelen bemenet
>>> fRes = osztokSzama(-20)
    helytelen bemenet
>>> print(fRes)
    None
>>> fRes = osztokSzama(20)
>>> fRes
    4
```

Az első meghívás esetében kikerül a képernyőre `60` osztóinak a száma. Ez azért lehetséges, mert a függvény a `return` utasításon keresztül visszaadja a `db` értékét a hívó egységnek. A második meghíváskor a hibaüzenet kerül ki a képernyőre, a függvény által meghatározott érték pedig `None` lesz. Figyeljük meg, hogy a függvényből való két kilépési ponton nem ugyanaz a visszatérési érték típusa, ez azért lehetséges, mert mindkét esetben tulajdonképpen egy objektumreferencia lesz a visszatérési érték. Az `osztokSzama` utolsó meghívásakor az eredmény az `fRes`-be kerül.

A függvényt parancssorból (terminálból) is meg lehet hívni, ehhez előbb ki kell választani azt a mappát, ahová az `PythonAlap1.py` állományt mentettük. Feltételezve, hogy ez a `C:\PythonProgramok` nevet viseli, a következők lesznek az utasítássorok:

```
> cd C:\PythonProgramok
> python -c "from PythonAlap1 import*; osztok(60)"
```

A meghívás eredményeként csak az osztók jelennek meg a képernyőn, ha azt szeretnénk, hogy az osztók száma is megjelenjen, a következőképpen kell meghívnunk a függvényt:

```
> python -c "from PythonAlap1 import*; print(osztok(60))"
```

A következőkben úgy módosítjuk a függvényt, hogy az osztók számának a meghatározása mellett kiírjuk egy sorba az osztókat is, illetve meghívjuk a függvényt 60-as bemeneti értékkel, hogy futtatáskor automatikusan végrehajtódjanak az `osztok` függvény műveletsorai.

```
def osztok(n):
   if n <= 0:
        print('helytelen bemenet')
        return
   db, i = 0, 2
   while i <= n // 2:
        if n % i == 0:
           db += 1
           print(i, end = ' ')
        i += 1
   print()
   return db
print(osztok(60))
```

Egy állomány több függvényt is tartalmazhat, meghívásuk pedig opcionális. A következőben további függvényeket írunk, amelyeket elhelyezhetünk a `PythonAlap1.py` állományba.

2.2. algoritmus. Írjunk egy Python függvényt, amely egy listában megadott számok közül kiválogatja a négyzetszámokat.

```
def tesztNegyzet(x):
    y = int(x ** 0.5)
    if float(y * y) == x: return True
    return False

def mainNegyzet(L = [49,345,6084,26542,4326154,17424]):
    for elem in L:
        if tesztNegyzet(elem): print(elem, end = ' ')
```

A feladatot két függvény definiálásával oldottuk meg. A `mainNegyzet` bemenete egy valós számokból álló lista, amely alapértelmezetten a megadott számokat fogja tartalmazni, de híváskor megadható más érték is. A `mainNegyzet` a megadott listaelemek közül fogja kiírni azokat, amelyek négyzetszámok. A `tesztNegyzet` meghívásával pedig meg lehet állapítani, hogy egy szám négyzetszám-e.

```
>>> mainNegyzet()
    49 6084 17424
>>> mainNegyzet([34.5, 121.0, 56.75, 12])
    121.0
>>> tesztNegyzet(49)
    True
>>> tesztNegyzet(49.5)
    False
```

A `tesztNegyzet` függvény meghatározza az `y` változóba a bemeneti paraméter négyzetgyökének egészrészét. Az `if`-ben megadott feltétel akkor teljesül, ha `y` négyzete megegyezik `x`-szel, de előtte a négyzetre emelt értéken típuskonverziót szükséges végrehajtani, ezért került alkalmazásra a `float`.

A `mainNegyzet` a `for` alkalmazásával oldja meg, hogy az `elem` rendre az `L` listában megadott értékeket jelölje, tesztelje. Amikor a tesztelés eredménye `True`, akkor kiíratásra kerül a megfelelő listaelem.

A következő `valogatNegyzet` függvény nem írja ki a képernyőre a négyzetszámokat, hanem megtalálásukkor egy új listába teszi őket, amely a függvény visszatérési értéke is lesz.

```
def valogatNegyzet(L):
   ujL = []
   for elem in L:
      if tesztNegyzet(elem): ujL += [elem]
   return ujL

>>> valogatNegyzet()
    ...
    TypeError: valogatNegyzet() missing ...
>>> R = valogatNegyzet([167, 169, 179, 189, 144])
>>> R
    [169, 144]
```

A `valogatNegyzet` függvényben az `ujL` kezdetben üres listát jelöl. A `for`-ban ez az üres lista fog kiegészülni azokkal a számokkal, amelyekre igaz az `if`-ben megadott feltétel.

Korábban megadtunk egy Python-műveletsort, amely meghatározta `n!` értékét. Lássuk most ezt más módszerekkel is. Egy feladatot sokféleképpen lehet megírni, a cél legtöbbször a gyorsaság, a minél kevesebb memóriahasználat, de fontos szempont a programkód tömörsége, olvashatósága [28].

2.3. algoritmus. Írjunk egy Python függvényt, amely meghatározza `n!`-t.

```
def myFactorialF(n):
    if n < 0:
        print('helytelen bemenet')
        return
    res = 1
    for i in range(1, n+1):
        res *= i
    return res
```

A helytelen bemenet tárgyalása után a `for` utasítás használatával oldjuk meg, hogy többször végrehajtsuk a `res *= i` műveletet. Az `i` ciklusváltozó értékadása és értékmódosítása a `range` miatt automatikus, a `for` ezért egyetlen utasítást tartalmaz.

A második `myFactorialR` változat megírásához másképp fogunk gondolkodni. A következő megoldás során a függvény önmagát hívja, a kódsor a jól ismert matematikai definíció gondolatmenetét tükrözi, Az `if` feltétel a triviális esetért van, az utolsó sor pedig az általános esetet kezeli. Az így megírt függvényeket **rekurzív** függvényeknek hívjuk, amelyek előnye

elsősorban a programok átláthatóságában, tömörségében rejlik. Habár futási időben nem érik utol az iterációt alkalmazó függvényeket, helyességük könnyebben átlátható.

```
def myFactorialR(n):
    if n == 0: return 1
    return n * myFactorialR(n-1)
```

A harmadik változat esetében nincs szükség különösebb algoritmikus tudásra, a Python beépített függvényét kell ismerni. A faktoriális értékét meghatározó függvény a `math` modulban van, és a következőképpen kell meghívni:

```
>>> from math import factorial
>>> factorial(25)
15511210043330985984000000
```

Egy függvény paraméterkezelésének a megértéséhez figyeljük meg a következő `foF, segedF1, segedF2, segedF3` függvényeket:

```
def foF():
    db = 0
    segedF1(db)
    print('a segedF1 utan:', db)

    db = 0
    db = segedF2(db)
    print('a segedF2 utan:', db)

    db = ['a', 'b', 'c', 'd']
    segedF3(db)
    print('a segedF3 meghivasa utan:', db)

    db = 'abcd'
    segedF3(db)
    print('a segedF3 meghivasa utan:', db)
```

```
def segedF1(x):
    x += 10

def segedF2(x):
    x += 10
    return x

def segedF3(x):
   x[0] = 'z'
```

```
>>> foF()
    a segedF1 utan: 0
    a segedF2 utan: 10
    a segedF3 meghivasa utan: ['z', 'b', 'c', 'd']
```

```
...
TypeError: 'str' object does not support ...
```

A `segedF1` függvény törzsében végzett módosítás nem érvényesül a `foF` hívó egységben. Ezzel szemben a `segedF2` függvény hívása megváltoztatja a `db` értékét, mert a `foF` *látja* a `segedF2`-ben végrehajtott változtatást:

- a `segedF2` visszatérési értékét a `return x`-en keresztül adtuk meg, és
- a `foF`-ben a `db` az `=` értékadó utasításon keresztül kapta meg a `segedF2` által visszatérített értéket.

A `segedF3` megírásával egy másik problémára szeretnénk felhívni a figyelmet. A `segedF3` megváltoztatja a `x` nulladik pozícióján található értéket. Ez abban az esetben, amikor a függvénybemenet `list` típusú megengedett, és a hívóegységben is látható lesz az értékváltoztatás, azonban amikor a bemenet `str` típusú, tilos az ilyen típusú művelet, ezért is kapunk hibaüzenetet.

Megállapítható, hogy ha egy függvény a függvénytörzsében megváltoztatja egy immutable típusú (például `int, str, tuple`) bemeneti paraméterének az értékét, akkor ahhoz, hogy ez a változás a hívóegységben is érvényesüljön, a `return` utasítást használva kell visszaadja a megváltoztatott értéket. Ekkor a paraméterátadás **érték szerint** történik. Mutable típusú (például `list`) bemenet esetén a paraméteren végzett bármilyen értékmódosítást a hívóegység is fogja látni, ekkor a paraméterátadás **objektumreferencia** szerint történik.

Korábban már láttunk példát arra vonatkozóan, hogy hogyan kell egy Python-modult importálni és egy függvényét használni. A fenti kódsor az importálás és függvényhasználat egy másik lehetséges módját is mutatja. Az ilyen módon importált függvény esetében az előny, hogy nem lesz importálva a teljes modul, csak a feltüntetett függvény, hátránya, hogy meghíváskor nincs explicit jelölve, hogy egy adott függvény milyen modulhoz tartozik.

2.4. algoritmus. A `honapL` lista a hónapok neveit, a `homersekletL` lista a hónapokhoz tartozó hőmérsékleti értékeket tárolják. Írjuk egy Python függvényt, amely meghatározza, hogy mely hónapokban volt negatív a hőmérséklet.

```
honapL = ['januar','februar','marcius','aprilis','majus',
       'junius','julius','augusztus','szeptember',
       'oktober','november','december']
homersekletL = [-7,-5,-1,4,8,10,12,12,9,4,0,-5]
```

```
def negativH1(hL, homL):
    resL = []
    for i in range(0,12):
        if homL[i] < 0: resL += [hL[i]]
    return resL

>>> negativH1(honapL, homersekletL)
    ['januar', 'februar', 'marcius', 'december']
```

A `negativH1`-ben az `i` változónak kettős szerepe van, segítségével számoljuk, hogy hányadik `if`-nél, azaz melyik hőmérsékleti értéknél tartunk, illetve kiválasztjuk a `hL` és `homL` listák `i`-dik sorszámú elemét. A feltételnek eleget tevő hónapok neveit egymás után fűzve hozzuk létre a `resL`-t.

A bemenet más formában is megadható. Legyen az `adatL` egy értékpárokból álló lista, ahol egy értékpár elemét a hónap neve és a hozzá tartozó hőmérsékleti érték adja. Ekkor módosítani kell a függvényt, az új implementáció a következő lesz:

```
adatL = [('januar', -7),('februar', -5),('marcius', -1),
    ('aprilis', 4), ('majus',8), ('junius', 10),
    ('julius',12),('augusztus',12),('szeptember',9),
    ('oktober', 4),('november',0,),('december',-5)]

def negativH2(L):
    resL = []
    for elem in L:
        honap, homerseklet = elem
        if homerseklet < 0: resL += [honap]
    return resL

>>> negativH2(adatL)
    ['januar', 'februar', 'marcius', 'december']
```

A `negativH2` függvényben használt `elem` változó a `for` szerkezete miatt rendre felveszi az `L` listában levő értékeket. A `for` ciklus most két utasítást tartalmaz, az első utasítás az értékpár elemeihez társítja a `honap`, illetve `homerseklet` változókat, a második, az `if` utasítás pedig a kiválogatást végzi. Az eredményt a `resL` változó fogja most is jelölni.

2.6. Adatbevitel

2.5. algoritmus. Írjunk egy Python függvényt, amely beolvas a billentyűzetről két értéket, és meghatározza az alap aritmetikai műveletek elvégzése utáni értékeket.

```
def foPrg():
    x = int(input('x: '))
    y = int(input('y: '))
    e1, e2, e3, e4, e5, e6 = muveletek(x, y)
    print('összeg: ', e1)
    print('különbség: ', e2)
    print('szorzat: ', e3)
    print('osztási egészrész: ', e4)
    print('osztási maradék: ', e5)
    print('valós osztás: ', e6)

def muveletek(x, y):
    return (x + y, x - y, x * y, x // y, x % y, x / y )
```

A fenti kódsor elvárja a programot futtató felhasználótól, hogy adatokat olvasson be a billentyűzetről. Az adatbeolvasás úgy történik, hogy a billentyűzetről az adatbevitel végén az `Enter` gombot is lenyomjuk. A `foPrg`-ben az `input`, majd az `int` függvények sikeres végrehajtása után az `x`, illetve `y` változók a billentyűzetről beolvasott egész számokat fogják jelölni. A sikeres végrehajtás azt is feltételezi, hogy a felhasználó egy olyan értéket visz be a billentyűzetről, amely átalakítható egész számmá. Az `input` mindig egy `str` típusú érték beolvasását teszi lehetővé, ezért, ha ezt az értéket a továbbiakban egész számként akarjuk kezelni, akkor át kell alakítsuk `int` típusú értékké, ahogy a fenti kódsorban is tettük.

A `muveletek` 6 aritmetikai műveletet végez: meghatározza az `x` és `y` változók által jelölt értékek összegét, különbségét, szorzatát, osztási egészrészét, osztási maradékát, illetve a valós osztás eredményét. Ezeket a `return` segítségével, mint egy ötelemű `tuple` visszatéríti a hívóegységnek, jelen feladat esetében a `foPrg`-nek.

A `foPrg` függvény bekér a billentyűzetről két értéket, amelyeket az `x` és `y` változók jelölnek, majd meghívja a `muveletek` függvényt ezekre a bemeneti értékekre, végül kiírja a képernyőre a `muveletek` által kiszámolt értékeket. A `foPrg`-nek nincs bemeneti paramétere, a `muveletek`-nek azonban két bemeneti értékét kell megadni.

Sikeres fordítás után bármelyik függvényt meghívhatjuk:

```
>>> muveletek(25, 3)
    (28, 22, 75, 8, 1, 8.333333333333334)

>>> foPrg()
    x: 91
    y: 42
    összeg:  133
    különbség:  49
    szorzat:  3822
    osztási egészrész:  2
    osztási maradék:  7
    valós osztás: 2.1666666666666665
```

2.6. algoritmus. Írjunk egy Python függvényt, amely meghatározza a billentyűzetről beolvasott számok legkisebbikét, azaz minimumát.

```
def sMin():
    n = int(input('n: '))
    x = int(input('x: '))
    m = x
    for i in range(n-1):
        x = int(input('x: '))
        if x < m: m = x
    return m
```

Sikeres fordítás után meghívható az `sMin` függvény. A kódsort futtató felhasználó első feladata, hogy az `n` változóba értéket olvasson be, azaz meghatározza, hogy hány szám közül kell a legkisebb értéket kiválasztani. Ezután további `n` darab értéket kell beolvasson a billentyűzetről, amelyeket rendre az `x` változó fog jelölni.

```
>>> sMin()
    n: 5
    x: 121
    x: 340
    x: 55
    x: 16
    x: 78
    16
```

Az `m` változónak az lesz a szerepe, hogy mindig az aktuálisan legkisebb elem értékét jelölje. A `for` utasítás előtti `m = x` értékadás algoritmikailag tehát azt jelenti, hogy kezdetben az elsőnek beolvasott érték lesz a legkisebb elem.

A `for` utasításon belül további `n-1` darab elemet olvasunk be, amelyek értékét rendre összehasonlítjuk az `m`-ben tárolt értékkel. Ha az `m`-nél egy kisebb értéket olvasunk be, akkor az `m` felveszi az `x`-ben tárolt értéket, ellenkező esetben nem kell semmit csinálni. A `for` végén az `m`, a beolvasott értékek közül a legkisebbet fogja jelölni, ezért az `sMin` függvény ezt az értéket téríti vissza. Vegyük észre, hogy a beolvasott értkékek, azaz a számok, amelyek között keressük a minimum elemet, nincsenek eltárolva, kivéve az utolsónak beolvasott elemet.

A feladatot megoldhatjuk másképp is. A beolvasott számok mindegyikét el fogjuk tárolni egy listába, és a legkisebb elem meghatározásának azután fogunk neki, miután minden elemet beolvastunk a billentyűzetről:

```
def sMinP():
    n = int(input('n: '))
    L = []
    for i in range(n):
        x = int(input('x: '))
        L += [x]

    m = L[0]
    print("%12s%12s" % ('x', 'm'))
    for x in L[1:]:
        print("%12s%12s" % (x, m))
        if x < m: m = x
    return m
```

A részszámítások eredményeit, ahogyan korábban is tettük, a `print`-tel végezzük, és a `%` operátort használjuk a kiíratás formázáshoz. Vegyük észre, hogy a `%` operátort már korábban is használtuk, de akkor más célból, ez a kettős funkció azonban nem jelent problémát a Pythonban, mert kontextustól függően meghatározható egy adott operátor szerepe.

```
>>> sMinP()
     n: 5
     x: 121
     x: 340
     x: 55
     x: 16
     x: 78
           x           m
         340         121
          55         121
```

```
          16          55
          78          16
    16
```

A következő műveletsorok az `%` operátor használatát mutatják be. A `%` operátor két paraméteres, bal oldali paramétere egy `str` típusú érték, amelyben meghatározzuk, hogy a jobb oldali paraméter milyen formátumú legyen. A jobb oldali paraméter lehet egy érték vagy egy `tuple`:

```
>>> "%7d" % 78
    '     78'
>>> import math
>>> "%7.2f" % math.pi
    '   3.14'
>>> x, y = 340, 120
>>> "%12s%12s" % (x, y)
    '         340         120'
```

A következő feladat nem számokkal, hanem karakterláncokkal végez műveleteket, ezeket írja ki a képernyőre, ezeket fűzi egymás után, hogy a kívánt eredményt kapja.

2.7. algoritmus. Írjunk egy Python függvényt, amely az Egyetem Napja alkalmából meghívót készít különböző diákok részére, ahol a diákok neveit a billentyűzetről olvassuk be.

```
def meghivo(n):
    diakok = []
    for i in range(n):
        print('nev: ', end= '')
        diakok += [input()]
    for d in diakok:
        s1 = 'Kedves ' + d + '! \n\n'
        s2 = 'Tisztelettel meghívjuk az Egyetem Napja\n'
        s3 = 'alkalmából tartott eloadássorozatra!\n\n'
        print(s1 + s2 + s3)
```

A `meghivo` függvény paramétere a meghívók számát, azaz a billentyűzetről beolvasni szükséges karakterláncok számát jelöli. Az első `for`-ban sor kerül a `diakok` lista elemeinek a beolvasására. A következő `for`-ban a `d` változó rendre a `diakok` listában megadott elemek értékét fogja felvenni. A meghívó szövegét három karakterlánc, az `s1, s2, s3` egymás után való

fűzéséből kapjuk, ahol a `\n` alkalmazásával sortörést, pontosabban egy üres sor beiktatását valósítjuk meg.

Sikeres fordítás, futtatás, adatbevitel után a képernyőn megjelennek a személyre szabott meghívók:

```
>>> meghivo(4)
    nev: Mari
    nev: Szabi
    nev: Kato
    nev: Feri
    Kedves Mari!

    Tisztelettel meghívjuk az Egyetem Napja
    alkalmából tartott eloadássorozatra!
    ...
```

2.7. Fordítási, logikai és futási hibák

Az eddig megírt kódsorok esetében is megfigyelhettük, hogy fordítás során **fordítási hibákba** ütközhetünk. Ezek kiküszöbölése egy gyakorlott programozó számára nem okoz nehézséget, de kezdő programozók számára sok fejtörést okozhatnak, elsősorban azért, mert nem értik a rendszer által kiírt hibaüzeneteket. Számos programszerkesztő még a szerkesztés során figyelmezteti a programozót, például eltérő színezést alkalmazva, hogy nem írt helyesen valamilyen műveletsort.

A következő kódsorok fordítási hibákat szemléltetnek. Például az `asciiKodokba` függvény definiálásakor, az első sor végére nem tettünk kettőspontot, illetve a `asciiKodokbol` esetében a `for`-nál sem tettük ki a sor végére a kettőspontot:

```
def asciiKodokba(L)
    ujL = []
    for elem in L:
        ujL += [ord(elem)]
    return ujL
```

```
def asciiKodokbol(L):
    ujL = ''
    for elem in L
        ujL += chr(elem)
    return ujL
```

Ha helytelenül adjuk meg a logikai feltételeket, akkor sem lesz sikeres a fordítás. Például a következő függvényben az `if`-ben megadott feltételek közé `and`-et kellett volna tenni:

```
def elsoBetuNagybetusitese(s):
    if s[0] >= 'a' s[0] <= 'z':
        s = chr(ord(s[0]) - 32) + s[1:]
    return s
```

A fenti függvényekkel kapcsolatban megjegyeznénk, hogy a különböző szimbólumokhoz az ASCII-kódolási eljárás segítségével számokat rendeltek, azért hogy azokat a számítógépek fel tudják dolgozni. Az `ord` megadja a paraméterként megadott karakterhez rendelt számot, azaz a karakter kódját, a `chr` pedig meghatározza a kódértékhez tartozó karaktert. Az ASCII-kódolási eljárásról bővebben az 5.5. fejezetben olvashatunk.

A következő kódsorban nem tettünk zárójelet a `T` elem definiálásakor a sor végére, fordításkor ezért hibaüzenetet fogunk kapni:

```
def tupleNagybetusit():
   T = ('keleti', 'deli', 'nyugati'
   for t in T:
      print(elsoBetuNagybetusitese(t))
```

Ha kijavítjuk a jelzett hibákat, akkor a fenti függvények meghívhatóak lesznek, például a következőképpen:

```
>>> asciiKodokba('diszkret')
    [100, 105, 115, 122, 107, 114, 101, 116]
>>> asciiKodokbol([65,66,67,68])
    'ABCD'
>>> elsoBetuNagybetusitese('diszkret')
    'Diszkret'
>>> tupleNagybetusit()
    Keleti
    Deli
    Nyugati
```

A fordítási hibák kiküszöbölése után további hibákkal szembesülhetünk, ezek lehetnek **logikai hibák**. A logikai hibák maghatározása, azaz annak a megállapítása, hogy helyesen működik a programunk, már nem mindig egyszerű feladat. Nagyobb alkalmazások esetében akár egy egész erre szakosodott csapat, a tesztelők foglalkoznak a program helyességének a megállapításával. Ha a korábban definiált `muveletek` függvényben a `*` operátor helyett `**`-ot írunk, akkor erre semmi sem figyelmeztet, és nem is biztos, hogy egyből észrevesszük, hogy szorzás helyett hatványozást végez a kódunk.

```
def muveletek(x, y):
    return (x + y, x - y, x ** y, x // y, x % y, x / y )

>>> foPrg()
    x: 2
    y: 10
    ...
    szorzat:  1024
    ...
```

Programírás során egy másik fontos feladat a **futási hibák** kivédése. Leggyakrabban az adatbevitel során léphetnek fel futási hibák, ahogyan ez a korábban megírt `foPrg` függvény futtatásakor is megtörténhet, például a billentyűzetről nem numerikus (egész, vagy valós) értéket olvasunk be:

```
>>> foPrg()
    x: 1
    y: d
    ...
    ValueError: invalid literal for int() with base 10...
```

Futási hiba akkor is adódhat, ha nem használjuk az `int` átalakító függvényt az `x` értékadása során, hiszen a beolvasási művelet után egy `str` és egy `int` típusú értéken akarjuk alkalmazni a + operátort:

```
def foPrg_bad1():
    x = input('x: ') #elhagytuk az int függvényt
    y = int(input('y:'))
    e1, e2, e3, e4, e5, e6 = muveletek(x, y)
    print('osszeg: ', e1)
    ...

>>> foPrg_bad1()
    x: 12
    y: 13
    ...
    TypeError: can only concatenate str(not "int") to str
```

Ahhoz, hogy elkerüljük az ilyen típusú futási hibákat, a Python lehetővé teszi, más programozási nyelvekhez hasonlóan, a `try...except` utasításblokk használatát. Alkalmazásával elkerülhető, hogy nem megfelelő bemeneti érték esetében leálljon a program. A `foPrg` (2.5) korábban megírt függvényt tehát érdemes a következőképpen átírni:

```
def foPrg_():
    try:
        x = int(input('x: '))
        y = int(input('y: '))
    except ValueError as hiba:
        print(hiba)
        return
    e1, e2, e3, e4, e5, e6 = muveletek(x, y)
    print('összeg: ', e1)
    ...
```

A `try` részbe azokat a műveletsorokat kell írni, amelyek futási hibát okozhatnak, az `except` részbe pedig a hibaüzenetet, illetve a függvényből, a programból való kilépést is meg lehet adni.

A módosítás során azonban vigyázzunk a tördelésre, logikai hibával szembesülhetünk. Például ha a fenti függvénynek az utolsó két sora a `return` utasítással egy szinten lesz tördelve, akkor a `foPrg_bad2` meghívása helyes bemeneti adatok esetében sem fog eredményt kiírni a képernyőre, és nem fogjuk érteni, hogy miért:

```
def foPrg_bad2():
    try:
        x = int(input('x: '))
        y = int(input('y: '))
    except ValueError as hiba:
        print(hiba)
        return
        e1, e2, e3, e4, e5, e6 = muveletek(x, y)
        print('összeg: ', e1)
        ...
```

2.8. Szövegállományok kezelése

A Python más programozási nyelvhez hasonlóan lehetővé teszi, hogy egy program eredményét állományba írjuk, illetve a program bemeneti adatait állományból olvassuk be. Ennek érdekében állománykezeléshez szükséges függvényeket biztosít. Az állományok kezeléséhez szükséges fogalmaknak a megértéséhez a 2.7. algoritmust fogjuk módosítani.

2.8. algoritmus. Írjunk egy Python függvényt, amely az Egyetem Napja alkalmából meghívót készít különböző diákok részére, és a meghívókat kiírja egy szövegállományba.

```
def meghivoW():
   diakok = ['Mari', 'Szabi', 'Kati', 'Feri']
   allomany = open('meghivo.txt', 'wt')
   for d in diakok:
      s1 = 'Kedves ' + d + '! \n\n'
      s2 = 'Tisztelettel meghívjuk az Egyetem Napja\n'
      s3 = 'alkalmából tartott eloadássorozatra!\n\n'
      allomany.write (s1 + s2 + s3)
   allomany.close()
```

```
>>> meghivoW()
```

A `diakok` listát most konstans elemekkel inicializáltuk. A `meghivoW` függvény végrehajtásakor a képernyőre nem lesz semmi kiírva, csak a prompt újbóli megjelenése jelzi, hogy lefutott a programunk. A függvény által létrehozott `meghivo.txt` állományt pedig abban a mappában kell keresnünk, ahol a Python-állományunk is található.

A kódsor az állománykezeléshez három függvényt, illetve metódust alkalmaz. Az `open` függvény segítségével kapcsolatot létesíthetünk egy változó és a fizikailag létrejött állomány között. Paraméterezése lehetővé teszi, hogy megadjuk az állománynevet, az állomány típusát és az állomány hozzáférési módját. A fenti kódsorban az `open` első paramétere annak az állománynak a neve lesz, amelyet fel szeretnénk dolgozni. Második paraméterének `w` (write) értéke azt jelenti, hogy automatikusan létrejön egy üres tartalommal rendelkező állomány, amibe adatokat írhatunk. Ha nem adunk meg elérési útvonalat, akkor az állomány az aktuális mappába jön létre, illetve ha létezik hasonló nevű állomány az aktuális mappában, akkor annak tartalma felülíródik. A `t` érték azt jelenti, hogy az állományba karaktereket, karakterláncokat írhatunk, ilyenkor azt is mondjuk, hogy text módban nyitottuk meg az állományt, azaz szövegállomány kerül feldolgozásra. A későbbiekben találkozni fogunk a `t` helyett a `b` értékkel is, ami az állományok tartalmának bináris feldolgozását jelenti.

A következő állománykezelő művelet a `write`. Segítségével a paraméterként megadott karakterláncot kiírjuk az állományba. A `write` azonban nem egy beépített függvény, hanem egy metódus, ahol vegyük észre, hogy nem is úgy használjuk, ahogyan korábban az `sqrt` vagy a `len` függvényeket meghívtuk. A metódusok szorosan kapcsolódnak az objektumorientált

programozási paradigmához, amire részletesen nem térünk ki, most csak arra szeretnénk felhívni a figyelmet, hogy meghívási módjuk eltér a függvények meghívási módjától. Egy metódust mindig egy adott objektumra, jelen esetben az `allomany` objektumra alkalmazunk, és ezt a pont `(.)` operátor alkalmazásával jelezzük.

A `close` metódust, ahogy a neve is jelzi, akkor használjuk, amikor már nem akarunk több állományműveletet végezni, ilyenkor be kell zárnunk az állományt, a memóriában levő adatok tulajdonképpen ezután kerülnek ki a külső tárolóra.

2.9. algoritmus. Írjunk egy Python függvényt, amely az Egyetem Napja alkalmából meghívót készít különböző diákok részére, ahol a diákok neveit egy szövegállományban találjuk, innen olvassuk be őket.

Első feladatunk az lesz, hogy a munkakönyvtárunkba egy szövegszerkesztővel (pl. Windows alatt a Notepaddel) létrehozzunk egy `nevek.txt` állományt. Ennek minden sorába írjunk egy nevet, tartalma lehet a következő:

```
Mari
Szabi
Kati
Feri
Zsuzsa
```

A megoldáshoz a `meghivo` függvényt úgy módosítjuk, hogy töröljük azt a sort, ahol a `diakok` listának kezdőértéket adtunk, majd kiegészítjük azokkal a kódsorokkal, amelyek a `nevek.txt` állomány tartalmának a beolvasásához szükségesek:

```
def meghivoR():
    allomany = open('nevek.txt', 'rt')
    temp = allomany.read()
    diakok = temp.split('\n')
    allomany.close()
    for d in diakok:
        s1 = 'Kedves ' + d + '! \n\n'
        s2 = 'Tisztelettel meghívjuk az Egyetem Napja\n'
        s3 = 'alkalmából tartott eloadássorozatra!\n\n'
        print(s1 + s2 + s3)

>>> meghivoR()
```

Az `open` függvény második paramétere most `rt` lesz, mert az állományt olvasásra (read) és text módban akarjuk megnyitni. Az adatok beolvasását a `read` függvény teszi lehetővé, az adatformázást pedig a `split`-tel végeztük.

A `read` a teljes állomány tartalmát beolvassa, amelyet a `temp` fog jelölni, típusa `str` lesz, mert az állomány text módban lett megnyitva. A továbbiakban a beolvasott karakterláncot formázni kell, jelen esetben szavakra kell bontani, pontosabban a `\n` szimbólumok mentén fel kell darabolni. Ezt a formázást a `split` végzi, melynek paraméterként meg kell adni azt a karakterértéket, ami mentén a felosztást szeretnénk megvalósítani. A felosztás eredményét, azaz az állományban szereplő neveket a `diakok` változó fogja jelölni.

A `split` megértéséhez próbáljuk ki a shellben a következő utasításokat:

```
>>> 'Mari\nSzabi\nKati\nFeri\nZsuzsa'.split('\n')
    ['Mari', 'Szabi', 'Kati', 'Feri', 'Zsuzsa']
>>> 'Mari Szabi Kati Feri Zsuzsa'.split('')
    ['Mari', 'Szabi', 'Kati', 'Feri', 'Zsuzsa']
```

2.10. algoritmus. Írjunk egy Python függvényt, amely az Egyetem Napja alkalmából meghívót készít különböző diákok részére, ahol a diákok neveit egy szövegállományban találjuk, a meghívókat pedig személyenként külön állományba írjuk ki.

```
def meghivoRW():
    try:
        aIn = open('nevek.txt', 'rt')
    except:
        print('Állomány megnyitási hiba!')
        return
    temp = aIn.read()
    diakok = temp.split('\n')
    aIn.close()
    for d in diakok:
        aOut = open('meghivo' + d + '.txt', 'wt')
        s1 = 'Kedves ' + d + '! \n\n'
        s2 = 'Tisztelettel meghívjuk az Egyetem Napja\n'
        s3 = 'alkalmából tartott eloadássorozatra!\n\n'
        s4 = '\t\tA szervező bizottság\n\n\n'
        aOut.write (s1 + s2 + s3 + s4)
        aOut.close()

>>> meghivoRW()
```

A kódsort kiegészítettük hibakezeléssel, hogy ne adódjon futási hiba, ha a `nevek.txt` állomány nem létezik, például ha más mappában található, vagy más néven mentettük el. A kódsorban, egy időben két állománnyal dolgoztunk, éppen ezért az állományok azonosítására különböző nevű változóneveket választottunk, a bemeneti állományt az `aIn`, míg a kimeneti állomány jelölésére az `aOut` változót vezettük be. A második `open` paraméterének a `'meghivo' + d + '.txt'` karakterláncot adtuk meg, azért hogy a meghívókat tartalmazó állományok neveit a diákok nevei alapján hozzuk létre. A függvényünk által létrehozott `meghivo`*`Nev`*`.txt` állományokat pedig abban a mappában kell keresnünk, ahol a Python-állományunk is található.

2.11. algoritmus. A `homerseklet.txt` a hónapok neveit, illetve a hónapokhoz tartozó hőmérsékleti értékeket tárolja. Olvassuk be ezeket az adatokat a `homerseklet.txt` állományból, és írjunk egy Python függvényt, amely a beolvasott adatok esetében meghatározza, hogy mely hónapokban volt negatív a hőmérséklet.

Az adatbeolvasás előtt fontos tisztázni, hogy milyen szerkezete van a `homerseklet.txt` állománynak. Legyen tehát a `homerseklet.txt` tartalma a következő:

```
januar -7
februar -5
marcius -1
aprilis 4
majus 8
junius 10
julius 12
augusztus 12
szeptember 9
oktober 4
november 0
december -5
```

Az állományban, tehát minden hónapnév és hónapnévhez tartozó hőmérsékleti érték külön sorban van, ahol a hónap és a hőmérsékleti érték között egy-egy szóköz is található.

A feladat megoldásához előbb egy `beolvas` függvényt adunk meg, az adatok feldolgozásához pedig a `negativH2` függvényt fogjuk használni, melynek kódsorát korábban adtuk meg (2.4 algoritmus):

```
 def homersekletF():
    L = beolvas()
    resL = negativH2(L)
    return resL

def beolvas():
    inf = open('homerseklet.txt', 'rt')
    L = []
    while True:
        temp = inf.readline()
        if not temp: break
        temp = temp.strip('\n')
        honap, homerseklet = temp.split('')
        L += [(honap, int(homerseklet))]
    inf.close()
    return L

>>> homersekletF()
    ['januar', 'februar', 'marcius', 'november']
```

A `beolvas` függvényben az adatok beolvasását soronként végeztük egy `while` ciklus keretén belül, a `readline` metódussal. A `readline` mindig egy sort olvas be az állományból, visszatérési értékének típusa `str`. Minden beolvasási művelet után megvizsgáltuk, hogy sikeres volt-e vagy sem ez a művelet. Ha nem sikerült a sort beolvasni, azaz teljesült a `not temp` feltétel, akkor a `break`-kel megszakítottuk a további műveletek végrehajtását, ez ugyanis azt jelenti, hogy minden sort beolvastunk az állományból. A beolvasott sor formázásához előbb a `strip` metódussal levágtuk a sor végét jelző `\n` szimbólumot, majd a szóközök mentén a `split`-tel szavakra tördeltük a sort. Mielőtt a kapott két szót mint értékpárt hozzáfűztük volna az `L` listához, a `homerseklet`-et `str` típusból átalakítottuk `int` típussá.

Próbáljuk ki a `strip`-et a shellben:

```
>>> '!Sapientia Egyetem!!'.strip('!')
    'Sapientia Egyetem'
>>> 'Diszkret Matek\t\t'.strip('\t')
    'Diszkret Matek'
```

Vegyük észre, hogy a `strip` levágja a megadott `str` típusú adat elejéről és végéről is a paraméterként megadott karaktert.

2.9. Kitűzött feladatok

2.1. feladat. Írjuk meg az `fMin`, illetve `fMax` Python függvényeket, amelyek a két bemeneti paraméter közül meghatározzák a kisebbiket, illetve a nagyobbikat.

```
>>> fMin(23, 56)
    56
>>> fMax('helo','elsoev')
    'helo'
```

2.2. feladat. Írjuk meg az `fEgyenlet` Python függvényt, amely meghatározza az $a \cdot x = b$ elsőfokú egyenlet gyökét, ha a két bemeneti paraméter `a` és `b`.

```
>>> fEgyenlet(12, 3)
    -0.25
>>> fEgyenlet(0, 3)
    'nincs megoldás'
```

2.3. feladat. Írjuk meg az `fAbs` Python függvényt, amely meghatározza a bemeneti paraméter abszolút értékét.

```
>>> fAbs(-10)
    10
```

2.4. feladat. Írjuk meg az `fElojel` Python függvényt, amely meghatározza egy szám előjelét.

```
>>> fElojel(-10)
    'negatív'
```

2.5. feladat. Írjuk meg az `fBajt` Python függvényt, amely meghatározza, hogy hány bájt szükséges egy `k`-bites adathalmaz eltárolásához.

```
>>> fBajt(9)
    2
>>> fBajt(32)
    4
```

2.6. feladat. Írjuk meg az `fTerm` Python függvényt, amely kiírja vesszővel elválasztva egymástól az `n` és `m` közötti természetes számokat.

```
>>> fTerm(4, 3)
    'nincs megoldás'
>>> fTerm(4, 10)
    4, 5, 6, 7, 8, 9, 10
>>> fTerm(4, 4)
    4
```

2.7. feladat. Írjuk meg az `fParos` Python függvényt, amelynek `n` paramétere ha negatív, akkor kiírja vesszővel elválasztva egymástól az `n`-nél nagyobb vagy vele egyenlő negatív páros számokat, ha pedig `n` nem negatív, akkor az `n`-nél kisebb vagy vele egyenlő nem negatív, páros számokat.

```
>>> fParos(-10)
    -2, -4, -6, -8, -10
>>> fParos(-9)
    -2, -4, -6, -8
>>> fParos(11)
    0, 2, 4, 6, 8, 10
>>> fParos(10)
    0, 2, 4, 6, 8, 10
```

2.8. feladat. Írjuk meg az `fNegyzet` Python függvényt, amely ha `n` nem negatív egész szám, akkor meghatározza az `n`-nél kisebb vagy vele egyenlő négyzetszámok listáját, ellenkező esetben pedig a függvény adjon hibaüzenetet.

```
>>> fNegyzet(11)
    [0, 1, 4, 9]
>>> fNegyzet(-10)
    'nincs megoldás'
```

2.9. feladat. Írjuk meg az `fNegyzetPar` Python függvényt, amely meghatározza az `n`-edik négyzetszámot. Az algoritmust úgy írjuk meg, hogy vegye figyelembe, hogy az `n`-edik négyzetszám felírható az első `n` páratlan szám összegeként.

Példa:

$$\begin{array}{lrcl} 1. & 1 & = & 1 \\ 2. & 4 & = & 1+3 \\ 3. & 9 & = & 1+3+5 \\ 4. & 16 & = & 1+3+5+7 \\ 5. & 25 & = & 1+3+5+7+9 \end{array}$$

2.10. feladat. Írjuk meg az `fParatlan` Python függvényt, amelynek paraméterként egy egész számokból álló listát adunk meg, és amely meghatározza a páratlan számok számát.

```
>>> fParatlan([3, 21, 21, 4, -5, 14, 10, 17])
    5
```

2.11. feladat. Írjuk meg az `fOsszeg` Python függvényt, amely meghatározza az első `n` szám összegét, ha `n` nem negatív egész szám. Minden más bemenetre adjon hibaüzenetet a függvény.

```
>>> fOsszeg(10)
    55
>>> fOsszeg(-10)
    'nincs megoldás'
>>> fOsszeg('y')
    'hibás bemenet'
```

2.12. feladat. Írjuk meg az `fOsszeg`, illetve `fSzorzat` Python függvényeket, amelyeknek paraméterként valós számokból álló listát adjunk meg, ahol az `fOsszeg` határozza meg a lista elemeinek összegét, az `fSzorzat` pedig az elemek szorzatát. Ne használjunk könyvtárfüggvényeket.

```
>>> fOsszeg([4.5, 6.75, 10, 8.5, 5.33])
    35.08
>>> fOsszeg([4.5, 't', 6.75, 10, 5.33])
    'hibás bemenet'
```

2.13. feladat. Írjuk meg az `fMin` Python függvényt, amelynek paraméterként egy listát adjunk meg, és amely meghatározza a lista elemei közül a legkisebbet, illetve a legkisebb elem sorszámát.

```
>>> fMin([3.5, 4.8, -5.2, -1, 10, 21.7])
    (-5.2, 2)
>>> fMin(['abab', 'aaa', 'bababa', 'bbbb', 'aab'])
    ('aaa', 1)
```

2.14. feladat. Írjuk meg az `fMax` Python függvényt, amelynek paraméterként egy listát adjunk meg, és amely meghatározza a lista elemei közül a legnagyobbat és a legnagyobb számok sorszámát.

```
>>> fMax([21, 3, 21, 4, -5, -1, 10, 21])
    (21, [0, 2, 7])
```

2.15. feladat. Írjuk meg az `fNegyzetT` Python függvényt, amelynek paraméterként egy egész számokból álló listát adjunk meg, és amely meghatározza egy `tuple` típusú listába a négyzetszámokat és a négyzetszámok pozícióit.

```
>>> fNegyzetT([64, 400, 15, 47, 9, 1001, 10, 29])
    [(64, 0), (400, 1), (9, 4)]
>>> fNegyzetT(list(range(30)))
    [(0, 0), (1, 1), (4, 4), (9, 9), (16, 16), (25, 25)]
```

2.16. feladat. Írjuk meg az `fAtlag` Python függvényt, amely beolvas `n` számot a billentyűzetről, ahol legyen `n` a függvény bemeneti paramétere. A függvény tárolja el ezeket a számokat egy listában, majd határozza meg a beolvasott számok átlagértékét. Ne használjunk könyvtárfüggvényeket.

```
>>> fAtlag(5)
    szam: 10
    szam: 9.5
    szam: 8.75
    szam: 10
    szam: 6.5
    8.95

>>> fAtlag(7)
    szam: 6
    szam: f
    'hibás bemenet'
```

2.17. feladat. Írjuk meg a `fuzStr` Python függvényt, amely beolvas `n` karakterláncot a billentyűzetről, ahol legyen `n` a függvény bemeneti paramétere. A beolvasott karakterláncokból a függvény készítsen egyetlenegy karakterláncot úgy, hogy a beolvasott értékeket fűzze egymás után, kötőjeleket téve közéjük.

```
>>> fuzStr(4)
    Sapientia
    erdelyi
    magyar
    tudomanyegyetem
    Sapientia-erdelyi-magyar-tudomanyegyetem
```

2.18. feladat. A `telefon.txt` állomány telefonok adatait tartalmazza. Minden telefonról egy sorban, szóközökkel elválasztva négy adat van eltárolva: márkanév, megjelenési év, darabszám, ár. Írjuk meg a `beolvas` Python

függvényt, amely beolvassa a `telefon.txt` tartalmát egy négyelemű `tuple` elemtípusú listába, majd írjuk meg:

- a `countTelefon` Python függvényt, amely meghatározza, hogy egy adott évben hány telefon jelent meg,
- a `minTelefon` Python függvényt, amely meghatározza azt a márkanevet, amelyből a legkevesebb van,
- az `averageTelefon` Python függvényt, amely meghatározza a telefonok átlagárát.

Ha a `telefon.txt` állomány tartalma a következő:

```
samsung2 2022 30 950
nokia 2021 25 500
iphone1 2020 40 2300
samsung1 2021 15 800
huawei 2021 35 1800
iphone2 2020 45 2500
```

akkor az eredmény:

```
>>> countTelefon(2021)
    3
>>> minTelefon()
    'samsung1'
>>> averageTelefon()
    1475.0
```

3. fejezet

Kombinatorika

Matematikai problémák egyik legalapvetőbb objektuma a halmaz, amely különböző elemek gyűjteményét jelenti. Halmazt alkothatnak egy egyetem diákjai, a betűk valamely ábécéből, a matematika legfontosabb halmazai azonban a természetes, egész, racionális stb. számok halmaza.

A kombinatorika a matematikának és informatikának is egy nagyon fontos területe, véges, illetve megszámlálható halmazok előállításával foglalkozik. Idetartoznak azon algoritmusok, amelyek a halmazelemek összeszámlálását, felsorolását, permutálását, rendezését stb. végzik [20].

A Python listakifejezéseivel nagyon elegánsan lehet kombinatorikai feladatokat megoldani, mert a matematikában megszokott jelölésrendszer szerint teszik lehetővé egy adott halmaz elemeinek a megadását. A fejezet első felében ezeket a nyelvi elemeket mutatjuk be.

3.1. Listakifejezések

Egy Python-listakifejezéssel (angolul list comprehension) listák elemeinek megadását lehet nagyon tömör formában megvalósítani. Szögletes zárójelbe egy kifejezést kell írnunk, a kifejezés után pedig egy `for` ciklusban a generálási szabályt kell megadni. Az új listaelemek egy már létező lista alapján adhatók meg.

A generálási szabály megadásakor használhatunk feltételes kifejezéseket is. Az ilyen szerkezetek számos előnnyel rendelkeznek, például a klasszikus módon megadott `for` ciklussal szemben helyet és időt takaríthatunk meg,

tömörebb kódot kaphatunk, ahol az iterációt gyakorlatilag képletté alakítjuk [13].

3.1. algoritmus. Írjunk egy Python-listakifejezést, amely meghatározza az első `n` páros szám listáját.

```
def parosLista(n):
    return [x for x in range(0, 2*n, 2)]

>>> parosLista(10)
    [0, 2, 4, 6, 8]
```

3.2. algoritmus. Írjunk egy Python-listakifejezést, amely meghatározza az első `n` négyzetszám listáját.

```
def negyzetLista(n):
    return [x * x for x in range(0, n)]

>>> negyzetLista(10)
    [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

3.3. algoritmus. Legyenek egy lista elemei nagyobbak vagy egyenlők, mint 0 és kisebbek vagy egyenlők, mint 15, azaz a listában egy 16-os számrendszerben megadott számsorozatnak az alakját adjuk meg. Írjunk egy Python-listakifejezést, amely bemenetként egy ilyen típusú listát kap és meghatározza azt a listát, ahol a számértékeket a 16-os számrendszerben használt szimbólumokkal helyettesítjük.

A számítástechnikában használt számrendszerekkel a 5.1. fejezetben fogunk foglalkozni, elöljáróban csak azt mondjuk el, hogy a tizenhatos számrendszerben a `0,1, ...,9,A,B,C,D,E,F` szimbólumokat használjuk, amelyek rendre a `0,1,...,9,10,11,12,13,14,15`-ös számértékeknek felelnek meg, illetve a nagybetűk helyett a megfelelő kisbetűk is használhatók.

```
def hexaSz(c):
    if c >= 0 and c <= 15:
        if c >= 10 and c <= 15: return chr(c + 55)
        else: return chr(c + 48)
    else: return None

def hexaLista(L):
    temp = [hexaSz(x) for x in L]
    print('tempList: ', temp)
```

```
    return ''.join(temp)

>>> hexaSz(13)
    'D'

>>> hexaLista([12, 4, 5, 15, 7, 0, 11, 4])
    tempList:  ['C', '4', '5', 'F', '7', '0', 'B', '4']
    'C45F70B4'
```

A feladatot két függvénnyel oldottuk meg, ahol a `hexaSz` egyetlenegy számértéket helyettesít a megfelelő hexa szimbólummal, a `hexaLista` pedig a listakifejezést használva alkalmazza a `hexaSz`-t minden egyes elemére. Az így kapott eredménylistára meghívtuk a `join` beépített metódust, amely a listaelemekből egy `str` típusú értéket hoz létre.

Hívjuk meg a `join` metódust a shellben, hogy jobban megértsük a működését:

```
>>> ' '.join(['C', '4', '5', 'F', '7', '0', 'B', '4'])
    'C 4 5 F 7 0 B 4'
>>> '#'.join(['C', '4', '5', 'F', '7', '0', 'B', '4'])
    'C#4#5#F#7#0#B#4'
```

3.4. algoritmus. Írjunk egy Python-listakifejezést, amely meghatározza az `n`-nél kisebb pitagoraszi számhármasok listáját, ahol pitagorászi számhármast alkot az a három x, y, z egész szám, amelyekre igaz, hogy $x^2+y^2 = z^2$.

```
def pitagoraszLista(n):
    return [(x, y, z) for x in range(1, n)
                      for y in range(x, n)
                      for z in range(y, n)
                      if z * z == x * x + y * y]

>>> pitagoraszLista(25)
    [(5, 4, 3), (10, 8, 6), (13, 12, 5), (15, 12, 9) ...
```

3.5. algoritmus. Írjunk egy Python-listakifejezést, amely a bemeneti karakterláncot szavakra bontja, minden szót nagybetűsít, és minden szó esetében meghatározza a szóhosszt.

```
def szohosszLista(kLanc = 'Emte Sapientia 2001 oktober'):
    szoStr = kLanc.split()
    return [(szo.upper(), len(szo)) for szo in szoStr]
```

```
>>> szohosszLista()
 [('EMTE',4),('SAPIENTIA',9),('2001',4),('OKTOBER',7)]
```

3.6. algoritmus. Írjunk egy Python-programot, amely összehasonlítja egy listakifejezést alkalmazó függvény és egy `for` ciklust alkalmazó függvény időigényét.

```
import time
def forCiklus(n):
    result = []
    for i in range(n):
        result += [i * i]
    return result

def listaKifejezes(n):
    return [i*i for i in range(n)]

def idomeres():
    st = time.time()
    forCiklus(10**7)
    fs = time.time()
    print('for ciklus idoigenye: ', round(fs - st, 4))
    st = time.time()
    listaKifejezes(10**7)
    fs = time.time()
    print('listakifejezes idoigenye: ', round(fs - st, 4))

>>> idomeres()
    for ciklus idoigenye:  1.7461
    lista kifejezes idoigenye:  0.9401
```

A futási idő meghatározásához a `time` modul `time` függvényét alkalmaztuk: a `forCiklus` meghívása előtt is eltároltuk a számítógép által mutatott időértéket. és a meghívás után is, az előbbit az `st` változóba, míg az utóbbit az `fs` változóba tettük. A mért idők közti különbség mutatja a futási időt, amelyet a `round` segítségével felkerekítve írtunk ki a képernyőre. Ugyanígy jártunk el a `listaKifejezes` függvény meghívásakor is.

A `forCiklus` és `listaKifejezes` függvények mindegyike létrehozza az első `n` természetes szám négyzetének a listáját. Az időt egy-egy 10^7 elemszámú lista kigenerálása esetén mértük le, jól látható, hogy a listakifejezést alkalmazó kódsor sokkal hatékonyabb. Természetesen különböző számítógépen végezve az időméréseket különböző időértékeket fogunk kapni, a két időérték közötti különbség azonban a számítógépektől független.

3.2. Összeszámlálási alapelvek

A kombinatorikában leggyakrabban az összeadás szabályát és a szorzás szabályát alkalmazzák annak érdekében, hogy valamely feladat megoldásszámát, azaz a lehetséges eseteket meg tudjuk határozni [20, 28].

Az **összeadás szabálya** a számlálási feladatok közül a legegyszerűbb. Azt jelenti, hogy ha egy feladat megoldható n_1 vagy n_2-féleképpen, ahol az n_1 és n_2 megoldáshalmazoknak nincs közös elemük, akkor a feladat $n_1 + n_2$-féleképpen oldható meg. Az összeadás szabálya általánosítható $n_1, n_2, \ldots, n_k$ részfeladatra, ekkor a megoldásszám: $n_1 + n_2 \cdots + n_k$ lesz.

Az összeadás szabálya a halmazelméletben alkalmazott terminológiával is megfogalmazható: ha $H_1, H_2, \ldots, H_m$ véges elemszámú diszjunkt halmazok, akkor az egyesített halmazok elemszáma megegyezik a halmazok elemszámának összegével:

$$|H_1 \cup H_2 \cup \cdots \cup H_m| = |H_1| + |H_2| + \cdots + |H_m|,$$

ahol $A_i \cap A_j = \emptyset$, minden i, j-re.

Példa: Ha egy diák elsőként választ diplomadolgozat témát, és összesen 5 vezetőtanár témái közül választhat, ahol az első tanár 5 témát, a második 9, a harmadik 11, a negyedik 3 és az ötödik 4 témát javasolt, akkor a diák összesen $5 + 9 + 11 + 3 + 4 = 32$ téma közül választhat.

Példa: Egy egyetemen a hallgatói önkormányzatba ki kell választani egy képviselőt. Hányféleképpen lehet kiválasztani a képviselőt, ha a hallgató kiválasztható 45 informatikus vagy 50 számítástechnikus hallgató közül? Válasz: $45 + 50 = 95$-féleképpen.

A **szorzás szabálya** azt jelenti, ha egy feladat felbontható két részfeladatra, ahol az első részfeladatot n_1-féleképpen oldhatjuk meg, és ezek közül mindegyik feladatot n_2-féleképpen oldhatjuk meg, akkor a feladatot $n_1 \times n_2$ módon lehet megoldani. Természetesen ez a szabály is általánosítható, ekkor a megoldásszám: $n_1 \times n_2 \cdots \times n_k$ lesz.

A szorzás szabálya is megfogalmazható halmazelméleti terminológiával: ha $H_1, H_2, \ldots, H_m$ véges elemszámú halmazok, akkor ezen halmazok Descartes-szorzatának elemszáma megegyezik a halmazelemszámok szorzatával. A szorzás szabállyal való kapcsolat pedig azáltal látható be, hogy egy elem kiválasztása a Descartes-szorzatból azt jelenti, hogy kiválasztunk egy elemet a H_1-ből, a H_2-ből, ..., a H_m-ből:

$$|H_1 \times H_2 \times \cdots \times H_m| = |H_1| \cdot |H_2| \cdot \cdots \cdot |H_m|.$$

Példa: Ha egy egyetemen két oktató n darab egyágyas szoba között választhat, akkor $n \cdot (n-1)$ -féleképpen foglalhatják el a szobákat.

3.7. algoritmus. Írjunk egy Python-listakifejezést, amely meghatározza, hogy ha két oktató n darab egyágyas szoba között választhat, akkor melyek lesznek a lehetséges választások.

```
def szamol1(n):
    return [(x, y) for x in range(1, n + 1)
                   for y in range(1, n + 1)
                   if x != y]

>>> len(szamol1(4)), szamol1(4)
    (12, [(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), ...])
```

A fenti lekérdezés egyszerűen értelmezhető, ha figyelembe vesszük, hogy a szobák 1-től `n`-ig vannak számozva. A generált lista kételemű `tuple`-ket tartalmaz, ahol az első érték az egyik oktató szobaszáma, a második érték pedig a második oktató szobaszámát jelenti.

Példa: Az egyetemi raktárnyilvántartásban minden számítógépet felcímkéznek, ami egy angol ábécébeli nagybetűből és egy kétjegyű egész számból áll. Hányféle felcímkézési mód lehetséges, és melyek ezek? Tudva azt, hogy összesen 26 angol ábécébeli betű és 90 kétjegyű szám létezik, a válasz: $26 \cdot 90 = 2340$-féleképpen történhet a címkézés.

3.8. algoritmus. Írjunk egy Python-listakifejezést, amely meghatározza az előző példában megfogalmazott szabály szerint kigenerálható címkéket, illetve a különböző címkék számát.

```
def szamol2():
    return [chr(x) + str(y) for x in range(65, 91)
                            for y in range(10, 100)]

>>> len(szamol2()), szamol2()[:5]
    (2340, ['A10', 'A11', 'A12', 'A13', 'A14'])
```

Példa: Ha egy személygépkocsi számtáblája öt szimbólumból áll, ahol az első két szimbólum a 10 lehetséges számjegy valamelyike, és a következő három szimbólum pedig az angol ábécé 26 lehetséges karaktere, akkor összesen $10 \cdot 10 \cdot 26 \cdot 26 \cdot 26 = 1757600$ különböző számtáblát lehet kibocsátani.

3.9. algoritmus. Írjunk egy Python-listakifejezést, amely meghatározza, hogy melyek a lehetséges számtáblák, amelyek az előző példában megfogalmazott szabály szerint generálhatók ki. Határozzuk meg a generált lista elemszámát, illetve paraméterezzük a függvényt, hogy lehetőségünk legyen a kigenerált lista a és b közötti elemeit kiíratni.

```
def szamol3(a, b):
    L = [str(x) + str(y) + chr(i)+ chr(j)+ chr(k)
              for x in range(0,10)
              for y in range(0,10)
              for i in range(65, 91)
              for j in range(65, 91)
              for k in range(65, 91)]
    return len(L), L[a : b]
```

```
>>> szamol3(859000, 859010)
(1757600, ['48WSM','48WSN','48WSO','48WSP','48WSQ','48WSR'
'48WSS','48WST','48WSU','48WSV'])
```

A **kivonás szabálya** azt jelenti, hogy ha egy feladat megoldható n_1-féleképpen vagy n_2-féleképpen, akkor a feladat $n_1 + n_2$ módon lesz megoldható, minusz az az érték, amely mindkét megoldási menetnek része.

A szabály halmazelméletben alkalmazott terminológiával is megfogalmazható. Legyenek H_1, H_2 véges elemszámú halmazok, ahol feltételezzük, hogy a H_1 halmazból $|H_1|$-féleképpen tudunk kiválasztani egy elemet és a H_2 halmazból pedig $|H_2|$-féleképpen. Egy elem kiválasztása a H_1 vagy H_2 halmazból, azaz a $H_1 \cup H_2$ halmazból $|H_1 \cup H_2|$-féleképpen lehetséges, ami egyenlő azzal, hogy az elemet $|H_1|$-féleképpen tudjuk kiválasztani a H_1 halmazból $|H_2|$-féleképpen a H_2 halmazból, levonva ebből az összeggből a $|H_1 \cap H_2|$ értéket ami a közösen kiválasztható elemszámot jeleni:

$$|H_1 \cup H_2| = |H_1| + |H_2| - |H_1 \cap H_2|.$$

Példa: Hány olyan 8 hosszúságú bitsorozat szerkeszthető, amely vagy '1'-el kezdődik, vagy '00'-val végződik? Válasz:

- A 8 hosszúságú '1'-gyel kezdődő bitsorozatok száma: $2^7 = 128$.
- A 8 hosszúságú '00'-val végződő bitsorozatok száma: $2^6 = 64$.
- A fenti két bitsorozat között lesznek olyan bitsorozatok amelyek '1'-gyel kezdődnek, ugyanakkor pedig '00'-val végződnek, ezek lesznek a közös elemek, ezek száma: $2^5 = 32$.
- A megfelelő bitsorozatok száma: $128 + 64 - 32 = 160$.

3.10. algoritmus. Írjunk egy Python függvényt, amely meghatározza, hogy melyek azok a bitsorozatok, amelyek az előző példában megfogalmazott szabály szerint generálhatók ki. Határozzuk meg a generált lista elemszámát.

```
def szamol4(b):
    L = '01'
    for k in range(b - 1):
        L = [b + X for X in L for b in '01']
    L = [X for X in L if X[0] == '1' or X[-2:] =='00']
    return L

>>> len(szamol4(8))
    160
```

A `for` ciklusban létrehozzuk az összes lehetséges `b` hosszúságú bitsorozatot, a `for` ciklus után pedig ezek közül kiválogatjuk azokat a sorozatokat, amelyek vagy 1-es bittel kezdődnek, vagy két nullás bitben végződnek.

Az **osztás szabálya** azt jelenti, hogy egy feladat akkor lesz n/d-féleképpen megoldható, ha az `n` megoldásszám közül minden m megoldás esetében d lesz azon megoldások száma, amelyek az m megoldásnak felelnek meg.

Példa: Hányféleképpen ültethető le n személy egy kerek asztalhoz, ha azt az ültetést egyformának tekintjük, amikor minden embernek ugyanaz a bal és jobb szomszédja? Például ha 4 személyt szeretnénk leültetni, és a székeket 1-től 4-ig számozzuk, akkor az $1234, 2341, 3412, 4123$ ültetések ugyanazokat az ültetéseket fogják jelenteni. Válasz:

- az egyik széket kitüntetettnek fogjuk tekinteni, ez lesz az első hely.
- 4-féleképpen választható ki egy személy az első helyre.
- 3-féleképpen választható ki egy személy a második helyre.
- 2-féleképpen választható ki egy személy a harmadik helyre.
- 1-féleképpen választható ki egy személy a negyedik helyre.
- összesen $4! = 24$ sorrend lehetséges, és ezekből ki kell szűrni az egyformának számító ültetéseket.
- $\frac{4!}{4} = \frac{24}{4} = 6$ lesz a lehetséges megoldások száma, mert az első hely kiválasztása, amely négyféleképpen lehetséges, tulajdonképpen egyforma ültetést eredményez.
- n személy $\frac{n!}{n} = (n-1)!$-féleképpen ültethető le.

3.11. algoritmus. Írjunk egy Python függvényt, amely meghatározza, hogy melyek azok az ültetések, amelyek az előző példában megfogalmazott szabály szerint generálhatók ki. Határozzuk meg a generált lista elemszámát.

```
def szamol5(D = '1234'):
    L = [d for d in D]
    for k in range(len(D)-2):
        L = [d + X for X in L for d in D if d not in X]
    L = [D[0] + X for X in L if D[0] not in X]
    return L

>>> len(szamol5('1234567'))
    720
>>> szamol5('1234567')[100 : 110]
   ['1546372', '1456372', '1653472', '1563472', '1635472',
   '1365472', '1536472', '1356472', '1643572', '1463572']
```

Számos számlálási feladat megoldásszámát úgy tudjuk meghatározni, ha a megadott szabályokat kombináljuk. A következő feladat megoldásszámának meghatározásához az összeadás és szorzás szabályát is kell alkalmazni.

Példa: Ha egy weboldalra való bejelentkezéshez egy felhasználó 6, 7, 8 vagy 9 karakterből álló jelszót kell megadjon, ahol egy jelszóban a 10 lehetséges számjegy, vagy az angol ábécé kiss és nagybetűi szerepelhetnek, akkor a következőképpen határozhatjuk meg a lehetséges jelszavak számát:

- megállapítható, hogy összesen $10 + 26 + 26 = 52$ szimbólum közül tud választani a felhasználó,
- 6-os hosszúságú jelszóból $52^6 = 19,770,609,664$ van,
- 7-es hosszúságú jelszóból $52^7 = 1,028,071,702,528$ van,
- 8-as hosszúságú jelszóból $52^8 = 53,459,728,531,456$ van,
- 9-es hosszúságú jelszóból $52^9 = 2,779,905,883,635,712$ van.

Tehát összesen $52^6 + 52^7 + 52^8 + 52^9 = 2,834,413,454,479,360$ jelszó közül választhat a felhasználó.

Példa: Az internet számítógépek sokaságának kapcsolata, ahol minden számítógépnek egy egyedi címe van: Internet Address. Az Internet Protokoll 4-es verzióban (IPv4) egy címet 32 biten tárolunk, és aszerint, hogy mekkora a halózat, különböző bitméretű lesz a netid-, illetve hostid-érték. A Class A típusú címeket nagy kiterjedésű hálózatokban használják, a Class B típusúakat közepes, míg a Class C típust kisméretű hálózatokban használják. Lehetséges még Class D és Class E típusú címeket is generálni, ahol a Class

D-t multicasting esetében használják, azaz amikor egyszerre több számítógépet szükséges egy időben megcímezni, a Class E típus pedig tartalékként van fenntartva.

A címek megadásakor a következő szabályok érvényesek:

1. Class A: '0' + 7 bit netid + 24 bit hostid,
2. Class B: '10' + 14 bit netid + 16 bit hostid,
3. Class C: '110' + 21 bit netid + 8 bit hostid,
4. a Class A esetében a netid értéke nem lehet '1111111',
5. egyetlen csoport esetében sem lehetséges az, hogy a hostid csupa egyesekből, illetve csupa nullásokból álló bitsor legyen,
6. a cím vagy Class A, vagy Class B, vagy Class C típusú lehet.

A fenti szabályokat figyelembe véve hány különböző IPv4 cím adható meg? Válasz:

Az 1., 4., 5. szabályok alapján a Class A címek száma:

$$(2^7 - 1) \cdot (2^{24} - 2) = 2,130,706,178.$$

A 2., 5. szabályok alapján a Class B címek száma:

$$2^{14} \cdot (2^{16} - 2) = 1,073,709,056$$

A 3., 5. szabályok alapján pedig a Class C címek száma:

$$2^{21} \cdot (2^8 - 2) = 532,676,608.$$

A 6. szabály alapján összesen tehát $2,130,706,178+1,073,709,056+532,676,608 = 3,737,091,842$ IPv4 cím van.

3.3. Kombinatorikai alapalgoritmusok

Számos feladat úgy oldható meg, ha megtaláljuk egy adott halmaz elemeinek egy bizonyos szabály szerinti elrendezési módját. A fejezet azokat a tipikus algoritmusokat mutatja be, amelyek az ilyen jellegű feladatokra adnak megoldást [8, 12, 15].

Példa: Hányféleképpen tudunk öt könyvet feltenni egy polcra, illetve melyek lesznek ezek az elrendezések? A válasz: 5! = 120-féleképpen, az elrendezéseket pedig egy Python függvénnyel fogjuk kigenerálni.

A példa a középiskolából is ismert permutációk meghatározását kéri tőlünk, amikor is $n = 5$.

3.1. értelmezés. Egy n elemű halmaz elemeinek egy **permutációjának** a meghatározása azt jelenti, hogy a sorrendet szem előtt tartva kiválasztjuk a halmaz összes elemét Egy n elemű halmaz egymástól különböző permutációinak száma $n!$ lesz.

3.12. algoritmus. Írjunk egy Python függvényt, amely meghatározza az $\{1, 2, \ldots, n\}$ elemek összes permutációját.

```
def permutacio(D = '12345'):
    L = ['']
    for k in range(len(D)):
        #print(L)
        L = [X + elem for X in L for elem in D
                 if elem not in X]
    return L

>>> permutacio('abc')
    ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> len(permutacio())
    120
```

A `permutacio` a paraméterként megadott karakterlánc elemeit tekinti halmazelemeknek, és meghatározza a karakterek permutációit. Először előállítjuk az összes egyelemű karakterláncot, majd ez alapján előállítjuk az összes lehetséges kételemű karakterláncot, majd a kételemű karakterláncok alapján a háromeleműeket, és így tovább. Egy új karakter hozzáadását csak abban az esetben végezzük, ha az nem szerepel a már kigenerált karakterláncban. A függvényben `print`-tel nyomon lehet követni az `L` listának a bővítési folyamatát.

Példa: Öt sportoló között hányféleképpen osztható ki az első, második, illetve harmadik hely? Melyek lesznek ezek a kiosztások? A válasz: $5 \cdot 4 \cdot 3 = 60$, a kiosztásokat pedig egy Python függvénnyel fogjuk kigenerálni.

A példa a középiskolából is ismert n elem m-ed rendű variációjának a meghatározását kéri tőlünk, amikor is $n = 5$ és $m = 3$.

3.2. értelmezés. Egy n elemű halmaz elemeinek egy m-ed rendű **variációjának** a meghatározása azt jelenti, hogy a sorrendet szem előtt tartva kiválasztjuk a halmaz m különböző elemét. Egy n elemű halmaz m-ed rendű variációinak a száma: $n \cdot (n-1) \cdot \ldots \cdot (n-m+1)$ lesz.

3.13. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy n elemű halmaz m-ed rendű variációit.

```
def variacio(D, m):
    L = ['']
    for k in range(m):
        L = [X + elem for X in L for elem in D
                 if elem not in X]
    return L

>>> variacio('abc', 2)
    ['ab', 'ac', 'ba', 'bc', 'ca', 'cb']
>>> len(variacio('abcde', 3))
    60
```

A `variacio` függvény a `permutacio` függvénynek egy módosított változata, amelyen csak annyi változtattunk, hogy bevezettük az m-et második függvényparaméternek, és módosítottuk a `for` ciklus lépésszámát, amely most az m értékétől függ.

Példa: Négy diák közül hány fajta 3 diákból álló diákképviseleti testületet lehet létrehozni? Melyek lesznek ezek a testületek? A válasz:

$$\frac{4!}{3! \cdot (4-3)!} = 4,$$

a testületeket pedig egy Python függvénnyel fogjuk kigenerálni.

A példa a középiskolából is ismert n elem m-ed rendű kombinációjának a meghatározását kéri tőlünk, amikor is $n = 4$ és $m = 3$.

3.3. értelmezés. Egy n elemű halmaz elemeinek egy m-ed rendű **kombinációjának** a meghatározása azt jelenti, hogy a sorrendet figyelmen kívül hagyva kiválasztjuk a halmaz m különböző elemét. Egy n elemű halmaz m-ed rendű kombinációinak a száma: $\dfrac{n!}{m! \cdot (n-m)!}$ lesz.

3.14. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy n elemű halmaz m-ed rendű kombinációit.

```
def kombinacio(D, m):
    L = ['']
    for k in range(m):
        L = [X + elem for X in L for elem in D
```

```
                if tesztKomb(elem, X)]
    return L

def tesztKomb(elem, X):
    return all(elem > x for x in X)

def tesztKomb_(elem, X):
    for x in X:
        if elem <= x: return False
    return True

>>> tesztKomb('e', 'abcf')
    False
>>> kombinacio('abcd', 3)
    ['abc', 'abd', 'acd', 'bcd']
```

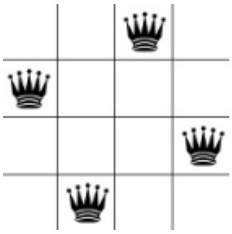
A korábban megadott `permutacio` függvénynél alkalmazott gondolatmenet alapján jártunk el. A bemeneti karakterlánc elemszáma fogja az n értékét jelenteni. A `tesztKomb_`-ban, illetve annak egy tömörebb változatában a `tesztKomb`-ban azt vizsgáljuk, hogy az `elem` hozzáadható-e az `X`-hez. Mindkét függvény kimenete akkor lesz `True`, ha az `elem` kisebb vagy egyenlő lesz a bemeneti karakterlánc minden egyes eleménél, ez azt fogja jelenteni, hogy az `X` elemei szigorúan növekvő sorrendben fognak előállni. Ha a logikai kifejezéseknél megcseréljük az egyenlőtlenség irányát, akkor csökkenő sorrendet kapunk.

Ha azt szeretnénk, hogy egész számokból álló listelemeken is kombinációt számoljon a függvény, akkor módosítanunk kell a `kombincio` függvényt:

```
def kombinacioL(D, m):
    L = [[]]
    for k in range(m):
        L = [X + [elem] for X in L for elem in D
                if tesztKomb(elem, X)]
    return L

>>> kombinacioL([1,2,3,4], 3)
    [[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]
```

Példa: Hogyan tudunk elhelyezni egy sakktáblán 8 királynőt úgy, hogy azok ne üssék egymást? Akkor mondjuk, hogy két királynő nem üti egymást, ha nincsenek ugyanabban a sorban, oszlopban, illetve átlón. A válasz: 92, az elhelyezéseket pedig egy Python függvénnyel fogjuk kigenerálni.

A feladat általánosan is megfogalmazható, amikor is m királynőt kell elhelyezni egy $m \times m$-es sakktáblán úgy, hogy azok ne üssék egymást [18]. A megoldás listák kigenerálását fogja jelenti, ahol a listaelemek egész számok lesznek, ahol az első listaelem az első sorban levő királynő oszloppozícióját fogja jelenteni, a második listaelem a második sorban levő királynő oszlop-pozícióját fogja jelenti, és így tovább. Egy 4×4-es sakktáblán a **`[3, 1, 4, 2]`** helyes megoldás azt fogja jelenteni, hogy a királynőt az első sorban a harmadik, a második sorban az első, a harmadik sorban a negyedik és a negyedik sorban a második oszlopba tettük:



Azzal, hogy megoldásnak a listaelemek ilyen formáját választjuk, megoldottuk a sorok szerinti ütközés problémáját. Megállapítható az is, hogy ha egy listában a kigenerált listaelemek különbözni fognak egymástól, akkor a királynők oszlop szerint sem fogják ütni egymást.

3.15. algoritmus. Írjunk egy Python függvényt, amely meghatározza az összes olyan elhelyezést, ahogyan m **királynőt** egy $m \times m$-es sakktáblára fel lehet tenni úgy, hogy azok ne üssék egymást.

```
def kiralyno(m):
    D = [str(d) for d in range(1,m+1)]
    ind, L = 0, ['']
    for k in range(m):
        L = [X + elem for X in L for elem in D
                if tesztKir(elem, X, ind)]
        ind += 1
    return L

def tesztKir(elem, X, ind):
    for x in X:
        if elem == x: return False
        if abs(int(elem) - int(x)) == ind: return False
        ind -= 1
    return True
```

```
>>> kiralyno(6)
    ['246135', '362514', '415263', '531642']
```

A korábban megadott `permutacio, kombinacio` függvényeknél alkalmazott gondolatmenet alapján jártunk most is el. A `kiralyno` függvényben meghívásra kerülő `tesztKir` függvényben vizsgáljuk, hogy az `elem` hozzáadható-e az `X` listához úgy, hogy a feladat feltételei teljesüljenek. Az átlók menti ütközés kizárását a `tesztKir`-ben az

```
if abs(int(elem) - int(x)) == ind: return False
```

sor biztosítja, ahol az `abs` függvény a paraméter abszolút értékét adja meg. Az `X` listában kigenerált `x` elemek közül az fog az `elem`-mel az átlók mentén ütközni, amely `ind` pozícióira helyezkedik el az `x`-től. Az `ind` kezdeti értékét a `kiralyno`-ben adjuk meg, ez 0 lesz. A `kiralyno` `for` ciklusában az `ind` értéke 1-esével nő, aszerint, hogy hányadik elemet tesszük be az `X` listába, a `tesztKir`-ben a `for`-ban pedig csökken. Például a 6×6-os sakktábla esetében az átlós ütközés miatt, amikor `ind = 4`, azaz a `4`-dik elemet kell elhelyezni és `X = [2,6,1,3]` akkor fenn kell álljon, hogy:

`abs(elem-2)` nem egyenlő `ind = 4`-gyel,
`abs(elem-6)` nem egyenlő `ind = 3`-mal,
`abs(elem-1)` nem egyenlő `ind = 2`-vel,
`abs(elem-3)` nem egyenlő `ind = 1`-gyel.

Tehát az `[2,6,1,3]` lista végére

- nem lehet `1, 2, 3` vagy `6`-osokat tenni az oszlopütközések miatt,
- nem lehet **`4`**-est tenni, mert `abs(`**`4`**` - 3) = 1`,
- lehet **`5`**-est tenni, mert:
  `abs(`**`5`**` - 2)` $\neq$ `4`,
  `abs(`**`5`**` - 6)` $\neq$ `3`,
  `abs(`**`5`**` - 1)` $\neq$ `2`,
  `abs(`**`5`**` - 3)` $\neq$ `1`.

Példa: Hányféleképpen lehet sorba rendezni a *sapientia* szót?

Vegyük észre, hogy a *sapientia* szó 9 betűből áll, az *a* és *i* betűk pedig kétszer is előfordulnak. Az elrendezéseket egy Python függvénnyel fogjuk kigenerálni, a megoldásszámot pedig a következő képlet adja:

$$\frac{9!}{2! \cdot 2!} = 90720.$$

A példa az ismétléses permutációk meghatározását kéri tőlünk.

3.4. értelmezés. n darab nem feltétlenül egyforma elem egy **ismétléses permutációjának** a meghatározása azt jelenti, hogy a sorrendet szem előtt tartva kiválasztjuk az összes elemet. Feltételezve, hogy m különböző elem áll a rendelkezésünkre, ahol az első elemből k_1 darab az egyforma, a másodikból k_2, és így tovább, míg az m-edik elemből k_m darab az egyforma, akkor a megoldásszámot a következő képlet szerint határozhatjuk meg:

$$\frac{n!}{k_1! \cdot k_2! \cdot ... \cdot k_m!}.$$

Példa: Az `1,2,3,3` elemek ismétléses permutációi, ahol mondjuk az 1-et egyszer, a 2-t egyszer, a 3-at pedig kétszer vehetjük, a következők:

`1233, 1323, 1332, 2133, 2313, 2331,3123, 3132, 3213, 3231, 3312, 3321`

A megoldásszám pedig:

$$\frac{4!}{1! \cdot 1! \cdot 2!} = 12.$$

3.16. algoritmus. Írjunk egy Python programot, amely meghatározza adott elemeknek az összes ismétléses permutációját.

Először próbáljuk ki a `set`, a `count` és az `enumerate` metódusokat, használni fogjuk őket az implementációban:

```
>>> set('sapientia')
    {'e', 'a', 't', 's', 'n', 'i', 'p'}
>>> 'sapientia'.count('a')
    2
>>> D = 'p', 'e', 's', 't', 'i', 'n', 'a'
>>> for i,elem in enumerate(D):
        print(i, elem, end = ',  ')
    0 p,  1 e,  2 s,  3 t,  4 i,  5 n,  6 a,
```

A feladat megoldásához három függvényt fogunk írni:

1. Az `ismPermutacio` a bemenetét átalakítja `set` típussá, eredményként a `D`-be a többször előforduló elemek csak egyszer fognak szerepelni. A `count`-ot alkalmazva az `S`-be meghatározza minden `D`-beli elem `L`-beli előfordulási számát.
2. Az `ismPermutacioAux` a `D, S, n` bemenetek alapján meghatározza az `L` elemeinek az összes lehetséges elrendezési módját.

3. A `tesztIsmPerm` függvény teszteli, hogy melyek azok az `elem`-ek, amelyeket hozzá lehet fűzni az `X`-hez.

```
def ismPermutacio(L):
    D = set(L)
    S = [L.count(elem) for elem in D]
    n = len(L)
    P = ismPermutacioAux(D, S, n)
    return P

def ismPermutacioAux(D = '123', S = [1,1,2], n = 4):
    L = ['']
    for k in range(n):
        L = [X + elem for X in L
                      for i,elem in enumerate(D)
                      if tesztIsmPerm(elem, X, S, i) ]
    return L

def tesztIsmPerm(elem, X, S, i):
    #print(elem, X)
    db = X.count(elem)
    if db < S[i]: return True
    return False


>>> ismPermutacioAux()
    ['1233','1323','1332','2133','2313','2331','3123',
    '3132','3213','3231','3312','3321']
>>> ismPermutacio('1323')
    ['3321','3312','3231','3213','3132','3123','2331',
    '2313','2133','1332','1323','1233']
>>> len(ismPermutacio('sapientia'))
    90720
```

Példa: Hányféle ötkarakteres szót lehet alkotni az `a, b, c` karakterekből, ahol a szóban egy karakter többször is előfordulhat? Az elrendezéseket egy Python függvénnyel fogjuk kigenerálni, a megoldásszám pedig $3^5 = 243$.

A példa az ismétléses variációk meghatározását kéri tőlünk.

3.5. értelmezés. n különböző elem k tagú **ismétléses variációjának** a meghatározása azt jelenti, hogy a sorrendet szem előtt tartva kiválasztunk k elemet, ahol egy elem többször is kiválasztható. Az összes ismétléses variációszám egyenlő n^k-val.

Példa: Az `a,b,c` elemek öttagú ismétléses variációi következők:

```
aaaaa, aaaab, aaaac, aaaba, aaabb, aaabc, aaaca,
aaacb, aaacc, aabaa, aabab, aabac, aabba, aabbb,
...
```

3.17. algoritmus. Írjunk egy Python függvényt, amely meghatározza n különböző elem k tagú ismétléses variációit.

```
def ismVariacio(D, m):
    L = ['']
    for k in range(m):
        L = [X + elem for X in L for elem in D]
    return L

>>> len(ismVariacio('abc', 5))
    243
```

A variációkat meghatározó 3.13. algoritmushoz képest az `ismVariacio`-ban elhagytuk a listakifejezésből az `if` feltételt.

Példa: Hányféle ötszálú virágcsokrot lehet készíteni az `a, b, c` fajta virágokból, ahol egy csokorba ugyanolyan fajta virágok is kerülhetnek? Az elrendezéseket egy Python függvénnyel fogjuk kigenerálni, a megoldásszám pedig

$$\frac{(5+3-1)!}{5!\cdot(3-1)!}=\frac{7!}{5!\cdot 2!}=21.$$

A példa az ismétléses kombinációk meghatározását kéri tőlünk.

3.6. értelmezés. n különböző elem k tagú **ismétléses kombinációjának** a meghatározása azt jelenti, hogy a sorrendtől eltekintve kiválasztunk k elemet, ahol egy elem többször is kiválasztható. Az ismétléses kombinációk száma a következő képlettel adható meg:

$$\frac{(k+n-1)!}{k!\cdot(n-1)!}.$$

.

Példa: Az `a,b,c` elemek öttagú ismétléses kombinációi a következők:

```
aaaaa, aaaab, aaaac, aaabb, aaabc, aaacc
abbbc, abbcc, abccc, acccc, bbbbb, bbbbc
bbbcc, bbccc, bcccc, ccccc
```

3.18. algoritmus. Írjunk egy Python függvényt, amely meghatározza n különböző elem k tagú ismétléses kombinációit.

```
def ismKombinacio(D, m):
    L = ['']
    for k in range(m):
        L = [X + elem for X in L for elem in D
                 if tesztIsmKomb(elem, X)]
    return L

def tesztIsmKomb(elem, X):
    return all(elem >= x for x in X)

>>> len(ismKombinacio('abc', 5))
    21
```

A kombinációkat meghatározó 3.14. algoritmushoz képest a feltételt megadó függvényben kellett módosítást végezni. A `tesztIsmKomb`-ben megengedjük, hogy egy `elem` egyenlő legyen valamely `X`-beli elemmel.

3.19. algoritmus. Írjunk egy Python függvényt, amely meghatározza a Pascal-háromszög első n sorát.

A **Pascal-háromszög** a binomiális együtthatók háromszög formában való rendezését jelenti, ahol a binomiális együttható azt a pozitív egész számot jelenti, amely az $(1 + x)^n$ polinom x^k tagjának az együtthatója. Ez az érték tulajdonképpen egyenlő lesz n elem k-ad rendű kombinációinak a számával [20]. A Pascal-háromszög kiíratása során az n értéke a sor, míg a k értéke az oszlop értékét jelöli, és fennáll: $n \geq k \geq 0$. A binomiális együttható értékét direkt módon a következő képlettel is meg tudjuk határozni:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

A binomiális együtthatók értékét azonban meg lehet határozni úgy is, hogy az $(n-1)$-edik sorban levő értékekből generáljuk az n-edik sorban levő értékeket, ahol a nulladik sorban az 1 érték található, amelyet $\binom{0}{0}$-val jelölünk. A bemutatásra kerülő algoritmus ezt a gondolatmenetet követi: kiindulunk az `L = [1]` listából, ebből előállítjuk az `L = [1,1]`, majd ez alapján az `L = [1,2,1]`, majd megint ez alapján az `L = [1,3,3,1]` stb. listákat. Ennek érdekében az `L` listából létrehozzuk az `L1` listát:

az L1 első elemét 1-re inicializáljuk, a többi elemet pedig az L listában levő egymás melletti két elem összeadásával kapjuk. Végül az L1 lista utolsó elemét is 1-re állítjuk. Ezután az L értékét felülírjuk az L1 értékével, és folytatjuk az iterációt. Például ha L = [1,3,3,1] akkor L1 = [1] + [1+3,3+3,3+1] + [1] = [1,4,6,4,1] lesz a következő L.

```
def pascalT(n):
    L = [1]
    print(L)
    for i in range(n):
        tempL = [1]
        tempL += [e1 + e2 for (e1, e2) in zip(L, L[1:])]
        tempL += [1]
        print(tempL)
        L = tempL
```

```
>>> pascalT(15)
    [1]
    [1, 1]
    [1, 2, 1]
    [1, 3, 3, 1]
    [1, 4, 6, 4, 1]
    ...
```

A pascalT függvényben a zip függvényt alkalmaztuk, hogy az L és L[1:] listákat összezipzárazzuk, eredményként egy tuple elemtípusú listát határozva meg. Próbáljuk ki a zip-et a shellben:

```
>>> L = [1,3,3,1]
>>> list(zip(L, L[1:]))
    [(1, 3), (3, 3), (3, 1)]
```

Egy B halmazt az A halmaz egy részhalmazának nevezünk, ha B minden eleme ugyanakkor eleme A-nak is. B tartalmazhatja A minden egyes elemét, de lehet üres halmaz is, amikor A egyetlenegy elemét sem tartalmazza. Az üres halmaz tulajdonképpen minden halmaznak a részhalmaza. Számos esetben fontos annak a kérdésnek a megválaszolása, hogy egy halmaznak hány részhalmaza van.

Példa: Legyen A = [1,2,3], az előállítható részhalmazok halmaza ekkor H = [[], [1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]. Az A halmaz részhalmazainak száma $2^3 = 8$. Általános esetben pedig: 2^n [20].

3.20. algoritmus. Írjunk egy Python függvényt, amely meghatározza az `A` halmaz összes részhalmazát.

Legyen `A = [1,2,3]`, ekkor az algoritmus lépései a következők:

1. a `H` lista első elemét üres halmazként inicializáljuk: `H = [[]]`
2. `H` minden egyes eleméhez hozzáfűzzük az `A` lista **első** elemét, és az így kapott elemmel kibővítjuk a `H`-t:

   `H = [[],[1]]`

3. `H` minden egyes eleméhez hozzáfűzzük az `A` lista **második** elemét, és az így kapott elemekkel kibővítjuk a `H`-t:

   `H = [[],[1],[2],[1,2]]`

4. `H` minden egyes eleméhez hozzáfűzzük az `A` lista **harmadik** elemét, és az így kapott elemekkel kibővítjuk a `H`-t:

   `H = [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`.

```
def reszhalmazok(A):
    H = [[]]
    for elem in A:
        H += [X + [elem] for X in H]
    return H

>>> reszhalmazok([1,2,3])
    [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

3.4. Kitűzött feladatok

3.1. feladat. A *jelszavakE.txt* állomány minden sorában két érték található szóközökkel elválasztva. Az első érték egy személy nevét jelölő karakterlánc, a második egy jelszót jelölő karakterlánc. Írjunk egy Python programot, amely megállapítja, hogy kinek van erős jelszava. Egy személynek akkor mondjuk, hogy erős a jelszava, ha a számjegyek és az angol ábécé kis- és nagybetűi mellett tartalmaz egyéb szimbólumokat is.

3.2. feladat. Írjunk egy Python programot, amely megvizsgálja egy adott bemenetre, hogy az lehet-e valamely IPv4 típusú cím.

3.3. feladat. Írjunk egy Python programot, amely kiírja egy állományba n elem lehetséges permutációit.

3.4. feladat. Írjunk egy Python programot, amely kiírja egy állományba n elem m-ed rendű lehetséges variációit.

3.5. feladat. Írjunk egy Python programot, amely kiírja egy állományba n elem m-ed rendű lehetséges kombinációit.

3.6. feladat. Írjunk egy Python programot, amely megvizsgálja egy adott bemenetre, hogy az egyenlő-e n elem permutációjával vagy sem. Bemenetként adjuk meg az n értékét és a listabeli elemeket, amelyekből a permutációk előállíthatók.

3.7. feladat. Írjunk egy Python programot, amely megvizsgálja egy adott bemenetre, hogy az egyenlő-e n elem valamelyik m-ed rendű variációjával vagy sem. Bemenetként adjuk meg az n, az m értékeit és a listabeli elemeket, amelyekből a variációk előállíthatók.

3.8. feladat. Írjunk egy Python programot, amely megvizsgálja egy adott bemenetre, hogy az egyenlő-e n elem valamelyik m-ed rendű kombinációjával, vagy sem. Bemenetként adjuk meg az n, az m értékeit és a listabeli elemeket, amelyekből a kombinációk előállíthatók.

3.9. feladat. Írjunk egy Python programot, amely megvizsgálja egy adott bemenetre, hogy az lehet-e az $n \times n$ királynő feladat valamelyik megoldása.

3.10. feladat. Írjunk egy Python programot, amely kiírja a Pascal-háromszög első n sorát egy szövegállományba, háromszög formában.

3.11. feladat. Írjunk egy Python programot, amely kiírja egy szövegállományba egy n elemű halmaz összes részhalmazát. Az első sorba az üres halmazt, a következőbe az egyelemű részhalmazokat, majd a következőbe a kételemű részhalmazokat és így tovább írjuk.

3.12. feladat. Írjunk egy Python programot, amely kiírja egy szövegállományba az angol ábécé első n betűjéből képezhető m hosszúságú szavakat, ahol nem megengedett a betűk többszöri felhasználása, és nem számít a betűk sorrendje. Formázzuk úgy a szövegállomány tartalmát, hogy minden sorba négy szót írjunk, és a szavak közé pedig tegyünk szóközöket.

4. fejezet

Számok, számtartományok

A számok, a különböző számtartományok tulajdonságainak feltérképezése és az ebből levonható következtetések hozták létre a matematikát és annak különböző szakterületeit. A számokat tulajdonságaik alapján osztályozhatjuk, így megkülönböztetünk természetes számokat, egész számokat, racionális számokat, irracionális számokat, valós számokat, komplex számokat, algebrai számokat és transzcendens számokat. A számokkal végzett alapműveletek az összegzés, a különbség, a szorzás, az osztás, a hatványozás és a logaritmálás. A számítástechnikai algoritmusok mindegyike visszavezethető számokkal végzett műveletekre, éppen ezért ebben a fejezetben a különböző számtartományokhoz kapcsolódó alapalgoritmusok kerülnek bemutatásra [27].

4.1. Természetes számok

A természetes számok halmazát $\mathbb{N}$-nel jelöljük, és elemeit a következőképpen adhatjuk meg: $\{0, 1, 2, 3, ...\}$. A természetes számokkal végzett műveletek tulajdonságai a következők:

1. az összeadás és a szorzás **kommutatív** művelet, ami azt jelenti, hogy bármely $a, b \in \mathbb{N}$ számokra igaz a következő:

$$a + b = b + a, \qquad a \cdot b = b \cdot a, \tag{4.1}$$

2. az összeadás és a szorzás **asszociatív** művelet, ami azt jelenti, hogy bármely $a, b, c \in \mathbb{N}$ számokra igaz a következő:

$$\begin{aligned}(a+b)+c &= a+(b+c),\\ (a\cdot b)\cdot c &= a\cdot(b\cdot c),\end{aligned} \tag{4.2}$$

3. az összeadás a szorzásra nézve **disztributív** művelet, mert fennáll, hogy bármely $a, b, c \in \mathbb{N}$ számok esetében:

$$(a+b)\cdot c = a\cdot c + b\cdot c, \tag{4.3}$$

4. a természetes számok halmaza **zárt** az összeadásra, szorzásra nézve, ami azt jelenti, hogy bármely két természetes szám úgy adható, szorozható össze, hogy közben az eredmény természetes szám lesz; ez nem igaz a különbség és osztás műveletekre,
5. a 0 az összeadásra, míg az 1 a szorzásra nézve lesz a **semleges elem**,
6. a természetes számok minden nem üres részhalmazának van egy legkisebb eleme, ez azt jelenti, hogy a halmaz **jól rendezett**.

A természetes számokkal végzett műveletek és a hozzájuk kapcsolódó algoritmusok a diszkrét matematika gerincét alkotják, éppen ezért a továbbiakban olyan algoritmusokat mutatunk be, amelyek természetes számokon végeznek különböző műveleteket [28, 27].

4.1. algoritmus. Írjuk egy Python függvényt, amely meghatározza, hány nullás számjegy van `n!` végén, anélkül, hogy kiszámolnánk `n!` értékét.

Az algoritmus gondolatmenete a következő: egy szám akkor végződik nullás számjegyben, ha osztható 10-zel, azaz osztható 2-vel és 5-tel. A 2-vel való oszthatóságot azonban nem szükséges megvizsgálni, mert `n`-ig minden második szám páros. Elegendő, ha megszámoljuk, hogy hány 5-tel, 5^2-tel, 5^3-nal stb. osztható szám van `n`-ig.

Példa: Ha `n` = 91, akkor 18 + 3 = 21 nullás számjegy van 91! végén, mert
- az 5-tel osztható számok száma: 91 / 5 = 18,
- az 5^2-tel osztható számok száma: 18 / 5 = 3, ez ugyanaz, mintha meghatároznánk 91 / 5^2-t,
- az 5^3-mal osztható számok száma: 3 / 5 = 0, ez ugyanaz, mintha meghatároznánk 91 / 5^3-t.

Az 5-tel való oszthatóságot a `//` operátorral végezzük, mert osztási egészrészekre van szükségünk. A feladat megoldására két egymástól alig

különböző függvényt is megadtunk, kiemelve azokat a részeket, ahol eltérnek egymástól a függvények:

```
def nullasF(n):
    res = 0
    while n > 0:
        temp = n // 5
        res += temp
        n = temp
    return res

>>> nullasF(1000)
    249
```

```
def nullasF_(n):
    res = 0
    while n > 0:
        res += n // 5
        n = n // 5
    return res
```

A `nullasF_` függvény többet számol, hiszen minden körben kétszer is kiszámolja az `n` 5-tel való osztási egészrészét. A `nullasF` függvény ezt az értéket minden körben csak egyszer számolja ki, mert a `temp` változóba eltárolja ezt az értéket, ezért több memóriát használ. A fenti megoldások esetében ezeknek nincsen különösebb jelentősége, de rávilágítanak arra a programozói dilemmára, amely abban áll, hogy több memóriát igénylő, de gyors, vagy kevesebb memóriát igénylő, de lassúbb kódot írjunk.

4.2. algoritmus. Írjunk egy Python függvényt, amely meghatározza x^n értékét, ahol az alap és a kitevő is természetes számok.

A faktoriális feladathoz hasonlóan itt is többféle megoldás létezik. Használhatjuk a Python `**` operátorát vagy a `pow` függvényt, ahogy azt a Python-bevezetőben láttuk, ha azonban a háttérben végbemenő számításokra vagyunk kíváncsiak, akkor meg kell írni az algoritmust.

A legkézenfekvőbb megoldás az lenne, ha `n`-szer összeszoroznánk az `x`-et. Ezt a gondolatmenetet követi a `myPowL` Python függvény:

```
def myPowL(x, n):
    res = 1
    for i in range(n):
        res *= x
    return res

>>> myPowL(2, 10)
    1024
```

Ennél természetesen létezik sokkal gyorsabb algoritmus is, amikor nem `n` szorzást végzünk, hanem kevesebbet. Kiindulásképpen vegyük észre, hogy

3^{100}-on egyenlő a következő kifejezéssel, ahol figyeljük meg, hogy a jobb oldali hatványkitevők mindegyike 2 egyik hatványa:

$$\begin{aligned} 3^{\mathbf{100}} &= 3^{\mathbf{4}} \cdot 3^{\mathbf{32}} \cdot 3^{\mathbf{64}} \\ 100 &= 4 + 32 + 64 = 2^2 + 2^5 + 2^6 \end{aligned}$$

Ez azt jelenti, hogy elég, ha meghatározzuk rendre a következő négyzeteket, majd összeszorozzuk azokat, ahol a hatványkitevők összege egyenlő: $4+32+64 = 100$-zal:

$$\begin{aligned} 3^2 &= 3^1 \cdot 3^1 & \qquad 3^{16} &= 3^8 \cdot 3^8 \\ \mathbf{3^4} &= 3^2 \cdot 3^2 & \qquad \mathbf{3^{32}} &= 3^{16} \cdot 3^{16} \\ 3^8 &= 3^4 \cdot 3^4 & \qquad \mathbf{3^{64}} &= 3^{32} \cdot 3^{32} \end{aligned}$$

Most már csak az kérdés maradt, hogy hogyan választjuk ki azokat az értékeket, amelyeket össze kell szorozni? A következő táblázatban feltüntettük a hatványkitevő 2-vel való osztási egészrészeit, és kiemeltük azokat a hatványértékeket, amelyeket össze kell szorozni:

100 // 2 =	50	3^2
50 // 2 =	$\mathbf{25}$	$\mathbf{3^4}$
25 // 2 =	12	3^8
12 // 2 =	6	3^{16}
6 // 2 =	$\mathbf{3}$	$\mathbf{3^{32}}$
2 // 2 =	$\mathbf{1}$	$\mathbf{3^{64}}$

Azt a megállapítást tehetjük, hogy abban az esetben, ha a hatványkitevő 2-vel való osztási egészrésze páratlan szám, akkor a négyzetre emelések mellett a hatványértékkel szorzást is kell végezni. Megállapíthatjuk, hogy 3^{100} meghatározásához 6 négyzetre emelést és 3 szorzást kell elvégezni. Ez összesen 9 szorzást jelent a 100 helyett.

A fenti gondolatmenetet követő algoritmust **gyors hatványozásnak** (fast exponentiation) hívjuk. Az algoritmusra szoktak még ismételt négyzetre emelés (repeated-squaring) megnevezéssel is hivatkozni.

4.3. algoritmus. Írjunk egy Python függvényt, amely a gyorshatványozás algoritmusa szerint határozza meg x^n-t.

```
def myPow_(x, n):
    res = 1
```

```
    while n != 0:
        if n % 2 == 1:
            res = res * x
        #print("%20s%20s%20s" % (x, n, res))
        n = n // 2
        x = x * x
    return res

>>> myPow_(3, 100)
    515377520732011331036461129765621272702107522001
```

Vegyük észre, hogy az algoritmus tulajdonképpen 7 négyzetre emelést és 3 szorzást végez, ha `x = 3` és `n = 100`. A következő táblázat az algoritmus változóinak a változását mutatja:

x	n	res
		1
3	100	1
$3^2 = 9$	50	1
$\mathbf{3^4 = 81}$	**25**	**81**
$3^8 = 6561$	12	81
$3^{16} = 43046721$	6	81
$\mathbf{3^{32}} = 1853020188851841$	**3**	**150094635296999121**
$\mathbf{3^{64}} = 343 \ldots 281$	**1**	$\mathbf{515} \ldots 001$
$3^{128} = 117 \ldots 961$	0	

A következő `myPow` függvény eggyel kevesebb négyzetre emelést végez, mert a megállási feltételt módosítottuk.

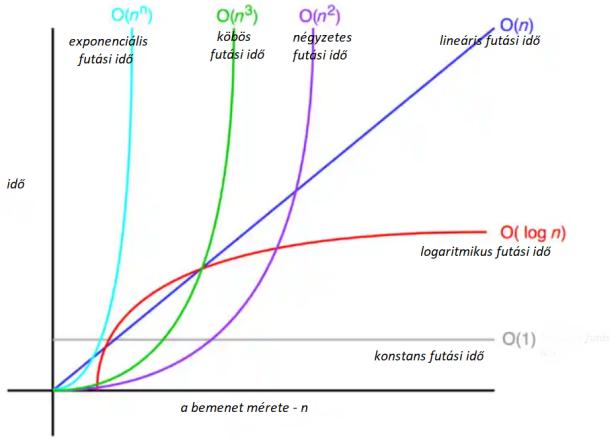
```
def myPow(x, n):
    res = 1
    while True:
        if n % 2 == 1:
            res = res * x
        n = n // 2
        if n == 0: break
        x = x * x
    return res
```

A `while`-ban feltételnek `True`-t adtunk, amely egy végtelen ciklus elindítását eredményezi. Ennek a leállítása a ciklustörzs bármelyik részén megoldható, helyes logikai feltétel megadásával. A `myPow` esetében a második `if`

teljesülése azt fogja eredményezni, hogy a `break` kerül végrehajtásra, azaz ezen a ponton kerül sor a ciklusból való kilépésre. Tulajdonképpen így tudtuk megszakítani a `while`-hoz tartozó műveletek ismételt elvégzését. Mivel a kilépési feltétel a négyzetre emelés előtt foglal helyet, megspóroljuk az utolsó körben sorra kerülő, legnagyobb számon végzett, de már fölösleges négyzetre emelést.

Megállapítható, hogy a `myPow_`, illetve a `myPow` függvények kevesebb számítást végeznek, azaz jobb a futási idejük, mint a `myPowL`-é. A `myPow_`, illetve a `myPow` algoritmusok futási ideje logaritmikus, míg a `myPowL` futási ideje lineáris.

Algoritmusok tervezése során fontos szempont, hogy egy algoritmus futási ideje minél jobb legyen. Pontosabban az a cél, hogy a bemeneti paraméterek értékét növelve, a futási idő minél kisebb mértékben növekedjen. A jegyzet keretén belül nem térünk ki különösebben az algoritmusok futási idejének az elemzésére, azonban a 4.1. ábrán megtekinthetjük a különböző futási idejű algoritmusok közötti kapcsolatot [8, 28].



4.1. ábra

A gyors hatványozás következő két implementációja rekurzív, ahol a lényegi különbség a két algoritmus között abban van, hogy az elsőnél a rekurzív függvényhívás paraméterein végezzük a négyzetre emelést, a második változatnál a rekurzív hívás után határozzuk meg ezt az értéket.

```
def myPowR1(x, n):
    if n == 0: return 1
    if n % 2 == 1: return x * myPowR1(x * x, n // 2)
    return myPowR1(x * x, n // 2)

def myPowR2(x, n):
    if n == 0: return 1
    temp = myPowR2(x, n // 2)
    if n % 2 == 1: return x * temp * temp
    else: return temp * temp
```

A következő feladatban úgy módosítjuk a `myPowR2` függvényt, hogy lehetőségünk legyen megszámolni a négyzetremeléseket, illetve szorzásokat.

4.4. algoritmus. Írjunk egy Python függvényt, amely meghatározza azoknak a szorzásoknak, négyzetre emeléseknek a számát, amelyeket a hatványozás során végzünk.

```
def myPowRSz(x, n):
    if n == 0: return 1, 0
    temp, nr = myPowRSz(x, n // 2)
    if n % 2 == 1: return x * temp * temp, nr + 2
    return temp * temp, nr + 1

def szorSzama(x, n):
    res, nr = myPowRSz(x, n)
    print('Szorzasok szama: ', nr)
    #print(res)
```

A módosítások után a `myPowRSz` függvény visszatérési értéke egy `tuple` típusú érték lesz. A `szorSzama` függvényben a `res` érték tehát a hatványozás eredményét jelöli, a `nr` értéke pedig a szorzások számát adja. Ha a hatványozás eredményét is ki szeretnénk íratni, akkor a `szorSzama` utolsó sora elejéről távolítsuk el a `#`-t.

```
>>> szorSzama(2, 1023)
    Szorzasok szama: 20
>>> szorSzama(2, 1024)
    Szorzasok szama:  12
>>> szorSzama(2, 1025)
    Szorzasok szama:  13
```

A fenti lekérdezések eredményei alapján több kérdést is feltehetünk: miért van ekkora eltérés a szorzások számában, hiszen a kitevőértékek között nincs

akkora különbség? Azt is megfigyelhetjük, hogy van, amikor a nagyobb kitevő esetén kevesebb szorzást végzünk. A választ úgy tudjuk megadni, ha meghatározzuk a kitevők kettes számrendszerbeli alakját. Ehhez használjuk a beépített `bin` függvényt:

```
>>> bin(1023)
    '0b1111111111'
>>> bin(1024)
    '0b10000000000'
>>> bin(1025)
    '0b10000000001'
```

Megfigyelhetjük, hogy az 1023 kettes számrendszerbeli alakjában van a legtöbb 1-es számjegy, és ebben az esetben végezzük a legtöbb szorzást is. Ha újra áttanulmányozzuk az algoritmust, akkor könnyen megállapíthajuk, hogy a szorzások számát az határozza meg, hogy a hatványkitevő kettes számrendszerbeli alakja hány bites (hányszor tudunk 2-vel osztani), illetve hány 1-es bit (hányszor lesz a 2-vel való osztási egészrész 1) szerepel benne.

4.5. algoritmus. Írjunk egy Python függvényt, amely egy listában meghatározza a Fibonacci-számsorozat első `n` tagját.

A Fibonacci[1]-számsorozat a matematika, az informatika, de a mindennapi élet területén is megjelenik. Például algoritmusok futási idejének elemzésénél, számok ábrázolásánál, kombinatorikai feladatoknál, a zene, a különböző művészetek területén, stb. [8, 12, 16, 28].

Definíció szerint a nulladik Fibonacci-szám a 0, az első az 1, a Fibonacci-számsorozat n-edik tagját pedig úgy határozzuk meg, hogy összeadjuk az $n-1$-edik, illetve $n-2$-edik tagot. Ezek szerint a Fibonacci-számsorozat első 11 tagja a következő: $0, 1, 2, 3, 5, 8, 11, 13, 21, 34, 55$.

```
def myFibList(n):
    if n == 0: return [0]
    L = [0, 1]
    for i in range(2, n):
        L += [L[i-1] + L[i-2]]
    return L

>>> myFibList(11)
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

1 Leonardo Fibonacci olasz matematikus (kb. 1170 – kb. 1250)

A `myFibList` első sora egy egyelemű listát térít vissza, amely a nulladik Fibonacci-számot fogja tartalmazni. A következő sorban egy kételemű listát inicializálunk, amelybe a Fibonacci-számsorozat első két tagját tesszük. A `for` keretén belül meghatározzuk a lista további elemeit úgy, hogy a lista végéhez fűzzük az összegzés eredményeként kapott új Fibonacci-számot.

A Fibonacci-számsorozat elemei a következő rekurziós képlettel is meghatározhatóak: $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$, ahol $F_0 = 0, F_1 = 1$. Ez az összefüggés más kezdőértékekkel alkalmazható a Lucas[2]-számsorozat elemei meghatározásához: $L_n = L_{n-1} + L_{n-2}$, ahol $\boldsymbol{L_0 = 2, L_1 = 1}$. Ezek szerint a Lucas-számsorozat első pár eleme a következő: $2, 1, 3, 4, 7, 11, 18, 29, 47, \ldots$.

4.6. algoritmus. Írjunk egy-egy Python függvényt, amelyek különböző rekurzív összefüggéseket alkalmazva meghatározzák az n-dik Fibonacci-számot.

A `fibR1` algoritmus a fent megadott rekurzív képletet alkalmazva határozza meg a n-dik Fibonacci-számot. Egy mai, átlagos számítógépen az algoritmus futási ideje nagyon rossz lesz, már a 30-dik Fibonacci-szám meghatározására is várnunk kell, a 30-nál nagyobb értékek esetében a várakozási idő pedig elfogadhatatlan. Az algoritmus futási ideje exponenciális. A `fibR2` futási ideje már jóval jobb, lineáris. Várakozási idő nélkül határozza meg az 1000-dik Fibonacci-számot:

```
def fibR1(n):
    if n == 0: return 0
    if n == 1: return 1
    return fibR1(n-1) + fibR1(n-2)

def auxFib(a, b, n):
    if n == 0: return a
    return auxFib(b, a + b, n-1)

def fibR2(n):
    return auxFib(0, 1, n)

>>> fibR1(30)
    832040
>>> fibR2(1000)
    434665...228875
```

2 François Édouard Anatole Lucas francia matematikus (1842–1891)

Az `auxFib`-re a kezdőértékek megadása miatt van szükség. Az `auxFib` első és második paramétere a nulladik és az első Fibonacci-számot, a harmadik paramétere pedig a sorszámot jelöli. A Pythonban a függvényparaméterek alapértelmezett kezdeti értékadásának lehetősége miatt az algoritmust egy függvénnyel is meg lehet adni:

```
def fibR3(n, a = 0, b = 1):
    if n == 0: return a
    return fibR3(n-1, b, a + b)
```

Az n-dik Fibonacci-szám meghatározására létezik logaritmikus futási idejű algoritmus is. A feladat visszavezethető a gyorshatványozás algoritmusára, ahol az alap egy speciális 2×2-es mátrix lesz, a kitevő pedig egy egész szám:

$$F_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}, \quad F_{\mathbf{5}} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\mathbf{4}} = \begin{pmatrix} \mathbf{5} & 3 \\ 3 & 2 \end{pmatrix}.$$

A fenti képlet szerint az n-dik Fibonacci-szám egyenlő lesz a hatványozás eredményeként kapott mátrix első sorának első elemével. Például az ötödik Fibonacci-szám meghatározásához az alapot negyedik hatványra kell emelni, és az eredmény a kapott mátrix első sorának első oszlopában lesz. Az algoritmusban szükséges meghatározni a következő szorzatokat:

$$\begin{pmatrix} a & \boldsymbol{b} \\ \boldsymbol{b} & d \end{pmatrix} \cdot \begin{pmatrix} a & \boldsymbol{b} \\ \boldsymbol{b} & d \end{pmatrix} = \begin{pmatrix} a^2 + b^2 & \boldsymbol{a \cdot b + b \cdot d} \\ \boldsymbol{a \cdot b + b \cdot d} & b^2 + d^2 \end{pmatrix},$$

$$\begin{pmatrix} a & \boldsymbol{b} \\ \boldsymbol{b} & d \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} a + b & \boldsymbol{a} \\ \boldsymbol{b + d} & b \end{pmatrix}.$$

Mivel a egyenlő $b + d$-vel, elegendő egy olyan függvényt írni, amely a következő típusú mátrixszorzást végzi:

$$\begin{pmatrix} a & \boldsymbol{b} \\ \boldsymbol{b} & d \end{pmatrix} \cdot \begin{pmatrix} e & \boldsymbol{f} \\ \boldsymbol{f} & g \end{pmatrix} = \begin{pmatrix} a \cdot e + b \cdot f & \boldsymbol{a \cdot f + b \cdot g} \\ \boldsymbol{a \cdot f + b \cdot g} & b \cdot f + d \cdot g \end{pmatrix}$$

A következő `fibF2` függvény hatékonyság miatt mátrix helyett egy számhármassal végzi a megfelelő műveleteket. Nincs értelme ugyanis, hogy a mátrix második sorának első elemével is végezzünk számításokat, hiszen ahogy fent jeleztük, ez megegyezik az első sor második elemével. A két mátrix szorzását a `szor` függvény végzi, ahol a számítások a mátrixszorzás szabályai szerint lettek megadva. A `fibF2` függvény első sorában a `res` változó az egységmátrixnak megfelelő számhármast jelöli.

```
def szor(t1, t2):
    a, b, d = t1
    e, f, g = t2
    return (a * e + b * f, a * f + b * g, b * f + d * g)

def fibF2(n):
    res = (1, 0, 1)
    alap = (1, 1, 0)
    n -= 1
    while n != 0:
        if n % 2 == 1: res = szor(res, alap)
        n = n // 2
        alap = szor(alap, alap)
    return res[0]

>>> fibF2(100)
    354224848179261915075
```

A következő táblázat az $(1,1,0)^{\mathbf{28}}$ értéket, azaz a **29**-dik Fibonacci-szám lépésenkénti meghatározását mutatja be:

alap	*n*	*res*
		$(1,0,1$
$(1,1,0)$	28	$(1,1,0)$
$(1,1,0)^2 = (2,1,1)$	14	$(1,1,0)$
$(1,1,0)^4 = (5,3,2)$	7	$(5,3,2)$
$(1,1,0)^8 = (34,21,13)$	3	$(233,144,89)$
$(1,1,0)^{16} = (1597,987,610)$	1	$(\mathbf{514229}, 317811, 196418)$

Az eredmény:

$$F_{\mathbf{29}} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\mathbf{28}} = \begin{pmatrix} \mathbf{514229} & 317811 \\ 317811 & 196418 \end{pmatrix}$$

4.2. Egész számok

Az egész számok halmazát $\mathbb{Z}$-vel jelöljük, és elemeit a következőképpen adhatjuk meg: $\{..., -3, -2, -1, 0, 1, 2, 3, ...\}$. Az egész számokkal végzett műveletek tulajdonságai pedig a következők:

1. az összeadás, szorzás kommutatív, illetve asszociatív műveletek,
2. az összeadás a szorzásra nézve disztributív művelet,
3. az egész számok halmaza zárt az összeadásra, kivonásra, szorzásra nézve, de ez nem igaz az osztásra,
4. a 0 az összeadásra nézve semleges elem, míg az 1 a szorzásra nézve lesz a semleges elem,
5. az összeadásra nézve minden elemnek van **inverz** eleme, azaz bármely $a \in \mathbb{Z}$, létezik a_1 úgy, hogy $a + a_1 = 0$.

Megállapítható, hogy $\mathbb{N} \subset \mathbb{Z}$, azaz a természetes számok halmaza benne van az egész számok halmazában, és a két halmaz számossága megegyezik. Két halmaz **számossága** akkor egyezik meg, ha a két halmaz között létezik egy bijekció, azaz egy kölcsönösen egyértelmű leképezés. Például az egész számok és természetes számok között a következő bijektív kapcsolat áll fenn:

$$f : \mathbb{Z} \to \mathbb{N},\ f(x) = \begin{cases} 2 \cdot x & \text{ha } x \geq 0 \\ -2 \cdot x - 1 & \text{ha } x < 0 \end{cases},$$

azaz a nem negatív számoknak a páros számokat, a negatív számoknak a páratlan számokat feleltethetjük meg.

Az egész számokon az osztás alkalmazásakor egy sor érdekes tulajdonság állapítható meg, amelyeknek az ókori görögök is külön figyelmet szenteltek [27, 33, 39]. Azt az algoritmust, amely két egész szám legnagyobb közös osztóját határozza meg, és amely az adatbiztonság területén ma is különösen fontos szerepet tölt be, Eukleidész [3] írta le először. Éppen ezért a fejezetet az oszthatóság értelmezésével és az oszthatósághoz kapcsolódó tulajdonságok bemutatásával folytatjuk, utána pedig bemutatjuk Eukleidész algoritmusát, az euklideszi algoritmust és annak kiterjesztett változatát.

4.1. értelmezés. Legyenek a és b egész számok, ahol $b \neq 0$. b akkor osztja a-t, ha létezik olyan q egész szám, amelyre fennáll: $a = q \cdot b$, jelölése: $b \mid a$. Ha b osztja a-t, akkor azt mondjuk, hogy a a b többszöröse. Azt az esetet, amikor b nem osztja a-t, $b \nmid a$-vel jelöljük.

4.1. tétel. (A maradékos osztás tétele) Legyen a egy egész szám, b pedig egy pozitív egész szám. Ekkor egyértelműen létezik két olyan q és r egész szám, amelyekre igaz $a = q \cdot b + r$, $0 \leq r < b$.

3 Alexandriai Eukleidész ókori görög matematikus (kb. i. e. 300 – ?)

Bizonyítás: A létezés bizonyításához válasszuk ki a q-t úgy, hogy $q \cdot b$ a legnagyobb többszöröse legyen b-nek, de amely kisebb vagy egyenlő, mint a.

Az egyértelműséghez tegyük fel, hogy a felírható $q_1 \cdot b + r_1$ alakba is, ahol $0 \leq r_1 < b$. Ekkor

$$(q - q_1) \cdot b + (r - r_1) = 0. \tag{4.4}$$

Ez azonban azt jelenti, hogy $r - r_1$ többszöröse b-nek. Másrészt fennáll, hogy $|r - r_1| < b$, tehát a (4.4) egyenlet csak akkor állhat fenn ha $r - r_1 = 0$, amiből következik, hogy q_1 is egyenlő q-val.

□

A 4.1. tétel által értelmezett a-t osztandónak, a b-t osztónak, a q-t hányadosnak és az r-t maradéknak hívjuk. A Pythonban, mint korábban már mutattuk, külön operátor alkalmazásával lehet meghatározni a hányadost, illetve a maradékot: $q = a$ `//` b, $r = a$ `%` b.

Az a és b egész számok **legnagyobb közös osztója** az a legnagyobb pozitív egész szám, amely osztja az a-t és b-t is. Jelölése többféleképpen is lehetséges: $lnko(a, b)$, (a, b), vagy $gcd(a, b)$. A tankönyvben mi a kerek zárójel változatot fogjuk leginkább használni.

4.2. tétel. Legyenek a, b, q, r egész számok, $a = q \cdot b + r$ és legyen $(a, b) = d$, azaz a és b legnagyobb közös osztója d. Ekkor igaz az, hogy:

$$d = (a, b) = (b, a - q \cdot b). \tag{4.5}$$

Bizonyítás: Az $a = q \cdot b + r$ egyenlőség alapján kijelenthető, hogy a és b minden közös osztója osztja $r = a - q \cdot b$-t is, tehát közös osztója lesz b-nek is és r-nek is. Másrészt az $a = q \cdot b + r$ egyenlőség alapján kijelenthető, hogy b és r minden közös osztója osztja a-t is, tehát közös osztója lesz a-nak is és b-nek is. Tehát a és b közös osztói ugyanazok lesznek, mint a b és r közös osztói, amiből következik, hogy a legnagyobb közös osztóik is meg kell egyezzenek.

□

Azt mondjuk, hogy a és b egész számok **relatív prímek**, ha a legnagyobb közös osztójuk 1.

Azt mondjuk, hogy az $a_1, a_2, \ldots, a_n$ egész számok kölcsönösen relatív prímek, ha $(a_1, a_2, \ldots, a_n) = 1$. Az $a_1, a_2, \ldots, a_n$ egész számok pedig páronként relatív prímek, ha $(a_i, a_j) = 1$, bármely $i, j \in \{1, 2, \ldots, n\}$.

Az a és b egész számok **legkisebb közös többszöröse** az a legkisebb pozitív egész szám, amely osztható a-val és b-vel is. Jelölése: $[a, b]$.

Az a és b egész számok szorzata egyenlő az a és b legnagyobb közös osztójának és legkisebb közös többszörösének a szorzatával, fennáll tehát:

$$(a, b) \cdot [a, b] = a \cdot b. \tag{4.6}$$

4.3. Az euklideszi algoritmus és változatai

A legnagyobb közös osztó meghatározására több algoritmus is ismert [26, 27, 32]. Az euklideszi algoritmus alapötlete a 4.1. tételen alapszik. Az a, b pozitív egész számokból kiindulva ismételten meghatározzuk az osztási hányadost és osztási maradékot. Az osztást az előzőleg meghatározott osztási hányados és maradék között végezzük, mindaddig, míg nulla nem lesz az osztási maradék. A 4.2. tétel alapján megállapítható, hogy a legnagyobb közös osztó egyenlő lesz a következő számítási sorozat során meghatározott utolsó nem nullás maradékkal, ahol $r_0 = a$, és $r_1 = b$:

$$\begin{aligned}
r_0 &= q_1 \cdot \boldsymbol{r_1} + r_2 && 0 \le r_2 < r_1,\\
\boldsymbol{r_1} &= q_2 \cdot r_2 + r_3 && 0 \le r_3 < r_2,\\
&\vdots\\
r_{n-2} &= q_{n-1} \cdot r_{n-1} + r_n && 0 \le r_n < r_{n-1},\\
r_{n-1} &= q_n \cdot r_n.
\end{aligned}$$

$$(a, b) = (r_0, r_1) = (r_1, r_2) = \ldots = (r_{n-2}, r_{n-1}) = (r_{n-1}, r_n) = (r_n, 0) = r_n.$$

Példa: Határozzuk meg 32 és 76 legnagyobb közös osztóját.

$$\begin{aligned}
(32, 76) &= (76, 32 - 0 \cdot 76) = (76, 32)\\
(76, 32) &= (32, 76 - 2 \cdot 32) = (32, 12)\\
(32, 12) &= (12, 32 - 2 \cdot 12) = (12, 8)\\
(12, 8) &= (8, 12 - 1 \cdot 8) = (8, 4)\\
(8, 4) &= (4, 8 - 2 \cdot 4) = (4, 0) = 4
\end{aligned}$$

4.7. algoritmus. Írjunk egy Python függvényt, amely az euklideszi algoritmussal határozza meg két egész szám legnagyobb közös osztóját.

```
def lnkoF(a, b):
    while True:
```

```
        r = a % b
        if r == 0: break
        a = b
        b = r
    return b

>>> lnkoF(-32, -76)
    4
>>> lnkoF(32.7, 76)
    'hibás bemenet'
```

Az algoritmus ismételten meghatározza `a` és `b` osztási maradékát. Minden egyes ciklusművelet során teszteli az osztási maradék értékét, ha ez nulla, akkor a `break` utasítással kilép a ciklustörzsből, és a `return` segítségével visszaadja a hívó egységnek a `b`, azaz az utolsó nem nullás maradék értékét.

Ha szeretnénk látni, hogy a számítások alapján hogyan változnak az `a, b, r` változókban tárolt értékek, akkor ki is lehet őket íratni. A következőkben ezt tesszük, illetve még kiegészítettük a kódsort további műveletekkel, lekezeltük azt az esetet, amikor a bemenet valamelyike nem egész szám, illetve a bemeneti paraméterek abszolút értékével dolgoztunk.

```
def lnkoP(a, b):
    if a != int(a) or b != int(b):
        return 'hibás bemenet'
    a = abs(a)
    b = abs(b)
    print("%4s%4s%4s" % ("a","b","r"))
    while True:
        r = a % b
        print("%4i%4i%4i" % (a, b, r))
        if r == 0: break
        a = b
        b = r
    return b

>>> lnkoP(32, 76)
         a   b   r
        32  76  32
        76  32  12
        32  12   8
        12   8   4
        8    4   0
    4
```

4.8. algoritmus. Írjuk egy Python függvényt, amely rekurzívan az euklideszi algoritmussal határozza meg két pozitív egész szám legnagyobb közös osztóját.

A rekurzív változat sokkal kompaktabb lesz. A hibakezelést, illetve az `abs` függvény alkalmazását most elhagytuk:

```
def lnkoR(a, b):
    temp = a % b
    if temp == 0: return b
    return lnkoR(b, temp)
```

4.9. algoritmus. Írjuk egy Python függvényt, amely kivonásos módszerrel határozza meg két pozitív egész szám legnagyobb közös osztóját.

Az algoritmus egyszerűsített változata a következő elgondoláson alapszik: addig csökkentjük a nagyobbik számot a kisebbikkel, amíg nem lesz egyenlő a két bemeneti paraméter értéke. Az algoritmus hatékonyabb változata szerint, ha valamelyik szám páros, akkor a kivonás művelete előtt még sor kerül 2-vel való osztásra is, amelyet addig ismétlünk, amíg páratlan számot nem kapunk [16].

```
def lnkoM(a, b):
    while a != b:
        if a > b: a = a - b
        else: b = b - a
    return a
```

Az euklideszi algoritmus hatékonyságát az osztások száma határozza meg, ezért fontos feltennünk azt a kérdést, hogy hány osztást végez az algoritmus? A kérdésre Lamé[4] tétele adja meg a választ:

4.3. tétel. (Lamé tétele) Az euklideszi algoritmus során végzett osztások száma nem lesz nagyobb, mint ötször a kisebbik szám számjegyeinek száma.

A legnagyobb közös osztó meghatározására legegyszerűbb persze a Python `math` moduljában található **`gcd`** függvény alkalmazása:

```
>>> from math import gcd
>>> gcd(60, 45)
    15
>>> gcd(1789, 100)
```

4 Gabriel Lamé francia matematikus (1795–1870)

```
    1
>>> gcd(-63, 45)
    9
```

Az a és b egész számok, illetve a d legnagyobb közös osztó esetében fennáll a következő összefüggés, ahol x, y egész számok:

$$d = x \cdot a + y \cdot b \tag{4.7}$$

d tehát az a legkisebb egész szám, amely egyenlő az a és b lineáris kombinációjával. A **kiterjesztett euklideszi algoritmus** a legnagyobb közös osztó mellett meghatározza az x és y együtthatókat is. Az euklideszi algoritmus `for` ciklusát kell ehhez kiegészíteni [8, 28]. A kiterjesztett euklideszi algoritmus a következő kezdeti értékekből indul ki:

$$x_0 = 1,\ x_1 = 0,\ y_0 = 0,\ y_1 = 1,$$

és minden egyes j-edik iterációnál még meghatározza a következőket is:

$$x_j = x_{j-2} - q_{j-1} \cdot x_{j-1}$$
$$y_j = y_{j-2} - q_{j-1} \cdot y_{j-1},$$

ahol q_j az euklideszi algoritmus során felülírt a és b értékek osztási egészrésze. Az utolsó körben meghatározott x_j, y_j értékek lesznek a keresett együtthatók.

4.10. algoritmus. Írjunk egy Python függvényt, amely a kiterjesztett euklideszi algoritmussal meghatározza azokat a `d, x, y` egész számokat, amelyekre fennáll: `d = x · a + y · b`.

```
def extEuclid(a, b):
    x0, x1, y0, y1 = 1, 0, 0, 1
    while True:
        q = a // b
        r = a - q * b
        if r == 0:
            return (b, x1, y1)
        a = b
        b = r
        x = x0 - q * x1
        y = y0 - q * y1
        x0, x1, y0, y1 = x1, x, y1, y

>>> extEuclid(54332, 9494)
    (94, 18, -103)
```

Az `extEuclid` függvényben az `r` meghatározásakor az `r = a - q * b` műveletet alkalmaztuk, mert figyelembe vettük, hogy a `q` értéke az előző műveletsorban meghatározásra került. Több számítást igényelt volna az `r = a % b` művelet, mert az osztás költsége nagyobb, mint egy szorzás és egy különbség költsége. Nagy számok esetében ez a költség befolyásolja a függvény futási idejét.

Példa: A kiterjesztett euklideszi algoritmust alkalmazva határozzuk meg 84 és 35 legnagyobb közös osztóját, illetve azokat az x és y egész számokat, melyekre fennáll: $84 \cdot x + 35 \cdot y = d$, ahol $d = (84,\ 35)$.

a	b	r	q	x_0	x_1	y_0	y_1
				1	0	0	1
84	35	14	2	0	1	1	-2
35	14	7	2	1	**-2**	-2	**5**
14	**7**	0	2				

Tehát a megoldás: $d = 7,\ x = -2,\ y = 5$, és fennáll a következő összefüggés: $84 \cdot (-2) + 35 \cdot 5 = 7$. A számításokból az is megállapítható, hogy 84 és 35 legnagyobb közös osztója 7.

Példa: A kiterjesztett euklideszi algoritmust alkalmazva határozzuk meg 45 és 31 legnagyobb közös osztóját, illetve azokat az x és y egész számokat, melyekre fennáll: $45 \cdot x + 31 \cdot y = d$, ahol $d = (45,\ 31)$.

a	b	r	q	x_0	x_1	y_0	y_1
				1	0	0	1
45	31	14	1	0	1	1	-1
31	14	3	2	1	-2	-1	3
14	3	2	4	-2	9	3	-13
3	2	1	1	9	**-11**	-13	**16**
2	**1**	0	2				

Tehát a megoldás: $d = 1,\ x = -11,\ y = -29$, és fennáll a következő összefüggés: $45 \cdot (-11) + 31 \cdot 16 = 1$. A számításokból az is megállapítható, hogy 45 és 31 relatív prímek.

A kiterjesztett euklideszi algoritmust egész együtthatós kétismeretlenes algebrai egyenletek megoldásánál is lehet alkalmazni. Azokat az egész

együtthatós többismeretlenes algebrai egyenleteket, amelyeknek megoldásai egész számok (ritkán természetes vagy racionális számok), diophantoszi[5] egyenleteknek hívjuk [27, 33, 39].

4.4. tétel. Legyenek a, m egész számok úgy, hogy $(a, m) = d$. Ha $d \not| b$, azaz ha d nem osztja b-t, akkor az $a \cdot x + m \cdot y = b$ egyenletnek nincs megoldása az egész számok körében, ha $d \mid b$, akkor végtelen sok megoldás van.

Az $a \cdot x + m \cdot y = b$ egyenlet megoldásainak a meghatározása a következő lépéseket jelenti:

1. kiterjesztett euklideszi algoritmussal meghatározzuk a $d, \hat{x}, \hat{y}$ értékeket úgy, hogy teljesüljön: $a \cdot \hat{x} + m \cdot \hat{y} = d$,
2. ha $d \mid b$, akkor meghatározzuk az egyenlet első megoldását, legyen ez x_0 és y_0:

$$x_0 = \hat{x} \cdot (b/d), y_0 = \hat{y} \cdot (b/d),$$

3. az egyenlet további megoldásait az $n = \ldots, -2, -1, 0, 1, 2, \ldots$ egész számok alapján a következőképpen tudjuk kiszámolni:

$$x = x_0 + n \cdot (m/d), y = y_0 - n \cdot (a/d).$$

Nagyon gyakran elsőfokú, kétismeretlenes diofantoszi egyenletek pozitív megoldásait kell meghatározni. Ennek érdekében megállapítható, hogy az $a \cdot x + m \cdot y = b$ egyenlet pozitív megoldásai esetében fenn kell álljon:

$$x_0 + n \cdot \frac{m}{d} \geq 0 \Rightarrow n \geq \frac{-x_0}{\frac{m}{d}}$$

$$y_0 - n \cdot \frac{a}{d} \geq 0 \Rightarrow n \leq \frac{y_0}{\frac{a}{d}}$$

Meg kell tehát keresnünk azokat az n természetes számokat, amelyek kielégítik a fenti két egyenlőtlenséget.

Példa: Egy elárusító 1676 lej értékben rendelt almát és körtét. Minden láda alma 36 lejbe, és minden láda körte 50 lejbe kerül. Hány láda almát és hány láda körtét rendelhetett?.

Tulajdonképpen a $36 \cdot x + 50 \cdot y = 1676$ diofantoszi egyenlet pozitív megoldásait kell megkeresnünk, amely a következő lépésekből áll:

5 Diophantosz ókori görög matematikus (kb. 200 – kb. 284)

1. a kiterjesztett euklideszi algoritmussal meghatározzuk a $d = 2, \hat{x} = 7, \hat{y} = -5$ értékeket, ekkor fennáll $7 \cdot 36 + (-5) \cdot 50 = 2$
2. mivel $d = 2$ osztja 1676-ot, meghatározzuk az egyenlet első megoldását:

$$x_0 = \hat{x} \cdot \frac{b}{d} = 7 \cdot \frac{1676}{2} = 7 \cdot 838 = 5866$$

$$y_0 = \hat{y} \cdot \frac{b}{d} = -5 \cdot \frac{1676}{2} = -5 \cdot 838 = -4190$$

3. fennáll: $5866 \cdot 36 - 4190 \cdot 50 = 1676$
4. most meg kell keressük a pozitív megoldásokat, fenn kell álljon:

$$5866 + n \cdot \frac{50}{d} = 5866 + n \cdot 25 \quad \geq 0 \quad \Rightarrow n \quad \geq -234.64$$

$$-4190 - n \cdot \frac{36}{d} = -4190 - n \cdot 18 \quad \geq 0 \quad \Rightarrow n \quad \leq -232.7$$

5. $n = -234$ és $n = -233$ értékekre kapunk tehát pozitív megoldásokat:

$$\begin{aligned} \Rightarrow \quad n &= -234 \\ x_1 &= 5866 + 25 \cdot (-234) = 16 \\ y_1 &= -4190 - 18 \cdot (-234) = 22 \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad n &= -233 \\ x_2 &= 5866 + 25 \cdot (-233) = 41 \\ y_2 &= -4190 - 18 \cdot (-233) = 4 \end{aligned}$$

Tehát az elárusító 16 láda almát és 22 láda körtét, vagy 41 láda almát és 4 láda körtét rendelhetett:

$$16 \cdot 36 + 22 \cdot 50 = 1676$$
$$41 \cdot 36 + 4 \cdot 50 = 1676.$$

Példa: Egy elárusító 549 lej értékben rendelt almát és körtét. Minden láda alma 18 lejbe, és minden láda körte 33 lejbe kerül. Mennyi az a minimális ládaszám, amit az elárusító rendelhetett?

Először a $18 \cdot x + 33 \cdot y = 549$ diofantoszi egyenlet pozitív megoldásait keressük, ahol a megoldás menete a következő:

1. a kiterjesztett euklideszi algoritmussal meghatározzuk a $d = 3, \hat{x} = 2, \hat{y} = -1$ értékeket, ekkor fennáll $2 \cdot 18 + (-1) \cdot 33 = 3$,
2. mivel $d = 3$ osztja-e 549-et, meghatározzuk az egyenlet első megoldását:
$$x_0 = \hat{x} \cdot \frac{b}{d} = 2 \cdot \frac{549}{3} = 2 \cdot 183 = 366$$
$$y_0 = \hat{y} \cdot \frac{b}{d} = -1 \cdot \frac{549}{3} = -1 \cdot 183 = -183$$
3. fennáll:
$$366 \cdot 18 - 183 \cdot 33 = 549$$
4. most meg kell keressük a pozitív megoldásokat, fenn kell álljon:
$$366 + n \cdot \frac{33}{d} = 366 + n \cdot 11 \quad \geq 0 \quad \Rightarrow n \quad \geq -33.2$$
$$-183 - n \cdot \frac{18}{d} = -183 - n \cdot 6 \quad \geq 0 \quad \Rightarrow n \quad \leq -30.5$$
5. $n = -33, -32, -31$ értékekre kapunk tehát pozitív megoldásokat:
$$\begin{aligned} \Rightarrow \quad n &= -33 \\ x_1 &= 366 + 11 \cdot (-33) = 3 \\ y_1 &= -183 - 6 \cdot (-33) = 15 \end{aligned}$$
$$\begin{aligned} \Rightarrow \quad n &= -32 \\ x_2 &= 366 + 11 \cdot (-32) = 14 \\ y_2 &= -183 - 6 \cdot (-32) = 9 \end{aligned}$$
$$\begin{aligned} \Rightarrow \quad n &= -31 \\ x_3 &= 366 + 11 \cdot (-31) = 25 \\ y_3 &= -183 - 6 \cdot (-31) = 3 \end{aligned}$$

Tehát az elárusító

25 láda almát és 3 láda körtét, összesen 28 ládát, vagy
14 láda almát és 9 láda körtét, összesen 23 ládát, vagy
3 láda almát és 15 láda körtét, összesen 18 ládát rendelhetett.

Az elárusító tehát 18 láda gyümölcsöt rendelt, mert a meghatározott értékek közül ez a legkisebb.

4.11. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy kétismeretlenes diofantoszi egyenlet pozitív megoldásait.

```
from math import ceil, floor
def diofant(a, m, b):
    (d, xk, yk) = extEuclid (a, m)
    if b % d != 0:
       print ('nincs megoldas')
       return
    x0 = xk * b//d
    y0 = yk * b//d
    bd = m // d
    ad = a // d
    n1 = int(ceil(-x0 / bd))
    n2 = int(floor(y0 / ad))
    if n1 <= n2:
       print('megoldasok:')
       for i in range(n1, n2 + 1):
          print (x0 + bd * i, y0 - ad * i)
    else: print('nincs pozitiv megoldas')

>>> diofant(36, 50, 1676)
    megoldasok:
    16 22
    41 4
```

4.4. Racionális számok

A racionális számok halmazát: $\mathbb{Q}$-val jelöljük. A halmaz bármelyik eleme a következőképpen írható fel: $\left\{\frac{a}{b} : a, b \in \mathbb{Z}, b \neq 0\right\}$. A racionális számok tehát tetszőleges egész számok hányadosaként adhatók meg, de tekinthetünk úgy is rájuk, mint egész számok rendezett párjaira. Az a számot számlálónak, míg a b-t nevezőnek hívjuk. A racionális számokkal végzett műveletek tulajdonságai a következők:

1. az összeadás, szorzás kommutatív, illetve asszociatív műveletek,
2. az összeadás a szorzásra nézve disztributív művelet,

3. a racionális számok halmaza zárt az összeadásra, kivonásra, szorzásra, osztásra nézve,
4. az összeadásra, szorzásra nézve minden elemnek lesz inverz eleme,
5. végtelen sok alakban felírhatóak, ezek között fontos szerepet kap az **irreducibilis alak**, ami azt jelenti, hogy a számláló és nevező legnagyobb közös osztója 1,
6. a racionális számok halmaza **megszámlálható**, azaz létezik egy számsorozat, amely a racionális számokból áll,
7. a racionális számok **sűrűn rendezett** halmazt alkotnak: bármely két racionális szám között van egy harmadik,
8. minden racionális szám felírható **tizedes alakba**, azaz véges vagy végtelen szakaszos tizedesjegyek segítségével.

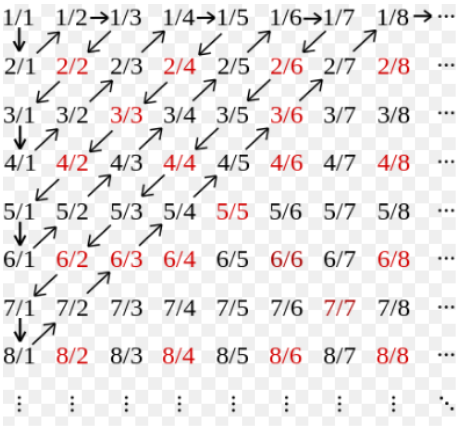
4.12. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy adott racionális szám irreducibilis alakját.

A következő kódsorban az $\frac{a}{b}$ racionális számot az `(a, b)` értékpárral jelöljük. Meghívásra kerül a korábban megadott `lnkoF` függvény, mert egy tört irreducibilis alakját úgy határozzuk meg, hogy kiszámoljuk a számláló és nevező legnagyobb közös osztóját, majd végigosztjuk mind a számlálót, mind a nevezőt ezzel az értékkel.

```
def iAlak(a, b):
    temp = lnkoF(a, b)
    return (a // temp, b // temp)

>>> iAlak(1024, 500)
    (256, 125)
```

A következőkben olyan módszereket mutatunk be, amelyek segítségével meghatározható a racionális számok egy sorozata, jelezve azt, hogy a racionális számok halmaza megszámlálható [28].

A generálás egyik módszere, ha elindulunk a 4.2. ábrán (kép forrása: *https://en.wikipedia.org/*) látható mátrix bal felső sarkában található elemtől, majd a nyilakat követjük. Pirossal azokat a számokat jelöltük az ábrán, amelyek nem irreducibilis alakban vannak, és korábban pedig már ki lettek generálva.



4.2. ábra

4.13. algoritmus. Írjunk egy Python függvényt, amely a 4.2. ábrán mutatott bejárási sorrend szerint bejárja a mátrixot, és kiírja az első $\frac{n \cdot (n+1)}{2}$ racionális számot.

A következő `auxRacionalis` függvény a fent megadott bejárás szerint a `k`-dik átló kiíratását valósítja meg. A racionális számokat számpárokként kezeli és aszerint, hogy páratlan vagy páros sorszámú átló kiíratásánál tart, a számláló értékeit növekvő vagy csökkenő sorrendbe generálja ki.

```
def auxRacionalis(k):
    for j in range(1, k+1):
        if k % 2 == 1: print((j, k+1-j), end = ' ')
        else: print((k+1-j, j), end = ' ')
    print()

>>> auxRacionalis(4)
    (4, 1) (3, 2) (2, 3) (1, 4)
>>> auxRacionalis(5)
    (1, 5) (2, 4) (3, 3) (4, 2) (5, 1)
```

A `racionalis_` függvény `n`-szer fogja meghívni az `auxRacionalis` függvényt ahhoz, hogy a kívánt számsorozatot kiírja:

```
def racionalis_(n):
    for k in range(1, n+1):
```

```
        auxRacionalis(k)

>>> racionalis_(10)
    (1, 1)
    (2, 1) (1, 2)
    ...
```

Az előző kódsorok megadhatók segédfüggvény nélkül, **for, egymásba ágyazott** ciklussal. Az `if` feltétel is elhagyható, mert a számok sorrendje az egyes sorokon belül nem számít. A `racionalis` függvény ezt az elgondolást követi:

```
def racionalis(n):
    for k in range(1, n+1):
        for j in range(1, k+1):
            print((j, k+1-j), end = ' ')
        print()
```

Ha szükségünk van a racionális számok mindegyikére, akkor el kell őket tárolni. A `racionalisL_` Python függvény egy `tuple` elemtípusú listába tárolja el a pozitív racionális számokat, visszatérési értéke pedig a létrehozott lista lesz:

```
def racionalisL_(n):
    L = []
    for k in range(1, n+1):
        for j in range(1, k+1):
            L = L + [(j, k+1-j)]
            n = n - 1
            if n == 0: return L
    return L

>>> racionalisL_(7)
    [(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4)]
```

A kiírt eredményt tanulmányozva vegyük észre, hogy lesznek olyan racionális számok, amelyek különböző alakban, azaz többször is megjelennek, ilyen például az `(1,1)` és a `(2,2)` stb. számpárok. Megállapítható, hogy ha az aktuálisan kigenerált racionális szám nincs irreducibilis alakban, akkor az korábban már kiíratásra került. Ennek a módosításnak a bevezetését azonban az olvasóra bízzuk.

A racionális számok egy másik sorozatát a következő elgondolás alapján is meghatározhatjuk:

– az első racionális szám $\frac{1}{1}$,
– az $\frac{x}{y}$ után következő racionális szám: $2 \cdot \left\lfloor \frac{x}{y} \right\rfloor + 1 - \frac{x}{y}$ reciproka, ahol $\lfloor\ \rfloor$ alsó egészrészt jelent.

Példák: A $\frac{5}{2}$ után következő racionális szám $\frac{2}{5}$, a $\frac{5}{3}$ utáni $\frac{3}{4}$, mert:

$$\frac{5}{2} \to 2 \cdot \left\lfloor \frac{5}{2} \right\rfloor + 1 - \frac{5}{2} = 2 \cdot 2 + 1\frac{5}{2} = 5 - \frac{5}{2} = \frac{10-5}{2} = \frac{5}{2} \to \frac{2}{5}$$

$$\frac{5}{3} \to 2 \cdot \left\lfloor \frac{5}{3} \right\rfloor + 1 - \frac{5}{3} = 2 \cdot 1 + 1 - \frac{5}{3} = 3 - \frac{5}{3} = \frac{9-5}{3} = \frac{4}{3} \to \frac{3}{4}$$

Figyeljük meg, hogy a számítás során nem végeztük el az osztást, helyette a közönséges törtalakokkal végeztünk műveleteket. Ezzel a módszerrel a következő sorrendben állnak elő a racionális számok:

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{3}{1}, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, \frac{4}{1}, \ldots$$

A feladat megoldásához a racionális számokat a Python `Fraction` típusaként fogjuk eltárolni [38]. Ehhez a **`fractions`** modult kell importálni. A következő példában a `rac1`, `rac2` egy-egy `Fraction` típusú változót jelölnek. A `Fraction` típus esetében az aritmetikai műveletek felüldefiniáltak, ezért a megszokott módon lehet őket használni. Lehetséges a számlálóra, illetve nevezőre való hivatkozás is:

```
from fractions import Fraction
>>> rac1, rac2 = Fraction(2,3), Fraction(4,5)
>>> rac1 * rac2
    Fraction(8, 15)
>>> print('szamlalo: ', (rac1*rac2).numerator)
    8
>>> print('nevezo: ', (rac1*rac2).denominator)
    15
```

4.14. algoritmus. Írjunk egy Python függvényt, amely meghatározza az első n pozitív racionális számot a fenti elgondolás alapján.

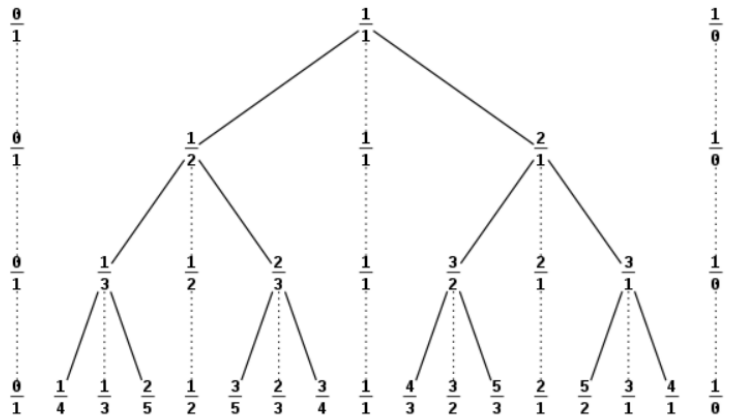
Két függvényt írunk, ahol a `nextRacFrac` az $\frac{x}{y}$ racionális szám utáni számot fogja meghatározni. A második függvény, a `racionalisL` kiindul az `(1,1)` számpárból, ami az első racionális számnak felel meg, és a következő racionális számot a `nextRacFrac` függvény meghívásával határozza meg.

```
from fractions import Fraction
def nextRacFrac(r):
    x, y = r.numerator, r.denominator
    temp = 2 * (x // y) + 1
    res = Fraction(temp, 1) - Fraction(x, y)
    return Fraction(res.denominator,  res.numerator)

def racionalisL(n):
    if n < 1: return []
    x = Fraction(1, 1)
    L = [(1, 1)]
    while n > 1:
        x = nextRacFrac(x)
        L += [(x.numerator, x.denominator)]
        n = n - 1
    return L

>>> nextRacFrac(Fraction(4,3))
    Fraction(3, 5)
>>> racionalisL(7)
    [(1, 1),(1, 2),(2, 1),(1, 3),(3, 2),(2, 3),(3, 1)]
```

A racionális számok sorozatát a Stern–Brocot-fa megszerkesztésével is ki lehet generálni, ahol a $\frac{0}{1}$ és az $\frac{1}{0}$ számok által alkotott kételemű sorozatból indulunk ki, amelyet folyamatosan bővítünk új racionális számokkal [12]. A sorozat bővítése azt jelenti, hogy az $\frac{a_1}{b_1}$ és $\frac{a_2}{b_2}$ elemek közé beszúrjuk az $\frac{a_1 + a_2}{b_1 + b_2}$ racionális számot. A 4-ed rendű Stern–Brocot-fa a 4.3. ábrán látható (kép forrása: *https://en.wikipedia.org/*).

A `nextRacFrac` Python függvény segítségével kigenerált sorozat tulajdonképpen a Stern–Brocot-fának egy speciális bejárási sorrendje lesz, ennek szemléltetése végett próbáljuk ki a következő függvényt:

```
def kiirFa(n):
    L = racionalisL(n)
    i, k = 0,1
    while i < n:
        print(L[i : i + k])
        i, k = i + k, k * 2

>>> kiirFa(20)
```



4.3. ábra

Megállapítható, hogy a Stern–Brocot-fa bal oldalán az 1-nél kisebb racionális számok találhatók, amely pedig a Farey[6]-sorozathoz kapcsolódik.

4.2. értelmezés. n-ed rendű Farey-sorozatnak nevezzük a $\frac{h}{k}$ típusú törtek halmazának növekvő sorrendben felírt elemeit, ahol $(h,k) = 1$ és $0 \leq h \leq k \leq k \leq n$. A sorozat első eleme: $\frac{0}{1}$, utolsó eleme pedig: $\frac{1}{1}$ [27].

Példák:

1. A negyedrendű Farey-sorozat: $\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}$.
2. Az ötödrendű Farey-sorozat: $\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}$.

Az n-ed rendű Farey-sorozat tehát megadja a 0 és 1 közötti azon racionális számok rendezett sorrendjét, amelyeknek nevezője kisebb vagy egyenlő, mint n.

4.15. algoritmus. Írjunk egy Python függvényt, amely meghatározza az n-ed rendű Farey-sorozatot.

A továbbiakban két algoritmust is bemutatunk, ahol az első módszert akkor érdemes használni, ha n-ig az összes sorozatra szükségünk van, a másodikat pedig akkor, amikor csak az n-ed rendűt szeretnénk kigenerálni.

6 John Farey Sr. angol geológus (1766–1826)

Mindkét módszernél a racionális számokat számpárokként kezeljük. Az első módszer szerint az $n+1$-ed rendű Farey-sorozatot az n-ed rendű alapján fogjuk előállítani:

– az n-ed rendű sorozat $\frac{a_1}{b_1}$ és $\frac{a_2}{b_2}$ elemei közé beszúrjuk az $\frac{a_1+a_2}{b_1+b_2}$ tört irreducibilis alakját, ha az így kapott tört nevezője kisebb vagy egyenlő, mint n.

A következő `auxFarey` függvény az n-ed rendű Farey-sorozat alapján, amelyet az `L` paraméterben kell megadni, meghatározza az `ujL` listába, az n-ed rendű Farey-sorozatot. Meghívásra fog kerülni az `iAlak` függvény, lásd a 4.12. algoritmust.

```
def auxFarey(L = [(0,1), (1,1)], n = 2):
    ujL = []
    for i in range(1, len(L)):
        (a1, b1) = L[i-1]
        (a2, b2) = L[i]
        temp = iAlak(a1 + a2, b1 + b2)
        ujL += [L[i-1]]
        if temp[1] <= n:
             ujL += [temp]
    ujL += [L[-1]]
    return ujL
```

Például az ötödrendű Farey-sorozatot a következő műveletsorok megadásakor kapjuk:

```
>>> L = [(0, 1),(1, 4),(1, 3),(1, 2),(2, 3),(3, 4),(1, 1)]
>>> auxFarey(L, 5)
    [(0, 1), (1, 5), (1, 4), (1, 3), (2, 5), (1, 2),...
```

Az `auxFarey`-t alkalmazva a `farey_` függvény az `L` listába meghatározza az n-ed rendű Farey-sorozatot:

```
def farey_(n):
    L = [(0,1), (1,1)]
    for i in range(1, n):
        L = auxFarey(L, i+1)
    return L

>>> farey_(6)
    [(0, 1), (1, 6), (1, 5), (1, 4), (1, 3), ...
```

A második algoritmus gondolatmenete szerint az n-ed rendű sorozat első két eleme mindig: $\frac{0}{1}, \frac{1}{n}$. A sorozat további elemeit pedig úgy határozzuk meg, hogy az új elem mindig a sorozat két utolsó eleme után kerül. Általánosan az $\frac{a_1}{b_1}, \frac{a_2}{b_2}$ elemek után következő $\frac{a_3}{b_3}$ elem a következő képlettel határozható meg:

$$\frac{a_3}{b_3} \;=\; \frac{a_2 \cdot k - a_1}{b_2 \cdot k - b_1}, \quad \text{ahol} \quad k \;=\; \left\lfloor \frac{n + b_1}{b_2} \right\rfloor$$

A következő `farey` Python függvény a fenti elgondolás alapján kiírja a képernyőre az `n`-ed rendű Farey-sorozatot.

```
def farey(n):
    a1, b1, a2, b2 = 0, 1, 1, n
    print((a1, b1), end = ' ')
    print((a2, b2), end = ' ')
    while b2 > 1:
        k = (n + b1) // b2
        a3, b3 = a2 * k - a1, b2 * k - b1
        print((a3, b3), end = ' ')
        a1, b1 = a2, b2
        a2, b2 = a3, b3

>>> farey(5)
    (0,1) (1,5) (1,4) (1,3) (2,5) (1,2) (3,5) (2,3) ...
```

Példa: A 4-ed rendű Farey-sorozat a fenti elgondolás alapján kiindul a $\frac{0}{1}, \frac{1}{4}$ elemekből, a többi elemet pedig következőképpen számítja ki:

$$\frac{0}{\mathbf{1}}, \frac{1}{\mathbf{4}} \xrightarrow{k=1} \frac{1 \cdot k - 0}{4 \cdot k - 1} = \frac{1}{3} \quad \text{ahol } k = \left\lfloor \frac{4 + \mathbf{1}}{\mathbf{4}} \right\rfloor = 1,$$

$$\frac{1}{\mathbf{4}}, \frac{1}{\mathbf{3}} \xrightarrow{k=2} \frac{1 \cdot k - 1}{3 \cdot k - 4} = \frac{1}{2} \quad \text{ahol } k = \left\lfloor \frac{4 + \mathbf{4}}{\mathbf{3}} \right\rfloor = 2,$$

$$\frac{1}{\mathbf{3}}, \frac{1}{\mathbf{2}} \xrightarrow{k=3} \frac{1 \cdot k - 1}{2 \cdot k - 3} = \frac{2}{3} \quad \text{ahol } k = \left\lfloor \frac{4 + \mathbf{3}}{\mathbf{2}} \right\rfloor = 3,$$

stb.

4.5. Lánctörtek

A lánctört (continued fraction) egy speciális *emeletes* tört, amely segítségével a valós számokat tudjuk egész számok sorozataként megadni. A lánctörtek ilyen módon mind a racionális, mind az irracionális számok ábrázolására alkalmasak. A racionális számok esetében a számot ábrázoló lánctört véges, míg az irracionális számok esetében végtelen jegyet tartalmaz.

Egy lánctört kétféle alakban is megadható, ahogyan azt a következő ábrák mutatják. A két alak átalakítható egymásba [27, 33].

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{\ddots}}}} \qquad d_0 + \cfrac{1}{d_1 + \cfrac{1}{d_2 + \cfrac{1}{d_3 + \cfrac{1}{\ddots}}}}$$

A jobb oldali alakot *egyszerű lánctörtnek*, a $[d_0, d_1, d_2, d_3, ...]$ számsorozatot pedig a lánctörtjegyeinek hívjuk, ahol d_0 egész szám, a számsorozat további elemei pedig pozitív egész számok. Ha a $[d_0, d_1, d_2, d_3, ...]$ számsorozat véges számú elemet tartalmaz, akkor *véges lánctörtről* beszélünk.

A korábban megadott 4.7. euklideszi algoritmus felhasználható racionális számok lánctörtalakjának a meghatározásához, mert a kiszámolt osztási egészrészek fogják a lánctörtjegyeket jelenteni.

Példa: Határozzuk meg $\frac{32}{76}$ lánctörtjegyeit és lánctört alakját.

a	b	q	r				
32	76	**0**	32	$32 = 0 \cdot 76 + 32$	$\rightarrow$	$\frac{32}{76}$	$= 0 + \frac{32}{76} = 0 + \frac{1}{76/32}$
76	32	**2**	12	$76 = 2 \cdot 32 + 12$	$\rightarrow$	$\frac{76}{32}$	$= 2 + \frac{12}{32} = 2 + \frac{1}{32/12}$
32	12	**2**	8	$32 = 2 \cdot 12 + 8$	$\rightarrow$	$\frac{32}{12}$	$= 2 + \frac{8}{12} = 2 + \frac{1}{12/8}$
12	8	**1**	4	$12 = 1 \cdot 8 + 4$	$\rightarrow$	$\frac{12}{8}$	$= 1 + \frac{4}{8} = 1 + \frac{1}{8/4}$
8	4	**2**	0	$8 = 2 \cdot 4 + 0$	$\rightarrow$	$\frac{8}{4}$	$= 2$

A fenti táblázat bal oldalán az a, b, q, r értékek változását láthatjuk, ahogyan azok a kiterjesztett euklideszi algoritmusban meghatározásra kerülnek. A jobb oldalon a lánctört alakhoz kapcsolódó összefüggések láthatók.
Felírható tehát:

$$\frac{32}{76} = 0 + \frac{1}{76/32} = 0 + \cfrac{1}{2 + \cfrac{1}{32/12}} = 0 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{12/8}}} =$$

$$= 0 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{8/4}}}} = 0 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{\mathbf{2}}}}}}$$

Azt mondjuk, hogy a meghatározott $[0, 2, 2, 1, 2]$ osztási egészrészek a $\frac{32}{76}$ racionális szám lánctörtjegyei.

Ha a lánctört utolsó jegye nem 1, akkor ez az érték helyettesíthető két további értékkel. x-szel jelölve az utolsó jegyet, akkor ehelyett az $x - 1, 1$ jegyek írhatóak, mert $\boldsymbol{x = x - 1 + \frac{1}{1}}$. A két alak ekvivalens, ahol a rövidebb alakot kanonikus alaknak is hívják. Ezek szerint $\frac{32}{76}$ lánctört alakja a következő is:

$$\frac{32}{76} = 0 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{\mathbf{1} + \cfrac{\mathbf{1}}{\mathbf{1}}}}}}$$

Példa: Határozzuk meg $\frac{61}{47}$ lánctörtjegyeit és lánctört alakját.

A lánctörtjegyek az: $[1, 3, 2, 1, \mathbf{4}]$ vagy $[1, 3, 2, 1, \mathbf{3, 1}]$ értékek lesznek, amelyeket az euklideszi algoritmus által meghatározott osztási egészrészek adnak, a lánctört alakok pedig a következők:

$$\frac{61}{47} = 1 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{\mathbf{4}}}}} = 1 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{\mathbf{3 + \cfrac{1}{1}}}}}}$$

4.16. algoritmus. Írjunk egy Python függvényt, amely meghatározza az $\frac{a}{b}$ racionális szám lánctörtjegyeit és tizedes alakját.

```
def lanct(a, b):
    if not float(a).is_integer()
        or not float(b).is_integer():
            print('hibás bemenet')
            return
    tAlak, L = a / b, []
    while True:
        q = a // b
        L += [q]
        r = a - b * q
        if r == 0: break
        a = b
        b = r
    return (tAlak, L)

>>> lanct(41, 11)
    (3.727272727272727, [3, 1, 2, 1, 2])
>>> lanct(89, 55)
    (1.6181818181818182, [1, 1, 1, 1, 1, 1, 1, 1, 2])
```

A fenti kódsor egy ellenőrző művelettel kezdődik, amellyel teszteljük, hogy a bemeneti paraméterek `int` típusú értékek vagy sem. Az `is_integer` `True`-t ad vissza, ha a bemeneti `float` típusú paraméter egész szám, ellenkező esetben `False` lesz a visszatérési érték.

```
>>> 3.5.is_integer()
    False
>>> 3.0.is_integer()
    True
```

4.6. Irracionális számok

Az irracionális számok halmazát azok a számok alkotják, amelyek **nem** írhatóak fel **két egész szám** hányadosaként. Egy szám nem lehet egyszerre racionális és irracionális is, tehát a racionális és irracionális számok halmazának metszete üres halmaz. A racionális és irracionális számok egyesített halmaza alkotja a valós számok halmazát.

Az irracionális számok halmazának nincs standard jelölési módja. Egyaránt használatosak a $\mathbb{Q}^*$, $\overline{\mathbb{Q}}$, $\mathbb{R}\backslash\mathbb{Q}$, $\mathbb{R}-\mathbb{Q}$ jelek, ahol $\mathbb{R}$ a valós számok halmazát jelöli.

Két irracionális számmal végzett aritmetikai művelet (összeadás, kivonás, szorzás, osztás) eredményezhet akár racionális, akár irracionális számot. Egy racionális és egy irracionális szám összege és különbsége mindig irracionális és egy nullától különböző racionális szám szorzata, osztása egy irracionális számmal is mindig irracionális.

Az irracionális számoknak is többféle alakja lehet, tizedestört alakjuk végtelen nem szakaszos tizedesjegyekből áll, lánctört alakjuk pedig végtelen lánctörtjegyből áll [33, 27].

Az egyik leghíresebb irracionális szám a $\sqrt{2} = 1.4142\ldots$, más néven a Püthagorasz-állandó [7], ami tulajdonképpen az egységnyi négyzet átlója, és amiről azt tartják, hogy Hippaszosz[8] látta be elsőként, hogy nem lehet racionális szám.

Egyéb *híres* irracionális számok:

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} = 1.6118\ldots, \text{az aranyarány} \\ \pi &= 3.1415\ldots, \text{a pi} \\ e &= 2.7182\ldots, \text{az Euler-féle szám.} \end{aligned}$$

Az irracionális számok között megkülönböztetünk **algebrai** és **transzcendens** számokat. Az algebrai számok azok a számok, amelyek valamely nem nulla, racionális együtthatójú polinomnak gyökei. A transzcendens számokról ugyanezt azonban nem tudjuk kijelenteni. A racionális számok mindegyike algebrai. A $\sqrt{2}$ az x^2-1 polinom egyik megoldása, míg a ϕ az x^2-x-1 polinom egyik megoldása lesz. A π esetében azonban nem tudunk olyan racionális együtthatójú polinomot szerkeszteni, amelynek π a gyöke

7 Püthagorasz görög matematikus (i. e. 570 – i. e. 495)
8 Hippaszosz görög matematikus (i. e. 530 – i. e. 450)

lenne. Ugyanilyen tulajdonságú az e szám is. Ez azt jelenti, hogy a $\sqrt{2}$ és ϕ amellett, hogy irracionális számok, algebrai számok is, a π és az e azonban nem algebrai számok, hanem transzcendens számok.

A számítástechnika az irracionális számok közelített értékét tudja meghatározni, és a gyakorlatban is ezekkel a közelített értékekkel dolgozik. Minél több tizedesjegyét határozzuk meg valamely irracionális számnak, annál pontosabban közelítjük meg a számot.

A Python a `float` típusú adatokat 15 tizedesjeggyel tárolja:

```
>>> 2**0.5
    1.4142135623730951
>>> (1 + 5**0.5)/2
    1.618033988749895
>>> from math import pi, e
>>> pi
    3.141592653589793
>>> e
    2.718281828459045
>>> len(str(e)) #az egészrésszel együtt a számjegyek száma
    17
```

Néha sokkal több tizedesjegyre is szükség lehet. Például a számítástechnikai rendszerek fejlettségét bizonyítja, hogy 2022-ben a π-nek több milliárd számjegyét sikerült kiszámítani. A tizedesjegyek számát Pythonban a **`decimal`** modulban található `Decimal` típus használatával, illetve az itt definiált `getcontext` metódus alkalmazásával tudjuk szabályozni.

A következő lekérdezések a `float` és `Decimal` típusú adatokon végzett műveletek esetében mutatják a különbségeket. A `decimal` modulban található függvények importálását most úgy adjuk meg, hogy a függvények hívásakor ne kelljen használni a modulnevet.

```
>>> 0.1 + 0.1 - 0.2
    0.0
>>> 0.1 + 0.1 + 0.1 - 0.3
    5.551115123125783e-17

>>> from decimal import *
>>> d1 = Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
>>> d1 - Decimal('0.3')
    Decimal('0.0')

>>> d2 = Decimal(0.1) + Decimal(0.1) + Decimal(0.1)
```

```
>>> d2 - Decimal(0.3)
    Decimal('2.77555756156515654042363166 8E-17')
```

Vegyük észre, hogy a kerekítési hibák miatt a második műveletsor nem ad helyes eredményt. Ugyancsak helytelen lesz az eredmény az utolsó műveletsor esetében is. Amikor egy `Decimal` típusú adatot egy `str` típusú adat alapján hozunk létre, kiküszöbölhetjük a kerekítési hibákat, míg ha `float` típusról alakítunk, megmaradnak a kerekítési hibák. Kerekítési hiba adódik a `log` függvény alkalmazásakor is, amelyet `decimal` modulban levő `log10` metódus használatával tudunk megoldani:

```
>>> from math import log
>>> log(10**6, 10)
    5.999999999999999
>>> (Decimal(10)**6).log10()
    Decimal('6')
```

A Python alapbeállítása szerint a `Decimal` típus 28 tizedesjeggyel dolgozik, ezt a `getcontext.prec()` segítségével tudjuk megváltoztatni:

```
>>> x, y = Decimal(1), Decimal(3)
>>> getcontext().prec = 10
>>> x / y
    Decimal('0.3333333333')
>>> getcontext().prec = 25
>>> x / y
    Decimal('0.3333333333333333333333333')
```

A következő algoritmusok irracionális számok számjegyeit generálják ki, lánctörtképletek alkalmazásával.

4.17. algoritmus. Írjunk egy Python függvényt, amely meghatározza $\sqrt{n}$ több tizedesjegyét.

Tetszőleges a értékre könnyedén belátható a következő egyenlőség:

$$\sqrt{n} = a + \frac{n - a^2}{a + \sqrt{n}}.$$

Ha a fenti képletben az a értékét 1-gyel, és a nevezőben a $\sqrt{n}$ értékét ismételten a képlet szerinti összefüggéssel helyettesítjük, akkor a következő lánctörtet kapjuk:

$$\sqrt{n} = 1 + \frac{n-1}{1+\sqrt{n}} = 1 + \frac{n-1}{1+1+\frac{n-1}{1+\sqrt{n}}} = 1 + \cfrac{n-1}{2+\cfrac{n-1}{2+\cfrac{n-1}{2+\cfrac{n-1}{2+\ldots}}}}$$

A kapott lánctört alkalmas lesz $\sqrt{n}$ több tizedesjegyének a meghatározásához:

```
def mySqrt_(n):
    temp = 0
    for x in range(0, 30):
        temp = (n - 1) / (2 + temp)
    return 1 + temp

>>> mySqrt_(3)
    1.7320508075688772
```

Ha több tizedesjegyet szeretnénk, akkor `Decimal` típussal kell dolgoznunk, és a `getcontext` segítségével meg kell adnunk a tizedesjegyek számát.

A következő `mySqrt` függvény 100 tizedes jeggyel számol, a `for` ciklus pedig 300-szor fog lefutni.

```
from decimal import *
def mySqrt(n):
    getcontext().prec = 100
    temp = Decimal(0)
    for x in range(0, 300):
        temp = (n - 1) / (2 + temp)
    return 1 + temp

>>> mySqrt(2)
    Decimal('1.41421356237309504880168872420969807...
```

$\sqrt{n}$ tizedesjegyeit meg lehet határozni a Newton-féle[9] módszerrel is, amelynek alapja a következő számítás [37]:

$$\begin{aligned} x_0 &= 1 \\ x_{i+1} &= \frac{1}{2} \cdot \left(x_i + \frac{n}{x_i}\right). \end{aligned}$$

9 Sir Isaac Newton angol fizikus, matematikus (1642 – 1727)

Az iterációt akkor kell leállítani, amikor az x_{i+1}-ben kiszámolt érték megegyezik x_i-vel. A keresett eredmény az utolsónak kiszámolt x_{i+1} érték lesz. A Python függvény a következő:

```
def newtonSqrt1(n):
    getcontext().prec = 100
    n = Decimal(n)
    x0 = 1
    while True:
        xi = (x0 + n/x0) / 2
        if xi == x0: return xi
        x0 = xi

>>> newtonSqrt1(2)
    Decimal('1.414213562373095048801688724209698078569671875376948073176679737990732478462107038850387534327641572')
```

A `while`-ban megadott számítási folyamatot leállíthatjuk akkor is, amikor az x_{i+1} és x_i közötti különbség kisebb lesz, mint ε, amely értéket akár a programozó, akár a felhasználó is meghatározhatja. Ez egy kis érték kell legyen, például 0.000000001. A következő függvényben a második paraméter az ε szerepét tölti be és alapértelmezett értéket kap. Ezt a felhasználó a `newtonSqrt2` meghívásakor tetszőleges értékre módosíthatja, ahogy az a második meghívásnál látható:

```
def newtonSqrt2(n, eps = Decimal('0.000000001')):
    getcontext().prec = 100
    n = Decimal(n)
    x0 = 1
    while True:
        xi = (x0 + n/x0) / 2
        if abs(xi - x0) < eps: return xi
        x0 = xi

>>> newtonSqrt2(2)
    Decimal('1.41421356237309504880168962350253024361498192577619742849828949862319582422892362178494183673583035 6')

>>> newtonSqrt2(2, 0.00000000000000000000000000001)
    Decimal('1.41421356237309504880168872420969807856967
```

```
    71875376948073176679737990732478462107038850387534327641602')
```

A számítási sorozat általánosítható, Newton módszere alkalmas a $\sqrt[k]{n}$ meghatározásához, a képlet a következő:

$$\begin{aligned} x_0 &= 1 \\ x_{i+1} &= \frac{1}{\boldsymbol{k}} \cdot \left((\boldsymbol{k-1}) \cdot x_i + \frac{n}{x_i^{\boldsymbol{k-1}}} \right) \end{aligned}$$

A legrégebbi értelmezések szerint az euklideszi geometriában a $\boldsymbol{\pi}$ a kör kerületének és átmérőjének hányadosa. E mellett más definíciók is léteznek, amelyek kihagyják a kört, azaz a kör fogalma nélkül határozzák meg a π értékét. Több, lánctörtek segítségével megadott képlet is létezik, amelyek bármelyikét jól lehet alkalmazni több ezer tizedesjegy meghatározására, ezek közül kettőt adunk meg bizonyítás nélkül:

$$\pi = \cfrac{\boldsymbol{4}}{\boldsymbol{1} + \cfrac{1^2}{\boldsymbol{3} + \cfrac{2^2}{\boldsymbol{5} + \cfrac{3^2}{\boldsymbol{7} + \ldots}}}} \qquad \pi = 3 + \cfrac{1}{6 + \cfrac{3^2}{6 + \cfrac{5^2}{6 + \cfrac{7^2}{6 + \ldots}}}}$$

4.18. algoritmus. Írjunk egy Python függvényt, amely meghatározza π első 500 tizedesjegyét.

A `myPi` az első lánctörtet használja, a képernyőre való kiíratást pedig tömbösítve, a számjegyeket tizes blokkokba csoportosítva, soronként öt blokkot írva a `myPrint` függvény végzi.

```
from decimal import *
def myPi():
    getcontext().prec = 500
    temp = Decimal(0)
    for x in range(1500, 0, -1):
        a = x * x
        b = 2 * x + 1
        temp = a / (b + temp)
    res = 4 / (1 + temp)
    myPrint(res)
```

```
def myPrint(val):
    strVal = str(val)
    print(strVal[:2], end = '\n')
    l = 0
    for i in range(2, len(strVal), 10):
        print(strVal[i:i+10], end = ' ')
        if l % 5 == 4: print()
        l += 1

>>> myPi()
    3.
    1415926535 8979323846 2643383279 5028841971 6939937510
    5820974944 5923078164 0628620899 8628034825 3421170679
    ...
    0744623799 6274956735 1885752724 8912279381 830119490
```

A `myPrint` első utasítása a paraméter `val` értékét alakítja át `str` típusú értékké. A π számjegyeit azért érdemes `str` típussá átalakítani, mert a Python szeletelési műveletei segítségével könnyedén meg lehet valósítani a kért kiíratási formát. A `myPrint` a `myPi` utolsó sorában kerül meghívásra, paramétere a `res` lesz.

Az iteratív megoldás mellett rekurzív algoritmus is megadható. Két függvényt fogunk írunk, a `myPiR` az inicializálásokat fogja végezni, és meghívja az `auxPi` függvényt, ami a tulajdonképpeni számításokat végzi. Ezzel a megoldással nem tudunk 500 számjegyet kigenerálni, mert az futási hibát eredményezne, ezért csak 100 számjegyet írunk, és az `x` értékét 1500 helyett 300-ra inicializáljuk:

```
def myPiR():
    getcontext().prec = 100
    return 4 / (1  + auxPi(1))

def auxPi(x):
    if x == 300: return Decimal(0)
    temp = auxPi(x + 1)
    a = x * x
    b = 2 * x + 1
    return a / (b + temp)

>>> myPrint(myPiR())
    ...
```

Az e szám a π mellett a természettudományok egyik legfontosabb mennyisége. Többféleképpen is lehet értelmezni, ezek közül az egyik a következő:

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n.$$

Az e tizedesjegyeit a következő lánctörttel is ki lehet számolni:

$$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{2}{3 + \cfrac{3}{4 + \ldots}}}}$$

4.19. algoritmus. Írjunk egy Python függvényt, amely kiírja egy állományba az Euler[10]-féle e szám első 10000 tizedesjegyét. A meghatározott számjegyeket tízes blokkokba csoportosítsuk, és az állományba soronként 10 blokkot írjunk.

A `myEuler` a megadott lánctörtet használja az e tizedesjegyeinek a meghatározásához. Az állományba a számjegyeket a `myPrintFile`-lal írjuk, ezért a számjegyeket `str` típusú értékké alakítjuk, majd a Python szeletelési operátorát használjuk a részszámjegyek meghatározásához. Minden tizedik blokk után az állományba a `out.write('\n')`-al egy újsort írunk:

```
from decimal import *
def myEuler():
    getcontext().prec = 10000
    temp = Decimal(0)
    for x in range(50000, 0, -1):
        a = x
        b = x + 1
        temp = a / (b + temp)
    res = 2 + 1 / (1 + temp)
    myPrintFile(res)

def myPrintFile(val):
    strVal = str(val)
    out = open('eulerDigit.txt', 'wt')
    out.write(strVal[:2] + '\n')
```

10 Leonhard Euler svájci matematikus (1707 – 1783)

```
    l = 0
    for i in range(2, len(strVal), 10):
        out.write(strVal[i : i+10] + ' ')
        if l % 10 == 9: out.write('\n')
        l += 1
    out.close()

>>> myEuler()
```

Futtatáskor legyünk türelmesek, mert a függvény futási ideje nagy, az eredményt pedig ne a képernyőn, hanem az aktuális mappában az `eulerDigit.txt` állományban keressük.

Lánctörtek segítségével a szögfüggvények értékei is meghatározhatóak. A **sin(*z*)** értékéről ismert, hogy egyenlő a következő lánctörttel:

$$\sin(z) = \cfrac{z}{1 + \cfrac{z^2}{2 \cdot 3 - z^2 + \cfrac{2 \cdot 3 \cdot z^2}{4 \cdot 5 - z^2 + \cfrac{4 \cdot 5 \cdot z^2}{6 \cdot 7 - z^2 + \ldots}}}}$$

Általában a programozási nyelvek radiánban várják a szögfüggvények bemenetét, a `math` könyvtárcsomagban található szögfüggvényeknek is radiánban kell megadni a bemeneti paraméter értékét, ellenkező esetben nem az elvárt eredményt kapjuk:

```
from math import *
>>> sin(60)
    -0.3048106211022167
>>> sin(60 * pi/180)
    0.8660254037844386
>>> sin(radians(60))
    0.8660254037844386
```

Az első lekérdezés eredménye helytelen, mert $\sin(60°) = \frac{\sqrt{3}}{2} = 0.8660254037844386$. Amint látható, a helyes lekérdezést kétféleképpen is meg lehet adni. Az első lekérdezésben képlet alkalmazásával alakítjuk át a 60°-t radiánná, a második esetben pedig a `math` csomagban található `radians` függvénnyel.

4.20. algoritmus. Írjunk egy Python függvényt, amely meghatározza a `sin(z)` első `n` tizedesjegyét, ahol `z` egy radiánban megadott értéket jelent.

```
from decimal import *
from math import *
def mySin(z, n):
    getcontext().prec = n
    temp = Decimal(0)
    p = 2 * n
    for x in range(p, 0, -2):
        a = (x+2) * (x+3) - z*z
        b = x * (x+1) * z*z
        temp = b / (a + temp)
    temp = z*z / (2*3 - z*z + temp)
    return z / (1 + temp)

>>> mySin(Decimal(60*pi/180), 25)
    Decimal('0.8660254037844385893455090')
>>> mySin(Decimal(radians(45)), 35)
    Decimal('0.70710678118654750275194295621751674')
>>> mySin(Decimal(radians(90)), 20)
    Decimal('1.0000000000000000000')
```

A `mySin` függvény második paramétere, az `n` a tizedesjegyek számát jelöli. A függvénytörzsben inicializált `p` a `for` ciklus lépésszámát, azaz a pontosságot jelöli, értékét azért duplázzuk meg, mert az eredmény nem lesz helyes páratlan `n` bemenet esetében.

A többi szögfüggvényre is léteznek lánctörtképletek, de értékük meghatározható a fenti szinuszfüggvény, illetve a trigonometrikus összefüggések alkalmazásával:

$$\sin^2(z) + \cos^2(z) = 1, \quad \mathrm{tg}(z) = \frac{\sin(z)}{\cos(z)}, \quad \mathrm{ctg}(z) = \frac{1}{\mathrm{tg}(z)}.$$

A φ az aranymetszés, az aranyarány (golden ratio) értéke megjelenik mind a matematikában, mind az építészetben, a művészetekben, a természetben stb. Matematikailag az mondjuk, hogy két mennyiség, legyenek ezek a és b, ahol $a > b$ az **aranymetszés** szerint aránylik egymáshoz, ha fennáll:

$$\varphi \stackrel{def}{=} \frac{a}{b} = \frac{a+b}{a}.$$

A φ meghatározása érdekében az egyenlet átalakítható:

$$\varphi = \frac{a}{b} = \frac{a+b}{a} = 1 + \frac{1}{\frac{a}{b}} = 1 + \frac{1}{\varphi}.$$

Az egyenlőség bal és jobb oldali tagjai alapján a következő másodfokú egyenletet kapjuk:

$$\varphi = 1 + \frac{1}{\varphi} \Leftrightarrow \varphi^2 = \varphi + 1$$

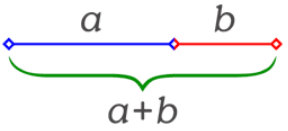
φ értékét tehát a fenti egyenletet megoldva lehet meghatározni:

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots \text{ és } \hat{\varphi} = \frac{1-\sqrt{5}}{2} = -0.61803\ldots$$

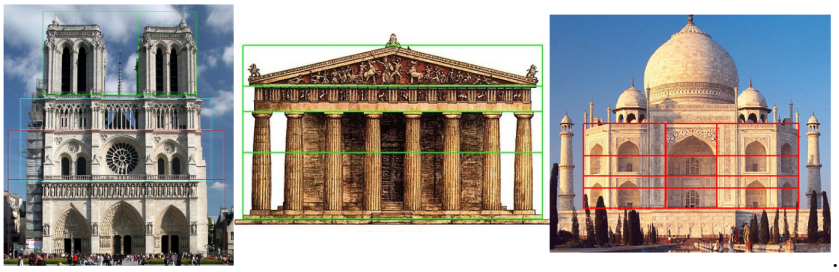
A korábban megírt `mySqrt` (4.17. algoritmus) tehát alkalmas lesz φ első 100 számjegyének a meghatározására:

```
>>> (mySqrt(5) + 1) / 2
    Decimal('1.6180339887498948482045868343656381177203091798057628621354486227052604628189024497072072041893911138')
```

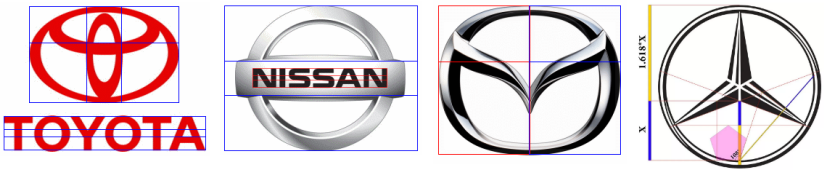
A φ arányt szemléletesen a következő ábra jeleníti meg:



Az aranymetszést számos helyen használják, mert mind a régi görögök, mind a mai művészek, dizájnerek is szépnek tartják azokat az alakzatokat amelyek felépítésében szerepet kap ez az érték. Az athéni Parthenon homlokzatának esetében az épület szélessége és magassága közötti arány egyenlő φ-vel, de az épületen további aranymetszés szerinti arányosságok is megfigyelhetők. A párizsi Notre-Dame, illetve az indiai Tádzs Mahal tervezői is az aranymetszés szabályai szerint jártak el, ahogyan ezek a következő képeken ki vannak emelve:

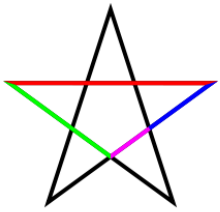


Sok esetben az autólogóknál alkalmazott arányok is az aranymetszés szabályait követik:



.

A Pentagramma, a szabályos ötszög átlóiból alkotott ötágú csillag szakaszainak aránya egyenlő φ-vel:



$$\frac{piros}{zold} = \frac{zold}{kek} = \frac{kek}{lila} = \varphi$$

.

A természetben a napraforgó, a toboz magjai, a pozsgás levelei által meghatározott bal, illetve jobb irányba tartó spirálok számának aránya megegyezik φ-vel:

.

A φ értékét felírhatjuk lánctörtek segítségével is. A $\varphi = 1 + \frac{1}{\varphi}$ rekurzív összefüggésből kiindulva, ha a nevezőbe ismételten a képletet helyettesítjük, akkor a következő végtelen lánctörtet kapjuk:

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{\varphi}} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}}.$$

4.21. algoritmus. Írjunk egy Python függvényt, amely meghatározza a φ első `n` tizedesjegyét a fenti lánctörtképlet alkalmazásával.

```
from decimal import *
def myPhiR():
    n = int(input('tizedesjegyek száma: '))
    x = 3 * n
    getcontext().prec = n
    return auxPhi(Decimal(1), x)

def auxPhi(temp, x):
    if x == 0: return temp
    return auxPhi(1 + 1/temp, x - 1)

>>> myPhiR()
    tizedesjegyek száma: 100
    Decimal('1.6180339887498948482045868343656381177203091798057628621354486227052604628189024497072072041893911
    37')
```

Az `auxPhi` rekurzív függvényben eltértünk a korábban megírt `auxPi`-nél alkalmazott elgondolástól. Az `auxPhi` a kívánt eredményt úgy határozza meg, hogy amikor önmagát hívja, az első paraméter értékét az új értékre módosítja, mondhatjuk azt, hogy akkor számol, amikor *megy be a rekurzióba*. A rekurzió legalsó szintjén a `temp` változóban meg lesz határozva az eredmény, ezért az `if == 0` feltétel teljesülésekor ezt az értéket adja vissza a függvény.

A φ közelített értéke két egymás utáni Fibonacci-szám arányaként is felírható.

4.3. értelmezés. Ha F_{n+1} az $n+1$-edik és F_n az n-edik Fibonacci-szám, akkor fennáll:

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \varphi.$$

4.22. algoritmus. Írjunk egy Python függvényt, amely megadott n és m értékek esetében meghatározza az F_i, F_{i-1} Fibonacci-számokat, és az $\dfrac{F_i}{F_{i-1}}$ arányokat, ahol $n \leq i < m$.

```
def fibPhi(n, m):
    f1 = 0
    f2 = 1
```

```
    for i in range(2, m):
        f = f1 + f2
        if n <= i:
            print("%4d%15d%15d%17.13f" % (i, f, f2, f/f2))
        f1 = f2
        f2 = f

>>> fibPhi(50, 55)
    50    12586269025     7778742049  1.6180339887499
    51    20365011074    12586269025  1.6180339887499
    52    32951280099    20365011074  1.6180339887499
    53    53316291173    32951280099  1.6180339887499
    54    86267571272    53316291173  1.6180339887499

>>> fibPhi(10, 15)
    10             55             34  1.6176470588235
    11             89             55  1.6181818181818
    12            144             89  1.6179775280899
    13            233            144  1.6180555555556
    14            377            233  1.6180257510730
```

A fenti lekérdezésekből jól látszik, hogy az 50-dik és 49-dik Fibonacci-számok aránya teljesen jól megközelíti a φ első 12 számjegyét, míg a 10-dik és 9-dik Fibonacci-számok aránya még nem pontos.

4.23. algoritmus. Írjunk egy Python függvényt, amely a következő képletet alkalmazva meghatározza az n-dik Fibonacci-számot.

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(1-\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \mathbf{round}\left(\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right)$$

```
from decimal import *
def fibF1(n):
    getcontext().prec = 500
    temp = Decimal('5')
    temp = temp.sqrt()
    phi = (1 + temp)/2
    return round((phi**n)/temp)

>>> fibF1(1000)
    434665...228875
```

A pontos érték meghatározásához a `Decimal` típusú értékek esetében alkalmazható `sqrt` metódust hívtuk meg. A `math` modulban található `sqrt` függvény alkalmazásával nagy számok esetében pontatlan eredményt kaptunk volna.

4.7. Valós számok

A valós számok halmazát a racionális és irracionális számok halmazának egyesítése alkotja, ugyanakkor egy szám egyszerre nem lehet racionális és irracionális is. A valós számok halmazát $\mathbb{R}$-rel jelöljük, és fennállnak tehát a következők: $\mathbb{R} = \mathbb{Q} \cup \mathbb{Q}^*$, $\mathbb{Q} \cap \mathbb{Q}^* = \emptyset$.

A valós számokkal végzett műveletek tulajdonságai a következők:

1. az összeadás, szorzás kommutatív, asszociatív műveletek,
2. az összeadás a szorzásra nézve disztributív művelet,
3. a valós számok halmaz zárt az összeadásra, kivonásra, szorzásra, osztásra nézve,
4. az összeadásra, szorzásra nézve minden elemnek lesz inverz eleme,
5. a valós számokhoz hozzárendelhető egy mindkét irányban végtelen egyenes egy pontja,
6. a valós számok halmaza nem megszámlálható.

A számítástechnikában a valós számok közelített értékével dolgozunk, ábrázolásuk eltér az egész számok ábrázolásától: lebegőpontos ábrázolási technika szerint tárolják őket, amelyre a jegyzet keretén belül nem térünk ki.

A matematikában a valós számok legismertebb alakja a tizedestört alak, de a valós számokat lánctört alakban is felírhatjuk, amelyet a tizedestört alak alapján könnyedén meg lehet határozni.

4.24. algoritmus. Írjunk egy Python függvényt, amely a bemenetként megadott valós szám tizedes alakja alapján meghatározza a megfelelő **egyszerű** lánctörtjegyeket.

```
def lanctValos(nr):
    L = []
    for i in range(len(str(nr))):
        eR = int(nr)
        L += [eR]
        tR = nr - eR
```

```
        if tR == 0: break
        nr = 1 / tR
    return L

>>> lanctValos(pi)
    [3,7,15,1,292,1,1,1,2,1,3,1,14,3,3,23,1]
>>> lanctValos(myPhiR())
    tizedesjegyek száma: 100
    [1, 1, 1, 1, 1, 1, 1...
```

A számok mellett egy fontos matematikai struktúra a polinom, amely formálisan a következő kifejezéssel adható meg:

$$A(c) = a_n \cdot c^n + \cdots + a_1 \cdot c + a_0,$$

ahol az $a_n, \ldots a_1, a_0$ számokat a polinom együtthatóinak mondjuk, a c szám pedig egy változó értéket jelöl. Ha $a_n \neq 0$, akkor az n természetes számot a polinom fokszámának nevezzük. A polinomokkal végezhető műveletekre és azok tulajdonságaira nem térünk ki a jegyzetben, viszont bemutatjuk, hogy a következő algoritmus hogyan alkalmazható matematikai függvények ábrázolására.

4.25. algoritmus. Írjunk egy Python függvényt, amely megadott értékre meghatározza egy adott polinom helyettesítési értékét.

A bemutatásra kerülő algoritmus a Horner[11] módszer nevet viseli. A `poly` függvénynek két bemenete lesz, az első a `c` változó, a helyettesítési értéket jelöli, a második pedig a polinom együtthatóit mint listaelemeket. A lista első eleme a polinom legnagyobb fokszámú elemének együtthatója lesz, például $A = [a_n, \ldots a_1, a_0]$. Az együtthatókat pedig a következőképpen dolgozzuk fel:

$$\begin{aligned} A(c) &= a_n \cdot c^n + \cdots + a_1 \cdot c + a_0 = \\ &= a_0 + c \cdot (a_1 + \cdots + c \cdot (a_{n-2} + c \cdot (a_{n-1} + c \cdot a_n)) \ldots). \end{aligned}$$

```
def horner(c, A):
    res = 0
    for a in A:
        res = res * c + a
    return res
```

11 William George Horner brit matematikus (1786–1837)

```
>>> horner(5, [4.3,7.1,0,8.5,2.2,1.7,9.5])
    90510.5
```

Függvények ábrázolásához szükség lesz a `numpy` és `matplotlib` Python-csomagokra. Ezeket is, mint minden más Python-csomagot, legegyszerűbben a `pip` telepítővel tudjuk feltenni. A Python újabb verziói tartalmazzák alapból a `pip`-et, de korábbi verziók esetében szükség lehet a `pip` telepítésére:

1. Windows alatt, ha eddig még nem tettük meg, állítsuk be a `PATH` környezetváltozót (`Edit the system environment variables`) úgy, hogy megadjuk a `python.exe` állomány elérési útvonalát (`Advanced/Environment Variables`)
2. töltsük le egy mappába a megfelelő állományt:

   https://bootstrap.pypa.io/get-pip.py

3. nyissunk meg egy `Command prompt` ablakot, válasszuk ki azt a mappát, amibe a `get-pip` állományt tettük, és adjuk ki a következő parancsokat:
   - `python get-pip.py`
   - `python -m install --upgrade pip`
   - `python -m pip install numpy`
   - `python -m pip install matplotlib`

A letöltött csomagból használni fogjuk a `plot` függvényt, amely a pontokat ábrázolja, a `show` függvényt, amely megjeleníti a képernyőn a teljes ábrát és a `linspace`-t, amely létrehoz egy `n` dimenziós `numpy` típusú tömböt. A `linspace`-nek paraméterként meg lehet adni a kiinduló és befejező elemet, illetve harmadik paraméterként az elemszámot. Az elemszám alapján egyenlő lépésközzel lesznek kigenerálva a tömb elemei. A következő műveletsorokban -5 és 5 között generálunk ki 10 számot:

Példa:

```
>>> import numpy as np
>>> X = np.linspace(-5, 5, 10)
>>> print(X)
    [-5.   -3.88888889 -2.77777778 -1.66666667 -0.55555556
    0.55555556 1.66666667  2.77777778  3.88888889  5.    ]
```

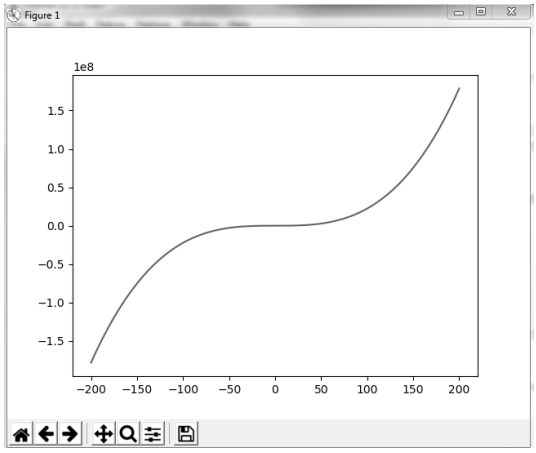
Az aritmetikai műveletek felül vannak írva. A következő példában ez azt fogja jelenteni, hogy a + művelettel X minden eleméhez automatikusan hozzáadunk 10-et, illetve a ** művelettel X minden elemét hatványozni fogjuk:

Példa:

```
>>> print(X + 10)
    [ 5.   6.11111111  7.22222222  8.33333333  9.44444444
    10.55555556 11.66666667 12.77777778 13.88888889 15.   ]
>>> print(X ** 3)
    [-125.     -58.81344307  -21.43347051   -4.62962963
    -0.17146776   0.17146776    4.62962963   21.43347051
    58.81344307  125.   ]
```

4.26. algoritmus. Írjunk egy Python függvényt, amely ábrázol egy matematikai függvényt, ahol a függvény valós típusú együtthatóit egy bemeneti lista elemeiként adjuk meg.

```
import numpy as np
import matplotlib.pyplot as plt

def myPlot(A):
    X = np.linspace(-200, 200, 300)
    F1 = horner(X, A)
    plt.plot(X, F1, label = "F1")
    plt.show()

>>> myPlot([22.3, 7.5, -9.2, 6.8])
```



A kódsorban az `X` a megadott paraméterek szerint összesen 300 egyenlő lépésközzel kigenerált értékeket fog tartalmazni, a legkisebb a -200-as, a legnagyobb a 200-as lesz. A `horner` függvény `X` minden elemére meghatározza a helyettesítési értéket, és eredményként egy `numpy ndarray`-t ad vissza, az `F1`-et.

4.8. Komplex számok

A komplex számok halmazát $\mathbb{C}$-vel jelöljük és elemeit a valós számhalmaz bővítésével kapjuk úgy, hogy értelmezzük a negatív számok esetében a gyökvonást. A komplex számok halmaza három modell alapján is értelmezhető: halmazelméleti, geometriai, illetve algebrai modell alapján.

A halmazelméleti modell alapján $\mathbb{C} = \{(a, b)|a \in \mathbb{R}, b \in \mathbb{R}\}$, azaz a komplex számok halmazára tekinthetünk úgy, mint valós számok rendezett párjaira. A komplex számok $a + bi$ alakban írhatóak fel, ahol a-t valós résznek, b-t imaginárius résznek, i-t pedig imaginárius egységnek hívjuk, és amelyre fennáll: $i^2 = -1$. Amikor $b = 0$, akkor valós számot kapunk [12].

Az $a = a_1 + a_2 i$ és $b = b_1 + b_2 i$ komplex számokon az arimetikai műveletek a következőképpen vannak értelmezve, :

$$\begin{aligned}
\text{abs}(a) &= \sqrt{a_1^2 + a_2^2}, \text{ a komplex szám „nagysága”} \\
a + b &= (a_1 + b_1) + (a_2 + b_2)i \\
a - b &= (a_1 - b_1) + (a_2 - b_2)i \\
a \cdot b &= (a_1 \cdot b_1 - a_2 \cdot b_2) + (a_2 \cdot b_1 + a_1 \cdot b_2)i \\
\frac{1}{b} &= \frac{b_1}{(b_1^2 + b_2^2)} - \frac{b_2}{(b_1^2 + b_2^2)}i \\
\frac{a}{b} &= a \cdot \frac{1}{b}
\end{aligned}$$

A komplex számok esetében értelmezettek a következő értékek:

- additív semleges elem: $z = 0 + 0 \cdot i$, inverz elem: $-z = -a - b \cdot i$,
- multiplikatív semleges elem: $z = 1 + 0i$, inverz elem:

$$\frac{1}{z} = \frac{a}{a^2 + b^2} - \frac{b}{a^2 + b^2} \cdot i, z \neq 0$$

A komplex számok esetében az összeadás, szorzás kommutatív, asszociatív műveletek, illetve az összeadás a szorzásra nézve disztributív.

Pythonban komplex számokkal a beépített `complex` típus alkalmazásával tudunk dolgozni, ahol az aritmetikai operátorok felül vannak írva [38].

```
>>> a = complex(2, 4)
>>> a.imag, a.real, abs(a)
    (4.0, 2.0, 4.47213595499958)
```

Értékadáskor nem szükséges explicit módon a `complex` típussal dolgozni, megengedett a következő is, hatványozáshoz azonban nem lehet a `pow`-t használni, csak a `**`-t:

```
>>> b = 1 + 10j
>>> a + b, a - b, a * b
    ((3+14j), (1-6j), (-38+24j))
>>> a / b
    (0.4158415841584159 - 0.15841584158415842j)
>>> a ** b
    (-6.463391276556521e-05-2.5654375165195912e-05j)
```

4.27. algoritmus. Írjunk egy Python függvényt, amely az a, b, c bemeneti paraméterek esetében meghatározza az $a \cdot x^2 + b \cdot x + c = 0$ másodfokú egyenlet gyökeit.

```
from math import *
def mEgyenlet (a, b, c):
    if a == 0:
        if b == 0: return 'nincs megoldás'
        else: return -c/b
    delta = b*b - 4*a*c
    if delta < 0:
        valosR = -b / (2*a)
        imagR = sqrt( abs (delta)) / (2*a)
        gy1 = complex(valosR, imagR)
        gy2 = complex(valosR, -imagR)
        return (gy1, gy2)
    if delta == 0:
        return -b / (2*a)
    if delta > 0:
        gy1 = (-b + sqrt(delta)) / (2*a)
        gy2 = (-b - sqrt(delta)) / (2*a)
        return (gy1, gy2)
```

```
>>> mEgyenlet(0, 4, 1)
    -0.25
>>> mEgyenlet(0, 0, 1)
    'nincs megoldás'
>>> (x1, x2) = mEgyenlet(1, 1, 1)
>>> x1, x2
    ((-0.5+0.866025403784438j), (-0.5-0.866025403784438j))
>>> print("%.2f + %.4fI" % (x1.real, x1.imag))
    -0.50 + 0.8660I
>>> print("%.2f + %.4fI" % (x2.real, x2.imag))
    -0.50 + -0.8660I
```

Az `mEgyenlet` első `if` feltétele azt az esetet kezeli, amikor a bemeneti paraméterek alapján egy elsőfokú egyenlet gyökét kell meghatározni. A második `if` feltétel a beépített `complex` típus segítségével határozza meg negatív `delta` esetében a komplex gyököket.

Ha a `python -m pip install sympy` paranccsal telepítjük a `sympy` modult, akkor a **`solve`** metódussal is meg tudjuk határozni egy másodfokú egyenlet gyökeit:

```
>>> import sympy
>>> x = sympy.Symbol('x', complex = True)
>>> sympy.solve(4*x + 1, x)
    [-1/4]
>>> sympy.solve(1, x)
    []
>>> sympy.solve(x**2 + x + 1, x)
    [-1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

A Python `sympy` moduljában található `solve` egy általános célú, különböző típusú egyenletek megoldására alkalmas metódus, két paraméteres, ahol első paraméterként az egyenletet kell megadni, másodiknak pedig azt a szimbólumot, ami szerint az egyenletet szeretnénk megoldani.

4.28. algoritmus. Írjunk egy Python függvényt, amely meghatározza a^n-t, ahol $a = (a_1, a_2)$ egy komplex, n pedig egy egész szám. A komplex számok kezelését valós számpárokkal végezzük, és ne használjuk a beépített `complex` típust.

A `my_powC` függvény a gyorshatványozás gondolatmenete alapján határozza meg az eredményt. Két komplex szám szorzatának, illetve egy komplex

szám négyzetének a meghatározásához pedig meghívja a `szorzatC`-t függvényt. A `szorzatC`-nek bemenetként egy számpárt kell megadni, amelyek egy-egy komplex szám valós, illetve imaginárius részét jelölik.

```
def szorzatC(a, b):
    a1, a2 = a
    b1, b2 = b
    vResz = a1 * b1 - a2 * b2
    iResz = a2 * b1 + a1 * b2
    return (vResz, iResz)

def my_powC (a, n):
    res = (1, 0)
    while True:
        if n % 2 == 1:
            res = szorzatC(res, a)
        if n == 1: break
        a = szorzatC(a, a)
        n = n // 2
    return res

>>> szorzatC((2.5, 5.4), (3.34, 101.5))
    (-539.75, 271.786)
>>> my_powC((2.5, 5.4), 3)
    (-203.07500000000002, -56.21400000000003)
```

Komplex számokat fraktálok szerkesztésekor is használunk [10]. A fraktálok különböző ismétlődő minta szerint megjelenített érdekes alakzatok. Az alakzat pontjait egy komplex számból kiindulva, egy iterációs folyamat során számítjuk ki. A kirajzolt alakzatot a választott komplex szám, illetve az iterációk száma határozza meg. A következőkben a Mandelbrot- és Julia-fraktálokkal fogunk foglalkozni.

A Mandelbrot-fraktál[12], azaz a Mandelbrot-halmaz elemeinek a meghatározásához a következő iterációs képletet kell használni, ahol a c egy komplex szám:

$$\begin{aligned} z_0 &= c \\ z_{n+1} &= z_n^2 + c, \end{aligned}$$

A Mandelbrot-halmaz azokat a c komplex számokat fogja tartalmazni, amelyekre a z_n sorozat nem tart a végtelenbe. A z_n sorozat a végtelenbe tart,

12 Benoit B. Mandelbrot lengyel születésű matematikus (1924–2010)

ha az iteráció során valamely z_n komplex szám abszolút értéke nagyobb lesz mint 2.

4.29. algoritmus. Írjunk egy Python függvényt, amely megjeleníti a Mandelbrot-halmaz elemeit a képernyőn. Egy halmazbeli elem esetén #-t, másképp szóközt írjunk a képernyőre.

```
def fMandl():
    L1 = [a * 0.07 for a in range(-15, 15)]
    L2 = [a * 0.04 for a in range(-40, 40)]
    for y in L1:
        L = ''
        for x in L2:
            z = complex(x, y)
            c = z
            for i in range(40):
                z = z**2 + c
                if abs(z) > 2:
                    L += ' '
                    break
            if abs(z) <= 2: L += '#'
        print(L)
```

A kódsorban az `L1, L2` listaelemek kombinálásával több, különböző komplex számot határoztunk meg, amelyek mindegyikét legtöbb 40-szer iteráltuk. Az `L1` halmazból kerülnek ki a számok imaginárius részei, míg az `L2` halmazból a valós részek. Ha az `L1, L2` intervallumokat módosítjuk, akkor kisebb vagy nagyobb, esetleg torzabb alakzatot kapunk. A függvény meghívása után megtekinthetjük a kirajzolt alakzatot.

A Julia[13]-halmaz iterációs képlete ugyanaz, mint a Mandelbrot-halmazé, azzal a különbséggel, hogy a c értéke konstans, és a függvény meghívásakor kell megadni mint bemeneti paramétert. Aszerint, hogy a c értékét milyen komplex számmal inicializáljuk, különböző alakzatokat kapunk.

4.30. algoritmus. Írjunk egy Python függvényt, amely megjeleníti a Julia halmaz elemeit a képernyőn. Egy halmazbeli elem esetén #-t, másképp szóközt írjunk a képernyőre.

```
def fJulia(c):
    L1 = [a * 0.07 for a in range(-15, 15)]
```

13 Gaston Maurice Julia francia-algériai matematikus (1893–1978)

```
    L2 = [a * 0.04 for a in range(-40, 40)]
    for y in L1:
        L = ''
        for x in L2:
            z = complex(x, y)
            for i in range(40):
                z = z**2 + c
                if abs(z) > 2:
                    L += ' '
                    break
            if abs(z) <= 2: L += '#'
        print(L)
```

Érdemes több bemeneti értékekre futtatni az `fJulia` függvényt, mert érdekesebbnél érdekesebb alakzatokat kapunk, próbáljuk ki a következő függvényhívásokat:

```
>>> fJulia(complex (0.285, 0.013))
>>> fJulia(complex (-0.295, -0.55))
>>> fJulia(complex (-0.624, 0.435))
>>> fJulia(complex (-1, -0))
```

Az `fJulia(`**`complex (-1, -0.25)`**`)` meghívás esetén a következő lesz a kirajzolt alakzat:

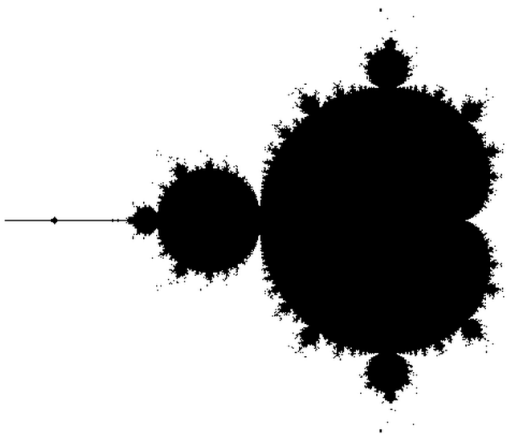
A grafikus, elegánsabb megjelenítéshez szükség lesz a `pygame` csomagra, amelynek telepítése hasonló módon történik, mint a `numpy` vagy más Python-csomag telepítése. A kódsor, amely megjeleníti grafikusan a Mandelbrot-fraktált, a következő, ahol nem térünk ki a grafikus megjelenítéshez használt függvények ismertetésére [23]:

```
from pygame.locals import *
import pygame
```

```
def gMandl():
    width, height = 600, 600
    screen = pygame.display.set_mode((width,height),DOUBLEBUF)
    xaxis, yaxis = width / 1.5, height / 2
    scale, iterations = 190, 40
    for iy in range(height):
        for ix in range(width):
            c = complex((ix - xaxis)/scale, (iy - yaxis)/scale)
            z = c
            for i in range(iterations):
                z = z**2 + c
                if abs(z) > 2:
                    color = (255, 255, 255)
                    break
            if abs(z) <= 2:
                color = (0, 0, 0)
            screen.set_at((ix, iy), color)
            screen.set_at((ix, height - iy), color)
    pygame.display.update()
    while True:
        event = pygame.event.poll()
        if (event.type == QUIT or
            (event.type == KEYDOWN and event.key == K_ESCAPE)):
            break
```

A függvényhívás megjeleníti a Mandelbrot-halmazt. A shellből való futtatás esetén, a megjelenítés után, az ablak bezárásához szükséges kiadni a `pygame.quit()` parancsot:

```
>>> gMandl()
>>> pygame.quit()
```



A Julia-fraktált a következő kódsorral tudjuk kirajzolni, ahol a színezésen is változtattunk:

```
from pygame.locals import *
import pygame
def gJulia(c):
    width, height = 600, 600
    screen = pygame.display.set_mode((width,height),DOUBLEBUF)
    xaxis, yaxis = width / 2, height / 2
    scale, iterations = 170, 40
    for iy in range(height):
        for ix in range(width):
            z = complex((ix - xaxis)/scale, (iy - yaxis)/scale)
            for i in range(iterations):
                z = z**2 + c
                if abs(z) > 2.0:
                    color = (i%16, i%16*8, i%16*16)
                    break
            if abs(z) <= 2:
                color = (0, 0, 0)
            screen.set_at((ix, iy), color)
            screen.set_at((width, height), color)
    pygame.display.update()
    while True:
        event = pygame.event.poll()
        if (event.type == QUIT or
            (event.type == KEYDOWN and event.key == K_ESCAPE)):
            break
```

```
>>> gJulia(c = complex (0, -0.8))
```

Az `gJulia` függvényt is meghívhatjuk ugyanazokra a komplex számokra, mint amelyeket az `fJulia` függvénynél használtunk.

4.9. Kitűzött feladatok

4.1. feladat. Írjunk egy Python függvényt, amely alkalmazva a gyorshatványozás algoritmusát, meghatározza több, a billentyűzetről beolvasott `x, n` érték esetében x^n értékét. Minden hatványozás esetében írassuk ki, hogy hány szorzást végzett az algoritmus.

4.2. feladat. Írjunk egy Python függvényt, amely meghatározza, hogy hány számjegyből áll egy szám 2-es számrendszerben anélkül, hogy átalakítanánk a számot 2-es számrendszerbe.

4.3. feladat. Írjunk egy Python függvényt, amely meghatározza, hogy hány számjegyből áll $n!$, anélkül, hogy meghatározná $n!$ értékét.

4.4. feladat. Írjunk egy Python függvényt, amely meghatározza, hogy hány számjegyből áll $n!$ 2-es számrendszerben anélkül, hogy meghatároznánk $n!$ értékét.

4.5. feladat. Írjunk egy Python függvényt, amely meghatározza, hogy melyik az a legnagyobb n érték, amelyre $n!$ kevesebb mint k számjegyet tartalmaz. Mennyi az eredmény, ha $k = 10000$?

4.6. feladat. Írjunk Python függvényeket, amelyek rendre kiírják egy-egy állományba a következő számsorozatok első `n` tagját, ahol `n` minden esetben a függvény bemeneti paramétere:

- n^3: `1, 8, 27, 64, 125, ...`
- 2^n: `1, 2, 4, 8, 16, ...`
- `1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...`
- `1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, ...`
- `1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, ...`
- `1, 0, 2, 0, 4, 0, 8, 0, 16, 0, ...`
- `2, 4, 16, 256, 65536, 4294967296, ...`

4.7. feladat. A *szamok.txt* állomány minden sorában egy-egy számpár található, szóközzel elválasztva. Írjunk egy Python programot, amely beolvassa ezeket a számokat, és az euklideszi, illetve a kivonásos algoritmussal meghatározza minden számpár esetében a legnagyobb közös osztót.

4.8. feladat. A *szamok.txt* állomány minden sorában egy-egy számpár található, szóközzel elválasztva. Írjunk egy Python programot, amely meghatározza az euklideszi algoritmus során végzett osztások számát, és a kapott értékeket összehasonlítja a Lamé-tétel (4.3. tétel) által meghatározott értékekkel. Az állományban a számpárok között vannak olyanok, amelyek egymást követő két Fibonacci-számból állnak. Ezen számpárok esetében mit veszünk észre az osztások számát illetően?

4.9. feladat. A *szamok.txt* állomány minden sorában egy-egy számpár található, szóközzel elválasztva. Írjunk egy Python programot, amely beolvassa ezeket a számokat, és meghatározza minden számpár esetében a legkisebb közös többszöröst.

4.10. feladat. Írjunk programot, amely meghatározza egy kétismeretlenes diofantoszi egyenlet pozitív megoldásait. Vizsgáljuk meg a kapott megoldásokat a következő feladatok esetében:

1. Egy erdélyi diák az USA-ba utazik, és bevált eurót, illetve lejt dollárra. A beváltás után összesen 17,06 dollárja lesz. Tudva azt, hogy minden euró után 59 dollárcentet és minden lej után 19 dollárcentet kapott, határozzuk meg, hogy mennyi pénze volt a beváltás előtt külön-külön mindkét pénznemből.
2. Egy forgalmazó 5100 euró értékben rendel Samsung és Iphone telefonokat. Tudva azt, hogy minden Samsung 200 euróba, és minden Iphone 500 euróba kerül, határozzuk meg, hogy maximálisan hány Samsung telefont, illetve hány Iphone telefont rendelhetett a forgalmazó.
3. Egy postahivatalban 1,4 és 2,1 lejes bélyegeket árulnak. Határozzuk meg, hogy melyik bélyegből hányat kell feltenni egy adott csomagra, hogy az összérték 7,77 lej legyen.
4. Egy étteremben egy adag vegetáriánus saláta 8 lej, egy adag húsos saláta 11 lej. Ha egy csapat fogyasztása után az összszámla értéke rendre 777 lej, mit tudunk mondani arra vonatkozóan, hogy hányan voltak a csapatból vegetáriánusok és hányan nem?

4.11. feladat. A *diofant.txt* állomány minden sorában egy-egy számhármas található, szóközzel elválasztva, jelöljük őket a, b, c-vel. Írjunk egy Python programot, amely beolvassa ezeket a számhármasokat és meghatározza az $a \cdot x + b \cdot y = c$ diofantoszi egyenlet pozitív megoldásait. Ha valamelyik számhármas esetében nincs pozitív megoldás, írjunk hibaüzenetet a képernyőre.

4.12. feladat. Írjunk Python függvényeket, amelyek rendre meghatározzák két racionális szám összegét, szorzatát, különbségét, hányadosát, ahol a racionális számokat egész számpárokként kezeljük.

4.13. feladat. Írjunk egy olyan Python függvényt, amely meghatározza $\left(\frac{x}{y}\right)^n$ értékét, ahol az $\frac{x}{y}$ racionális számot egész számpárként kezeljük és n egész szám.

4.14. feladat. A racionális számok sorozatba rendezésekor két módszert mutattunk be (4.13, 4.14 algoritmusok). Írjunk egy Python függvényt, amely megmondja, hogy egy adott racionális szám hányadik az egyik, illetve hányadik a másik módszer szerint előállított számsorozatban.

4.15. feladat. Írjunk egy Python függvényt, amely kiírja egy állományba az n-ed rendű Farey-sorozat elemeit, ahol $n > 1000$.

4.16. feladat. Írjunk egy Python függvényt, amely meghatározza az n-ed rendű Farey-sorozatban szereplő racionális számok lánctörtjegyeit.

4.17. feladat. Írjunk Python függvényeket, amelyek rendre kiírják egy állományba a $\sqrt{n}, \pi, e, \varphi, \ln(n), \sin(n), \cos(n), \mathrm{tg}(n), \mathrm{ctg}(n)$ számok első k tizedesjegyét, ahol $k > 100$. Az állományba a számjegyeket tömbösítve írjuk, pontosabban k_1-es blokkokba csoportosítva, soronként k_2 blokkot tegyünk, ahol a blokkok közé írjunk szóközt, és a k_1, k_2 értékeket a billentyűzetről olvassuk be.

4.18. feladat. Írjunk Python függvényeket, amelyek rendre meghatározzák a $\sqrt{n}, \pi, e, \varphi$, $\ln(n), \sin(n), \cos(n), \mathrm{tg}(n), \mathrm{ctg}(n)$ számok k-dik tizedesjegyét, ahol $k > 100$.

4.19. feladat. Határozzuk meg $\ln(z)$ első n tizedesjegyét a következő lánctört segítségével:

$$\ln(1+z) = \cfrac{z}{1 + \cfrac{1^2 \cdot z}{2 + \cfrac{1^2 \cdot z}{3 + \cfrac{2^2 \cdot z}{4 + \cfrac{2^2 \cdot z}{5 + \cfrac{3^2 \cdot z}{6 + \ldots}}}}}}$$

4.20. feladat. Írjunk Python függvényeket, amelyek rendre meghatározzák két komplex szám összegét, szorzatát, különbségét, hányadosát, ahol a komplex számokat valós számpárokként kezeljük.

4.21. feladat. Határozzuk meg $\ln(z)$ első n tizedesjegyét, alkalmazva a következő összefüggéseket:

$$\begin{aligned}\ln(z) &= (z-1) - \frac{(z-1)^2}{2} + \frac{(z-1)^3}{3} - \frac{(z-1)^4}{4} \ldots \\ &= \sum_{n=1}^{\infty} (-1)^{n+1} \cdot \frac{(z-1)^n}{n}\end{aligned}$$

$$\ln(z) = \lim_{n \to \infty} n \cdot (z^{1/n} - 1)$$

4.22. feladat. A `homersekletV.txt` állományban városok időjárásával kapcsolatos adatok vannak. Az állomány első sorában egy városnév van. A második sor a havi maximális átlag hőmérsékleti értékeket (a `maxH` értékeket), a harmadik sor pedig a havi minimális átlag hőmérsékleti értékeket

(a `minH` értékeket) tartalmazza. Ez után üres sor következik, majd további városokra vonatkozó adatok vannak megadva, hasonló módon, mint az első három sorban. Írjunk egy `homersekletF` Python függvényt, amely meghatározza:

1. városonként a legnagyobb `maxH` értékeket, illetve a hónapokat, amikor ezeket az értékeket mérték,
2. városonként azokat a hónapokat, amikor negatív volt a `minH`,
3. azokat a hónapokat, amikor minden városban negatív volt a `minH`.

Ha a `homersekletV.txt` állomány tartalma a következő:

```
Marosvasarhely
2,6,12,18,23,26,28,28,24,18,10,4
-5,-2,1,6,10,13,15,14,11,6,2,-2

Debrecen
1,4,10,17,22,25,27,26,22,17,9,3
-6,-3,1,5,10,13,14,14,10,5,1,-3

Budapest
1,4,10,16,21,24,26,26,22,16,8,3
-4,1,2,6,11,14,15,14,11,7,2,-2
```

akkor az eredmény:

```
>>> homersekletF()
    varosonkent a legnagyobb maxH ertekek:
    Marosvasarhely : 28, julius, augusztus
    Debrecen : 27, julius
    Budapest : 26, julius, augusztus

    varosonkent, amikor negativ volt a minH:
    Marosvasarhely : januar, februar, december
    Debrecen : januar, februar, december
    Budapest : januar, december

    a honapok, amikor minden varosban negativ volt a minH:
    januar, december
```

5. fejezet

Számrendszerek és kódolási technikák

5.1. Természetes szám alapú számrendszerek

A mindennapi életünkben a különböző mennyiségek ábrázolásához a tízes számrendszert használjuk, ezt pedig leginkább azzal lehet magyarázni, hogy az embereknek 10 ujjuk van. De a tízes helyett bármikor választhatunk egy másik értéket, azaz használható más számrendszer is. A különböző természetes szám alapú számrendszerek között létezik egy egyértelmű átalakítási folyamat. Éppen ezért a fejezet keretén belül ezekkel a számrendszerekkel fogunk foglalkozni. Bemutatjuk továbbá a tízes számrendszer és a számítástechnikában leggyakrabban használt számrendszerek közötti átalakítási algoritmusokat [17, 28].

A tízes vagy decimális számrendszerben 10 szimbólum van, amelyeket arab számjegyeknek is mondunk: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Az 53428 mennyiség pedig azt jelenti, hogy van 5 tízezresem, 3 ezresem, 4 százasom, két tízesem és nyolc egységem. Ez matematikailag azt jelenti, hogy a számrendszerem alapja tízes, és felírható a következő: $53428 = 5 \cdot 10^4 + 3 \cdot 10^3 + 4 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0$. A tíz szimbólum segítségével tehát tetszőlegesen nagy mennyiség leírható.

Az informatikai algoritmusok által a leggyakrabban használt számrendszerek a tízes mellett a kettes, a nyolcas, a tizenhatos és a kétszázötvenhatos. A kettes számrendszert **bináris**, a nyolcast **oktális**, a tizenhatost pedig **hexadecimális** számrendszernek is hívjuk. A különböző számrendszerekhez használt szimbólumok ugyancsak az arab számjegyeket használják, kivéve

a tizenhatos számrendszert. A kettes számrendszerben két szimbólum van, ezek a $0, 1$. Nyolcas számrendszerben a $0, 1, 2, 3, 4, 5, 6, 7$ jegyeket használjuk, összesen nyolcat. Tizenhatos számrendszerben tizenhat szimbólum van ezek a $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$ szimbólumok, illetve a kisbetűk helyett a megfelelő nagybetűket is használjuk. Kétszázötvenhatos számrendszerben 256 szimbólum van, ezeket a $0, 1, 2, \ldots 253, 254, 255$ számokkal jelöljük. Megállapítható, hogy a számítógépek bájtjai egy-egy 256-os számrendszerbeli jegynek felelnek meg.

Matematikai számítások során a számrendszer alapját indexként is fel szokták tüntetni, például 3256_{10} tízes számrenszerbeli értéket, míg 11010111_2 kettes számrendszerbeli értéket jelöl, de jelölhetjük a számot szögletes zárójelben is, a szimbólumok közé vesszőt téve: $[1, 1, 0, 1, 0, 1, 1, 1]_2$.

Példák:

1. 215_{10}, bináris alakja $[1, 1, 0, 1, 0, 1, 1, 1]_2$, és fennáll:

$$\begin{aligned} 215 &= 128 + 64 + 16 + 4 + 2 + 1 = \\ &= \mathbf{1} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{0} \cdot 2^5 + \mathbf{1} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{1} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0. \end{aligned}$$

2. 215_{10}, oktális alakja $[3, 2, 7]_8$, és fennáll:

$$215 = \mathbf{3} \cdot 8^2 + \mathbf{2} \cdot 8^1 + \mathbf{7} \cdot 8^0.$$

3. 7432_{10}, hexa alakja $[1, 13, 0, 8]$, vagy $[1, D, 0, 8]_{16}$, mert tizenhatos számrendszerben az $A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$ megfeleltetéseket szokás használni, és fennáll:

$$7432 = \mathbf{1} \cdot 16^3 + \mathbf{13} \cdot 16^2 + \mathbf{0} \cdot 16^1 + \mathbf{8} \cdot 16^0.$$

4. $876512123863001 26_{10}$, 256-os számrendszerbeli alakja a következő: $[1,\ 55, 102, 80, 50, 106, 240, 222]_{256}$, és fennáll:

$$\begin{aligned} 87651212386300126 &= \mathbf{1} \cdot 256^7 + \mathbf{55} \cdot 256^6 + \mathbf{102} \cdot 256^5 + \mathbf{80} \cdot 256^4 + \\ &+ \mathbf{50} \cdot 256^3 + \mathbf{106} \cdot 256^2 + \mathbf{240} \cdot 256^1 + \mathbf{222}. \end{aligned}$$

A Python több beépített függvénnyel is rendelkezik, amelyek a számrendszerek közötti átalakításokat végzik. A `bin` a kettes, az `oct` a nyolcas, míg a `hex` a tizenhatos számrendszerbeli alakot határozza meg a tízes számrendszerbeli érték alapján.

```
>>> bin(215)
    '0b11010111'
>>> oct(215)
    '0o327'
>>> hex(7432)
    '0x1d08'
```

```
>>> bin(215)[2:]
    '11010111'
>>> oct(215)[2:]
    '327'
>>> hex(7432)[2:]
    '1d08'
```

Figyeljük meg, hogy mindhárom függvény kimenetének típusa `str`, ahol az első két karakter jelölő szerepet tölt be, az `0b` a bináris, az `0o` az oktális, az `0x` a hexadecimális alakot jelzi, amelyektől, szükség esetén, a szeletelési operátorral, ahogyan a bal oszlopban láthatjuk, könnyedén meg tudunk szabadulni.

A fordított műveletet az `int` függvénnyel végezzük, ahol a bemenetet `str`-ként kell megadni, és jelezni kell, hogy mire alakítunk:

```
>>> int('11010111', 2)
    215
>>> int('327', 8)
    215
>>> int('1d08', 16)
    7432
```

A `format` függvénnyel is alakíthatunk tízes számrendszerből kettes, nyolcas, tizenhatos számrendszerbe. Ekkor az eredmény előtt nem lesznek jelölő karakterek:

```
>>> format(1023, 'b')
    '1111111111'
>>> format(1023, 'o')
    '1777'
```

```
>>> format(1023, 'x')
    '3ff'
>>> format(1023, 'X')
    '3FF'
```

Általános esetben egy nr tízes számrendszerbeli szám egyértelműen megadható b számrendszerben a következőképpen, ahol $b > 1$, pozitív egész szám:

$$nr = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \cdots + a_1 \cdot b^1 + a_0 \cdot b^0.$$

$\mathbb{Z}^*$-al jelölve a nem negatív egész számok halmazát fennáll továbbá, hogy $k \in \mathbb{Z}^*$, $a_i \in \mathbb{Z}^*$, $a_i < b$, minden $i \in \{0, \ldots, k\}$ értékre és $a_k \neq 0$. Az $a_k, a_{k-1}, \ldots, a_1, a_0$ értékeket a b számrendszerbeli számjegyeknek hívjuk.

A 4.1. tétel alapján a nr tízes számrendszerbeli számot a következőképpen tudjuk alakítani b számrendszerbe:

– nr-t elosztjuk b-vel, ekkor fennáll: $nr = b \cdot q_0 + a_0$, ahol $0 \leq a_0 < b$,

- q_0-t elosztjuk b-vel, ekkor fennáll: $q_0 = b \cdot q_1 + a_1$, ahol $0 \leq a_1 < b$,
- q_1-et elosztjuk b-vel, ekkor fennáll: $q_1 = b \cdot q_2 + a_2$, ahol $0 \leq a_2 < b$,
- mindig az osztási egészrészt osztva b-vel, addig folytatjuk az osztási folyamatot, amíg az osztási egészrész nem lesz kisebb, mint b; ez előbb-utóbb bekövetkezik, és fennáll, hogy $n > q_0 > q_1 > \cdots \geq 0$.

Az első egyenletből kiindulva, megfelelő helyettesítéseket végezve kapjuk:

$$\begin{aligned} nr &= b \cdot q_0 + a_0 \\ &= b \cdot (b \cdot q_1 + a_1) + a_0 = b^2 \cdot q_1 + a_1 \cdot b + a_0 \\ &= b^2 \cdot (b \cdot q_2 + a_2) + a_1 \cdot b + a_0 = b^3 \cdot q_2 + a_2 \cdot b^2 + a_1 \cdot b + a_0 \\ &\ldots \\ &= a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \cdots + a_1 \cdot b + a_0. \end{aligned}$$

Az osztási folyamat során előállt maradékok alkotják a b számrendszerbeli számjegyeket: $a_k, a_{k-1}, \ldots a_1, a_0$.

Példa: Alakítsuk át 7432-t 16-os számrendszerbe:

$$\begin{array}{lrcl} \text{7432-t osztjuk 16-tal:} & 7432 & = & 464 \cdot 16 + \mathbf{8} \\ \text{464-t osztjuk 16-tal:} & 464 & = & 29 \cdot 16 + \mathbf{0} \\ \text{29-t osztjuk 16-tal:} & 29 & = & \mathbf{1} \cdot 16 + \mathbf{13} \end{array}$$

Az utolsó osztásnál kisebb osztási egészrészt kaptunk, mint 16, nem folytatjuk tovább az osztási folyamatot. Az $[1, 13, 0, 8]$ osztási maradékok képezik a 16-os számrendszerbeli alakot.

5.1. algoritmus. Írjunk egy Python függvényt, amely meghatározza a `nr` szám `b` számrendszerbeli alakját.

A függvény kimenete egy egész számokból álló lista lesz, amelynek elemei a `b` számrendszerbeli számjegyek lesznek.

```
def nrToBaseB(nr, b):
    L = []
    while nr >= b:
        L = [nr % b] + L
        nr = nr // b
    L = [nr] + L
    return L

>>> nrToBaseB(7432, 16)
    [1, 13, 0, 8]
```

Az `L = [nr % b] + L` sor azt jelenti, hogy a kapott osztási maradékokat, mindig a lista elejére fűzzük, ezért a legjobboldali pozícióra az utolsónak kiszámolt maradék kerül. Ha azonban az `L = [nr % b] + L` sor helyett az `L = L + [nr % b]` sort írjuk, akkor fordított sorrendet kapunk. Amikor 256-os számrendszerbe alakítunk, azaz amikor egy egész szám értékeit bájtok sorozataként adjuk meg, akkor a kétféle sorrend külön nevet kap. A lista elejére való fűzés a *big order* nevet kapja, a másik a *little order*-t.

Az eredményt `str` típusként is meg lehet határozni, ekkor elválasztó szerepet játszó szimbólumokat kell a számjegyek közé beszúrni. A következő függvény szóközöket használ elválasztó szimbólumként:

```
def nrToBaseB_(nr, b):
    L = ''
    while nr >= b:
        L = ' ' + str(nr % b) + L
        nr = nr // b
    L = str(nr) + L
    return L

>>> nrToBaseB_(53428, 2)
    '1 1 0 1 0 0 0 0 1 0 1 1 0 1 0 0'
```

A következő algoritmus meghatározza egy szám b számrendszerbeli alakja alapján a 10-es számrendszerbeli alakot. Figyeljük meg a hasonlóságot a 4.25. algoritmussal.

- az $a_k, a_{k-1}, \ldots a_1, a_0$ elemekből hatványösszeget számolunk,
- kiindulva a $nr = 0$ kezdőértékből, minden $a_i \in \{a_k, a_{k-1}, \ldots a_1, a_0\}$ elemre elvégezzük a $nr = nr \cdot b + a_i$ műveletet:

$$\begin{aligned}
nr &= 0 \\
nr &= nr \cdot b + a_k = a_k \\
nr &= nr \cdot b + a_{k-1} = a_k \cdot b + a_{k-1} \\
nr &= nr \cdot b + a_{k-2} = (a_k \cdot b + a_{k-1}) \cdot b + a_{k-2} = \\
&= a_k \cdot b^2 + a_{k-1} \cdot b + a_{k-2} \\
&\ldots \\
nr &= nr \cdot b + a_0 = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \ldots a_1 \cdot b + a_0
\end{aligned}$$

Példa: Alakítsuk át $1D08$-t 10-es számrendszerbe, azaz az $L = [1, 13, 0, 8]$ lista elemeiből számoljuk ki a megfelelő hatványösszeget:

$$\begin{aligned} nr &= 0 \\ nr &= 0 \cdot 16 + 1 \\ nr &= 1 \cdot 16 + 13 = 29 \\ nr &= (1 \cdot 16 + 13) \cdot 16 + 0 = 29 \cdot 16 = 464 \\ nr &= (1 \cdot 16^2 + 13 \cdot 16 + 0) \cdot 16 + 8 \\ &= 1 \cdot 16^3 + 13 \cdot 16^2 + 0 \cdot 16 + 8 = 464 \cdot 16 + 8 = 7432 \end{aligned}$$

5.2. algoritmus. Írjunk egy Python függvényt, amely egy b számrendszerbeli számot átalakít tízes számrendszerbe.

A függvény bemenete egy lista lesz, amely tartalmazza a b számrendszerbeli számjegyeket, kimenete pedig egy egész szám.

```
def nrFromBaseB(L, b):
    nr = 0
    for elem in L:
        nr = nr * b + elem
    return nr

>>> nrFromBaseB([1, 13, 0, 8], 16)
    7432
```

A következő két algoritmus azt az esetet kezeli, amikor 16-os számrendszerbe, illetve amikor 16-os számrendszerből kell alakítani.

5.3. algoritmus. Írjunk egy Python függvényt, amely meghatározza a `nr` szám 16-os számrendszerbeli alakját. A függvény kimenete egy `str` típusú érték legyen, ahol a 16-os számrendszerbeli számjegyeket a $0, ..., 9, a, b, c, d, e, f$ szimbólumokkal adjuk meg.

```
def nrToBase16(nr):
   L, S = '', 'abcdef'
    while nr >= 16:
        temp = nr % 16
        if temp > 9: temp = S[temp - 10]
        L = str(temp) + L
        nr = nr // 16
    if nr > 9: nr = S[nr - 10]
    L = str(nr) + L
    return L
```

```
>>> nrToBase16(2035)
    '7f3'
```

Az algoritmus, ha a `temp`-be 9-nél nagyobb maradékot számol ki, akkor meghatározza az `S temp-10`-es sorszámú betűjét. Például 12-es maradék esetén a `temp-10` egyenlő lesz 2-vel, ezért a `temp` felülírt értéke a `c` betű lesz.

5.4. algoritmus. Írjunk egy Python függvényt, amely egy 16-os számrendszerben megadott számot átalakít tízes számrendszerbe. A függvény bemenete egy `str` típusú érték legyen, amely tartalmazza a 16-os számrendszerbeli számjegyeket, ahol a 16-os számrendszerbeli számjegyeket az $0, ..., 9, a, b, c, d, e, f$ szimbólumokkal adjuk meg.

```
def nrFromBase16(L):
   nr, S = 0, 'abcdef'
   for elem in L:
      try:
        elem = S.index(elem) + 10
      except:
        elem = int(elem)
      nr = nr * 16 + elem
   return nr

>>> nrFromBase16('7f3')
    2035
```

Az `if`-fel teszteljük, hogy az `elem` `S`-beli betűt jelöl vagy sem. Ha igen, akkor az `elem` egyenlő lesz a betű indexének az értékével plusz 10. Például ha az `elem f` betűt jelöl, akkor az `elem` egyenlő lesz 5 + 10-zel. Ehhez az `index` metódust használtuk, amely megadja azt a legelső sorszámot, amelyen egy adott karakter előfordul egy karakterláncban.

Példa: A következő kódsorok az `index` használatát mutatják:

```
>>> S = 'abcdef'
>>> S.index('f')
    5
>>> S.index('z')
    ...ValueError: substring not found
```

5.2. Vegyes alapú számrendszerek

A korábban tárgyalt pozicióalapú számrendszerktől eltérő módon is megoldható a különböző mennyiségek ábrázolása. A vegyes alapú számrendszerek esetében a számrendszer alapszáma változó. Például az időmérést is vegyes alapú számrendszer segítségével végezzük. Jelöljék a $\mathbf{27}_{52}\mathbf{3}_{7}\mathbf{6}_{24}\mathbf{20}_{60}\mathbf{15}_{60}$ ábrázolás esetén az index-számok az alapszámot, a vastagított számok pedig, hogy az alapszámból mennyit használunk fel. Ekkor ez az ábrázolás egy adott év 27. hetét, 3. napját, 6. óráját, 20. percét, 15. másodpercét jelenti.

A következőkben két vegyes alapú számrendszert mutatunk be, a faktoriális számrendszert, illetve a Fibonacci-számrendszert [28].

A **faktoriális számrendszert** Cantor-ábrázolásnak is hívják. Alapját az képezi, hogy bármely pozitív egész szám egyértelműen felírható a faktoriális függvény értékeinek segítségével:

$$nr = a_k \cdot k! + a_{k-1} \cdot (k-1)! + \cdots + a_2 \cdot 2! + a_1 \cdot 1!,$$

ahol $a_i \in [0, 1, \ldots, i]$, bármely $i \in [1, 2, \ldots, k]$ és az $a_k, a_{k-1}, \ldots, a_2, a_1$ értékek képezik a faktoriális számrendszerbeli számjegyeket.

A 4.1. tétel alapján kijelenthető, hogy nr a következő lépések szerint egyértelműen felírható faktoriális számrendszerbe:

1. minden nr természetes szám esetén létezik egy n egész szám, amelyre fennáll: $n! \leq nr < (n+1)!$,
2. nr-t elosztjuk $n!$-al, ekkor fennáll: $nr = a_n \cdot n! + r_n$, ahol $0 \leq a_n \leq n$ és $0 \leq r_n < n!$,
3. r_n-t elosztjuk $(n-1)!$-al, ekkor fennáll: $r_n = a_{n-1} \cdot (n-1)! + r_{n-1}$, ahol $0 \leq a_{n-1} \leq (n-1)$ és $0 \leq r_{n-1} < (n-1)!$,
4. a folyamatot addig ismételjük, amíg 1!-al is elvégezzük az osztást,
5. az osztási folyamat során meghatározásra kerülő $a_n, a_{n-1}, \ldots a_1$ értékek fogják a számrendszerbeli számjegyeket jelölni.

Példa: Határozzuk meg $(8172)_{10}$ faktoriális-számrendszerbeli alakját.

$$\begin{aligned} 8172 &= \mathbf{1} \cdot 7! + \mathbf{4} \cdot 6! + \mathbf{2} \cdot 5! + \mathbf{0} \cdot 4! + \mathbf{2} \cdot 3! + \mathbf{0} \cdot 2! + \mathbf{0} \cdot 1! \\ 8172 &= 1 \cdot 5040 + 4 \cdot 720 + 2 \cdot 120 + 2 \cdot 6 \\ 8172 &= [\mathbf{1, 4, 2, 0, 2, 0, 0}]_{faktBase} \end{aligned}$$

Egy nr szám faktoriális számrendszerbeli alakját azonban a következő hatékonyabb algoritmussal fogjuk megadni:

1. nr-t elosztjuk **2**-vel, ekkor fennáll: $nr = 2 \cdot q_1 + a_1$,
2. q_1-et elosztjuk **3**-mal, ekkor fennáll: $q_1 = 3 \cdot q_2 + a_2$,
3. folytatjuk az osztási folyamatot a **4, 5, ...** értékekkel, amíg az osztási egészrész nem lesz 0,
4. az osztási folyamat során kiszámolt maradékok lesznek nr faktoriális számrendszerbeli számjegyei.

Az algoritmus helyessége belátható, a következő egyenlőség alapján:

$$\begin{aligned} nr &= 2 \cdot (3 \cdot (4 \cdot (5 \cdot (\ldots) + a_4) + a_3) + a_2) + a_1 \\ &= \ldots \cdot 4! \cdot \boldsymbol{a_4} + 3! \cdot \boldsymbol{a_3} + 2! \cdot \boldsymbol{a_2} + \boldsymbol{a_1} \\ nr &= [\boldsymbol{a_n}, \boldsymbol{a_{n-1}}, \ldots, \boldsymbol{a_2}, \boldsymbol{a_1}]_{faktBase} \end{aligned}$$

Példa: 8172 faktoriális számrendszerbe való átalakításához felírható:

$$\begin{aligned} 8172 &= 2 \cdot (3 \cdot (4 \cdot (5 \cdot (6 \cdot (7 \cdot (8 \cdot 0 + \mathbf{1}) + \mathbf{4}) + \mathbf{2}) + \mathbf{0}) + \mathbf{2}) + \mathbf{0}) + \mathbf{0} \\ 8172 &= \mathbf{1} \cdot 7! + \mathbf{4} \cdot 6! + \mathbf{2} \cdot 5! + \mathbf{0} \cdot 4! + \mathbf{2} \cdot 3! + \mathbf{0} \cdot 2! + \mathbf{0} \cdot 1! \\ 8172 &= [1, 4, 2, 0, 2, 0, 0]_{faktBase} \end{aligned}$$

5.5. algoritmus. Írjunk egy Python Programot, amely a fenti elgondolás alapján meghatározza egy szám faktoriális számrendszerbeli számjegyeit.

```
def digitToFactB(nr):
    L = []
    i = 2
    while nr != 0:
        q = nr // i
        a = nr - q * i
        nr = q
        L = [a] + L
        i = i + 1
    return L

>>> digitToFactB(45389)
    [1, 1, 0, 0, 1, 0, 2, 1]
```

A **Fibonacci-számrendszert** Zeckendor-ábrázolásnak is hívják, alapját az képezi, hogy bármely nr természetes szám egyértelműen felírható Fibonacci-számok összegeként:

$$nr = a_n \cdot F_n + a_{n-1} \cdot F_{n-1} + \cdots + a_2 \cdot F_2 + a_1 \cdot F_1,$$

ahol $a_i \in [0, 1]$, bármely $i \in [1, 2, \ldots, n]$, és $F_n, F_{n-1}, \ldots, F_2, F_1$ a 0 és 1 kezdőértékeket elhagyva kapott Fibonacci-számsorozat (87. oldal). Fibonacci-számrendszerben egy számot tehát az $F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \ldots$-al kezdődő Fibonacci-számsorozat elemeivel írhatunk fel. A számrendszerbeli számjegyek nulla vagy egyes értékeket vehetnek fel, de a kapott ábrázolást nem szabad összetéveszteni egy szám kettes számrendszerbeli alakjával, ahol a számjegyek szintén nullás vagy egyes értékek lehetnek.

Példa: Határozzuk meg 100 Fibonacci-számrendszerbeli számjegyeit.

1. meghatározzuk a 100-nál kisebb vagy vele egyenlő Fibonacci-számok listáját: **89**, 55, 34, 21, 13, **8**, 5, **3**, 2, 1,
2. kiválasztjuk a kapott Fibonacci-számok listájából az első számot, a 89-est, az eredménylista első eleme tehát `'1'` lesz, majd meghatározzuk a $100 - 89 = 11$ különbséget,
3. mivel a következő 4 szám, azaz az 55, 34, 21, 13-as értékek mindegyike nagyobb, mint 11, az eredménylistát kiegészítjük 4 nullással: `'10000'`,
4. a fennmaradó elemek közül kiválasztjuk az első számot, a 8-ast, az eredménylistát kiegészítjük 1 egyessel: `'100001'`, majd meghatározzuk a $11 - 8 = 3$ különbséget,
5. a fennmaradó elemek közül csak az 5-ös nagyobb, mint a 3-as, az eredménylistát kiegészítjük 1 nullással: `'1000010'`,
6. a fennmaradó elemek közül kiválasztjuk az első számot, a 3-ast, az eredménylistát kiegészítjük 1 egyessel: `'10000101'`, majd meghatározzuk a $3 - 3 = 0$ különbséget,
7. a fennmaradó elemek közül a 2-es és 1-es számok mindegyike nagyobb mint 0, ezért kiegészítjük 2 nullással az eredménylistát: `'1000010100'`
8. mivel az utolsó különbség értéke 0, véget ér a számítási folyamat.

Tehát felírható:

$$\begin{aligned} 100 &= \mathbf{1} \cdot 89 + 0 \cdot 55 + 0 \cdot 34 + 0 \cdot 21 + 0 \cdot 13 + \mathbf{1} \cdot 8 + 0 \cdot 5 + \\ &\quad \mathbf{1} \cdot 3 + 0 \cdot 2 + 0 \cdot 1 \\ 100 &= [1000010100]_{fibonacciBase} \end{aligned}$$

5.6. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy szám Fibonacci-számrendszerbeli alakját. Az eredményt egy `str` típusú változóba generáljuk ki.

Két függvényt írunk, ahol a `myFibListNr` az `L` változóba meghatározza a bemeneti paraméterénél kisebb vagy vele egyenlő Fibonacci-számok csökkenő listáját, ahol az utolsó két Fibonacci-szám a 2 és az 1 lesz. A `fibBaseStr` pedig a példánál bemutatott gondolatmenetet követi.

```
def myFibListNr(nr):
    if nr <= 0: return []
    if nr <= 1: return [1]
    if nr <= 2: return [2, 1]
    L = [2, 1]
    while True:
        temp = L[0] + L[1]
        if temp > nr: return L
        L = [temp] + L
    return L

def fibBaseStr(nr):
    L = myFibListNr(nr)
    lenL = len(L)
    bL, i = '', 0
    while i < lenL:
        nr -= L[i]
        bL += str(1)
        i += 1
        while i < lenL and L[i] > nr:
            bL += str(0)
            i += 1
    return bL

>>> fibBaseStr(100)
    '1000010100'
```

5.3. Bitműveletek

A programsorok aritmetikai műveleteit gyakran helyettesíteni szokták bitműveletekkel [6]. Ekkor a számítógép a számításokat közvetlenül a biteken, bájtokon hajtja végre, anélkül, hogy a programozó előzetesen elvégezné a 2-es számrendszerbe való átalakítást. Ilyenkor hatékonyabb lesz a kód, éppen ezért a legtöbb programozási nyelv rendelkezik biteket közvetlenül manipuláló operátorokkal. Python esetében ezek a következők, ahol a táblázat második oszlopában feltüntetjük az angol megnevezéseket [13, 38]:

```
nr << k     left shift     nr bitjei k pozícióval balra tolódnak
nr >> k     right shift    nr bitjei k pozícióval jobbra tolódnak
nr1 | nr2       OR         bitenkénti vagy
nr1 & nr2       AND        bitenkénti és
nr1 ^ nr2       XOR        bitenkénti kizáró vagy
```

A következő táblázatban először 4 pozícióval balra, majd 3 pozícióval jobbra látjuk a `nr` számon alkalmazott eltolások eredményét. A második, harmadik és negyedik oszlopban a bináris alakok vannak:

```
nr       2-es alak      nr << 4          nr >> 3
17         10001       10001 0000          10
67        1000011      10001 0000         1000
```

Kipróbálhatjuk Pythonban is a műveletsorokat:

```
>>> nr = 17
>>> r1, r2 = nr << 4, nr >> 3
>>> format(nr, 'b'), format(r1, 'b'), format(r2, 'b')
    ('10001', '10001 0000', '10')

>>> nr = 67
>>> r1, r2 = nr << 4, nr >> 3
>>> format(nr, 'b'), format(r, 'b')
    ('1000011', '100011 0000', '1000')
```

A bitenkénti `vagy`, `és`, illetve `kizáró vagy` műveleteket az `x` és `y` bitértékek esetében a következő táblázat mutatja:

```
    x   y   x | y   x & y   x ^ y
    0   0     0       0       0
    0   1     1       0       1
    1   0     1       0       1
    1   1     1       1       0
```

5.7. algoritmus. Írjunk egy `bitmuveletek` függvényt, amely két egész szám esetén kiírja a számok bináris alakját, a számokat, majd a bitenkénti `OR, AND, XOR` utáni bitkonfigurációkat és a tízes számrendszerbeli eredményeket.

```
def bitmuveletek(nr1, nr2):
   res1 = nr1 | nr2
   res2 = nr1 & nr2
   res3 = nr1 ^ nr2
```

```
    print("%17s%6d" % (format(nr1, 'b'), nr1))
    print("%17s%6d" % (format(nr2, 'b'), nr2))
    print(" OR: %12s%6d" % (format(res1, 'b'), res1))
    print("AND: %12s%6d" % (format(res2, 'b'), res2))
    print("XOR: %12s%6d" % (format(res3, 'b'), res3))

>>> bitmuveletek(1025, 1026)
          10000000001  1025
          10000000010  1026
     OR:  10000000011  1027
     AND: 10000000000  1024
     XOR:          11     3
```

2, illetve 2 hatványaival végzett műveletek során lehet és érdemes bitműveleteket használni, mert hatékonyabb lesz a számítási folyamat. Egy egész szám 2-vel való szorzása helyettesíthető 1 pozícióval történő balra shifttel. Egy egész szám 2^k-nal való szorzása helyettesíthető k pozícióval történő balra shifttel:

```
>>> 17 * 2, 17 << 1
    (34, 34)
>>> format(17, 'b'), format(34, 'b')
    ('10001','10001**0**')

>>> 17 * 16, 17 << 4
    (272, 272)
>>> format(17, 'b'), format(272, 'b')
    ('10001','10001**0000**')
```

Egy egész szám 2-vel való osztása helyettesíthető 1 pozícióval történő jobbra shifttel. Egy egész szám 2^k-al való osztása helyettesíthető k pozícióval történő jobbra shifttel:

```
>>> 59 // 2, 59 >> 1
    (29, 29)
>>> format(59, 'b'), format(29, 'b')
    ('11101**1**','11101')

>>> 59 // 16, 59 >> 4
    (3, 3)
>>> format(59, 'b'), format(3, 'b')
    ('11**1011**', '11')
```

Egy egész számnak, ha meg akarjuk határozni 2-vel való osztási maradékát, akkor a `%` operátor helyett hatékonyabb, ha bitenként `és`-t végzünk a szám és 1 között. Ha 2^k-val akarjuk meghatározni az osztási maradékot, akkor a `%` operátor helyett hatékonyabb, ha bitenként `és`-t végzünk a szám és $2^k - 1$ között.

5.8. algoritmus. Írjunk egy Python függvényt, amely megvizsgálja egy számról, hogy páros vagy páratlan.

```
def parosF(szam):
    if szam & 1:
        print('paratlan szam')
    else: print('paros szam')
```

5.9. algoritmus. Írjunk egy Python függvényt, amely meghatározza, hogy a paramétereként megadott természetes szám hány 1-es bitet tartalmaz, más megfogalmazásban határozzuk meg egy adott szám **Hamming**[1]-súlyát.

```
def HammingW1(nr):
    db = 0
    #itNr = 0
    #print(db, format(nr, 'b'))
    while nr:
        db += (nr & 1)
        nr >>= 1
        #print(db, format(nr, 'b'))
        #itNr += 1
    #print("iterációszám: ", itNr)
    return db
```

```
>>> HammingW1(67)
    0 1000011
    1 100001
    2 10000
    2 1000
    2 100
    2 10
    2 1
    3 0
iterációszám:  7
3
```

Figyeljük meg, hogy az 1-es bitek számának a meghatározásához annyi iterációra volt szükség, ahány biten ábrázolható a szám. A `while`-ban először bitenkénti `és` műveletet hajtunk végre a `nr` és 1 között, majd következik egy `jobbra shift`. Ennek eredményeképpen a `db` értéke annyi 1-gyel fog növekedni, ahány 1-es van a szám bináris alakjában.

A Hamming-súly meghatározható hatékonyabb algoritmussal is. A következő `HammingW2` csak annyi iterációt végez, ahány 1-es bit van a szám bináris alakjában. Az algoritmus megértése végett érdemes nyomon követni

1 Richard Wesley Hamming amerikai matematikus (1915–1998)

a `nr, nr-1`, illetve `nr & (nr-1)` értékeinek változását, ezért ki is írathatjuk a részértékeket.

```
def HammingW2(nr):
    db = 0
    #itNr = 0
    #print('%10s%10s%14s' % ('nr', 'nr-1', 'nr & (nr-1)'))
    while nr:
        #print('%10s' % format(nr, 'b'), end = '')
        #print('%10s' % format(nr-1, 'b'), end = '')
        #print('%14s' % format(nr & (nr-1), 'b'))
        nr &= nr - 1
        db += 1
        #itNr += 1
    #print("iterációszám: ", itNr)
    return db

>>> HammingW2(149)
          nr         nr-1       nr & (nr-1)
    10010101      10010100      10010100
    10010100      10010011      10010000
    10010000      10001111      10000000
    10000000      1111111             0
    iteráció szám:  4
    4
```

Figyeljük meg, hogy a `nr &= nr - 1` művelet ismételt végrehajtása után az eredmény bitkonfigurációja annyiban fog eltérni a `nr` bitkonfigurációjától, hogy a legjobboldalibb egyes helyett mindig nullás fog megjelenni. Éppen ezért az egyesek számát az fogja meghatározni, hogy a `nr &= nr - 1` művelet hányszor kerül végrehajtásra. Hívjuk meg a függvényt különböző bemenetekre!

Pythonban a `bit_count` metódus meghatározza a bináris alakban található egyesek számát, míg a `bit_length()` megadja egy egész szám bináris alakjához szükséges bitek számát:

```
>>> nr = 129
>>> bin(nr)
    '0b10000001'
>>> nr.bit_count()
    2
>>> nr.bit_length()
    8
```

5.10. algoritmus. Írjunk egy Python függvényt, amely a paraméterként megadott `nr` számot átalakítja 256-os számrendszerbe, illetve egy másikat, amely egy 256-os számrendszerben megadott szekvenciát átalakít tízes számrendszerbe.

A feladatot korábban már megoldottuk általánosan, lásd az 5.1., 5.2. algoritmusokat, de a hatékonyabb számítási folyamat megvalósítása érdekében a következő két függvény bitműveleteket fog használni:

```
def nrToBytes(nr):
    L = []
    while nr >= 256:
       L = [nr & 255] + L
       nr = nr >> 8
    L = [nr] + L
    return L

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

>>> nrToBytes(72627813131)
    [16, 232, 244, 123, 11]
>>> nrFromBytes([16, 232, 244, 123, 11])
    72627813131
```

5.11. algoritmus. Írjunk egy Python programot, amely véletlenszerűen generál egy k bájtos bájtszekvenciát, meghatározza a bájtszekvencia bájtjainak hexadecimális alakját, és elvégzi a fordított műveletsort is, azaz a hexa karakterláncból visszaállítja a bájtszekvenciát.

Pythonban véletlenszerű értékek meghatározására a `random` csomagban található függvényeket használhatjuk. A feladat megoldása során a tízes és 16-os számrendszerben megadott értékek közötti átalakításokhoz a beépített `format`, illetve `int` függvényeket fogjuk használni.

```
from random import randint
def bytesFromHex(hStr):
    hL = hStr.split(' ')
    L = []
    for elem in hL:
        L += [int(elem, 16)]
```

```
    return L

def bytesToHex(L):
    hStr = ''
    for elem in L:
      hStr += format(elem, 'x') + ' '
    return hStr[:-1]

def mainConv(k):
    bajtSz = []
    for i in range(k):
        bajtSz += [randint(0, 255)]
    print('generalt bajtszekvencia: ', bajtSz)

    hStr = bytesToHex(bajtSz)
    print('a bajtszekvencia hexaban: ', hStr)
    bajtSz_ = bytesFromHex(hStr)
    print('az eredeti bajtszekvencia: ', bajtSz_)

>>> mainConv(10)
    ...
```

Korábban nem tértünk ki a b_1 és b_2 számrendszerek közötti direkt átalakításokra, de mivel a p és p^k számrendszerek közötti átalakítások, ahol k pozitív egész szám, közvetlenül is megoldhatók, a következő részben ezzel fogunk foglalkozni [28].

p számrendszerből p^k számrendszerbe való átalakítás során hátulról k-as csoportokat kell formálunk, és a csoportokon belül p hatványértékeiből hatványösszeget kell számolnunk:

$p^k = 2^3$-os esetén hátulról **3-as** csoportokat formálunk:

$$\begin{aligned}[1,0\ \ 1,1,1,\ \ 0,1,1]_2 \to [2,7,3]_8,\ \text{mert}\\ 1,0 \to 1\cdot 2^1 + 0\cdot 2^0 = 2\\ 1,1,1 \to 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 7\\ 0,1,1 \to 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 3\end{aligned}$$

$p^k = 2^4$-es esetén **4-es** csoportokat formálunk:

$$\begin{aligned}[1,1,\ \ 1,0,0,1,\ \ 1,0,1,1]_2 \to [3,9,B]_{16},\ \text{mert}\\ 1,1 \to 1\cdot 2^1 + 1\cdot 2^0 = 3\\ 1,0,0,1 \to 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 9\\ 1,0,1,1 \to 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 11 = B\end{aligned}$$

$p^k = 2^8$-os esetén **8-as** csoportokat formálunk:

$$[1,1,1,0,0,1,\ \ 0,1,1,0,1,0,1,1] \to [57,107],\ \text{mert}$$
$$1,1,1,0,0,1 \to 1\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 57$$
$$0,1,1,0,1,0,1,1 \to 0\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + $$
$$+1\cdot 2^1 + 1\cdot 2^0 = 107$$

5.12. algoritmus. Írjunk egy Python függvényt, amely egy 2-es számrendszerbeli értéket átalakít 2^k számrendszerbe. A függvénynek két bemeneti paramétere legyen. Az első paraméter legyen egy lista, amely a 2-es számrendszerbeli jegyeket tartalmazza, a második egy egész szám, amely a k értékét jelöli. A kimenet is, amely a 2^k-os számrendszerbeli jegyeket tartalmazza, lista típusú legyen.

```
def conv2To2K(L, k):
    nr, nL = 0, []
    m = len(L) % k
    for elem in L[:m]:
        nr = (nr << 1) + elem
    if nr != 0: nL = [nr]
    nr, i = 0, 0
    for elem in L[m:]:
        nr = (nr << 1) + elem
        i += 1
        if i == k:
            nL = nL + [nr]
            nr, i = 0, 0
    return nL
```

2-es számrendszerből 256-os számrendszerbe, illetve 8-as számrendszerbe alakítanak a következő függvényhívások:

```
>>> conv2To2K([1,1,1,0,0,1,0,1,1,0,1,0,1,1], 8)
    [57, 107]
>>> conv2To2K([1, 1, 0, 1, 0, 1], 3)
    [6, 5]
```

5.13. algoritmus. Írjunk egy Python függvényt, amely egy 2^k-as számrendszerbeli értéket átalakít 2-es számrendszerbe. A függvénynek két bemeneti paramétere legyen, ahol az első egy lista, amely a 2^k-as számrendszerbeli jegyeket tartalmazza, a második egy egész szám, amely a k értékét jelöli. A kimenet is, amely a 2-es számrendszerbeli jegyeket tartalmazza, lista típusú legyen.

```
def conv2From2K(L, k):
    nL = []
    for elem in L:
        sL = []
        for i in range(0, k):
            sL = [elem & 1] + sL
            elem = elem >> 1
        nL += sL
    return nL
```

$32 = 2^5$-es számrendszerből, illetve $16 = 2^4$-es számrendszerből alakítanak 2-es számrendszerbe a következők függvényhívások:

```
>>> conv2From2K([30, 11, 27, 13], 5)
    [1,1,1,1,0,0,1,0,1,1,1,1,0,1,1,0,1,1,0,1]
>>> conv2From2K([2, 15, 7, 13], 4)
    [0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1]
```

A két függvényt együtt is tudjuk alkalmazni:

```
>>> conv2To2K(conv2From2K([7, 6, 5, 4, 6], 3), 3)
    [7, 6, 5, 4, 6]
```

A kriptográfiában különösen fontosak azok az algoritmusok, amelyek megoldják valamely adathalmaz rejtjelezését, titkosítását. Ahhoz, hogy ez megvalósítható legyen, a rendszerek mindig egy kulcs segítségével végzik a rejtjelezést, ami egy szigorúan titkos információ. A visszafejtéshez is szükséges a kulcs, a titkos információ. Az ilyen típusú algoritmusok a **szimmetrikus kriptográfia** részét képezik [2, 22, 36]. Megjegyezzük, hogy az alkalmazott kulcs megosztását nem oldják meg az ilyen típusú rendszerek, ez az aszimmetrikus kriptográfia feladata, amelyre a 6.8., 6.10. és 6.13. fejezetekben még visszatérünk. A következőkben a szimmetrikus kriptográfia területére való bevezetésként egy olyan egyszerű algoritmust mutatunk be, amely egy kulcsot használva titkosít és visszafejt egy tetszőleges bájtszekvenciát. A bemutatott rejtjelezési eljárást azonban a gyakorlatban nem lehet használni, mert olyan formában, ahogy alkalmazni fogjuk, nem biztonságos. Ahhoz, hogy megfelelő biztonságú legyen, a következő tényezőket kellene figyelembe venni:

- a kulcs értékét véletlenszerűen kell kiválasztani,
- a kulcs bájthossza ugyanakkora kell legyen, mint a rejtjelezendő bájtszekvencia hossza,

– a kulcsot nem szabad többször felhasználni, minden egyes titkosításhoz más kulcsot kell használni,
– meg kell oldani a felek közti biztonságos kulcsmegosztást,
– meg kell oldani a felek kölcsönös hitelesítését.

5.14. algoritmus. Írjunk egy Python programot, amely titkosít és visszafejt egy tetszőleges karakterláncot egy megadott kulcsérték alapján. A titkosítást az `XOR` művelettel végezzük, amelyet alkalmazzunk a karakterlánc karakterei és kulcs karakterei között. A kulcs karaktereit egymás után, körkörösen vegyük, ami azt jelenti, hogy ha elfogynak a kulcs karakterei, akkor a kulcs első elemével folytassuk a műveletvégzést, majd a következővel és így tovább.

```
def crypt(myStr, key):
    crStr = ''
    i = 0
    n = len(key)
    for elem in myStr:
        crStr += cryptB(elem, key[i])
        if i == n-1: i = -1
        i += 1
    return crStr

def cryptB(kar, crKar):
    return chr(ord(kar) ^ ord(crKar))

>>> crText = crypt('Jó reggelt elso/másod év!!', 'passwd')
>>> crText
    ':\x92S\x01...'
```

A titkosítást a `crypt` függvény végzi, ahol a függvény első paramétere az a karakterlánc lesz, amelyet rejtjelezni szeretnénk, a második pedig a kulcs. A `crypt` visszatérési értéke a rejtjelezett karakterlánc lesz. A `cryptB` függvény egyetlenegy karakter rejtjelezését végzi. A `chr, ord` függvények alkalmazása azért szükséges, mert az `^` operátor `int` típusú értékekre alkalmazható.

A visszafejtés annyiban különbözik a rejtjelezéstől, hogy a `crypt` függvényt más paraméterekkel fogjuk meghívni. Az első paraméter most a rejtjelezett szöveg lesz, a második pedig ugyanaz a kulcsérték kell legyen, mint amit a rejtjelezésnél használtunk.

```
>>> crypt(crText, 'passwd')
    'Jó reggelt elso/másod év!!'
```

A titkosítás, visszafejtés helyessége az XOR művelet algebrai tulajdonságából következik. A matematikában az XOR jelölésére az $\oplus$ szimbólumot használják. A művelet algebrai tulajdonságai a következők:

$$x \oplus y = y \oplus x, x \oplus (y \oplus z) = (x \oplus y) \oplus z, x \oplus 0 = x, x \oplus x = 0$$

A következő `mainCR` függvény a billentyűzetről kéri be annak a karakterláncnak az értékét, amelyet rejtjelezni szeretnénk, illetve a kulcs értékét is a billentyűzetről kell beolvasni. A függvény elvégzi a beolvasott karakterlánc titkosítását, majd a rejtjelezett szöveg visszafejtését is. A rejtjelezett szöveg karaktereit hexadecimális számrendszerben írja ki a képernyőre, mert mint korábban észrevételezhettük, ezek értelmetlen karakterek lesznek.

```
def mainCR():
    szoveg = input("kerek egy karakterlancot: ")
    kulcs = input("kerek egy kulcsot: ")
    crText = crypt(szoveg, kulcs)
    print("a titkositott szoveg: ")
    for c in crText:
        print(hex(ord(c)), end = " ")
    print()
    vSzoveg = crypt(crText, kulcs)
    print("a visszafejtett szoveg: ", vSzoveg)
```

5.4. Kódolási technikák

A számítástechnikában többféle kódolási technika létezik, amelyeknek az a célja, hogy a feldolgozandó adatot, amely lehet szöveg, kép, hang stb., átalakítsa egy jól meghatározott megfeleltetési szabály szerint természetes számok sorozatává. A természetes számok, mint korábban láttuk könnyedén átalakíthatóak bináris szekevenciákká, azaz bit-, illetve bájtsorozatokká, amelyeket a számítógép már fel tud dolgozni [34]. Fontosnak tartjuk kiemelni, hogy a kódolási technikák (encoding technics) mást jelentenek, más a céljuk, mint a rejtjelezési technikáknak (encryption technics).

Többféle kódolási technika létezik, például:

- a gépi kód bináris formában tárolja az adatokat és utasításokat,
- az ASCII-kód és az Unicode szabvány természetes számokat rendel hozzá a különböző írásjelekhez, karakterekhez,
- az UTF-8 kódolás az Unicode karakterek tárolási módját határozza meg,

– a HTML, a LaTex leíró nyelvek olyan kódokat tartalmaznak, amelyek leírják, hogy az állomány tartalmát hogyan kell megjeleníteni, hogyan kell feldolgozni,
– a Huffman-kód adattömörítést megvalósító kód,
– a Hamming-kód hibajelzésre és hibajavításra alkalmas kód,
– a Base64 kód 64 szimbólumot alkalmaz a bájtszekvenciák ábrázolásához.

5.5. Az ASCII kódolás

Az ASCII (American Standard Code for Information Interchange) kódot először 1963-ban írták le. A standardot a korábbi távközlésben használt 7 bites kódolás alapján tervezték. Kezdetben 128 szimbólum kódját adták meg, később jelent meg az extended ASCII-kód, amely már 256 szimbólumot kódolt a természetes számok első 256 értékével. 33 nem nyomtatható, tulajdonképpen kontrollkódot és 95 nyomtatható szimbólumot kezel. A kontrollkódok között vannak olyanok, amelyek a mai számítástechnikában nem használatosak már, de egy jó pár közülük mai napig is fontos szerepet tölt be. A nyomtatható karakterek között megtaláljuk a számjegyeket, az angol ábécé betűit, írásjeleket stb. A szövegszerkesztők ASCII-kódoláson alapulnak, az újabb verziók azonban más kódolást is ismernek [34].

5.15. algoritmus. Írjunk egy Python függvényt, amely kiírja a képernyőre az ASCII-szimbólumokat és a hozzájuk tartozó kódértékeket.

```
def myASCIItable0():
    for i in range(128):
        print(chr(i), i)
```

Ha paraméterezzük a függvényt, akkor tetszőleges intervallumból tudjuk kiíratni az ASCII-kódokat:

```
def myASCIItable1(n = 0, m = 256):
    for i in range(n, m):
        print(chr(i), i)
```

Az angol ábécé nagybetűi és a megfelelő ASCII-kódok a következő függvényhívással írathatók ki:

```
>>> myASCIItable1(65, 91)
```

A számjegyek és a megfelelő ASCII-kódok a következő függvényhívással írathatók ki:

```
>>> myASCIItable1(48, 58)
```

A nem nyomtatható karakterek és a megfelelő ASCII-kódok a következő függvényhívással írathatók ki:

```
>>> myASCIItable1(0, 33)
```

5.16. algoritmus. Írjunk egy Python függvényt, amely kiírja a képernyőre a nyomtatható ASCII-szimbólumokat, a hozzájuk tartozó kódértékeket, és ezek bináris, oktális, illetve hexadecimális alakját.

```
def myCharTable(n, m):
    i1 = 0
    for i in range(n, m):
        print("%3s%7i" % (chr(i), i), end = ''),
        print("%18s" % format(i, 'b'), end = '')
        print("%7s" % format(i, 'o'), end = '')
        print("%5s" % format(i, 'X'), end = '')
        print("%8s" % ' ', end = '')
        if i1 % 2 == 0: print(end = '\n')
        i1 += 1

>>> myCharTable(33, 128)
```

5.6. Az Unicode szabvány és az utf-8 kódolás

A különböző nyelvek írásjeleinek a kódolását, megjelenítését, kezelését teszi lehetővé az Unicode szabvány [6]. Segítségével a világon használt összes szimbólumnak megfeleltethető egy egész szám, pontosabban egy kódpont. 144697 karaktert kezel, amelyben benne vannak a különböző népcsoportok által használt hagyományos és modern írásjelek, nem nyomtatható, vezérlő karakterek, hangulatjelek, műszaki jelek stb. A kódpontok nem mindig tárolhatóak el egy bájton, az alkalmazott kódolási eljárások határozzák meg, hogy a kódpontok hogyan alakíthatóak át bájtokká. Az egyik leggyakrabban alkalmazott kódolási eljárás az ASCII-kompatibilis UTF-8 kódolás [6].

Példa: Az előző 5.16. algoritmus segítségével különböző írásjeleket tudunk megjeleníteni, próbáljuk ki a következő lekérdezéseket:

```
>>> myCharTable(51700, 51710)
>>> myCharTable(630, 688)
```

5.17. algoritmus. Írjunk egy Python függvényt, amely véletlenszerűen generál `n` darab karaktert, amelyek `cKod` kódjára fennáll: `k1`≤`cKod`≤`k2`, és amely kiírja a karaktereket, a karakterek kódját, a kategóriáját és nevét.

```
import random, unicodedata
def unicode(n, k1, k2):
    for i in range(0, n):
        cKod = random.randint(k1, k2)
        try:
            c = chr(cKod)
            print(i, c, cKod, end = ' ')
            print(unicodedata.category(c), end = ' ')
            print(unicodedata.name(c), end = '\n')
        except:
            print ('hiba', end = '\n')

>>> unicode(5, 65, 91)
    ...
>>> unicode(10, 10000, 12000)
    ...
```

Az utf-8 kódolás lehetővé teszi bármilyen Unicode kódpont tárolását, ezt változó hosszúságú karakterkódolási technikával oldja meg. A 7-bites ASCII-kódokat saját kódértékükkel tárolja, a nagyobb értékű kódpontokat a kódpont nagyságától függően kettő, három vagy négy bájton tároja. Egybájtos tárolás esetén az első bit jelölő bit, kétbájtos tároláskor az első bájt első három bitje és a második bájt első két bitje lesz a jelölő bit, `110`, illetve `10` értékekkel. Hárombájtos tárolás esetén az első bájt első négy bitje, a második és harmadik bájt első két bitje tölti be a jelölő bitek szerepét, `1110`, illetve `10` értékekkel. Négybájtos tárolás esetén az első bájt jelölő bitjeinek értéke `11110`, a további három bájt mindegyik jelölőbitjének értéke `10`. Az 5.1. táblázat a kódpontok értéke alapján összegzi a fent leírtakat.

Az 5.2. táblázat az `ű` betű kódolását mutatja, amelynek kódértéke 369. Az 5.2. táblázat feltünteti az utf-8-as bitértékek hexadecimális és tízes számrendszerbeli értékeit is.

A Python `str` típusa Unicode kódpontokat kezel:

bájtszám	bitszám	1. kódpont	2. kódpont	1. bájt	2. bájt	3. bájt	4. bájt
1	7	U+0000	U+007F	0bbbbbbb			
2	11	U+0080	U+07FF	110bbbbb	10bbbbbb		
3	16	U+0800	U+FFFF	1110bbbb	10bbbbbb	10bbbbbb	
4	21	U+10000	U+10FFFF	11110bbb	10bbbbbb	10bbbbbb	10bbbbbb

5.1. táblázat

karakter	kód	bin	1. bájt			2. bájt		
			bin	hex	int	bin	hex	int
ű	369	101 110001	110 00101	0xc5	197	10 110001	0xb1	177

5.2. táblázat. Az ű betű utf-8 kódolása

```
>>> ord('ű')
    369
>>> type('ű')
    <class 'str'>
```

A tulajdonképpeni bájtszekvenciák kezelését a Python a **bytes** típussal végzi. Az `str` típusú értékek természetesen átalakíthatóak `bytes` típusú értékekké. Az átalakítást az `encode`, a visszaalakítást pedig a `decode` metódusokkal végezhetjük. Az átalakítás után a `b` prefix használatával jelzi a Python, hogy az eredmény nem `str`, hanem `bytes` típusú:

```
>>> 'ű'.encode()
    b'\xc5\xb1'
>>> type('ű'.encode())
    <class 'bytes'>
>>> b'\xc5\xb1'.decode()
    'ű'
```

A csak ASCII-kódokat tartalmazó stringből a `b` prefix használatával `bytes` típusú adatokat tudunk létrehozni. Nem lesz sikeres a konverzió, ha a bemenet nem csak ASCII-karaktereket tartalmaz:

```
>>> mStr = b'muszaki'
>>> type(mStr)
    <class 'bytes'>
>>> mStr = b'műszaki'
    SyntaxError: bytes can only contain ASCII characters.
```

`str` típusról `bytes` típusra úgy is alakíthatunk, ha a `bytes` függvényt használjuk, de fel kell tüntetni, hogy milyen kódolásról akarunk alakítani, ellenkező esetben, vagy ha nem lehetséges, akkor hibaüzenetet kapunk:

```
>>> mStr = bytes('muszaki', 'ascii')
>>> mStr
    b'muszaki'
>>> mStr = bytes('műszaki')
    ... TypeError: string argument without an encoding...
>>> mStr = bytes('műszaki', 'ascii')
    ... UnicodeEncodeError: 'ascii' codec can't encode...
>>> mStr = bytes('műszaki', 'utf-8')
>>> mStr
    b'm\xc5\xb1szaki'
```

A `bytes` függvény megfelelő szerkezetű `list` típusú adat átalakítására is alkalmas. Ha a listaelemek között nagyobb érték is szerepel, mint 255, akkor hibaüzenettel szembesülünk:

```
>>> bytes([109, 197, 177, 115])
    b'm\xc5\xb1s'
>>> bytes([109, 300, 177, 115])
    ...ValueError: bytes must be in range(0, 256)
```

A számítástechnikában a bájtsorrend (endianness) az adatok, a bájtok szekvenciáinak a memóriában való tárolási módját jelenti, amely kétféleképpen is lehetséges [34]. A **big-endian** tárolás esetében a legnagyobb helyi értékű bájt, azaz a legbaloldalibb bájt, angolul MSB (most significant byte) lesz a legalacsonyabb címen eltárolva, míg a **little-endian** esetében a legkisebb helyi értékű bájt, azaz legjobboldalibb bájt, angolul LSB (least significant byte) lesz a legalacsonyabb címen eltárolva.

Példa: A `3D F1 49 D1` egybájtos, illetve kétbájtos tárolása a `32`-es memóriacímtől kezdődően, ahol a legnagyobb helyi értékű bájt a `3D`, az 5.3. táblázatban látható.

A big-endian és little-endian elnevezések eredete Jonathan Swift[2] *Gulliver utazásai* című könyvében található. A könyv szereplőinek egy vitája alkalmával felvetődik a kérdés, hogy melyik végén kell megtörni egy lágytojást, a tojás hegyesebb vagy laposabb végén? Természetesen a szereplők nem tudnak megegyezni a helyes válaszban.

A számítógépes hálózatok protokolljai a csomagok címeiben a big-endian tárolást alkalmazzák, míg a legtöbb számítógép, például az Intel

2 Jonathan Swift ír szatirikus író (1667–1745)

big-endian, 1 bájtos tárolás

memória cím	32	33	34	35
érték	3D	F1	49	D1

big-endian, 2 bájtos tárolás

memória cím	32		33	
érték	F1	3D	D1	49

little-endian, 1 bájtos tárolás

memória cím	32	33	34	35
érték	D1	49	F1	3D

little-endian, 2 bájtos tárolás

memória cím	32		33	
érték	D1	49	F1	3D

5.3. táblázat. A `3D F1 49 D1` egybájtos, illetve kétbájtos tárolása

alapú gépek a little-endian szerint kezelik a bájtokat. A kompatibilitás miatt az adatok hálózatra történő kiküldése előtt, illetve a hálózatról érkező adatok fogadása után a bájtokon konverziót hajtanak végre.

A Python bájtsorrendje a számítógép processzora által használt bájtsorrendet használja. A következő kódsor végrehajtása után megtudhatjuk, hogy a számítógépünk processzora milyen bájtsorrendet alkalmaz:

```
>>> import sys
>>> if sys.byteorder == 'little':
        print('A bájtsorrend little-endian!')
    else:
        print('A bájtsorrend big-endian!')
```

Pythonban egy `int` típus érték `bytes` típusú értékre való átalakítására, illetve visszaalakítására beépített függvényeket használhatunk. A `to_bytes()` egy egész számot alakít át 256-os számrendszerbe, az eredmény egy bájt szekvencia lesz hexadecimális alakban, amelynek típusa `bytes`. Szükséges megadni a kimeneti bájtszekvencia hosszát és a bájtsorrendet, amelyre szintén a `big` és `little` megnevezéseket kell használni [38]:

```
>>> nr = 63587321362
>>> bitLen = nr.bit_length()
>>> bajtLen = bitLen // 8 + 1
>>> nr.to_bytes(bajtLen, byteorder = 'big')
```

```
    b'\x0e\xce\x19\x86\x12'
>>> nr.to_bytes(10, byteorder = 'big')
    b'\x00\x00\x00\x00\x00\x0e\xce\x19\x86\x12'
>>> nr.to_bytes(bajtLen, byteorder = 'little')
    b'\x12\x86\x19\xce\x0e'
```

A korábban bemutatott `nrToBaseB` függvénnyel (5.1. algoritmus) a `big` bájtsorrend szerinti átalakított értéket kapjuk:

```
>>> nrToBaseB(nr, 256)
    [14, 206, 25, 134, 18]
>>> for elem in b'\x0e\xce\x19\x86\x12':
        print(int(elem), end = ' ')
    14 206 25 134 18
```

Matematikailag ez azt jelenti, hogy az eredménylista első elme a legnagyobb hatványkitevőn levő tag együtthatója lesz:

$$63587321362 = 14 \cdot 256^4 + 206 \cdot 256^3 + 25 \cdot 256^2 + 134 \cdot 256^1 + 18 \cdot 256^0.$$

A visszaalakítást a `from_bytes`-al kell végezni, amely az `int` osztály egy metódusa, ezért a meghívás módja eltér a `to_bytes` meghívás módjától:

```
>>> bajtB = b'\x0e\xce\x19\x86\x12'
>>> int.from_bytes(bajtB, byteorder = 'big')
    63587321362
>>> bajtL = b'\x12\x86\x19\xce\x0e'
>>> int.from_bytes(bajtL, byteorder = 'little')
    63587321362
```

A `nrToBaseB`-vel kapott listát is megadhatjuk a `from_bytes`-nak bemenetként, de előtte szükséges a listát `bytes` típusra alakítani:

```
>>> bajtI = bytes([14, 206, 25, 134, 18])
>>> int.from_bytes(bajtI, byteorder = 'big')
    63587321362
```

5.18. algoritmus. Írjunk egy Python függvényt, amely az XOR műveletet alkalmazva titkosít egy `bytes` típusú tartalmat, egy szintén `bytes` típusú kulcs segítségével, úgy, ahogyan azt korábban már leírtuk (5.14. algoritmus).

```
def cryptFunc(bStr, bKey):
    C = bytes([])
    i, n = 0, len(bKey)
```

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

>>> cryptFunc('Sapientia'.encode(), 'kulcs2021'.encode())
    b'8\x14\x1c\n\x16\\D[P'
```

A `cryptFunc` függvényben az `XOR` művelet alkalmazásakor a listaelemeket nem kell átalakítani az `ord` függvénnyel, emiatt a `chr` függvényt sem kellett használni. A `cryptFunc` kimenete `bytes` típusú érték lesz.

A visszafejtéshez meg kell adni a rejtjelezett szöveget és a kulcsot, mindkettő `bytes` típusú kell legyen:

```
>>> crList = b'8\x14\x1c\n\x16\\D[P'
>>> cryptFunc(crList, 'kulcs2021'.encode())
     b'Sapientia'
```

Ha azt szeretnénk, hogy a visszafejtés eredménye `str` típusú érték legyen, akkor még a `decode` függvényt is kell alkalmazni:

```
>>> cryptFunc(crList, 'kulcs2021'.encode()).decode()
    'Sapientia'
```

A rejtjelezett értéket gyakran tizenhatos számrendszerben tárolják, ehhez a `hex` metódust lehet alkalmazni. A fordított művelethez pedig a `bytes.fromhex` metódust használhatjuk:

```
>>> crList.hex()
    '38141c0a165c445b50'
>>> bytes.fromhex(crList.hex())
    b'8\x14\x1c\n\x16\\D[P'
```

5.7. A Base64 kódolás

A Base64 kódolás segítségével tetszőleges bájtszekvenciát lehet szöveges típusú tartalommá átalakítani, amelynek segítségével például képfájlok, hangfájlok ágyazhatóak be különböző HTML fájlokba. Egy másik alkalmazási területe a kriptográfia, ahol a rejtjelezett szöveget szokták tovább

alakítani Base64-es kódolással. A kriptográfiai kulcsok tárolását is sok esetben Base64-es kódolást alkalmazva tárolják, itt is a nyomtatható tartalom elérése a cél.

A kódolást alkalmazva egy tetszőleges bájtsorozatot tudunk átalakítani egy olyan bájtsorozattá, amelyben csak angol ábécébeli nagybetűk, kisbetűk, számjegyek és három speciális szimbólum található [35]. A kódolt szöveg előállításához összesen tehát 64 + 1 fajta szimbólumot használnak. A következő műveletsorokban a sorszám, azaz a kódindex megadásával kiválasztásra kerül az `ABC64`-ből a megfelelő karakter, illetve az `index`-szel meg tudjuk határozni egy adott karakter kódindexét:

```
>>> ABC64 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> ABC64 += 'abcdefghijklmnopqrstuvwxyz'
>>> ABC64 += '0123456789+/='
>>> ABC64[20]
    U
>>> ABC64.index('2')
    54
```

A kódolás során az algoritmus a bemeneti `bytes` típusú sorozat minden 3 bájtjából 4 bájtot hoz létre, ezért a kimenet akár 33 százalékkal is nagyobb lehet, mint a bemeneti bájtszekvencia. A = karakternek speciális szerepe van, a kódolt adatsor végén foglal helyet, és azt jelzi, hogy a bemeneti bájtsorozat 3-mal való osztási maradéka 0, 1 vagy 2.

szöveg	S		a		p		i					
karakterkód	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		=	=

5.4. táblázat. A `Sapi` szó Base64-es kódolása

Az 5.4. táblázat a `Sapi` szó Base64 kódolását mutatja. A táblázat első sorában a karakterértékek vannak, a második sorban a karakterek kódjai (adott esetben az ACSII-kódok), ahol az utolsó két oszlop értéke nulla, ami ebben az esetben azt jelzi, hogy a Base64-es alakban az utolsó két karakter = lesz. A harmadik sorban a karakterek kódjainak megfelelő bináris értékek

láthatóak, majd a negyedik sorban ugyanezeket az értékeket látjuk, de 6 bitenként csoportosítva. A **kódindex** nevű sorban az átcsoportosítás után kapott bináris alakoknak a 10-es számrendszerbeli értékei láthatóak, amely értékek mindig kisebbek lesznek, mint 64, mert 6 biten 63 a legnagyobb érték, amit el lehet tárolni. A **base64 kód** nevű sorban a számokhoz rendelt karakterek vannak feltüntetve, amelyek a kódindex mint sorszám alapján az `ABC64` karakterláncból kerültek kiválasztásra.

A Pythonban a `base64` könyvtárcsomag importálásával lehet elérni a megfelelő függvényeket. A `b64encode` lesz a kódoló függvény, a `b64decode` pedig a dekódoló függvény. Mindkét függvény bemenetként `bytes` típusú értéket vár, és a kimenetük is `bytes` típusú érték lesz.

```
>>> from base64 import b64encode, b64decode
>>> codedStr = b64encode('Sapientia'.encode())
>>> codedStr
    b'U2FwaWVudGlh'
>>> type(codedStr)
    <class 'bytes'>
>>> b64encode('Sapientia')
    ...TypeError: a bytes-like object is required... 'str'
>>> b64decode(codedStr)
    b'Sapientia'
>>> b64decode(codedStr).decode()
    'Sapientia'
```

Példa: A `'Sapientia'` szó lépésenkénti Base64 kódolását a következő Python kód végrehajtásával tudjuk nyomon követni.

```
from base64 import b64encode, b64decode
def nyomonkovetesBase64(myStr):
    lStr = len(myStr)
    for i in range(0, lStr):
        codedStr = b64encode(myStr[:i+1].encode())
        print('%10s%s' % (myStr[:i+1], codedStr.decode()))

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

5.19. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy `bytes` típusú bemenet Base64-es alakját.

```
from string import ascii_uppercase, ascii_lowercase
from string import digits
```

```
1   ABC64 = ascii_uppercase + ascii_lowercase
2   ABC64 += digits + '+/='
3   def myB64Encode(mB):
4       E = b''
5       n = len(mB)
6       for i in range (0, n, 3):
7           b = (mB[i] & 0xFC) >> 2
8           E += ABC64[b].encode()
9           b = (mB[i] & 0x03) << 4
10          if i + 1 < n:
11              b = b | ((mB[i + 1] & 0xF0) >> 4)
12              E += ABC64[b].encode()
13              b = (mB[i + 1] & 0x0F) << 2
14              if i + 2 < n:
15                  b = b | (( mB[i + 2] & 0xC0) >> 6)
16                  E += ABC64[b].encode()
17                  b = mB[i + 2] & 0x3F
18                  E += ABC64[b].encode()
19              else:
20                  E += ABC64[b].encode()
21                  E += b'='
22          else:
23              E += ABC64[b].encode()
24              E += b'=='
25      return E
```

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

Az `E bytes` típusú változóba állítjuk elő a Base64-es kódolt szöveget, kezdetben ez egy üres `bytes`. A `b` változóba a kódindex-értékeket fogjuk meghatározni, bitműveleteket alkalmazva a megfelelő `mB[i], mB[i+1], mB[i+2]` értékeken:

– A 7. sorban `mB[i]` legkisebb helyi értékű **két** bitjét állítjuk nullásra, ahol `0xFC = 0b111111`**`00`**,

– A 9. sorban `mB[i]` legnagyobb helyi értékű **hat** bitjét állítjuk nullásra, ahol `0x03 = 0b`**`000000`**`11`,
– A 11. sorban az `mB[i + 1]` legkisebb helyi értékű **négy** bitjét állítjuk nullásra, ahol `0xF0 = 0b1111`**`0000`**,
– A 13. sorban az `mB[i + 1]` legnagyobb helyi értékű **négy** bitjét állítjuk nullásra, ahol `0x0F = 0b`**`0000`**`1111`,
– A 15. sorban az `mB[i + 2]` legkisebb helyi értékű **hat** bitjét állítjuk nullásra, ahol `0xC0 = 0b11`**`000000`**,
– A 17. sorban `mB[i + 2]` legnagyobb helyi értékű **két** bitjét állítjuk nullásra, ahol `0x3F = 0b`**`00`**`111111`.

A `>>` műveletek eredményeként megszabadulunk a megfelelő számú legkisebb helyi értékű bitektől, a 7. sorban kettő, a 11. sorban négy, a 15. sorban ez hat bitet jelent. A `<<` elvégzése után a legnagyobb helyi értékű bitek fognak kiesni, a 9. sorban négy, a 13. sorban 2 bit. A 11. sorban a `|` művelet alkalmazásával a bal oldali operandus legkisebb helyi értékű két bitjét és a jobb oldali operandus legnagyobb helyi értékű négy bitjét *rakjuk össze*. A 15. sorban a `|` alkalmazásával a bal oldali operandus legkisebb helyi értékű négy bitjét és a jobb oldali operandus legnagyobb helyi értékű két bitjét *rakjuk össze*.

5.20. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy Base64-es formában megadott bemeneti paraméter eredeti alakját. Alkalmazni fogjuk az előző feladatnál megadott `ABC64` karakterláncot

```
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
```

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

A `myB64Decode` négyesével dolgozza fel az `eB` elemeit. Az `index`-szel határozzuk meg egy adott szimbólum sorszámát az `ABC64` ábécéből. A `chr` függvény alkalmazásával `bytes` típusról alakítunk `str` típusra. A `temp`-be bitműveletek alkalmazásával hozzuk létre az eredeti szimbólumot, amely egy 256-nál kisebb érték kell legyen, ezért alkalmazzuk a 255-tel való `&` műveletet. Mielőtt a `D`-hez fűznénk a `temb`-be létrehozott értéket, szükséges a `bytes` típusra való konverzió.

5.8. Bináris állományok

Szöveges tartalmak mellett bináris adatokat, mint például kép-, videó- és hang-fileokat is fel tudunk dolgozni. Ezek kezelése bináris állományok feldolgozását jelenti [6]. Hasonlóan a szövegállományokhoz, egy bináris állományból is tudunk bájtokat beolvasni, tudunk bájtokat kiírni, és tudunk hozzá is adni bájtokat. Ezekhez a már bemutatott Python `open, read, write, close` stb. függvényeket/metódusokat használjuk [38].

5.21. algoritmus. Írjunk egy Python függvényt, amely kiírja egy `hexImage.txt` nevű szövegállományba egy bináris állomány bájtjait hexadecimális számrendszerbe.

```
def fileToHexa():
    fnev = input('kerek egy allomany nevet: ')
    byteL, chunk = bytes([]), 32
    try:
        inf = open(fnev, 'rb')
        out = open('hexImage.txt', 'wt')
        while True:
            byteL = inf.read(chunk)
            if not byteL: break
            for byte in byteL:
                if byte < 16: temp = '0' + format(byte, 'X')
                else: temp = format(byte, 'X')
                out.write(temp + ' ')
        inf.close()
        out.close()
```

```
    except:
        print('megnyitasi/olvasasi hiba!!')
        return

>>> fileToHexa()
    kerek egy allomany nevet: valami.pdf
```

A `fileToHexa` nem ír ki eredményt a képernyőre, helyette az aktuális mappába létrehozza a `hexImage.txt` szövegállományt. A korábban ismertetett `open` függvény második paramétere most `b`, ezzel jeleztük, hogy az állományban levő adatokat bájtok sorozataként kell kezelni. Az állomány tartalmát 32 bájtonként dolgoztuk fel, amelyet a `chunk = 32` értékadás tett lehetővé. A bináris állomány bájtjainak az átalakításához a beépített `format` függvényt használtuk, és minden egyes hexa érték után egy szóközt is fűztünk.

5.22. algoritmus. Írjunk egy Python függvényt, amely egy `hexImage.txt` nevű szövegállományban található hexa értékek alapján meghatározza az állomány bináris tartalmát

```
def fileFromHexa():
    fnev = input('kerek egy allomany nevet: ')
    chunk = 3 * 10
    try:
        inf = open('hexImage.txt', 'rt')
        out = open(fnev, 'wb')
        while True:
            strL = inf.read(chunk)
            if not strL: break
            strL = strL.split()
            byteL = bytes()
            for s in strL:
                temp = int(s, 16)
                byteL += temp.to_bytes(1, byteorder = 'big')
            out.write(byteL)
        inf.close()
        out.close()
    except:
        print('megnyitasi/olvasasi hiba!!')
        return

>>> fileFromHexa()
    kerek egy allomany nevet: valami.pdf
```

A `hexImage.txt` tartalmát 3×10 bájtonként dolgoztuk fel, mert minden bájt kiírásához két szimbólumot használtunk plusz a szóközt. Az `int` alkalmazásával minden egyes szót átalakítottunk 16-os számrendszerből 10-esbe, majd a `to_bytes`-sal létrehoztuk a `bytes` típusú értékeket, amelyeket kiírtunk a file-ba.

5.23. algoritmus. Írjunk egy Python programot, amely a következő menüpontok segítségével:

- `kilépés`: kilép a programból,
- `kulcs generálás/mentés`: véletlenszerűen generál egy `k` bájtos kulcsot, és kiírja egy bináris állományba a kulcs Base64-es alakját,
- `titkosítás`: beolvassa a megfelelő kulcs értékét egy bináris állományból, és `XOR` műveletet alkalmazva, ahogyan korábban is leírtuk, meghatározza egy billentyűzetről beolvasott karakterlánc rejtjelezett értékét, amelynek Base64-es alakját kiírja egy bináris állományba,
- `visszafejtés`: beolvassa a megfelelő kulcs értékét és a rejtjelezett szöveg értékét egy-egy bináris állományból, és visszafejti a rejtjelezett szöveget.

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

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

```
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!')
```

A `keyRead` beolvassa a kulcsot, a `keyWrite` pedig kiírja a megadott állományba azt a `k` bájtos kulcsot, amelyet a `keyGen`-nel hoz létre. A `keyGen` a kulcsot a `random` modulban található `sample` függvénnyel hozza létre. A Base64-es kódolt alakokat a `base64` modul `b64encode`, illetve `b64decode` függvényeivel oldottuk meg, de használhattuk volna a saját implementációinkat is. A `bytes` típusú szöveg titkosítását a korábban megírt `cryptFunc` függvénnyel végeztük (5.14. algoritmus). A rejtjelezett szöveg Base64-es alakjának állományba való kiírását a `cryptWrite` függvénnyel végeztük, a rejtjelezett szöveg beolvasását pedig a `crytpRead`-del. Hibakezelést alkalmazva, illetve a `keyRead`, `cryptRead` függvények -1-es visszatérési értékén keresztül biztosítottuk az állománymegnyitási hibákból adódó leállás elkerülését.

```
def keyGen(k):
    key = bytes(sample(range(0, 256), k))
    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

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
```

5.9. Kitűzött feladatok

5.1. feladat. Írjunk egy-egy Python függvényt, amely meghatározza egy természetes szám

1. első számjegyének értékét `b` számrendszerben,
2. számjegyeinek számát `b` számrendszerben,
3. számjegyeinek összegét `b` számrendszerben,
4. páros számjegyeinek számát `b` számrendszerben.

5.2. feladat. Írjunk egy-egy Python függvényt, amely meghatározza:

1. egy szám `b` számrendszerbeli számjegyei alapján a b^k számrendszerbeli alakot,
2. egy szám b^kszámrendszerbeli számjegyei alapján a `b` számrendszerbeli alakot,
3. egy szám 2-es számrendszerbeli számjegyei alapján, hogy melyik a szám legnagyobb számjegye 2^k számrendszerben,

4. egy szám 2^k számrendszerben megadott értéke alapján, hogy hány 1-es számjegy van a szám 2-es számrendszerbeli alakjában,
5. hogy két szám 2^k számrendszerben megadott értéke alapján melyik számnak a 2-es számrendszerbeli alakjában van több 1-es.

5.3. feladat. Írjunk egy-egy rekurzív függvényt, amely átalakít

1. egy 10-es számrendszerben megadott számot `b` számrendszerbe,
2. egy `b` számrendszerben megadott számot 10-es számrendszerbe,
3. egy `b` számrendszerben megadott számot b^k számrendszerbe,
4. egy b^k számrendszerben megadott számot `b` számrendszerbe.

5.4. feladat. Írjunk egy Python függvényt, amely egy adott szám esetében meghatározza azokat a Fibonacci-számokat, amelyek összegeként felírható a szám.

5.5. feladat. Írjunk egy Python függvényt, amely szövegállományban található számok mindegyikére meghatározza a Fibonacci-számrendszerbeli alakokat.

5.6. feladat. Írjunk egy Python függvényt, amely beolvas stringként egy bináris számsorozatot, amely feltételezhetően egy szám Fibonacci-számrendszerbeli számjegyeit jelöli, majd meghatározza a megfelelő tízes számrendszerbeli számot.

5.7. feladat. Írjunk egy Python függvényt, amely meghatározza az `n`-dik Fibonacci-szám első és utolsó számjegyét.

5.8. feladat. Írjunk egy Python függvényt, amely beolvas stringként egy számsorozatot, amely feltételezhetően egy szám faktoriális számrendszerbeli számjegyeit jelölik, majd meghatározza a megfelelő tízes számrendszerbeli számot.

5.9. feladat. Írjunk egy Python függvényt, amelynek 1 legyen a kimenete, ha a függvény `int` típusú paramétere páros számú, 1-es bitet tartalmaz. Ha páratlan az 1-es bitek száma, akkor a kimenet legyen -1.

5.10. feladat. Írjunk egy Python függvényt, amely véletlenszerűen generál egy-egy `n` bites számot, eredményként meghatározza a két szám bitenkénti `AND, OR, XOR` értékeit, majd meghatározza mindhárom eredményben az 1-es és 0-ás bitek számát. Alkalmazzuk a megírt Python függvényt több számpárra, és minden számpár esetében határozzuk meg az `AND, OR, XOR` elvégzése után kapott eredményekben az 1-es és 0-ás bitek száma közötti különbségeket! Mit veszünk észre?

5.11. feladat. Írjunk egy Python függvényt, amely `n` különböző értékeire meghatározza, hogy hány 1-es bit van a 2^n, 2^{n-2}, 2^{n+1} számokban.

5.12. feladat. Írjunk egy Python függvényt, amely kiírja egy szövegállományba egy bináris állomány bájtjait hexa számrendszerben. Minden hexa értéket két szimbólummal jelenítsünk meg, és egy sorba 16 értéket tegyünk, az értékek közé pedig tegyünk egy-egy szóközt. Például a 15-ös bájtértéket a `0F` szimbólumokkal írassuk ki.

5.13. feladat. Írjunk egy Python függvényt, amely kiírja egy szövegállományba egy bináris állomány bájtjait bináris számrendszerben. Minden bináris értéket nyolc szimbólummal és egy space-szel jelenítsünk meg, és egy sorba 4 értéket tegyünk. Például a 23-as bájtértéket a `00010111` szimbólumokkal írassuk ki.

5.14. feladat. Írjunk egy-egy Python függvényt, amely az előző két feladatnál létrehozott szövegállományt visszaalakítja bináris állománnyá.

5.15. feladat. Írjunk egy Python függvényt, amely kiírja egy állományba a nyomtatható ASCII-szimbólumokat, a hozzájuk tartozó kódértékeket, és ezek bináris, oktális, illetve hexadecimális alakját. Formázzuk tetszés szerint a kimeneti file-t.

5.16. feladat. Írjunk egy Python függvényt, amely nagybetűsíti egy szövegállomány tartalmát, azaz minden angol ábécébeli kisbetűt átalakít nagybetűvé, a többi betűt pedig változatlanul hagyja.

5.17. feladat. Írjunk egy Python függvényt, amely meghatározza, hogy egy szövegállományban hány nyomtatható karakter van.

5.18. feladat. A *jelszavak.txt* állomány minden sorában két érték található szóközökkel elválasztva. Az első érték egy személy nevét jelölő karakterlánc, a második egy jelszót jelölő karakterlánc Base64-es alakja. Írjunk egy Python programot, amely megállapítja, hogy kinek van erős jelszava. Egy személynek akkor mondjuk, hogy erős a jelszava, ha a számjegyek és az angol ábécé kis- és nagybetűi mellett tartalmaz egyéb szimbólumokat is.

5.19. feladat. Írjunk egy Python függvényt, amely véletlenszerűen generál egy legalább 1024 bites számot, amelynek meghatározza 256-os számrendszerben a számjegyeit, majd ennek a Base64-es alakját kiírja egy szövegállományba. Írjuk meg a visszaalakító műveletsort is: olvassuk ki az állományból a kódolt értéket, dekódoljuk, majd a kapott 256-os számrendszerbeli számot alakítsuk vissza 10-es számrendszerbe.

5.20. feladat. Írjunk egy Python programot, amely meghatározza egy tetszőleges bináris állomány titkosított értékét. A titkosítást az XOR művelettel végezzük, amelyet alkalmazzunk a bináris állomány bájtjai és egy paraméterként megadott kulcs értékei között. A kulcs bájtjait egy lista típusú változóban adjuk meg, amelynek elemeit egymás után, körkörösen vegyük, ahogyan azt korábban is tettük (5.14. algoritmus). Az ellenőrzéshez hívjuk meg úgy a függvényt, hogy az fejtse vissza a titkosított állományt.

5.21. feladat. A *cryptXOR* állomány egy pdf-állomány titkosított értéke, ahol a titkosított állomány a korábban leírt technikával, XOR-műveletet használva lett előállítva (5.14. algoritmus). Az *adatokXOR.txt* állomány több személynevet és a hozzájuk tartozó kulcsértékét tartalmazza, ahol az állományban a kulcsértékek Base64-es alakja lett eltárolva. Írjunk Python programot, amely megmondja, hogy ki végezte a titkosítást, meghatározva a kulcsot és meghatározva az eredeti pdf-t is, tudva, hogy egy pdf első négy bájtja mindig a következő: `0x25, 0x50, 0x44, 0x46`.

6. fejezet

Számelmélet

A számelmélet az egész számok tulajdonságaival foglalkozik, amelyekről korábbi fejezetekben is volt már szó [27, 33, 39]. Különösen fontos szerepet tölt be a mai számítástechnikában, hiszen a kódelmélet, a kriptográfia nem érthető meg alapvető számelméleti ismeretek nélkül. A számelmélet témaköreivel szinte elsőként találkozik a kisiskolás diák, mert a prímszámok, a legnagyobb közös osztó, a legkisebb közös többszörös fogalmai már a közlépiskolában is bevezetésre kerülnek. Jelen fejezet alapvető számelméleti ismeretek és a hozzájuk kapcsolódó tételek bemutatásával indul, később azonban bonyolultabb fogalmak, gyakorlati alkalmazások ismertetésére fog sor kerülni.

6.1. Prímszámok

6.1. értelmezés. Prímszámoknak hívjuk azokat az 1-nél nagyobb pozitív egész számokat, amelyek nem oszthatóak csak 1-gyel és önmagukkal.

A fenti értelmezés alapján kijelenthetjük, hogy a pozitív összetett számok azok az 1-nél nagyobb pozitív egész számok, amelyek nem prímszámok.

6.1. tétel. Minden 1-nél nagyobb pozitív egész számnak van egy prímosztója [27].

m

Bizonyítás: Tegyük fel az állítás ellenkezőjét, azaz hogy létezik egy olyan pozitív egész szám, amelyiknek nincs prímosztója. Ezek a számok meghatároznak egy nem üres halmazt, legyen ez M. A természetes számok halmaza azonban jól rendezett, azaz minden nem üres részhalmazának van egy legkisebb eleme. Legyen M-nek a legkisebb eleme n. Tehát n lesz az a legkisebb olyan pozitív egész szám, amelyiknek nincs prímosztója. Mivel $n > 1$ és n-nek nincs prímosztója, n nem lehet prímszám, ezért felírható, hogy: $n = a \cdot b$, ahol $1 < a, b < n$. Mivel $a < n$ következik, hogy a-nak van prímosztója. De a bármely osztója osztja n-t, ez pedig ellentmondás.

□

6.2. tétel. Végtelen sok prímszám létezik [39].

A tételre több bizonyítás is létezik, a következő bizonyítást még Euklidész írta le *Elemek* című könyvében.

Bizonyítás: Feltételezzük, hogy a prímszámok halmaza véges. Jelöljük ezt a halmazt P-vel, elemeit pedig $p_1, p_2, \ldots, p_n$-nel. Legyen

$$Q = p_1 \cdot p_2 \cdot \cdots \cdot p_n + 1.$$

Bebizonyítható, hogy Q-nak van egy olyan prímosztója, amelyik nem szerepel a P halmazban. Ennek érdekében induljunk ki az ellenkezőjéből, legyen q egy osztója a Q-nak, és tegyük fel, hogy ez megegyezik valamelyik p_j-vel, $j \in \{1, \ldots n\}$. Ez azonban azt jelenti, hogy q osztja $Q - p_1 \cdot p_2 \cdot \cdots \cdot p_n = 1$-t, azaz 1-t. Ez ellentmondás, mert egy prímszám nem oszthatja az 1-et.

□

Példa: Ha összeszorozzuk az első 6 prímszámot, akkor kapunk egy olyan számot, amelynek biztosan lesz egy olyan prímszám osztója, amely nem egyezik meg az első 6 prímszám egyikével sem:

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30031 = \mathbf{59} \cdot 509.$$

Példa: Lenstra[1] viccesen *bizonyítja*, hogy 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 🙂.

6.3. tétel. Ha n egy összetett szám, akkor n-nek van egy olyan prímosztója, amelyik kisebb vagy egyenlő, mint $\sqrt{n}$ [33].

Bizonyítás: Ha n egy összetett szám, akkor $n = a \cdot b$, ahol $1 < a, b < n$, és tegyük fel, hogy $a \leq b$. Ekkor $a \cdot a \leq a \cdot b = n = \sqrt{n} \cdot \sqrt{n}$, fennáll tehát,

1 Hendrik Lenstra dán matematikus (1949–)

hogy $a \leq \sqrt{n}$. A 6.1. tétel alapján azonban a-nek van egy prímosztója. Mivel $n = a \cdot b$, ez a prímosztó n-nek is osztója lesz. Másfelől fennáll, hogy $a \leq \sqrt{n}$, tehát n-nek van egy olyan prímosztója, amelyik kisebb vagy egyenlő, mint $\leq \sqrt{n}$.

□

A kriptográfiában különösen fontosak azok az algoritmusok, amelyek nagy, legalább 1024 bites (300 számjegyű) prímszámok kigenerálására alkalmasak, pontosabban képesek leellenőrizni, hogy egy véletlenszerűen generált nagy páratlan szám prímszám-e vagy sem. Az ilyen típusú algoritmusokat prímtesztelő algoritmusoknak hívják. A számelmélet több ilyen algoritmus kidolgozását is lehetővé teszi.

Az **osztási próba módszere** (trial division) egy **nyers erő** (brute-force) típusú algoritmus. Annak érdekében, hogy egy páratlan n számról megállapíthassuk, hogy prímszám-e vagy sem, sorra kell próbálni az oszthatóságot a páratlan számokkal. Páros számokkal fölösleges az oszthatóságot vizsgálni, hiszen egyetlenegy páros prímszám létezik, a 2. A 6.3. tétel alapján pedig elégséges az oszthatóságot $\sqrt{n}$-ig vizsgálni. Egy átlagos számítógépen az algoritmus futási ideje 10^{15}-nél nagyobb számokra azonban elfogadhatatlan [22].

Eratosztenész szitája[2] segítségével meghatározhatjuk egy adott n-ig az összes prímszámot. Általában az első 10^6 prímszámot szokták ezzel a módszerrel kigenerálni [11, 39].

A **Miller**[3]**–Rabin-**[4] és a **Solovay**[5]**–Strassen**[6]-prímtesztelő algoritmusokat nagy, például 1024 bites számok esetében alkalmazzák. Mindkét algoritmus érdekessége, hogy nem egyértelműen állapítja meg egy páratlan számról, hogy prímszám. Valószínűségi prímteszteknek hívják őket, mert csak nagy valószínűséggel állítják a tesztelendő páratlan számról, hogy prímszám. A gyakorlatban széles körben alkalmazzák őket, de elsősorban a Miller–Rabin-algoritmust használják, annak jobb tulajdonságai miatt [14, 22, 35].

Az **AKS** (Agrawal[7]–Kayal[8]–Saxena[9]) prímtesztelő algoritmus [1] polinomiális időben képes megállapítani egy tetszőleges számról, hogy prímszám-e

2 Kürénei Eratoszthenész görög matematikus (i. e. 276–i. e. 194)
3 Gary Lee Miller amerikai infomatikus
4 Michael O. Rabin izraeli matematikus (1931–)
5 Robert M. Solovay amerikai matematikus (1938–)
6 Volker Strassen német matematikus (1936–)
7 Manindra Agrawal indiai informatikus (1966–)
8 Neeraj Kayal, indiai matematikus, informatikus
9 Nitin Saxena indiai matematikus, informatikus(1981–)

vagy sem. Csupán elméleti jelentősége nagy, gyakorlatban nem használják elfogadhatatlan futási ideje miatt.

A fejezet következő részében az osztási próba módszerét és Eratoszthenész szitáját mutatjuk be. A Miller–Rabin-algoritmusnak a későbbiekben egy teljes fejezetet szentelünk, a Solovay–Strassen- és AKS-algoritmusokkal azonban nem foglalkozunk, mivel a gyakorlatban ritkábban alkalmazzák őket.

6.1. algoritmus. Írjunk egy Python függvényt, amely megvizsgálja az osztási próba módszerével, hogy a bemenet, amely egy pozitív egész szám. prímszám-e vagy sem.

```
def tDPrimeTest_(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
```

Az algoritmus első két műveletsora két triviális tesztet kezel, ha ezek nem állnak fenn, akkor sorra próbálja a páratlan számokkal való oszthatóságot, és az első talált osztónál kijelenti, hogy a szám nem prímszám. Ha a tesztelendő szám négyzetgyökéig nem találunk osztót, akkor kijelenthető, hogy a szám prímszám. Ez akkor fog fennállni, amikor `i * i <= n`, amely ugyanazt jelenti, mint `i <=` $\sqrt{\texttt{n}}$. Megállapítható, hogy időigényesebb annak az eldöntése, hogy a szám prím, mint hogy összetett.

Példa: A fenti algoritmus gondolatmenetét követve vizsgáljuk meg, hogy 101 prímszám-e vagy sem:

- megállapítható, hogy 101 nem osztható-e 3, 5, 7, 9-cel,
- 11-gyel már nem kell oszthatóságot vizsgálni, mert $11 \cdot 11 = 121 \leq 101$,
- kijelenthető, hogy 101 prímszám.

Az osztási próba módszerének futási idejét némileg javíthatjuk, ha a 2-vel való oszthatóság mellett a 3-mal való oszthatóságot is a `while` előtt vizsgáljuk, és figyelembe vesszük, hogy a 3-nál nagyobb prímszámok csak $6i + 5$ vagy $6i + 1$ alakúak lehetnek, mert a $6i, 6i + 2, 6i + 3, 6i + 4$ alakúak vagy 2-vel, vagy 3-mal, vagy 6-tal oszthatóak.

Példa:
$6i + 5$ alakú számok: $5, 11, 17, 23, 29, 35, 41, \ldots$
$6i + 1$ alakú számok: $7, 13, 19, 25, 31, 37, 43, \ldots$

A módosított függvény, a `tDPrimeTest` a következő:

```
def tDPrimeTest(nr):
    if nr == 2 or nr == 3: return True
    if nr & 1 == 0 or 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

>>> tDPrimeTest(163625217465571)
    True
```

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

```
from random import randint
def primeGenTD(k):
    while True:
        nr = randint(0, k)
        if tDPrimeTest(nr): return nr

>>> primeGenTD(10**15)
```

A `while` keretén belül addig generálunk véletlen számokat, amíg a `tDPrimeTest` kimenete nem lesz `True`. Figyeljük meg, hogy az algoritmus időigénye hogyan nő, ha nagyobb bemeneti értéket adunk meg.

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

```
def eratL_(n):
    L = [True] * (n + 1)
    i = 3
    while i * i <= n:
        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

>> eratL_(70)
   [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67]
```

Az algoritmus a `while` ciklusban az `L` lista elemeit állítja be, amely aszerint fog `True` vagy `False` értékeket tartalmazni, hogy az adott sorszámú elem prímszám-e vagy sem. Ennek érdekében kezdetben az `L` lista minden egyes elemét `True`-ra állítjuk, és a `while`-ban levő `for` ciklusban az összetett sorszámú elemeket átállítjuk `False`-ra. Az összetett sorszámok meghatározása úgy történik, hogy ha az `i` sorszám prím, akkor az `L` listában az `i` többszörösének a pozícióin `i*i`-től kezdődően `i`-esével lépegetve az értékeket `False`-ra állítjuk. Megjegyezzük, hogy az `i*i` előtti összetett sorszámok már korábban átállításra kerülnek. A `while`-ban az `i`-t kettesével növeljük, mert a páros értékeket kihagyhatjuk. A `while` utáni `for` ciklusban tesszük át a `Prim` listába a prímszámokat az `L`-ben levő értékek alapján, ahol szintén figyelmen kívül hagyjuk a páros sorszámokat.

Példa: A fenti algoritmus gondolatmenetét követve meghatározzuk Eratosztenész szitájával 100-ig a prímszámokat.

Az első páratlan prímszám a 3, ennek összes többszöröse összetett szám lesz. Az algoritmus szerint az `L` listában ezen sorszámú elemek értékét `False`-ra állítjuk. A következő táblázatban, amely csak a páratlan sorszámok értékét mutatja, ezeket az értékeket áthúztuk. Az első többszörös, amit át kell húzni, az a $3 \cdot 3 = 9$-es lesz.

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 soron következő `True` értékű elem sorszáma 5. Ez a következő prímszám, az összes többszöröse összetett szám lesz. Az első többszöröse, aminek sorszámát át kell állítani `False`-ra, az a $5 \cdot 5 = 25$. A következő táblázatban, amelybe nem írtuk át az előzőleg áthúzott elemeket, ezeket az értékeket

áthúztuk.

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							

A soron következő `True` értékű elem sorszáma 7, ez a következő prímszám, az összes többszöröse összetett szám lesz. Az első többszöröse, aminek sorszámát át kell állítani `False`-ra, az a $7 \cdot 7 = 49$. A következő táblázatban ezeket az értékeket áthúztuk.

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 soron következő `True` értékű elem sorszáma 11, ez a következő prímszám, az összes többszöröse összetett szám lesz. Az első többszöröse, aminek sorszámát át kell állítani `False`-ra, az a $11 \cdot 11 = 121$. Ilyen sorszámú elem azonban nem szerepel a listában. A megmaradt páratlan sorszámú elemek mindegyike prímszám lesz, ezek a következők:

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 a következők lesznek, ahol az egyetlen páros prímszámot, a 2-est külön fűzzük az eredménylistához:

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	

Az `eratL_` függvény futási idején is javíthatunk, ha a tesztelést csak a $6i+5$ és $6i+1$ alakú számokkal végezzük:

```
def eratL(n):
    L = [True] * (n + 1)
    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
```

6.4. tétel. (A számelmélet alaptétele) Bármely 1-nél nagyobb pozitív egész szám a szorzótényezők sorrendjétől eltekintve, egyértelműen felírható prímszámok szorzataként. A kapott szorzatot **prímtényezős** vagy törzstényezős felbontásnak hívjuk [27, 33, 39].

Bizonyítás: A létezés bizonyításához legyen a egy 1-nél nagyobb pozitív egész szám, amelynek jelöljük p_1-gyel a legkisebb prímosztóját. Ekkor felírható: $a = p_1 \cdot b_1$. Most vegyük b_1 legkisebb prímosztóját, amelyet jelöljünk p_2-vel. Ekkor hasonlóan felírható: $b_1 = p_2 \cdot b_2$. A felírást addig folytathatjuk, amíg találunk egy olyan b_n számot, amely egyenlő 1-gyel, ekkor $b_{n-1} = p_n$-nel. Ha az egyenlőségekben visszahelyettesítjük a megfelelő értékeket, kapjuk: $a = p_1 \cdot p_2 \ldots p_n$.

Az egyértelműség bizonyításához tegyük fel, hogy a-nak létezik egy másik felírása, legyen ez: $q_1 \cdot q_2 \ldots q_s$, melyre fennáll a következő egyenlőség:

$$p_1 \cdot p_2 \ldots p_n = q_1 \cdot q_2 \ldots q_s.$$

Mivel a fenti egyenlőség jobb oldala osztható q_1-gyel, akkor a bal oldal is osztható kell legyen q_1-gyel, amiből következik, hogy a bal oldal egyik tényezője meg kell egyezzen q_1-gyel, és ez lehet akár p_1 is. Ha egyszerűsítünk, akkor a következő egyenlőséget kapjuk:

$$p_2 \cdot p_3 \ldots p_n = q_2 \cdot q_3 \ldots q_s.$$

Ismételve alkalmazva az egyszerűsítéseket kapjuk, hogy az egyik oldalon minden tényezővel egyszerűsítettünk, legyen ez a bal oldal. De ekkor a másik oldalon is egyszerűsítenünk kellett minden tényezővel, mert $q_{n+1}, \ldots q_s$ 1-nél nagyobb értékekre az nem állhat fenn, hogy $1 = q_{n+1} \cdot q_{n+2} \cdot \cdots \cdot q_s$. Amiből következik, hogy a két felírás egyforma.

□

Egy a szám prímtényezős felbontásában egy prímszám többször is előfordulhat. A prímtényezős felbontás azon alakját, amelyben minden prímszámot csak egyszer, a megfelelő hatványértéken írjuk, **kanonikus alaknak** hívjuk. Ekkor felírható:

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

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. Az adatbiztonság területén fontos szerepet töltenek be azok az algoritmusok, amelyek képesek hatékonyan megadni egy szám prímtényezős felbontását. Nagy számok esetében a prímtényezős felbontás meghatározására, ha a számításokat nem kvantumszámítógépen végezzük, **nem ismert hatékony eljárás**. A gyakorlatban sok fejlesztésre van még szükség ahhoz, hogy a kvantumszámítógépek átvegyék a mai számítógépek helyét.

Az egyik legkézenfekvőbb, ugyanakkor csak kis számokra működő prímfaktorizációs módszer a prímtesztelésnél bemutatott osztási próba módszeréhez hasonlít. Ez is brute-force típusú algoritmus, és a szakirodalomban erre is trial division néven hivatkoznak. A kriptográfia több hatékonyabb prímfaktorizációs algoritmust tart számon, ezekre a 6.12. fejezetben visszatérünk.

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

```
def tDPrimeFact_(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 > 1: Prim += [(nr, 1)]
    return Prim
```

Az első `while` ciklusban megszámoljuk, hogy hányszor osztható a szám 2-vel, ha osztható, akkor el is végezzük az osztást. A következő `while` keretén belül a páratlan számokkal való oszthatóságot vizsgáljuk, illetve ha lehetséges, akkor most is elvégezzük a megfelelő számmal való osztást. Az eredményt egy `tuple` elemtípusú listában határozzuk meg, ahol egy `tuple` elem első értéke az adott prímszámot, a második értéke pedig annak a hatványkitevőnek az értéket jelöli, amelyen a prímszám megjelenik a prímtényezős felbontásban.

A `tDPrimeFact_` függvény utolsó `if` művelete azt figyeli, hogy az osztások miatt módosított `nr`-nek van-e a `nr` négyzetgyökénél nagyobb prímosztója. Abban az esetben, ha van, akkor maximum egy ilyen osztója lehet a `nr`-nek, amely ugyanakkor prímszám is, tehát további oszthatósági vizsgálatok nem szükségesek, a `Prim` listába betehetjük a `(nr, 1)` tuple-elemet.

A `tDPrimeFact_` függvény futási idején is javíthatunk, ha a tesztelést csak $6i + 5$ vagy $6i + 1$ 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 > 1: 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
```

```
>>> tDPrimeFact(311887381743)
    [(3, 4), (7, 2), (78580847, 1)]
```

A `tDPrimeFact`-nál egy adott `x` prímszámmal az oszthatóságot a `szamolOszthatosag` függvényben vizsgáljuk, amely módosítja a `nr` és `Prim` változó tartalmát. A `nr`-t addig osztjuk `x`-szel, amíg osztható. A `Prim`-hez pedig hozzáfűzzük a megfelelő `tuple` elemet: a `x` számot és a hatványkitevőt, amelyen `x` a prímtényezős felbontásban szerepel.

Érdemes kipróbálni a függvényt különböző bemenetekre, és megfigyelni, hogy milyen futási időket mér a Python:

```
from time import time
def idomeres():
    st = time()
    print(tDPrimeFact(8330059312218275892805601363123 2))
    fs = time()
    print('idoigeny: ', round(fs - st, 4))

    st = time()
    print(tDPrimeFact(10838496441610139787933089695749 83))
    fs = time()
    print('idoigeny: ', round(fs - st, 4))

    st = time()
    print(tDPrimeFact(199715263670994809))
    fs = time()
    print('idoigeny: ', round(fs - st, 4))

>>> idomeres()
    [(2, 8), (101, 7), (1789, 3), (530063, 1)]
    idoigeny:  0.0501
    [(17, 8), (83, 7), (1789, 3)]
    idoigeny:  0.0105
    [(206864963, 1), (965437843, 1)]
    idoigeny:  19.3885
```

Vegyük észre, hogy a legrosszabb futási időt akkor kaptuk, amikor a legkisebb számot próbáltuk faktorizálni. Ha azonban figyelmesek vagyunk, akkor észrevehetjük, hogy ebben az esetben a bemeneti szám két nagyobb prímszám szorzatából áll, míg az első két meghívás esetében a prímtényezők legtöbbje kicsi szám. Egy szám kis prímtényezőinek a meghatározása tehát

nem igényel sok számítási kapacitást, éppen ezért az igazi kihívást a prímfaktorizációs algoritmusok terén az jelenti, hogy nagy prímszámok szorzatából álló számnak határozzuk meg a prímtényezőit.

Két szám prímtényezős felbontása alapján meghatározható a két szám legnagyobb közös osztója és legkisebb közös többszöröse is. Nagy számok esetében ez a módszer azonban nem hatékony, mert a prímfaktorizációra nem ismert hatékony algoritmus.

$$\text{Ha } a = \prod p_i^{a_i} \text{ és } b = \prod p_i^{b_i}, \text{ akkor } (a,b) = \prod p_i^{min\{a_i,b_i\}}.$$

$$\text{Ha } a = \prod p_i^{a_i} \text{ és } b = \prod p_i^{b_i}, \text{ akkor } [a,b] = \prod p_i^{max\{a_i,b_i\}}.$$

Példák:

$$48708 = 2 \cdot 3^3 \cdot 11^2 \cdot 41$$
$$275684 = 2^2 \cdot 41^3$$
$$(48708, 257684) = 2 \cdot 41 = 82$$
$$[48708, 257684] = 2^2 \cdot 3^2 \cdot 11^2 \cdot 41^3 = 163756296$$

A fejezet hátralevő részében olyan értelmezéseket és tételeket adunk meg, amelyek a prímszámok számával, eloszlásával kapcsolatosak.

6.2. értelmezés. $\pi(x)$ az x-nél nem nagyobb prímszámok számát jelöli, ahol x egy pozitív valós szám [11, 27].

Példák: A 10-nél nem nagyobb prímszámok száma 4, a 100-nál nem nagyobb prímszámok száma 25, jelölésük pedig a következő:

$$\pi(10) = 4, \pi(100) = 25.$$

A prímszámtétel a prímszámok aszimptotikus eloszlását írja le a pozitív egész számok között. A tételt Gauss[10] adta meg 1793-ban mint sejtést, bizonyítani azonban Hadamard[11] bizonyította 1896-ban. A tételt bizonyítás nélkül adjuk meg [27].

6.5. tétel. (A prímszámtétel)

$$\lim_{x \to \infty} \frac{\frac{\pi(x)}{x}}{\frac{}{\ln(x)}} = 1,$$

ahol $\ln(x)$ a természetes logaritmusfüggvény.

10 Carl Friedrich Gauss német matematikus (1777–1855)
11 Jacques Salomon Hadamard francia matematikus (1865–1963)

A tétel aszimptotikus jelöléssel a következőket jelenti:

$$\pi(x) \sim \frac{x}{\ln(x)},$$

A következő megközelítések nagy számok esetében pontosabbak:

$$\pi(x) \sim \frac{x}{\ln(x)-1}, \quad \pi(x) \sim \frac{x}{\ln(x)-1-\frac{1}{\ln(x)}}.$$

Példák:

$$\pi(10) = 4, \quad \frac{10}{\ln(10)} = 4.34, \quad \frac{10}{\ln(10)-1} = 7.67.$$

$$\pi(1000000) = 78498,$$

$$\frac{1000000}{\ln(1000000)} = 72382.41, \quad \frac{1000000}{\ln(1000000)-1} = 78030.44.$$

6.5. algoritmus. Írjunk egy Python függvényt, amely Eratosztenész szitájával, illetve a fenti megközelítéseket használva megadja a prímszámok számát x-ig.

```
from math import log
def primszT (x):
    L = eratL(x)
    k = len(L)
    lg = log(x)
    print('primszamok szama:', k)
    print('kozelitett szamitasok: ')
    temp1, temp2, temp3 = x/lg, x/(lg-1), x/(lg-1-1/lg)
    print('%10.2f%10.2f%10.2f' % (temp1, temp2, temp3))

>>> primszT(10000000)
    primszamok szama: 664579
    kozelitett szamitasok:
     620420.69 661458.97 664184.67
```

A prímszámtétel alapján kijelenthető, hogy a következő két képlet valamelyikével is meghatározható az x-dik prímszám közelített értéke:

$$x \cdot \ln(x),$$
$$x \cdot (\ln(x) + \ln(\ln(x) - 1)).$$

Példák: A 10-dik prímszám 29, és fennáll:

$$10 \cdot \ln(10) = 23.02,$$
$$10 \cdot (\ln(10) + \ln(\ln(10) - 1)) = 25.66.$$

A 10000-dik prímszám 104729, és fennáll:

$$10000 \cdot \ln(10000) = 92103.40,$$
$$10000 \cdot (\ln(1000) + \ln(\ln(10000) - 1)) = 113157.34.$$

6.1. sejtés. (Goldbach-sejtés) Minden 2-nél nagyobb páros szám felírható két prímszám összegeként [11, 33].

Goldbach[12] az 1700-as évek közepén egy Eulernek írt levélben említi meg a fenti sejtést, amelyre Euler válaszol is, amelyben az áll, hogy nem tudja a bizonyítást. A sejtést azóta sem sikerült bizonyítani, így ez a matematikatörténet egyik legrégebbi, legismertebb meg nem oldott problémája. Több változata is ismert.

Példák:

$$4 = 2 + 2, \qquad 10 = 7 + 3,$$
$$6 = 3 + 3, \qquad 12 = 7 + 5,$$
$$8 = 5 + 3, \qquad 14 = 7 + 7.$$

6.3. értelmezés. Ikerprímeknek nevezzük azokat a $(p, p + 2)$ számpárokat, ahol p és $p + 2$ is prímszám [11, 33].

Példák: (3, 5) ikerprímek, mert 3 és 5 is prímszám. (17, 19) ikerprímek, mert 17 és 19 is prímszám.

6.2. sejtés. (Ikerprím-sejtés) Végtelen sok ikerprím létezik [11, 33].

Az ikerprím-sejtés tulajdonképpen úgy is megfogalmazható, hogy a szomszédos prímek közötti különbség végtelen sokszor nagyon kicsi. A prímek között az ikerprímek ritkán fordulnak elő. Például 10^{18}-nál kevesebb mint $8 \cdot 10^{15}$ pár található. A 2016-ban felfedezett, aktuálisan legnagyobb ikerprímpárok 388342 számjeggyel rendelkeznek.

12 Christian Goldbach német matematikus (1690–1764)

6.2. Kongruenciák

A kongruenciák alapjait Gauss dolgozta ki a 19. században. Az egész számokat vizsgálta, abból a szempontból, hogy milyen maradékot ad egy szám, ha elosztjuk egy megadott m pozitív egész számmal. A maradékos osztás tétele alapján (6.1. tétel) m-mel való osztáskor minden egész számnak egy jól meghatározott maradék felel meg. Ha az a és b egész számoknak m-mel való osztásakor ugyanaz a maradék felel meg, akkor azt mondjuk, hogy a **kongruens** b-vel modulo m szerint, és ezt a tényt $a \equiv b \pmod{m}$-el jelöljük, az m-et pedig **modulusnak** hívjuk [27]. Az az állítás, hogy $a \equiv b \pmod{m}$ egyenértékű a következővel:

$$a \text{ felírható } b + k \cdot m \text{ alakba, ahol } k \text{ egész szám.}$$

Ugyanakkor a következővel is egyenértékű:

$$m \text{ osztója } a - b\text{-nek, amelyet } m \mid (a - b)\text{-vel jelölünk.}$$

Ha a és b nem ugyanazt a maradékot adják az m egész számmal való osztáskor, akkor azt mondjuk, hogy a **inkongruens** b-vel modulo m szerint, ezt a tényt $a \not\equiv b \pmod{m}$-el jelöljük.

Példák:

$$\begin{aligned}
&21 \equiv 3 \pmod{9} \iff 9 \mid (21 - 3)\\
&21 \equiv 3 \pmod{9} \iff 21 = 3 + 2 \cdot 9\\
&4 \equiv -5 \pmod{9} \iff 4 = (-5) + 1 \cdot 9\\
&14 \not\equiv 5 \pmod{12} \iff 12 \not| (14 - 5) = 9
\end{aligned}$$

A kongruenciák megjelennek a mindennapi életünkben is, például egy nap órákra való felosztása történhet $\pmod{12}$ vagy $\pmod{24}$ szerint, az évet $\pmod{7}$ szerint osztjuk hetekre stb.

Az alaptulajdonságai alapján a kongruencia **ekvivalencia** reláció, mert fennáll a következő három tulajdonság:

1. **reflexív**, mert ha a egy egész szám, akkor $a \equiv a \pmod{m}$,
2. **szimmetrikus**, mert ha a, b egész számok és $a \equiv b \pmod{m}$, akkor $b \equiv a \pmod{m}$,
3. **tranzitív**, mert ha a, b, c egész számok és $a \equiv b \pmod{m}$, $b \equiv c \pmod{m}$, akkor $a \equiv c \pmod{m}$.

Példa:

$$21 \equiv 48 \pmod 9, 48 \equiv 3 \pmod 9 \Longrightarrow 21 \equiv 3 \pmod 9,$$

fennáll:

$$... \equiv -15 \equiv \mathbf{-6} \equiv \mathbf{3} \equiv 12 \equiv 21 \equiv ... \equiv 48 \equiv ... \pmod 9.$$

Az osztási maradékok között kiemelt szerepet tölt be legkisebb pozitív, illetve a legnagyobb negatív maradék. A következő példában a reláció bal oldalán a **legkisebb pozitív maradék**, míg a jobb oldalán a **legnagyobb negatív maradék** található:

Példa:

$$\begin{aligned} 1 &\equiv -8 \pmod 9 \\ 2 &\equiv -7 \pmod 9 \\ 3 &\equiv -6 \pmod 9 \\ &... \\ 8 &\equiv -1 \pmod 9 \end{aligned}$$

Legyenek a, b, c, m egész számok, ahol $m > 0$ és $a \equiv b \pmod m$. Ekkor a kongruencia mindkét oldalához hozzáadhatjuk, mindkét oldalából kivonhatjuk ugyanazt a számot, és szorozhatunk is ugyanazzal a számmal:

1. $a + c \equiv b + c \pmod m$,
2. $a - c \equiv b - c \pmod m$,
3. $a \cdot c \equiv b \cdot c \pmod m$.

Példa: $79 \equiv 7 \pmod 9$ esetében igazak a következők is:

$$\begin{aligned} 79 + \mathbf{3} &\equiv 7 + \mathbf{3} \pmod 9, \text{ azaz } 82 \equiv 10 \pmod 9 \\ 79 - \mathbf{4} &\equiv 7 - \mathbf{4} \pmod 9, \text{ azaz } 75 \equiv 3 \pmod 9 \\ 79 \cdot \mathbf{2} &\equiv 7 \cdot \mathbf{2} \pmod 9, \text{ azaz } 158 \equiv 14 \pmod 9 \end{aligned}$$

Az osztás esetében másképp kell eljárni. Legyenek a, b, c, m egész számok, ahol $m > 0$, és $d = (c, m)$. Ekkor

$$a \cdot c \equiv b \cdot c \pmod m \Longleftrightarrow a \equiv b \pmod{m/d}.$$

Egy kongruencia mindkét oldalát tehát el szabad osztani ugyanazzal a c számmal, ha a modulus osztható d-vel, az m és c legnagyobb közös osztójával. Ekkor a modulust is el kell osztani d-vel.

Példák:

1. Legyen $14 \equiv 8 \pmod 6$.
   - $(2, 6) = 2$, tehát a kongruenciát végig tudjuk osztani 2-vel, és **osztanunk kell a modulust is 2-vel**: $7 \equiv 4 \pmod 3$.
   - A kongruenciát nem tudjuk végigosztani 2-vel úgy, hogy a modulust ne változtatnánk meg: **nem igaz** az, hogy $7 \equiv 4 \pmod 6$.
2. Legyen $50 \equiv 20 \pmod{15}$.
   - $(\mathbf{10}, 15) = \mathbf{5}$, tehát a kongruenciát végig tudjuk osztani **10**-zel, de osztanunk kell a modulust is **5**-tel: $5 \equiv 2 \pmod 3$.
   - $(\mathbf{5}, 15) = 5$, tehát a kongruenciát végig tudjuk osztani **5**-tel, de osztanunk kell a modulust is **5**-tel: $10 \equiv 4 \pmod 3$. Ez utóbbi kongruenciát tovább oszthatjuk 2-vel, ekkor a modulust nem kell osztani, mert $(2, 3) = 1$. Így megkapjuk az előző pontnál meghatározott kongruenciát: $5 \equiv 2 \pmod 3$.
3. Legyen $42 \equiv 77 \pmod 5$.
   - $(7, 5) = 1$ tehát a kongruenciát végig tudjuk osztani 7-tel, ahol a modulust nem kell osztani: $6 \equiv 11 \pmod 5$.

Legyenek a, b, c, d, m egész számok, ahol $m > 0$, $a \equiv b \pmod m$ és $c \equiv d \pmod m$. Ekkor fennállnak a következők:

1. $a + c \equiv b + d \pmod m$,
2. $a - c \equiv b - d \pmod m$,
3. $a \cdot c \equiv b \cdot d \pmod m$,
4. $a^n \equiv b^n \pmod m$, ahol n pozitív egész szám,
5. $a \equiv b \pmod m \Longrightarrow (a, m) = (b, m)$.

Példa: Ha $4 \equiv -5 \pmod 9$ és $79 \equiv 7 \pmod 9$, akkor

$$\begin{aligned}
4 + 79 &\equiv -5 + 7 \pmod 9, \text{ azaz } 83 \equiv 2 \pmod 9 \\
4 - 79 &\equiv -5 - 7 \pmod 9, \text{ azaz } -75 \equiv -12 \equiv 6 \pmod 9 \\
4 \cdot 79 &\equiv (-5) \cdot 7 \pmod 9, \text{ azaz } 316 \equiv -35 \equiv 1 \pmod 9 \\
4^6 &\equiv (-5)^6 \pmod 9, \text{ azaz } 4096 \equiv 15625 \equiv 1 \pmod 9
\end{aligned}$$

A megadott tulajdonságok alapján megállapítható, hogy a moduláris hatványozás implementálható a gyorshatványozás algoritmusa (4.3) szerint. A szorzás és négyzetre emelések során azonban meg kell határozni az

adott modulus szerinti osztási maradékot. A részszorzatok és az eredmény nagyságrendjét a modulus értéke határozza meg, az algoritmus futási ideje logaritmikus lesz [14, 22].

Példa: $3^{43} \pmod{100}$ értékének a meghatározásához meg kell határozni a következő szorzatokat:

$$3^{43} = 3^{32} \cdot 3^8 \cdot 3^2 \cdot 3^1 = 41 \cdot 61 \cdot 9 \cdot 3 = 27 \pmod{100}$$

és a következő négyzetre emeléseket:

$$\begin{aligned} 3 &\equiv 3 \pmod{100} \\ 3^2 &\equiv 9 \pmod{100} \\ 3^4 \equiv 9^2 &\equiv 81 \pmod{100} \\ 3^8 \equiv 81^2 &\equiv 61 \pmod{100} \\ 3^{16} \equiv 61^2 &\equiv 21 \pmod{100} \\ 3^{32} \equiv 21^2 &\equiv 41 \pmod{100} \end{aligned}$$

6.6. algoritmus. Írjunk egy Python függvényt, amely meghatározza $x^n \pmod{m}$-t, ahol x, n, m pozitív egész számok.

```
def modPower(x, n, m):
    res = 1
    while n != 0:
        if n % 2 == 1: res = (res * x) % m
        x = (x * x ) % m
        n = n // 2
    return res

>>> modPower(3, 43, 100)
    27
```

A Python beépített `pow` függvénye alkalmas arra, hogy moduláris hatványozást végezzünk vele. Három paraméterrel kell meghívni, ahol a harmadik paraméter a modulus értékét fogja jelölni. Hasonlóan jártunk el mi is a `modPower` függvény paraméterezésekor.

```
>>> pow(3, 43, 100)
    27
```

Habár a végeredmény ugyanaz lesz, fontos, hogy különbséget tegyünk az előző és a következő két meghívás között:

```
>>> pow(3, 43) % 100
>>> (3**43) % 100
```

A fenti két műveletsor először kiszámítja a 3^{43} hatványt, és csak utána határozza meg a 100-al való osztási maradékot. Ez nagy számok esetében igencsak megnöveli a futási időt. A `pow(3, 43, 100)` meghívás során minden részszorzat kiszámításakor meghatározásra kerül a 100-zal való osztási maradék is, ezért az algoritmus jóval kisebb nagyságrendű számokkal fog dolgozni, jóval kisebb lesz a futási idő. Fontos tehát, hogy moduláris hatványozás esetén a `pow` függvény három paraméteres változatát használjuk.

Az m szerinti osztási maradékok alapján a számokat különböző halmazokba csoportosíthatjuk. Ha egy halmazba tesszük azokat a számokat amelyek egymással kongruensek, azaz ugyanazt az osztási maradékot adják a modulus szerint, akkor m darab halmaz fog létrejönni. Ezeket a halmazokat modulo m **maradékosztályoknak** hívjuk [27, 39].

Példa: Modulo 5 szerint öt maradékosztályt különböztetünk meg:

$$\begin{aligned}
\cdots &\equiv -10 \equiv -5 \equiv \mathbf{0} \equiv 5 \equiv 10 \equiv \ldots \pmod 5\\
\cdots &\equiv -9 \equiv -4 \equiv \mathbf{1} \equiv 6 \equiv 11 \equiv \ldots \pmod 5\\
\cdots &\equiv -8 \equiv -3 \equiv \mathbf{2} \equiv 7 \equiv 12 \equiv \ldots \pmod 5\\
\cdots &\equiv -7 \equiv -2 \equiv \mathbf{3} \equiv 8 \equiv 13 \equiv \ldots \pmod 5\\
\cdots &\equiv -6 \equiv -1 \equiv \mathbf{4} \equiv 9 \equiv 14 \equiv \ldots \pmod 5
\end{aligned}$$

Egy adott maradékosztályt a legkisebb nem negatív osztási maradékkal szokták reprezentálni. A fenti példában félkövéren jelennek meg ezek a számok. a-nak $\pmod m$ szerint a legkisebb nem negatív osztási maradéka r, ha fennáll: $a = q \cdot m + r$, ahol $0 \leq r \leq m - 1$. A maradékosztályok reprezentálhatóak a legnagyobb negatív értékű maradékokkal is, a fenti példában ezek dőlten jelennek meg.

Ha minden maradékosztályból veszünk egy reprezentánst, akkor a kapott halmazt **modulo** m **teljes maradékrendszernek** hívjuk. Bármely m darab szám, amelyek inkongruensek modulo m szerint, teljes maradékrendszert alkotnak az m modulus szerint.

Példa: A következő halmazok teljes maradékrendszert alkotnak a megadott modulus szerint:

$\{0, 1, 2, ..., m-1\}$ modulo m szerint,
$\{0, 1, 2, 3, 4\}$ modulo 5 szerint,
$\{-2, -1, 0, 1, 2\}$ modulo 5 szerint,
$\{60, 21, 52, 73, 34\}$ modulo 5 szerint,
$\{0, 1, 2, \ldots, 255\}$ modulo 256 szerint,
$\{-127, \ldots, -2, -1, 0, 1, 2, \ldots, 128\}$ modulo 256 szerint.

Modulo m redukált maradékrendszert a teljes maradékrendszer azon a_i elemei alkotnak, amelyekre fennáll $(a_i, m) = 1$. Tehát a redukált maradékrendszer elemei relatív prímek a modulussal.

Példa:

$\{1, 5, 7, 11\}$ modulo 12 redukált maradékrendszert alkotnak,
$\{0, 1, 2, ..., 11\}$ modulo 12 teljes maradékrendszert alkotnak.

6.4. értelmezés. Legyen m egy pozitív egész szám, ekkor az **Euler-függvény** meghatározza azoknak a pozitív egész számoknak a számát, amelyek nem nagyobbak, mint m, és relatív prímek m-nel. Az Euler-függvényt $\phi(m)$-nel jelöljük [27, 39].

Példa: A következő táblázat megadja az Euler-függvény értékeit $m = 1, 2, \ldots, 12$-re:

m	1	2	3	4	5	6	7	8	9	10	11	12
$\phi(m)$	1	1	2	2	4	2	6	4	6	4	10	4

6.6. tétel. Ha p egy prímszám, akkor $\phi(p) = p - 1$. A tétel fordítottja is igaz, azaz ha p egy pozitív egész szám, amelyre fennáll, hogy $\phi(p) = p - 1$, akkor p prímszám.

Példa: $\phi(61) = 60$.

Egy modulo m redukált maradékrendszer elemszáma $\phi(m)$ lesz, és azoknak a maradékosztályoknak a száma is $\phi(m)$ lesz, amelyek m-hez relatív prímeket tartalmaznak. Ha megadunk $\phi(m)$ darab tetszőleges számot, amelyek páronként inkongruensek (nem kongruensek) modulo m szerint és amelyek legnagyobb közös osztója a modulussal 1, akkor ezek a számok modulo m redukált maradékrendszert alkotnak.

6.7. tétel. Ha p egy prímszám és a egy pozitív egész szám, akkor

$$\phi(p^a) = p^a - p^{a-1}.$$

Bizonyítás: A p^a-val relatív prímek számát úgy lehet meghatározni, hogy az $1, 2, \dots p^a$ számsorozatból elhagyjuk a p-vel osztható számokat, és meghatározzuk a megmaradt számok számát. A $1, 2, \dots p^a$ számsorozatból a p-vel osztható számok a következők: $p, 2 \cdot p, \dots, p^{a-1} \cdot p$, számuk pedig p^{a-1}. Tehát a megmaradt számok száma $p^a - p^{a-1}$ lesz, amiből következik a tétel állítása.

□

Példa: $\phi(256) = \phi(2^8) = 2^8 - 2^7 = 128$.

6.8. tétel. Ha a és b relatív prímek, akkor $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$.

Bizonyítás: Legyenek $r_1, r_2, \dots, r_n$ azok a számok, amelyek modulo a redukált maradékrendszert alkotnak, ekkor $n = \phi(a)$. A következő táblázat az $a \cdot b$-nél kisebb, a-hoz relatív prímeket tartalmazza:

$$\begin{array}{cccc} r_1 & r_2 & \cdots & r_n \\ a + r_1 & a + r_2 & \cdots & a + r_n \\ 2 \cdot a + r_1 & 2 \cdot a + r_2 & \cdots & 2 \cdot a + r_n \\ \vdots & \vdots & \vdots & \vdots \\ (b-1) \cdot a + r_1 & (b-1) \cdot a + r_2 & \cdots & (b-1) \cdot a + r_n \end{array}$$

A táblázat egy tetszőleges i oszlopában található $r_i, a + r_i, 2 \cdot a + r_i, \dots, (b-1) \cdot a + r_i$ számokról megállapítható, hogy ezek azokat a számokat jelentik, amelyeket úgy is meg kaphatunk, ha a $0, 1, 2, \dots, b-1$ modulo b teljes maradékrendszert alkotó számokat rendre megszorozzuk a-val, majd ehhez hozzáadunk r_i-t. Mivel a és b relatív prímek, kijelenthető, hogy a kapott számok modulo b teljes maradékrendszert alkotnak. Tudjuk azonban, hogy egy modulo b teljes maradékrendszerben azoknak a számoknak a száma, amelyek relatív prímek b-hez $\phi(b)$ lesz. Tehát minden oszlopban $\phi(b)$ darab olyan szám van, amely relatív prím b-hez. Ugyanakkor a sorokban található számok száma $\phi(a)$, tehát a táblázatban összesen $\phi(a) \cdot \phi(b)$ olyan szám található, amely mindegyike relatív prím b-hez.

Másfelől a táblázatban azokat a pozitív számokat számoltuk össze, amelyek nem nagyobbak, mint $a \cdot b$ és relatív prímek a-hoz is, és b-hez is, azaz $a \cdot b$-hez, amely értelmezés szerint $\phi(a \cdot b)$-vel egyenlő.

□

Példa: $\phi(75) = \phi(3 \cdot 25) = \phi(3) \cdot \phi(25) = 2 \cdot (5^2 - 5) = 40.$

6.9. tétel. Ha m prímtényezős felbontása: $m = p_1^{a_1} \cdot p_2^{a_2} \cdot ... \cdot p_n^{a_n}$, akkor

$$\phi(m) = m \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \cdots \cdot \left(1 - \frac{1}{p_n}\right)$$

Példa:

$$\begin{aligned} \phi(60) &= (2^2 - 2^1) \cdot (3 - 1) \cdot (5 - 1) = 16, \text{ mert } 60 = 2^2 \cdot 3 \cdot 5 \\ \phi(100) &= 100 \cdot \left(1 - \frac{1}{2}\right) \cdot \left(1 - \frac{1}{5}\right) = 40, \text{ mert } 100 = 2^2 \cdot 5^2. \end{aligned}$$

Bizonyítás: Könnyen belátható, hogy $\phi(m)$ a következő alakba is felírható:

$$\phi(m) = (p_1^{a_1} - p_1^{a_1 - 1}) \cdot (p_2^{a_2} - p_2^{a_2 - 1}) \ldots (p_n^{a_n} - p_n^{a_n - 1}).$$

A 6.8. tétel alapján, mivel a $p_1^{a_1}, p_2^{a_2}, \ldots p_n^{a_n}$ számok páronként relatív prímek, teljes indukciót alkalmazva bizonyítható, hogy

$$\phi(p_1^{a_1} \cdot p_2^{a_2} \ldots p_n^{a_n}) = \phi(p_1^{a_1}) \cdot \phi(p_2^{a_2}) \ldots \phi(p_n^{a_n}).$$

Alkalmazva ezután a 6.7. tételben megadott képletet, kapjuk a kívánt összefüggést.

□

6.7. algoritmus. Írjunk egy Python-függvényt, amely meghatározza az Euler-függvény értékét, azaz meghatározza azoknak a pozitív egész számoknak a számát, amelyek nem nagyobbak, mint `m` és relatív prímek `m`-mel.

Az ϕ értékének a meghatározásához a Python-kódban a következő összefüggést használjuk, amelyről egyszerű belátni, hogy egyenlő a 6.9. tételben megadottal.

$$\phi(m) = m \cdot \left(\frac{p_1 - 1}{p_1}\right) \cdot \left(\frac{p_2 - 1}{p_2}\right) \ldots \left(\frac{p_n - 1}{p_n}\right).$$

```
def euler(m):
    eulerF = m
    m, eulerF = szamolEuler(2, m, eulerF)
    m, eulerF = szamolEuler(3, m, eulerF)
    x = 5
    while x * x <= m:
        m, eulerF = szamolEuler(x, m, eulerF)
        m, eulerF = szamolEuler(x + 2, m, eulerF)
        x += 6
```

```
    if m > 1: eulerF = eulerF // m * (m - 1)
    return eulerF

def szamolEuler(x, m, eulerF):
    ok = 0
    while m % x == 0:
        m = m // x
        ok = 1
    if ok != 0: eulerF = eulerF // x * (x - 1)
    return m, eulerF

>>> for i in range(1, 13):
        print(euler(i), end = ' ')
```

Az `euler` függvény az Euler-függvény értékét hasonló elgondolás alapján számolja ki, mint ahogy a 6.4. algoritmus meghatározta a bemenet prímtényezős felbontást. A lényeges különbség, hogy most nem szükséges megszámolni, hogy hányszor osztható a bemenet egy adott prímszámmal.

Fontos megjegyezni, hogy nagy számok esetében nem ismert hatékony algoritmus az Euler-függvény értékének a meghatározására.

A következőkben bemutatásra kerülő kis Fermat-[13] és Euler-tétel mindegyike nagyon fontos számelméleti eredmény. Számos kriptográfiai algoritmusnak képezik az alapját. Egy adott hatvány értékére vonatkozóan tesznek egy-egy megállapítást, ahol a kis Fermat-tétel esetében a hatványozás egy prímmodulus szerint történik, az Euler-függvény esetében a modulus pedig egy tetszőleges összetett szám lesz [11, 27, 33, 39].

6.10. tétel. (A kis Fermat-tétel) Ha m egy prímszám és x egy pozitív egész szám úgy, hogy $1 \leq x \leq m-1$, akkor

$$x^{m-1} \equiv 1 \pmod{m}.$$

6.11. tétel. (Az Euler-tétel) Ha $m > 1$ és x egy pozitív egész szám úgy, hogy $(x, m) = 1$, akkor

$$x^{\phi(m)} \equiv 1 \pmod{m}.$$

Megállapítható, hogy az Euler-tétel a kis Fermat-tétel általánosítása, hiszen ha az m modulus prímszám, akkor fennáll, hogy $\phi(m) = m - 1$. Az Euler-tétel bizonyítása előtt, annak jobb megértése végett kövessük figyelmesen a következő példákat.

13 Pierre de Fermat francia jogász (1607–1665)

Példák:

1. Legyen $m = 11, x = 3$, ekkor fennáll: $3^{10} = 59049 \equiv 1 \pmod{11}$. Vegyük észre, hogy felírhatók a következők:

$$\begin{aligned} 1 \cdot x &= 1 \cdot 3 \equiv 3 \pmod{11} & 6 \cdot x &= 6 \cdot 3 \equiv 7 \pmod{11} \\ 2 \cdot x &= 2 \cdot 3 \equiv 6 \pmod{11} & 7 \cdot x &= 7 \cdot 3 \equiv 10 \pmod{11} \\ 3 \cdot x &= 3 \cdot 3 \equiv 9 \pmod{11} & 8 \cdot x &= 8 \cdot 3 \equiv 2 \pmod{11} \\ 4 \cdot x &= 4 \cdot 3 \equiv 1 \pmod{11} & 9 \cdot x &= 9 \cdot 3 \equiv 5 \pmod{11} \\ 5 \cdot x &= 5 \cdot 3 \equiv 4 \pmod{11} & 10 \cdot x &= 10 \cdot 3 \equiv 8 \pmod{11} \end{aligned}$$

2. Legyen $m = 12, x = 5$, ekkor fennáll $(5, 12) = 1, 5^{\phi(12)} = 625 \equiv 1 \pmod{12}$, ahol $\phi(12) = 4$. A 12-vel relatív prímek a következők: 1, 5, 7, 11 és felírhatók a következők:

$$\begin{aligned} 1 \cdot x &= 1 \cdot 5 \equiv 5 \pmod{12} \\ 5 \cdot x &= 5 \cdot 5 \equiv 1 \pmod{12} \\ 7 \cdot x &= 7 \cdot 5 \equiv 11 \pmod{12} \\ 11 \cdot x &= 11 \cdot 5 \equiv 7 \pmod{12} \end{aligned}$$

Bizonyítás: Az Euler-tétel bizonyításához válasszuk ki úgy az $r_1, r_2, \ldots, r_n$ számokat, hogy azok modulo m redukált maradékrendszert alkossanak, ahol $n = \phi(m)$. Most határozzuk meg a $q_1 = x \cdot r_1, q_2 = x \cdot r_2, \ldots, q_n = x \cdot r_n$ szorzatokat, ahol $(x, m) = 1$. A kapott $q_1, q_2, \ldots, q_n$ számok is modulo m redukált maradékrendszert fognak alkotni, és ugyanazokat a számokat fogják jelölni, mint az $r_1, r_2, \ldots, r_n$ számok, csak más sorrendben. Ha az $x \cdot r_1 \equiv q_1 \pmod{m}, x \cdot r_2 \equiv q_2 \pmod{m}, \ldots, x \cdot r_n \equiv q_n \pmod{m}$ kongruenciákat összeszorozzuk, akkor a következő kongruenciát kapjuk:

$$x^n \cdot r_1 \cdot r_2 \cdot \cdots \cdot r_n \equiv q_1 \cdot q_2 \cdot \cdots \cdot q_n \pmod{m}.$$

Elosztva a kongruenciát $r_1 \cdot r_2 \cdot \cdots \cdot r_n = q_1 \cdot q_2 \cdot \cdots \cdot q_n$ szorzattal kapjuk:

$$x^{\phi(m)} \equiv 1 \pmod{m}.$$

□

6.3. Lineáris kongruenciák

6.5. értelmezés. Az $a \cdot x \equiv b \pmod{m}$ típusú egyenletet lineáris kongruenciának hívjuk, ahol x ismeretlen.

Ha x_0 megoldása a fenti kongruenciának és $x_1 \equiv x_0 \pmod{m}$, akkor x_1 is megoldás [11, 27, 33, 39].

Példa: Határozzuk meg azt az x számot, amelyre fennáll: $3 \cdot x \equiv 14 \pmod{19}$. Az x értékét brute-force módszerrel a következő elgondolást alkalmazva tudjuk meghatározni: az x helyébe rendre az $1, 2, \ldots 18$ értékeket helyettesítjük, és megvizsgáljuk, mikor áll fenn az egyenlőség. A következő algoritmus eszerint jár el.

6.8. algoritmus. Írjunk egy Python függvényt, amely az a, b, m egész számok esetében, ahol $m > 0$ meghatározza azokat az x értékeket, amelyekre teljesül: $a \cdot x \equiv b \pmod{m}$.

```
def linKongruencia(a, b, m):
    db = 0
    for x in range(1, m):
        if (a * x) % m == b:
            print(x, end = ' ')
            db += 1
    print('\nMegoldásszám: ', db)

>>> linKongruencia(9, 12, 45)
    Megoldásszám:  0
>>> linKongruencia(9, 12, 15)
    3 8 13
    Megoldásszám:  3
>>> linKongruencia(3, 14, 28892719)
    9630911
    Megoldásszám:  1
```

A fenti meghívások eredményeit vizsgálva megfigyelhetjük, hogy a lineáris kongruenciának lehet egy vagy több megoldása, illetve van, amikor nem oldható meg. Az utolsó meghívás esetében pedig megfigyelhetjük azt is, hogy a meghatározáshoz szükséges idő nagyobb, mint az előző két esetben.

6.12. tétel. Legyenek a, b, m egész számok, ahol $m > 0$ és $(a, m) = d$. Ha $d \not| b$, akkor az $a \cdot x \equiv b \pmod{m}$ kongruenciának **nincs megoldása**. Ha $d | b$, akkor az $a \cdot x \equiv b \pmod{m}$ kongruenciának d **inkongruens** megoldása van $\pmod{m}$ szerint.

Megállapítható, hogy az $a \cdot x \equiv b \pmod{m}$ kongruencia ekvivalens az $a \cdot x - m \cdot y = b$ egyenlettel, amelynek x, y megoldásait a kiterjesztett euklideszi algoritmussal tudjuk meghatározni, (4.10. algoritmus). Tegyük fel, hogy ezek az értékek $\hat{x}, \hat{y}, d$, azaz fennáll, hogy: $a \cdot \hat{x} + m \cdot \hat{y} = d$. Ha $d|b$, akkor az $a \cdot x \equiv b \pmod{m}$ kongruencia megoldásai a következő értékek lesznek:

$$x_0 = \hat{x} \cdot (b/d)$$
$$x_i = x_0 + i \cdot (m/d), i = 1, 2, ..., d - 1.$$

Példa: Határozzuk meg a $9 \cdot x \equiv 12 \pmod{15}$ kongruencia megoldásait. Megállapítható, hogy $(9, 15) = 3$ és $3|12$, amiből következik, hogy a kongruenciának 3 inkongruens megoldása van $\pmod{15}$ szerint. A kiterjesztett euklideszi algoritmussal kapjuk: $9 \cdot 2 + 15 \cdot (-1) = 3$, azaz $\hat{x} = 2$, $\hat{y} = -1$. A kongruencia megoldásai pedig a következő értékek lesznek:

$$x_0 = 2 \cdot (12/3) = 8,$$
$$x_1 = 8 + 1 \cdot (15/3) \equiv 13 \pmod{15},$$
$$x_2 = 8 + 2 \cdot (15/3) \equiv 3 \pmod{15}.$$

A következőkben az $a \cdot x \equiv b \pmod{m}$ egyenletnek egy sajátos esetét tekintjük át, nevezetesen azt, amikor $b = 1$.

6.6. értelmezés. Az $a \cdot x \equiv 1 \pmod{m}$ kongruenciának a megoldását az a szám **multiplikatív inverzének** vagy egyszerűen inverzének hívjuk, ahol a, m egész számok, $m > 0$ és $(a, m) = 1$.

Példák: 4 inverze $\pmod{7}$ szerint egyenlő 2-vel, mert $4 \cdot 2 = 1 \pmod{7}$. 7 inverze $\pmod{31}$ szerint egyenlő 9-cel, mert $7 \cdot 9 = 1 \pmod{31}$.

Az a szám $\pmod{m}$ szerinti multiplikatív inverzét szokás a^{-1}-el jelölni. Értéke többféleképpen is meghatározható. A brute-force algoritmus sorra próbálja az összes lehetséges értéket, ehhez a 6.8. algoritmusban a második paraméter értékét 1-re kell állítani:

```
>>> linKongruencia(7, 1, 31)
    9
    Megoldásszám:  1
>>> linKongruencia(4, 1, 22)
    Megoldásszám:  0
```

Megállapítható, hogy az $a \cdot x \equiv 1 \pmod{m}$ kongruencia ekvivalens az $a \cdot x - m \cdot y = 1$ egyenlettel. Éppen ezért a multiplikatív inverz, ha létezik, akkor a kiterjesztett euklideszi algoritmussal is meghatározható.

6.9. algoritmus. Írjunk egy Python függvényt, amely a 4.10. algoritmust alkalmazva meghatározza a inverzét $\pmod{m}$ szerint.

```
def modInverse_(a, m):
    d, x, y = extEuclid(a, m)
    if d != 1:
        print('nincs inverz')
        return -1
    return x % m

>>> modInverse_(7, 31)
    9
```

Az első műveletsorban meghatározásra kerülő `x` és `y` kimeneti értékek lehetnek negatív számok is, ezért ha azt szeretnék, hogy a multiplikatív inverzet a legkisebb pozitív egész szám reprezentálja, akkor szükséges alkalmazni a Python `%` operátorát, ahogyan arra az utolsó műveletsorban sor is került.

Példa:

```
>>> -1 % 9
    8
```

A kiterjesztett euklideszi algoritmust, ha csak inverz számításra használjuk, akkor a 4.10. algoritmusban az `y0, y1, y` változókkal végzett műveletek elhagyhatóak, hiszen az eredmény meghatározásához nincs szükség ezekre az értékekre. A következő `modInverse` függvényt eszerint adjuk meg, amelynek így a futási ideje is jobb lesz, mint a `modInverse_`-nek.

```
def modInverse(a, m):
    x0, x1 = 1, 0
    b = m
    while True:
        q = a // b
        r = a - b * q
        if r == 0:
            if b != 1: return -1
            else: return x1 % m
        x = x0 - q * x1
        x0, x1 = x1, x
        a, b = b, r

>>> modInverse(6789783, 288927593)
    786172
```

A multiplikatív inverz az Euler-tétel alapján is meghatározható. Ha $(a, m) = 1$, akkor a multiplikatív inverz a következő:

$$a^{\phi(m)-1} \pmod{m}.$$

Nehézséget jelenthet azonban a $\phi(m)$ értékének a meghatározása, hiszen nagy m esetében erre nem ismert hatékony eljárás. Ha a modulus prímszám, akkor a multiplikatív inverz könnyen meghatározható, mert ekkor $\phi(m) = m - 1$, és a multiplikatív inverz a következő:

$$a^{m-2} \pmod{m}.$$

Példa: Az Euler-tétel segítségével határozzuk meg a $13 \cdot x \equiv 1 \pmod{36}$ kongruencia egy megoldását, azaz határozzuk meg 13 multiplikatív inverzét $\pmod{36}$ szerint.

Megállapítható, hogy $(13, 36) = 1$, tehát a kongruenciának egy megoldása van $\pmod{36}$ szerint. Meghatározzuk $\phi(36)$-t:

$$\phi(36) = \phi(2^2 \cdot 3^2) = \phi(2^2) \cdot \phi(3^2) = (2^2 - 2^1) \cdot (3^2 - 3^1) = 2 \cdot 6 = 12.$$

Tehát 13 inverze $\pmod{36}$ szerint: $13^{\phi(36)-1} = 13^{11} \equiv 25 \pmod{36}$. A Python shellben ezt könnyedén le tudjuk ellenőrizni:

```
>>> pow(13, 11, 36)
    25
>>> (13 * 25) % 36
    1
```

Pythonban az a multiplikatív inverzét $\pmod{m}$ szerint, ha az létezik a `pow` függvényt -1-es kitevővel használva tudjuk meghatározni. Általánosan ez a következő meghívást jelenti: `pow(a, -1, m)`.

```
>>> pow(13, -1, 36)
    25     #13 inverze
>>> pow(6789783, -1, 288927593)
    786172     #6789783 inverze
>>> pow(4, -1, 22)
    ...Error: base is not invertible for the given modulus
```

Az Euler-tétel segítségével a $a \cdot x \equiv b \pmod{m}$ lineáris kongruencia megoldását is meg tudjuk határozni. Először a inverzét kell meghatározni $\pmod{m}$ szerinti, majd a kapott értéket meg kell szorozni b-vel.

Példa: Határozzuk meg a $7 \cdot x \equiv 22 \pmod{31}$ lineáris kongruencia megoldását.

Megállapítható, hogy $(7, 31) = 1$, tehát a kongruenciának egy megoldása van $\pmod{31}$ szerint. Mivel 31 prímszám 7 inverze $7^{29} \equiv 9 \pmod{31}$ lesz. A megoldás meghatározásához még szoroznunk kell 22-vel: $x = 9 \cdot 22 = 198 \equiv 12 \pmod{31}$. Az eredmény 12 lesz.

6.4. A kínai maradéktétel

Ebben a fejezetben a legegyszerűbb kongruenciarendszer megoldhatóságát fogjuk megvizsgálni, amelyben egy ismeretlen lesz. A kínai Sun-tzu Suan-ching[14] matematikai kéziratban található meg az a feladat, hogy határozzuk meg azt a számot, amelynek ismert különböző modulusokkal az osztási maradéka. A kínai maradéktétel elnevezés tehát innen származik. Először feltételezni fogjuk, hogy a modulusok páronként relatív prímek, majd megvizsgáljuk az általános esetet is [7, 27, 39].

A probléma a következő feladaton keresztül könnyebben megérthető: ha egy tojásokkal teli kosárból kivesszük a tojásokat 2, 3, 4, 5, majd 6-osával, akkor rendre 1, 2, 3, 4, 5 tojás marad mindig a kosárban. Ha 7-esével vesszük ki, nem marad egy tojás sem. Hány tojás van a kosárban?

A feladat az alábbi kongruenciarendszerrel modellezhető, ahol a kérdés, hogy meghatározható-e az x, és ha igen, akkor hogyan:

$$\begin{aligned} x &\equiv 1 \pmod{2} \\ x &\equiv 2 \pmod{3} \\ x &\equiv 3 \pmod{4} \\ x &\equiv 4 \pmod{5} \\ x &\equiv 5 \pmod{6} \\ x &\equiv 0 \pmod{7} \end{aligned}$$

Először is vegyük észre, hogy a feladat az alábbi kongruenciarendszerre vezethető vissza:

$$\begin{aligned} x &\equiv 4 \pmod{5} \\ x &\equiv 11 \pmod{12} \\ x &\equiv 0 \pmod{7} \end{aligned}$$

14 Az ismeretlen kínai Sun Mester harmadik és ötödik században íródott matematikai értekezése

mert

- az $x \equiv 3 \pmod 4$ egyenlet megoldásai kielégítik az $x \equiv 1 \pmod 2$ egyenlet megoldásait,
- az $x \equiv 5 \pmod 6$ egyenlet megoldásai kielégítik az $x \equiv 2 \pmod 3$ egyenlet megoldásait,
- az $x \equiv 11 \pmod{12}$ egyenlet megoldásai kielégítik az $x \equiv 3 \pmod 4$ és $x \equiv 5 \pmod 6$ egyenletek megoldásait.

Az utóbbi kongruenciarendszer megoldását pedig a következő tétel alapján fogjuk tudni megadni:

6.13. tétel. Legyenek $m_1, m_2, ..., m_r$ pozitív egész számok, amelyek páronként relatív prímek. Ekkor az

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_r \pmod{m_r} \end{aligned} \tag{6.1}$$

kongruenciarendszer az $M = m_1 \cdot m_2 \cdot ... \cdot m_r$ modulus szerint megoldható, és a megoldása a következőképpen határozható meg:

1. minden $k \in \{1, 2, \ldots, r\}$-re
   - meghatározzuk az $M_k = M/m_k$ értékeket, amelyekre igaz az is, hogy $M_k = m_1 \cdot m_2 \cdot ... \cdot m_{k-1} \cdot m_{k+1} \cdot ... \cdot m_r$,
   - meghatározzuk az M_k értékek inverzét $\pmod{m_k}$ szerint, jelöljük ezeket M_k^{-1}-el, fennáll tehát $M_k \cdot M_k^{-1} \equiv 1 \pmod{m_k}$,
2. legyen $x_0 = a_1 \cdot M_1 \cdot M_1^{-1} + a_2 \cdot M_2 \cdot M_2^{-1} + ... + a_r \cdot M_r \cdot M_r^{-1}$,
3. a 6.1. rendszer megoldását a következő kongruencia megoldása adja:

$$x \equiv x_0 \pmod M. \tag{6.2}$$

Bizonyítás: Vegyük észre, hogy minden $k \in \{1, 2, \ldots, r\}$-re minden M_k-tól különböző M_i osztható m_k-val, tehát

$$x_0 \equiv a_k \cdot M_k \cdot M_k^{-1} \equiv a_k \pmod{m_k}.$$

Az x_0 tehát megoldása az 6.1. egyenletrendszernek. Ezek alapján a 6.1. ekvivalens a következő kongruenciarendszerrel:

$$\begin{aligned} x &\equiv x_0 \pmod{m_1}, \\ x &\equiv x_0 \pmod{m_2}, \\ &\vdots \\ x &\equiv x_0 \pmod{m_r}, \end{aligned} \tag{6.3}$$

Másfelől, ha egy kongruencia fennáll különböző modulusokra, akkor fennáll a modulusok legkisebb közös többszörösére is: igaz az, hogy $x_0 \equiv x \pmod{[m_1, m_2, \dots m_r]}$. Ez utóbbi belátható úgy, hogy ha $x \equiv x_0 \pmod{m_1}, x \equiv x_0 \pmod{m_2}, \dots, x \equiv x_0 \pmod{m_r}$, akkor az $x - x_0$ különbség osztható a modulusok mindegyikével, azaz osztható a modulusok legkisebb közös többszörösével.

Az M viszont osztója a modulusok legkisebb közös többszörösének, tehát a kongruencia fennáll $\pmod{M}$ szerint is: $x_0 \equiv x \pmod{M}$.

A 6.3. kongruenciarendszernek tehát azok és csak azok az x értékek tesznek eleget, amelyek kielégítik a 6.2. kongruenciát [39].

□

Példák:

1. A tojásos feladat esetében, ahol

   $$\begin{aligned} a_1 = 4, a_2 = 11, a_3 = 0, \\ m_1 = 5, m_2 = 12, m_3 = 7, \end{aligned}$$

   a következők a számítási eredmények:

   $$\begin{aligned} M = 420, \quad M_1 = 84, \quad M_2 = 35, \quad M_3 = 60, \\ M_1^{-1} = 4, \quad M_2^{-1} = 11, \quad M_3^{-1} = 2. \end{aligned}$$

   A megoldás: $x = 84 \cdot 4 \cdot 4 + 11 \cdot 11 \cdot 35 + 5 \cdot 60 \cdot 0 \equiv 119 \pmod{420}$.

2. Oldjuk meg a következő kongruenciarendszert:

   $$\begin{aligned} x &\equiv 6 \pmod{10} \\ x &\equiv 8 \pmod{13}. \end{aligned}$$

   A számítási eredmények a következők:

   $$\begin{aligned} M = 130, \quad M_1 = 13, \quad M_2 = 10, \\ M_1^{-1} = 7, \quad M_2^{-1} = 4. \end{aligned}$$

   A megoldás: $x = 13 \cdot 7 \cdot 6 + 10 \cdot 4 \cdot 8 \equiv 86 \pmod{130}$.

A 6.1. kongruenciarendszer megoldhatósága meghatározható abban az esetben is, ha az $m_1, m_2, \ldots, m_r$ pozitív egész számok páronként **nem** relatív prímek [27]. Ekkor az algoritmus a következő:

1. megoldjuk az első két kongruenciából alkotott rendszert:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \end{aligned} \tag{6.4}$$

   – a kiterjesztett euklideszi algoritmussal meghatározzuk azokat a $d, \hat{x}, \hat{y}$ értékeket, amelyekre fennáll, hogy: $d = m_1 \cdot \hat{x} + m_2 \cdot \hat{y}$,
   – ha d nem osztja az $a_1 - a_2$ különbséget, akkor a kongruenciarendszernek nincs megoldása, és nem folytatjuk tovább,
   – ellenkező esetben meghatározzuk:

$$a_0 = \frac{\hat{x} \cdot m_1 \cdot a_2}{d} + \frac{\hat{y} \cdot m_2 \cdot a_1}{d},$$

   – a 6.4. rendszer megoldását a következő kongruencia adja:

$$a \equiv a_0 \pmod{[m_1, m_2]},$$

2. a kapott megoldással és a harmadik egyenlettel folytatjuk:

$$\begin{aligned} x &\equiv a \pmod{[m_1, m_2]} \\ x &\equiv a_3 \pmod{m_3} \end{aligned}$$

3. a kapott megoldással és a negyedik egyenlettel folytatjuk, majd így tovább, amíg minden egyenletet fel nem dolgoztunk,
4. a 6.1. kongruenciarendszernek ha van megoldása, akkor az a fenti eljárás szerint kapott utolsó két egyenletből álló rendszer megoldása lesz.

A fenti algoritmus helyességének bizonyításához csak a 6.4. kongruenciarendszer megoldásmenetével fogunk foglalkozni, mert a 6.1. kongruenciarendszer megoldásának menete ez alapján matematikai indukcióval levezethető.

Bizonyítás: Legyen x a 6.4. kongruenciarendszer egy megoldása, ekkor létezik egy k egész szám, amelyre: $x = a_1 + k \cdot m_1$. Behelyettesítve a második kongruenciába kapjuk:

$$k \cdot m_1 \equiv (a_2 - a_1) \pmod{m_2}, \tag{6.5}$$

amely akkor és csakis akkor oldható meg, ha $d \mid a_2 - a_1$, ahol $d = (m_1, m_2)$ Ha k_0 egy megoldása a 6.5. egyenletnek, akkor a további inkongruens megoldásokat a $k_0 + i \cdot \frac{m_2}{d}$ értékek adják, ahol i egész szám (4.4). Tudva, hogy $m_1 \cdot m_2 = d \cdot [m_1, m_2]$, felírható:

$$x = a_1 + k \cdot m_1 = a_1 + k_0 \cdot m_1 + i \cdot \frac{m_1 \cdot m_2}{d} = a_1 + k_0 \cdot m_1 + i \cdot [m_1, m_2].$$

Ez alapján a 6.4. kongruenciarendszernek $\pmod{[m_1, m_2]}$ szerint egy megoldása van, amely a következő: $x \equiv a_1 + k_0 \cdot m_1 \pmod{[m_1, m_2]}$. Ez után a k_0 kiszámításához a kiterjesztett euklideszi algoritmussal (4.4) meghatározzuk azokat az $\hat{x}, \hat{y}$ értékeket, amelyre: $\hat{x} \cdot m_1 + \hat{y} \cdot m_2 = d$.

Ekkor $k_0 = \hat{x} \cdot \frac{a_2 - a_1}{d}$, a 6.4. kongruenciarendszer megoldása pedig:

$$x \equiv a_1 + \hat{x} \cdot \frac{a_2 - a_1}{d} \cdot m_1 \equiv \frac{\hat{x} \cdot m_1 \cdot a_2}{d} + \frac{\hat{y} \cdot m_2 \cdot a_1}{d} \pmod{[m_1, m_2]}.$$

□

6.10. algoritmus. Írjunk egy Python függvényt, amely meghatározza a 6.1. kongruenciarendszer megoldását, abban az esetben, ha az $m_1, m_2, \ldots, m_r$ pozitív egész számok páronként nem relatív prímek.

```
from math import gcd
def kinaiMarTetel(aL, mL):
    a, m = aL[0], mL[0]
    for i in range(1, len(aL)):
        d, x, y = extEuclid(m, mL[i])
        if abs(a - aL[i]) % d != 0:
            print("nincs megoldas")
            return -1
        else:
            lcm = lcmFg(m, mL[i])
            a = (a * mL[i] * y)//d + (aL[i] * m * x)//d
            a, m = a % lcm, lcm
    return a

def lcmFg(a, b):
    return abs(a * b) // gcd(a, b)

>>> kinaiMarTetel([8, 18, 13, 10], [9, 35, 20, 17])
    5093
```

A `kinaiMarTetel` függvényben meghívásra került a korábban megadott `extEuclid` (4.10. algoritmus). Az `lcmFg` a paraméterként megadott két pozitív egész szám legkisebb közös többszörösét határozza meg.

6.5. Másodfokú kongruenciák

Ebben a fejezetben a legegyszerűbb másodfokú kongruencia egyenletekkel kapcsolatos tulajdonságokat adjuk meg [7, 27, 39]. A tulajdonságok megadása előtt azonban vegyük észre, hogy ha négyzetre emeljük az $\{1, 2, \ldots, 10\}$ számok mindegyikét $\pmod{11}$ szerint, akkor a következőket kapjuk:

$$\begin{array}{lllll} 1^2 \equiv 1 & 5^2 \equiv 3 & 2^2 \equiv 4 & 4^2 \equiv 5 & 3^2 \equiv 9 \\ 10^2 \equiv 1 & 6^2 \equiv 3 & 9^2 \equiv 4 & 7^2 \equiv 5 & 8^2 \equiv 9. \end{array}$$

Ezek alapján megállapíthatjuk, hogy az alábbi kongruenciák mindegyike megoldható, és minden esetben két megoldás van:

$$\begin{array}{ll} x^2 \equiv 1 \pmod{11}, & \text{a megoldások: } 1, 10, \\ x^2 \equiv 3 \pmod{11}, & \text{a megoldások: } 5, 6, \\ x^2 \equiv 4 \pmod{11}, & \text{a megoldások: } 2, 9, \\ x^2 \equiv 5 \pmod{11}, & \text{a megoldások: } 4, 7, \\ x^2 \equiv 9 \pmod{11}, & \text{a megoldások: } 3, 8. \end{array}$$

Az alábbi kongruenciák pedig egyike sem oldható meg:

$$\begin{array}{ll} x^2 \equiv 2 \pmod{11}, & x^2 \equiv 8 \pmod{11}, \\ x^2 \equiv 6 \pmod{11}, & x^2 \equiv 10 \pmod{11}, \\ x^2 \equiv 7 \pmod{11}. & \end{array}$$

6.7. értelmezés. Legyen m egy pozitív egész szám, és a egy egész szám. Az $\boldsymbol{a}$ számot **kvadratikus maradéknak** vagy négyzetes maradéknak nevezzük $\pmod{m}$ szerint, ha létezik olyan x, amelyre az $x^2 \equiv \boldsymbol{a} \pmod{m}$ kongruencia megoldható, ahol $(a, m) = 1$. Ebben az esetben x-et az $\boldsymbol{a}$ négyzetgyökének hívjuk. Az $\boldsymbol{a}$ számot, ha nem kvadratikus maradék, akkor **kvadratikus nemmaradéknak** hívjuk.

Példa: $\pmod{11}$ szerint:

– az $1, 4, 3, 5, 9$ számok kvadratikus maradékok,
– a $2, 6, 7, 8, 10$ számok kvadratikus nemmaradékok.

6.14. tétel. Egy p **prímszám** esetében az $x^2 \equiv a \pmod{p}$ kongruenciának két inkongruens megoldása van.

Bizonyítás: Ha a kvadratikus maradék, akkor az $x^2 \equiv a \pmod{p}$ kongruenciának van legalább egy megoldása, legyen ez x_1. Másfelől igaz, hogy $(-x_1)^2 \equiv x_1^2 \pmod{p}$, tehát $-x_1$ is megoldása a $x^2 \equiv a \pmod{p}$ kongruenciának. $-x_1$ azonban nem lehet kongruens x_1-el, mert ha igaz lenne $x_1 \equiv -x_1 \pmod{p}$, akkor $2 \cdot x_1 \equiv 0 \pmod{p}$ is igaz lenne, ez azonban nem lehetséges, mert $(x_1, p) = (2, p) = 1$.

□

6.15. tétel. Egy p **prímszám** esetében ugyanannyi szám kvadratikus maradék, mint amennyi kvadratikus nemmaradék, számuk:

$$\frac{p-1}{2}.$$

Bizonyítás: Ahhoz, hogy meghatározzuk az összes kvadratikus maradékot, emeljük rendre négyzetre az $x \in \{1, 2, \ldots, p-1\}$ számok mindegyikét, és határozzuk meg $(\text{mod } p)$ szerint minden esetben a legkisebb pozitív osztási maradékot. Másfelől az $x^2 \equiv a \pmod{p}$ kongruenciának, ahol $(x, p) = 1$ két inkongruens megoldása van, tehát az $1, 2, \ldots p-1$ legkisebb pozitív osztási maradékok között a négyzetes maradékok száma egyenlő lesz $\frac{p-1}{2}$-vel. A fennmaradó $p - 1 - \frac{p-1}{2}$ számok kvadratikus nemmaradékok lesznek.

□

Példa: Határozzuk meg a $\frac{11-1}{2} = 5$ kitevőn az $\{1, 2, \ldots, 10\}$ számok hatványértékét $(\text{mod } 11)$ szerint:

$$\begin{aligned} 1^{\mathbf{5}} &\equiv 1 \pmod{11} & 2^5 &\equiv 10 \equiv -1 \pmod{11} \\ 3^{\mathbf{5}} &\equiv 1 \pmod{11} & 6^5 &\equiv 10 \equiv -1 \pmod{11} \\ 4^{\mathbf{5}} &\equiv 1 \pmod{11} & 7^5 &\equiv 10 \equiv -1 \pmod{11} \\ 5^{\mathbf{5}} &\equiv 1 \pmod{11} & 8^5 &\equiv 10 \equiv -1 \pmod{11} \\ 9^{\mathbf{5}} &\equiv 1 \pmod{11} & 10^5 &\equiv 10 \equiv -1 \pmod{11} \end{aligned}$$

6.16. tétel. Egy p **prímszám** esetében egy a szám, ahol $(a, p) = 1$

1. akkor és csakis akkor **kvadratikus maradék** $(\text{mod } p)$ szerint, ha $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$,

2. akkor és csakis akkor **kvadratikus nemmaradék** $(\text{mod } p)$ szerint, ha $a^{\frac{p-1}{2}} \equiv p-1 \equiv -1 \pmod{p}$.

6.8. értelmezés. Egy p prímszám esetében a **Legendre-szimbólum** jelzi, hogy az a szám kvadratikus maradék vagy sem $(\text{mod } p)$ szerint. A Legendre-szimbólumot $\left(\frac{a}{p}\right)$-vel jelöljük, ahol:

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} = \begin{cases} 0, & \text{ha } a \equiv 0 \pmod{p}, \\ 1, & \text{ha } a \text{ kvadratikus maradék} \\ -1, & \text{ha } a \text{ kvadratikus nemmaradék} \end{cases}$$

6.9. értelmezés. Egy $m = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \dots p_n^{\alpha_n}$ **összetett** szám esetében $\left(\frac{a}{m}\right)$-mel jelöljük a **Jacobi-szimbólum**, ahol:

$$\left(\frac{a}{m}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdot \left(\frac{a}{p_2}\right)^{\alpha_2} \dots \left(\frac{a}{p_n}\right)^{\alpha_n}.$$

Ha $m = p \cdot q$, akkor:

$$\left(\frac{a}{m}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{a}{q}\right).$$

A Jacobi-szimbólum értéke alapján **nem lehet** következtetést levonni arra vonatkozóan, hogy egy a szám kvadratikus maradék-e $(\text{mod } m)$ szerint vagy sem. Ha m prímszám, akkor a Jacobi-szimbólum ugyanaz, mint a Legendre-szimbólum.

6.17. tétel. Egy $m = p \cdot q$ **összetett szám** esetében

1. egy a szám akkor és csakis akkor **kvadratikus maradék** $(\text{mod } m)$ szerint, ha kvadratikus maradék $(\text{mod } p)$ és kvadratikus maradék $(\text{mod } q)$ szerint is, fennáll:

$$\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = \left(\frac{a}{m}\right) = 1,$$

2. ha egy a szám kvadratikus maradék, akkor az $x^2 \equiv a \pmod{m}$ kongruenciának négy inkongruens megoldása van,
3. a kvadratikus maradékok száma: $\dfrac{(p-1)\cdot(q-1)}{4}$,

4. egy a szám akkor és csakis akkor **kvadratikus nemmaradék** (mod m) szerint, ha

$$\left(\frac{a}{p}\right) = -1 \quad \text{és} \quad \left(\frac{a}{q}\right) = -1, \quad \text{ekkor} \quad \left(\frac{a}{m}\right) = 1, \text{ vagy}$$
$$\left(\frac{a}{p}\right) = -1 \quad \text{és} \quad \left(\frac{a}{q}\right) = 1, \quad \text{ekkor} \quad \left(\frac{a}{m}\right) = -1, \text{ vagy}$$
$$\left(\frac{a}{p}\right) = 1 \quad \text{és} \quad \left(\frac{a}{q}\right) = -1, \quad \text{ekkor} \quad \left(\frac{a}{m}\right) = -1.$$

Általánosan csak abban esetben állapítható meg egy a számról, hogy kvadratikus maradék (mod m) szerint, ha az m prímtényezős felbontása alapján meghatározzuk a prímtényezők szerinti Legendre-szimbólumok értékét. Ha a Legendre-szimbólumok **mindegyike** 1 lesz, akkor az a szám kvadratikus maradék lesz (mod m) szerint, minden más esetben a kvadratikus nemmaradék lesz.

6.10. értelmezés. A **kvadratikus maradék probléma** azt jelenti, hogy egy m pozitív összetett szám esetében állapítsuk meg egy a egész számról hogy az kvadratikus maradék vagy sem, azzal a feltételezéssel élve, hogy nem ismertek az m prímosztói, illetve fennáll, hogy $\left(\frac{a}{m}\right) = 1$.

Nagy számok esetében a kvadratikus maradék problémára nem ismert hatékony algoritmus, ezt hívjuk **kvadratikus maradék feltételezésnek**. Biztonsággal ez jelenleg minimum 1024 bites összetett számok esetében jelenthető ki.

A Jacobi-szimbólum meghatározására azonban létezik hatékony algoritmus nagy számok esetében is. Meghatározása, a kvadratikus reciprocitás tétel alapján lehetséges. A tételnek több formája és bizonyítása is ismert. A kvadratikus reciprocitás tételt 1783-ben Euler fogalmazta meg, és egy téves bizonyítás Legendre-től is fenn maradt. Gauss is megfogalmazta a tételt 1795-ben, és az első helyes bizonyítás is tőle származik, amelyet 1796-ban publikált. Élete nagy felfedezésének tartotta.

6.18. tétel. (Kvadratikus reciprocitás tétel) Ha p, q páratlan prímszámok, akkor fennáll:

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)\cdot(q-1)}{4}}.$$

A 6.18. tételt és következő tulajdonságokat bizonyítás nélkül adjuk meg, ahol m egy pozitív egész szám és a egy egész szám.

6.1. tulajdonság.

$$\left(\frac{0}{m}\right) = \begin{cases} 1, \text{ ha } m = 1, \\ 0, \text{ ha } m \neq 1, \end{cases} \qquad \left(\frac{2}{m}\right) = \begin{cases} 1, \text{ ha } m \equiv 1 \pmod 8 \\ 1, \text{ ha } m \equiv 7 \pmod 8 \\ -1, \text{ ha } m \equiv 3 \pmod 8 \\ -1, \text{ ha } m \equiv 5 \pmod 8 \end{cases}$$

6.2. tulajdonság. (redukció)

$$\left(\frac{a}{m}\right) = \left(\frac{a \ mod \ m}{m}\right), \text{ ha } a \geq m.$$

6.3. tulajdonság. (multiplikativitás)

$$\left(\frac{a \cdot b}{m}\right) = \left(\frac{a}{m}\right) \cdot \left(\frac{b}{m}\right).$$

6.4. tulajdonság. (leválasztás)

$$\left(\frac{a}{m}\right) = \left(\frac{2}{m}\right) \cdot \left(\frac{a \ / \ 2}{m}\right), \text{ ha } a \equiv 0 \pmod 2.$$

6.5. tulajdonság. (reciprocitás)

$$\left(\frac{a}{m}\right) = \begin{cases} -1 \cdot \left(\frac{m}{a}\right), \text{ ha } a \equiv m \equiv 3 \pmod 4 \\ \left(\frac{m}{a}\right), \text{ másképp.} \end{cases}$$

Példa: Határozzuk meg $a = 41, m = 77$ esetében a Jacobi-szimbólumot.

$$
\begin{aligned}
&(6.5): \quad 77 \equiv 1 \pmod 4, 41 \equiv 1 \pmod 4 \Rightarrow \left(\frac{41}{77}\right) = \left(\frac{77}{41}\right), \\
&(6.2): \quad \left(\frac{77}{41}\right) = \left(\frac{77 \pmod{41}}{41}\right) = \left(\frac{36}{41}\right), \\
&(6.3): \quad \left(\frac{36}{41}\right) = \left(\frac{4 \cdot 9}{41}\right), \\
&(6.4): \quad 41 \pmod 8 = 1 \Rightarrow \left(\frac{2}{41}\right)^2 = 1 \cdot 1 = 1, \\
&(6.5): \quad \left(\frac{9}{41}\right) = \left(\frac{41}{9}\right), \\
&(6.2): \quad \left(\frac{41}{9}\right) = \left(\frac{41 \pmod 9}{9}\right) = \left(\frac{5}{9}\right), \\
&(6.5): \quad \left(\frac{5}{9}\right) = \left(\frac{9}{5}\right), \\
&(6.2): \quad \left(\frac{9}{5}\right) = \left(\frac{9 \pmod 5}{5}\right) = \left(\frac{4}{5}\right), \\
&(6.4): \quad 5 \pmod 8 = 5 \Rightarrow \left(\frac{2}{5}\right)^2 = (-1) \cdot (-1) = 1. \\
&\text{Tehát:} \quad \left(\frac{41}{77}\right) = 1.
\end{aligned}
$$

Másfelől tudjuk 77 prímtényezős felbontását, ezért a következőképpen is meghatározható a Jacobi-szimbólum értéke:

$$
\begin{aligned}
&\left(\frac{41}{77}\right) = \left(\frac{41}{7}\right) \cdot \left(\frac{41}{11}\right) = \left(\frac{6}{7}\right) \cdot \left(\frac{8}{11}\right) = (-1) \cdot (-1) = 1. \\
&\left(\frac{6}{7}\right) = \left(\frac{2}{7}\right) \cdot \left(\frac{3}{7}\right) = 1 \cdot (-1) = -1 \\
&\left(\frac{8}{11}\right) = \left(\frac{2}{11}\right)^3 = (-1)^3 = -1.
\end{aligned}
$$

Ugyanakkor az is megállapítható, hogy 41 kvadratikus nemmaradék (mod 77) szerint, mert 41 kvadratikus nemmaradék (mod 7) és (mod 11) szerint is:

$$\left(\frac{41}{7}\right) = -1, \quad \left(\frac{41}{11}\right) = -1.$$

Példa: Határozzuk meg $a = 23, m = 77$ esetében a Jacobi-szimbólumot.

$$(6.5): \quad 77 \equiv 1 \pmod 4, 23 \equiv 3 \pmod 4 \Longrightarrow \left(\frac{23}{77}\right) = \left(\frac{77}{23}\right),$$

$$(6.2): \quad \left(\frac{77}{23}\right) = \left(\frac{77 \pmod{23}}{23}\right) = \left(\frac{8}{23}\right),$$

$$(6.3): \quad \left(\frac{8}{23}\right) = \left(\frac{2}{23}\right)^3,$$

$$(6.4): \quad 23 \pmod 8 = 7 \Longrightarrow \left(\frac{2}{23}\right) = 1.$$

Tehát $\left(\frac{23}{77}\right) = 1.$

Másfelől tudjuk 77 prímtényezős felbontását, ezért a következőképpen is meghatározható a Jacobi-szimbólum:

$$\left(\frac{23}{77}\right) = \left(\frac{23}{7}\right) \cdot \left(\frac{23}{11}\right) = \left(\frac{2}{7}\right) \cdot \left(\frac{1}{11}\right) = 1.$$

Ugyanakkor az is megállapítható, hogy 23 kvadratikus maradék $\pmod{77}$ szerint, mert 23 kvadratikus maradék $\pmod 7$ és $\pmod{11}$ szerint is:

$$\left(\frac{23}{7}\right) = \left(\frac{2}{7}\right) = 1$$
$$\left(\frac{23}{11}\right) = \left(\frac{1}{11}\right) = 1.$$

6.11. algoritmus. Írjunk egy Python függvényt, amely meghatározza az $\left(\frac{a}{m}\right)$ Jacobi-szimbólumot.

```
def Jacobi(a, m):
    while True:
        if a == 0: return 0
        if a == 1: return 1
        e, r = 0, a
        while r & 1 == 0:
            e, r = e + 1, r >> 1
        #print('a, m, e, r:', a, m, e, r)
        if e & 1 == 0: s = 1
        else:
```

```
            temp = m % 8
            if temp == 1 or temp == 7: s = 1
            elif temp == 3 or temp == 5: s = -1
        if m % 4 == 3 and r % 4 == 3: s = -s
        m = m % r
        if r == 1: return s
        else: return s * Jacobi(m, r)

>>> Jacobi(41, 77)
    1
```

Ahhoz, hogy a függvény által kezelt `a, m, e, r` értékek kikerüljenek a képernyőre, távolítsuk el a `print`-et tartalmazó sor elejéről a `#`-t.

6.19. tétel. Egy p **prímszám** esetében, ahol $p \equiv 3 \pmod 4$, az $x^2 \equiv a \pmod p$ kongruencia két megoldása, azaz a **két négyzetgyök** a következő:

$$x_1 \equiv a^{(p+1)/4} \pmod p, \qquad x_2 = p - x_1.$$

Bizonyítás: Mivel a kvadratikus maradék fennáll, hogy $a^{(p-1)/2} \equiv 1 \pmod p$. Másfelől $x_1^2 = a^{(p+1)/2} = a \cdot a^{(p-1)/2} \equiv a \cdot 1 \equiv a \pmod p$. Másrészt $x_2 = p - x_1 \equiv -x_1 \pmod p$ és $(-x_1)^2 = x_1^2 \pmod p$.

□

Példa: Az $x^2 \equiv 5 \pmod{11}$ kongruenciának két megoldása van:

$$x_1 \equiv 5^{(11+1)/4} \equiv 5^3 \equiv 4 \pmod{11}, \qquad x_2 = 11 - 4 = 7.$$

Ellenőrzésképpen számoljuk ki a $4^2 \pmod{11}$ és $7^2 \pmod{11}$ értékeket.

6.20. tétel. Egy m **összetett szám** esetében, ahol $m = p \cdot q$ és $p \equiv q \equiv 3 \pmod 4$, az $x^2 \equiv a \pmod m$ kongruencia négy megoldását, azaz a négyzetgyököket a következőképpen határozzuk meg:

1. kiszámítjuk p^{-1}-et p inverzét $\pmod q$ szerint, és q^{-1}-et q inverzét $\pmod p$ szerint,
2. kiszámítjuk a következő hatványértékeket:

$$x_p \equiv a^{(p+1)/4} \pmod p, \; x_q \equiv a^{(q+1)/4} \pmod q,$$

3. a négy megoldást az x_1, x_2, x_3, x_4 értékek jelentik, ahol:

$$x_1 \equiv (x_p \cdot q \cdot q^{-1} + x_q \cdot p \cdot p^{-1}) \pmod m, \qquad x_3 = m - x_1,$$
$$x_2 \equiv (x_p \cdot q \cdot q^{-1} - x_q \cdot p \cdot p^{-1}) \pmod m, \qquad x_4 = m - x_2.$$

Bizonyítás: A 6.19. tétel alapján az $x^2 \equiv a \pmod{p}$ egyenletnek két inkongruens megoldása van: $x_p \equiv a^{(p+1)/4} \pmod{p}$ és $p - x_p$. Hasonlóan a $x^2 \equiv a \pmod{q}$-nak is két megoldása van: $x_q \equiv a^{(q+1)/4} \pmod{q}$ és $q - x_q$.

Másodsorban ha a kvadratikus maradék $\pmod{p}$ szerint és kvadratikus maradék $\pmod{q}$ szerint is, akkor kvadratikus maradék lesz $\pmod{m}$ szerint is.

Harmadsorban $(p, q) = 1$, ezért az $x^2 \equiv a \pmod{m}$ négy inkongruens megoldását a következő négy kongruenciarendszer megoldása adja, amelyeket a kínai maradéktételt (6.13) alkalmazva tudunk meghatározni:

$$(a)\quad \begin{aligned} x &\equiv x_p \pmod{p} \\ x &\equiv x_q \pmod{q} \end{aligned} \qquad (c)\quad \begin{aligned} x &\equiv p - x_p \pmod{p} \\ x &\equiv x_q \pmod{q} \end{aligned}$$

$$(b)\quad \begin{aligned} x &\equiv x_p \pmod{p} \\ x &\equiv q - x_q \pmod{q} \end{aligned} \qquad (d)\quad \begin{aligned} x &\equiv p - x_p \pmod{p} \\ x &\equiv q - x_q \pmod{q} \end{aligned}$$

Ha x_1-gyel, és x_2-vel jelöljük az $(a), (b)$ kongruenciarendszerek megoldásait, akkor egyszerű észrevenni, hogy a $(c), (d)$ rendszerek megoldásai $m - x_2$, illetve $m - x_1$ lesznek.

□

Példa: Az $x^2 = 17 \pmod{2773}$ kongruencia négy megoldását, ahol $m = 2773 = 47 \cdot 59$, $p = 47 \equiv 3 \pmod{4}$ és $q = 59 \equiv 3 \pmod{4}$ a következőképpen határozzuk meg:

$$\begin{aligned}
&p^{-1} \equiv 54 \pmod{59}, \text{ mert fennáll: } 47 \cdot 54 \equiv 1 \pmod{59}, \\
&q^{-1} \equiv 4 \pmod{47}, \text{ mert fennáll: } 59 \cdot 4 \equiv 1 \pmod{47}, \\
&x_p = 17^{(p+1)/4} = 17^{12} \equiv 8 \pmod{47}, \\
&x_q = 17^{(q+1)/4} = 17^{15} \equiv 28 \pmod{59}, \\
&x_1 = 8 \cdot 59 \cdot 4 + 28 \cdot 47 \cdot 54 \equiv 854 \pmod{2773}, \\
&x_2 = 8 \cdot 59 \cdot 4 - 28 \cdot 47 \cdot 54 \equiv 149 \pmod{2773}, \\
&x_3 = 2773 - 854 = 1919, \\
&x_4 = 2773 - 149 = 2624.
\end{aligned}$$

6.6. A Miller–Rabin-prímteszt

A Miller–Rabin-prímtesztet 1980-ban publikálta Rabin [25]. Miller egy korábban megadott algoritmusát módosította olyan módon, hogy az a gyakorlatban is alkalmas lett annak a megállapítására, hogy egy több száz számjegyből álló páratlan szám prím-e vagy sem. A prímtesztet azok után szokták alkalmazni, hogy előzetesen megállapítják, hogy a tesztelendő számnak nincsenek kis prímosztói, ahol a prímszámok listáját Eratosztenész szitájával állítják elő.

A bemutatásra kerülő algoritmus biztosan megállapítja a bemenetről, hogy összetett szám, azonban hogy a bemenete prímszám, csak nagy valószínűséggel állítja. Az ilyen típusú algoritmusokat valószínűségi prímteszteknek hívják. Az algoritmus ötlete azon alapszik, hogy t darab véletlenszerűen generált szám esetében megvizsgálunk egy adott feltételt. Ha ez a feltétel mind a t esetben teljesül, akkor az algoritmus kimenete `True` lesz, azaz elfogadjuk, hogy a szám prím. Az első olyan esetben azonban, amikor a feltétel nem teljesül, akkor az algoritmus kimenete `False` lesz.

Az algoritmus kiindulópontját a kis Fermat- és Euler-tételek képezik. A kis Fermat-tétel alapján ha egy m számra **nem teljesül** a $x^{m-1} \equiv 1 \pmod{m}$ feltétel, akkor egyértelműen megállapítható, hogy az m összetett. Ha azonban **teljesül** a feltétel, akkor további x értékekre kell megvizsgálni a hatványértéket, mert nem lehet egyértelműen kijelenteni, hogy m prímszám. Vannak ugyanis olyan m összetett számok, amelyekre létezik olyan x szám, hogy $1 \leq x \leq m-1$ és $x^{m-1} \equiv 1 \pmod{m}$. Ezeket az m számokat ***x* alapú Fermat-féle álprímeknek** hívjuk [28].

Példa: Az $m = 341$-es szám 2-es alapú Fermat-féle álprím, mert fennáll: $2^{340} \equiv 1 \pmod{341}$, ugyanakkor 341 összetett szám, mert $341 = 11 \cdot 31$.

Vannak olyan olyan összetett számok, amelyek esetében minden x-re, ahol $(x, m) = 1$ és $1 \leq x \leq m-1$ fennáll, $x^{m-1} \equiv 1 \pmod{m}$. Az ilyen tulajdonságú számokat **Carmichael-számoknak** hívjuk. Bizonyítani lehet, hogy a Carmichael-számok száma végtelen [32].

Példa: A legkisebb Carmichael-szám: $561 = 3 \cdot 11 \cdot 17$.

A kis Fermat-tétel a Carmichael-számok miatt nem alkalmazható prímtesztelő algoritmusban, nagy lesz a tévedési aránya, ahogyan azt a következő algoritmus futtatásakor is észre lehet venni.

6.12. algoritmus. Írjunk egy Python függvényt, amely a kis Fermat-tétel alapján végzi a prímtesztelést.

Az algoritmus bemenete **m, egy páratlan** szám és egy **t** biztonsági paraméter. A t értékében azt adjuk meg, hogy hány véletlenszerűen generált **x** esetében vizsgáljuk meg, hogy fennáll-e a kis Fermat-tétel. A kimenet akkor lesz True, ha az összes x-re fennáll a kis Fermat-tétel. Az első olyan x-re, amelyre nem igaz a kis Fermat-tétel, az algoritmus False-t határoz meg.

```
from math import gcd
from random import randint
def fermatT(m, t):
    for i in range(0, t):
        x = randint(2, m - 1)
        if gcd(x, m) != 1: return False
        y = pow(x, m - 1, m)
        if y != 1: return False
    return True
```

Példa: $1615681 = 23 \cdot 199 \cdot 353$ Carmichael-szám. A fenti algoritmus tíz alkalommal való futtatása legalább annyiszor állítja a számról, hogy prím, mint ahányszor hogy összetett:

```
>>> for i in range(10):
        print(fermatT(1615681, 10))

    False
    True
    False
    ...
```

Ezek után rátérhetünk az Euler-kritérium bemutatására, amelyet a Miller–Rabin-prímtesztnél fogunk használni.

6.21. tétel. (Euler-kritérium) Ha az m prímszám, akkor minden olyan x szám esetében, ahol $(x,\ m) = 1$, és $m - 1 = 2^s \cdot r$, ahol r páratlan szám, fennáll a következő két kritérium közül valamelyik

$$\begin{aligned} & & x^r & \equiv 1 \pmod{m}, \\ \text{létezik olyan } j,\ 0 \le j \le s-1: & & x^{2^j \cdot r} & \equiv m-1 \pmod{m}. \end{aligned}$$

A tétel bizonyítása előtt két példát adunk:

Példák:

1. Legyen $x = 13$ és $m = 561$. Fennáll, hogy $(13, 561) = 1$, és $560 = 35 \cdot 2^4$. A hatványértékek pedig a következők lesznek:

$$13^{35} \equiv 208 \pmod{561},$$
$$13^{70} \equiv 208^2 \equiv 67 \pmod{561},$$
$$13^{140} \equiv 67^2 \equiv 1 \pmod{561},$$
$$13^{280} \equiv 1^2 \equiv 1 \pmod{561}.$$

   A hatványértékek alapján biztosan kijelenthető, hogy 561 nem prímszám, ugyanis egyik hatványérték esetében sem teljesül a 6.21. tételben megadott két kritérium közül valamelyik. Tulajdonképpen a harmadik hatványérték után már kijelenthető, hogy a szám biztosan összetett, mert az 1-es érték négyzetre emelésekor nem kaphatunk $m - 1 \equiv -1 \pmod{m}$ eredményt.
2. Legyen $x = 42$ és $m = 101$. Fennáll, hogy $(42, 101) = 1$, és $101 = 25 \cdot 2^2$. A hatványértékek pedig a következők:

$$42^{25} \equiv 91 \pmod{101},$$
$$42^{50} \equiv 91^2 \equiv 100 \pmod{101}.$$

   Az utolsó hatványérték meghatározása után megállapítható, hogy 101 valószínűleg prímszám.

Bizonyítás: Könnyen belátható, hogy bármely pozitív $m \geq 3$ páratlan egész szám felírható $m-1 = 2^s \cdot r$ alakba, ahol r páratlan szám. $m-1$ ugyanis páros szám, a 2-vel osztást pedig addig ismételjük, amíg az r nem lesz páratlan.

A kis Fermat-tétel alapján, ha az m prímszám, akkor fennáll $x^{m-1} \equiv 1 \pmod{m}$. Megfigyelhető, hogy az $x^{2^j \cdot r}, 0 \leq j \leq s-1$ számsorozat minden eleme, kezdve a második elemtől, egyenlő az előző szám négyzetével:

$$x^r, x^{2 \cdot r} = (x^r)^2, x^{2^2 \cdot r} = (x^{2 \cdot r})^2, \ldots x^{2^{s-1} \cdot r}, x^{2^s \cdot r} = (x^{2^{s-1} \cdot r})^2$$

Mivel a számsorozat utolsó eleme a kis Fermat-tétel alapján egyenlő 1-gyel, két eset lehetséges:

1. a számsorozat első eleme kongruens 1-gyel, azaz $x^r \equiv 1 \pmod{m}$,
2. a számsorozat első elemei nem lesznek kongruensek 1 modulo m-mel, de egy adott elemtől kezdődően azonban kongruensek lesznek 1 modulo m-el. Ez azt jelenti, hogy van egy olyan b szám a sorozatban, amelyre fennáll:

$$b \not\equiv 1 \pmod{m} \text{ és } b^2 \equiv 1 \pmod{m}.$$

Mivel m **prím**, ez a két kongruencia csak úgy állhat elő, ha $b \equiv (m-1) \equiv -1 \pmod{m}$, azaz

létezik $j, 0 \leq j \leq s-1$ úgy, hogy $b = x^{2^j \cdot r} \equiv m-1 \equiv -1 \pmod{m}$.

□

Megállapítható, hogy a tétel fordítottja nem igaz, mert vannak olyan m összetett számok is, melyekre létezik olyan x: $1 \leq x \leq m-1$, amelyre fennáll a 6.21. tételben megfogalmazott két kritérium közül valamelyik.

6.13. algoritmus. Írjunk egy Python függvényt, amely adott x, m pozitív egész számok esetében meghatározza az $x^{2^j r} \pmod{m}, 0 \leq j \leq s-1$ sorozat elemeit, ahol fennáll, hogy: $m - 1 = 2^s \cdot r$

```
def eulerKrit(x, m):
    r, s = m - 1, 0
    while r & 1 == 0:
        r = r >> 1
        s += 1
    print('r:', r, ', s: ', s)
    y = pow(x, r, m)
    for i in range(s):
        print(x, '**', r*2**i, '=',y)
        y = (y * y) % m
```

Ha megvizsgáljuk a `eulerKrit(103, 561)` meghívás eredményét, azt állapíthatjuk meg, hogy a 561 valószínűleg prím, mert már az első hatványérték esetében teljesül az egyik kritérium:

```
>>> eulerKrit(103, 561)
    r: 35 , s:  4
    103**35 = 1
    103**70 = 1
    103**140 = 1
    103**280 = 1
```

Ha azonban az `eulerKrit(107, 561)` meghívás eredményét vesszük figyelembe, akkor biztosan kijelenthetjük, hogy 561 összetett szám:

```
>>> eulerKrit(107, 561)
    r: 35 , s:  4
    107**35 = 329
    107**70 = 529
    107**140 = 463
    107**280 = 67
```

A 6.12. algoritmusnál alkalmazott prímteszttel szemben a 6.21. tétel a gyakorlatban mégis **sikeresen** alkalmazható, mert az m összetett szám esetében maximum $\phi(m)/4$ az olyan x számok száma, amelyek eleget tesznek a 6.21. tételben megfogalmazott két kritérium valamelyikének [32].

Példa: $m = 91$ esetében azon x-ek száma, amelyek szerint 91 úgy viselkedik, mint egy prímszám egyenlő $\phi(91)/4 = 18$, ezek pedig a következők: $\{1, 9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82, 90\}$.

Példa: $m = 561$ esetében azon x-ek száma, amelyek szerint 561 úgyf viselkedik, mint egy prímszám, kisebb mint $\phi(561)/4 = 80$, ezek pedig a következők: $\{1, 50, 101, 103, 256, 305, 458, 460, 511, 560\}$.

Megállapítható az is, hogy annak a valószínűsége, hogy egy összetett m szám t darab különböző x esetében megfeleljen a 6.21 tételben megfogalmazott két kritériumnak, kisebb mint $(1/4)^t$ [32]. A Miller–Rabin-algoritmus esetében tehát szükséges egy $t \geq 1$ biztonsági paramétert választani, amely meghatározza, hogy hány különböző random x-re vizsgáljuk meg az Euler-kritériumot. A gyakorlatban elég, ha $\boldsymbol{t = 10}$ különböző x értékre tesztelünk, mert ebben az esetben a **tévedés valószínűsége** $\boldsymbol{(1/4)^{10} \sim 10^{-6}}$, amely egy elfogadható értéknek számít.

6.14. algoritmus. Írjunk egy Python függvényt, amely a 6.21. tétel alapján végzi a prímtesztelést, azaz adjuk meg a **Miller–Rabin-algoritmust**.

```
from random import randint
def millerRabinT(m, t = 10):
    s, r = 0, m - 1
    while r % 2 == 0:
        s, r = s + 1, r // 2
    for i in range(0, t):
        x = randint(2, m - 1)
        y = pow(x, r, m)
        if y == 1 or y == (m - 1): continue
        for j in range(1, s):
            y = (y * y) % m
            if y == 1: return False
```

```
            if y == m - 1: break
        if y != (m - 1): return False
    return True
```

Az algoritmusnak két bemenete lesz, **m egy páratlan** szám és **t** egy biztonsági paraméter. Az algoritmus kimenete **`True`** lesz, ha teljesül az Euler-kritérium t különböző x-re, ami azt fogja jelenteni, hogy $(1/4)^t$ tévedési arány mellett az `m` prím. Az algoritmus kimenete **`False`** lesz, ha az m biztosan összetett.

Példa: Legyen $m = 61$ és $t = 4$, ekkor $\boldsymbol{m - 1 = 60 = 2^2 \cdot 15}$. A következő táblázat 4 különböző x-re vizsgálja, hogy fennáll-e a két kritérium közül valamelyik:

x	j	
3	0	$3^{\mathbf{15}} \equiv \mathbf{60} \pmod{61}$
7	0	$7^{15} \equiv 11 \pmod{61}$
	1	$7^{\mathbf{2\cdot 15}} \equiv \mathbf{60} \pmod{61}$
42	0	$42^{\mathbf{15}} \equiv \mathbf{1} \pmod{61}$
24	0	$24^{15} \equiv 11 \pmod{61}$
	1	$24^{\mathbf{2\cdot 15}} \equiv \mathbf{60} \pmod{61}$

Mind a 4 véletlenszerűen választott x szám esetében fennáll a kritérium, megállapítható, hogy a szám $(1/4)^4$ tévedési arány mellett prím.

Példa: Legyen $m = 91$ és $t = 4$, ekkor $\boldsymbol{m - 1 = 90 = 2 \cdot 45}$.

x	j	
16	0	$16^{45} \equiv \mathbf{1} \pmod{91}$
75	0	$75^{45} \equiv \mathbf{90} \pmod{91}$
13	0	$13^{45} \equiv \mathbf{13} \pmod{91}$

Nem kell $\mathbf{t = 4}$ tesztet elvégezni, mert a harmadik $\mathbf{x = 13}$ érték esetében nem áll fenn egyik kritérium sem, tehát megállapítható, hogy a szám biztosan összetett.

6.15. algoritmus. Írjunk egy Python függvényt, amely a Miller–Rabin-algoritmus segítségével generál egy $\boldsymbol{k}$ bites prímszámot.

```
from random import getrandbits
def primeGen(k):
    while True:
```

```
        nr = getrandbits(k)
        if not (nr & 1): nr += 1
        if millerRabinT(nr): break
    return nr

>>> primeGen(1024)
    ...
```

A `nr` a `getrandbits` meghívása után egy `k` bites számot fog jelölni, ha ez páros, akkor eggyel növeljük az értékét, hogy páratlan számot kapjunk, mert csak ezután hívhatjuk meg a `millerRabinT` függvényt.

6.16. algoritmus. Írjunk egy Python függvényt, amely a Miller–Rabin-algoritmus segítségével generál egy k bites prímszámot, amelynek Base64-es alakját kiírja egy állományba.

```
from base64 import b64encode
def primToB64(k, fname = 'primeB64.txt'):
    nr = primeGen(k)
    bajtLen = k // 8 + 1
    m_str = nr.to_bytes(bajtLen, byteorder = 'big')
    b64str = b64encode(m_str)
    out = open(fname, 'wt')
    out.write(b64str.decode())
    out.close()
    return nr

>>> primToB64(512)
```

A Base64-es alak létrehozásához a generált számot a `to_bytes` metódussal először átalakítottuk 256-os számrendszerbe, amelynek eredménye egy `bytes` típusú érték. Ezután pedig alkalmazhatjuk a `b4encode` átalakító függvényt, amely `bytes` típusú bemenetet vár.

6.17. algoritmus. Írjunk egy Python függvényt, amely az előző algoritmus során létrehozott file-ból beolvassa a kiírt primszám Base64-es alakját, majd visszaalakítja tízes számrendszerbeli számmá.

```
from base64 import b64decode
def primFromB64(fname = 'primeB64.txt'):
    inf = open(fname, 'rt')
    m_str = inf.read()
    inf.close()
```

```
    b64str = b64decode(m_str.encode())
    nr = int.from_bytes(b64str, byteorder = 'big')
    return nr

>>> primFromB64()
```

A fejezetet egy determinisztikus prímteszt bemutatásával zárjuk, amely Wilson[15] tételén alapszik.

6.22. tétel. (Wilson tétel) Egy p szám akkor és csakis akkor prímszám, ha

$$(p-1)! \equiv p-1 \equiv -1 \pmod{p}.$$

A tétel első bizonyítása Lagrange-tól[16] származik 1770-ből, de a matematikatörténet számontartja, hogy a tételt korábban Ibn al-Haytham[17] is megfogalmazta.

Példák:

$6! = 720 \equiv 6 \pmod{7} \iff 7$ prímszám,

$22! = 1124000727777607680000 \equiv 22 \pmod{23} \iff 23$ prímszám,

$1788! \equiv 1788 \pmod{1789} \iff 1789$ prímszám.

Bizonyítás: A $p = 2$ eset megállapítható a következőből: $(p-1)! \equiv 1 \equiv -1 \pmod{p}$.

A $p > 2$ esethez a tétel direkt állításának bizonyításához először belátjuk, hogy ha p prímszám, akkor a $2, 3, \ldots, p-2$ számokat, ha párosával összeszorozzuk, akkor a kapott szorzatok mindegyike kongruens lesz 1 (mod p)-vel. Ebből pedig következni fog, hogy a $\mathbf{1}\cdot\mathbf{2}\ldots(\boldsymbol{p-2})\cdot(p-1) \equiv p-1 \pmod{p}$, azaz igaz a tétel direkt állítása.

Egy $2 \leq a \leq p-2$ szám párját úgy tudjuk meghatározni, ha megoldjuk az $a \cdot x \equiv 1 \pmod{p}$ kongruenciát. Ennek a kongruenciának egy megoldása lesz, mert $(a, p) = 1$. Legyen ez a megoldás b, ahol igaz, hogy $2 \leq b \leq p-2$, mert ha $x = 0, 1, p-1$, akkor $a \cdot x \not\equiv 1 \pmod{p}$-vel. Az is megállapítható, hogy a-nak nem lehet önmaga a párja, mert az azt jelentené, hogy fennáll: $a^2 \equiv 1 \pmod{p}$. Ebből pedig az következne, hogy $p \mid (a-1)\cdot(a+1)$. Mivel p prímszám, ebből pedig következne, hogy $p \mid a-1$ vagy $p \mid a+1$, azaz $a \equiv 1 \pmod{p}$ vagy $a \equiv p-1 \pmod{p}$. Ez pedig ellentmondás. Tehát a párja b lesz, ahol $a \neq b$.

15 John Wilson angol matematikus, jogász (1741–1793)
16 Joseph-Louis Lagrange olasz matematikus (1736–1813)
17 Ibn al-Haytham arab matematikus (965–1040)

A tétel fordított állításának bizonyításához tegyük fel, hogy p összetett szám: $p = a \cdot b$, ahol $1 < a < p$ és $1 < b < p$. Megállapítható, hogy $a \mid (p-1)!$, mert $a < p$, ami azt jelenti, hogy a-nak mint szorzótényezőnek szerepelnie kell a $(p-1)!$-ban. Másfelől $(p-1)! \equiv p-1 \equiv -1 \pmod{p}$, amiből következik, hogy $p \mid (p-1)!+1$, tehát a is osztója $(p-1)!+1$-nek. Ezek szerint kijelenthető, hogy a osztója $(p-1)!+1-(p-1)! = 1$. Ez azonban ellentmondás, mert feltételeztük, hogy $a > 1$.

□

Példa: p = 11 esetében 2 párja 6, mert $2 \cdot 6 \equiv 1 \pmod{11}$. 3 párja 4, mert $3 \cdot 4 \equiv 1 \pmod{11}$. 5 párja 9, mert $5 \cdot 9 = 1 \pmod{11}$. 7 párja 8, mert $7 \cdot 8 \equiv 1 \pmod{11}$. Fennáll tehát: $2 \cdot 6 \cdot 3 \cdot 4 \cdot 5 \cdot 9 \cdot 7 \cdot 8 \equiv 1 \pmod{11} \Longrightarrow 9! \cdot 10 = 10! \equiv 10 \pmod{11}$.

6.18. algoritmus. Írjunk egy Python függvényt, amely Wilson tételét alkalmazva megállapítja egy számról, hogy prímszám-e.

```
def tesztWilson(nr):
    f = 1
    for i in range(1, nr):
        f = (f * i) % nr
    if f == nr - 1:
        return True
    return False

>>> tesztWilson(626887)
    True
```

6.19. algoritmus. Írjunk egy Python függvényt, amelyben lemérjük a `tesztWilson` futási idejét a következő prímszámokra: 626887, 3276599, 42877453.

```
from time import time
def idomeresWilson():
    st = time()
    tesztWilson(626887)
    fs = time()
    print(round(fs - st, 4))

    st = time()
    tesztWilson(3276599)
    fs = time()
    print(round(fs - st, 4))

    st = time()
    tesztWilson(42877453)
    fs = time()
    print(round(fs - st, 4))
```

```
>>> idomeresWilson()
    0.06800413131713867
    0.3310239315032959
    4.427326917648315
```

Megállapíthatjuk, hogy a Wilson-tétel egy szükséges és elégséges feltételt határoz meg arra vonatkozóan, hogy mikor prím egy szám [27, 33]. Ennek ellenére nagy számok esetében mégsem alkalmazható, mert elfogadhatatlan a futási ideje.

6.7. Hatványok és generátor elemek

A kis Fermat-, illetve Euler-tétel mellett számos egyéb tulajdonság figyelhető meg az $x^n \pmod m$ értékek meghatározásakor [2].

Példa: Határozzuk meg $x^n \pmod m$, értékeit $m = 11, x = 2, 3, \ldots 10$-re illetve $n = 1, 2, \ldots 10$-re:

$2^1 = 2$	$3^1 = 3$	$4^1 = 4$	$5^1 = 5$	$6^1 = 6$	$7^1 = 7$	$8^1 = 8$	$9^1 = 9$	$10^1 = 10$
$2^2 = 4$	$3^2 = 9$	$4^2 = 5$	$5^2 = 3$	$6^2 = 3$	$7^2 = 5$	$8^2 = 9$	$9^2 = 4$	$\mathbf{10^2 = 1}$
$2^3 = 8$	$3^3 = 5$	$4^3 = 9$	$5^3 = 4$	$6^3 = 7$	$7^3 = 2$	$8^3 = 6$	$9^3 = 3$	$10^3 = 10$
$2^4 = 5$	$3^4 = 4$	$4^4 = 3$	$5^4 = 9$	$6^4 = 9$	$7^4 = 3$	$8^4 = 4$	$9^4 = 5$	$\mathbf{10^4 = 1}$
$\mathbf{2^5 = 10}$	$\mathbf{3^5 = 1}$	$\mathbf{4^5 = 1}$	$\mathbf{5^5 = 1}$	$\mathbf{6^5 = 10}$	$\mathbf{7^5 = 10}$	$\mathbf{8^5 = 10}$	$\mathbf{9^5 = 1}$	$10^5 = 10$
$2^6 = 9$	$3^6 = 3$	$4^6 = 4$	$5^6 = 5$	$6^6 = 5$	$7^6 = 4$	$8^6 = 3$	$9^6 = 9$	$\mathbf{10^6 = 1}$
$2^7 = 7$	$3^7 = 9$	$4^7 = 5$	$5^7 = 3$	$6^7 = 8$	$7^7 = 6$	$8^7 = 2$	$9^7 = 4$	$10^7 = 10$
$2^8 = 3$	$3^8 = 5$	$4^8 = 9$	$5^8 = 4$	$6^8 = 4$	$7^8 = 9$	$8^8 = 5$	$9^8 = 3$	$\mathbf{10^8 = 1}$
$2^9 = 6$	$3^9 = 4$	$4^9 = 3$	$5^9 = 9$	$6^9 = 2$	$7^9 = 8$	$8^9 = 7$	$9^9 = 5$	$10^9 = 10$
$\mathbf{2^{10} = 1}$	$\mathbf{3^{10} = 1}$	$\mathbf{4^{10} = 1}$	$\mathbf{5^{10} = 1}$	$\mathbf{6^{10} = 1}$	$\mathbf{7^{10} = 1}$	$\mathbf{8^{10} = 1}$	$\mathbf{9^{10} = 1}$	$\mathbf{10^{10} = 1}$

A kis Fermat-tétel kimondja, hogy ha m prím és $(x, m) = 1$, akkor $x^{m-1} = 1 \pmod m$. A fenti táblázat alapján megállapíthatjuk, hogy vannak

olyan x számok, amelyeknek $m-1$-nél kisebb hatványkitevőn vett hatványértékei is kongruensek lesznek 1-gyel.

6.11. értelmezés. Legyenek x, m pozitív egész számok úgy, hogy $(x, m) = 1$. x **rend**-jén, $\pmod m$ szerint, azt a legkisebb k kitevőt értjük, amelyre fennáll: $x^k \equiv 1 \pmod m$, és $\boldsymbol{ord_m x}$-el jelöljük.

Példák:

1. $\pmod{11}$ szerint 2 rendje 10: $ord_{11}2 = 10$,
2. $\pmod{11}$ szerint 3 rendje 5: $ord_{11}3 = 5$.

6.20. algoritmus. Írjunk egy Python függvényt, amely brute-force módszerrel meghatározza x rendjét $\pmod m$ szerint.

```
def ordElemBF(x, m):
    for i in range(2, m):
        temp = pow(x, i, m)
        if temp == 1: return i
    return -1

>>> ordElemBF(22, 35)
    4
```

6.23. tétel. Legyenek a, m egész számok úgy, hogy $(a, m) = 1$ és $m > 0$. Az $a^x \equiv 1 \pmod m$ kongruenciának x_0 akkor és csakis akkor megoldása, ha $ord_m a$ osztja x_0-t [27].

Példa: $3^{40} \equiv 1 \pmod{41}$ ekkor 3 rendjét 40 osztói között kell keresnünk:

$$\begin{aligned} 3^2 &\equiv 9 \pmod{41}, \\ 3^4 &\equiv 40 \pmod{41}, \\ 3^5 &\equiv 38 \pmod{41}, \\ 3^8 &\equiv 1 \pmod{41}, \text{ azaz } ord_{41}3 = 8. \end{aligned}$$

Példa: $ord_{44}9 = 5$, 5 osztja a 10-t, 15-t, 20-t, stb. $\Longrightarrow$

$$\begin{aligned} 9^{10} &\equiv 1 \pmod{44}, \\ 9^{15} &\equiv 1 \pmod{44}, \\ 9^{20} &\equiv 1 \pmod{44}, \text{ stb.} \end{aligned}$$

Bizonyítás: Tegyük fel, hogy $a^x \equiv 1 \pmod m$ a tétel direkt állításának bizonyításához. A maradékos osztás 4.1. tétele alapján felírhatjuk: $x = q \cdot ord_m a + r,\ 0 \leq r < ord_m a$. Hatványértéket számolva, kapjuk:

$$a^x = a^{q \cdot ord_m a + r} = (a^{ord_m a})^q \cdot a^r \equiv a^r \pmod{m}.$$

Mivel $a^x \equiv 1 \pmod{m}$, következik, hogy $a^r \equiv 1 \pmod{m}$. De $0 \leq r < ord_m a$ és $ord_m a$ az a legkisebb pozitív kitevő, amelyre $a^{ord_m a} \equiv 1 \pmod{m}$. Tehát $r = 0$, de ekkor $ord_m a$ osztja x-et.

A fordított állítás bizonyításához tegyük fel, hogy $ord_m a$ osztja x_0-et. Ebből következik, hogy $x_0 = q \cdot ord_m a$. Hatványértéket számolva kapjuk:

$$a^{x_0} = a^{q \cdot ord_m a} = (a^{ord_m a})^q \equiv 1 \pmod{m}.$$

Tehát x_0 megoldása az $a^x \equiv 1 \pmod{m}$ kongruenciának.

□

6.24. tétel. Legyenek x, m egész számok úgy, hogy $(x, m) = 1$ és $m > 0$. Ekkor $x^i \equiv x^j$ **(mod m)**, akkor és csakis akkor, ha $i \equiv j$ **(mod $ord_m x$)**, ahol i, j pozitív egész számok [27].

Példák:

1. $ord_{35}17 = 12$, ekkor $7 \equiv 43 \pmod{12} \iff 17^7 \equiv 17^{43} \pmod{35}$,
2. $ord_{41}11 = 40$, ekkor $23 \equiv 103 \pmod{40} \iff 11^{23} \equiv 11^{103} \pmod{41}$.

Bizonyítás: A tétel direkt állításának bizonyításához tegyük fel, hogy $x^i \equiv x^j \pmod{m}$, és legyen $j \leq i$. Mivel $(x, m) = 1$, következik, hogy $(x^j, m) = 1$, azaz a következő kongruenciát végig lehet osztani x^j-vel: $x^i \equiv x^j \cdot x^{i-j} \equiv x^j \pmod{m}$. Kapjuk, hogy $x^{i-j} \equiv 1 \pmod{m}$. Ekkor azonban $ord_m x$ osztója kell legyen $i - j$-nek, amiből következik, hogy $i \equiv j \pmod{ord_m x}$.

A fordított állítás bizonyításához tegyük fel, hogy $i \equiv j \pmod{ord_m x}$ és legyen $j \leq i$. A maradékos osztás 4.1. tétele alapján felírható, hogy $i = j + q \cdot ord_m x$. Hatványértéket számolva kapjuk:

$$x^i \equiv x^j \cdot x^{q \cdot ord_m x} \equiv x^j \pmod{m}.$$

□

6.25. tétel. Legyenek x, m egész számok úgy, hogy $(x, m) = 1$ és $m > 0$. Ekkor $ord_m x \mid \phi(m)$.

Bizonyítás: $(x, m) = 1$ következik, hogy $x^{\phi(m)} \equiv 1 \pmod{m}$. A 6.23. tétel alapján pedig $ord_m x \mid \phi(m)$.

□

Példa: Határozzuk meg $\pmod{35}$ szerint 17 rendjét. Ennek érdekében kiszámítjuk $\phi(35) = 4 \cdot 6 = 24 \Longleftrightarrow$ 17 rendjét, tehát 24 pozitív osztói között kell keresni:

$$\begin{aligned} 17^2 &\pmod{35} = 9, & 17^6 &\pmod{35} = 29, \\ 17^3 &\pmod{35} = 13, & 17^8 &\pmod{35} = 16, \\ 17^4 &\pmod{35} = 11, & 17^{12} &\pmod{35} = 1. \end{aligned}$$

A kiszámolt hatványértékek alapján megállapítható, hogy $ord_{35}17 = 12$.

6.12. értelmezés. Legyenek x, m egész számok úgy, hogy $(x, m) = 1$ és $m > 0$. Ha $ord_m x = \phi(m)$, akkor x-et **primitív gyöknek** vagy **generátor elemnek** hívjuk.

6.21. algoritmus. Írjuk egy Python függvényt, amely brute-force módszerrel meghatározza az m egész szám generátor elemeit.

A következő `genGeneratorBF` függvényben meghívásra kerülnek a 6.7, illetve 6.20 algoritmusok.

```
def genGeneratorBF(m):
    phi = euler(m)
    ok = False
    for x in range(2,m):
        if gcd(x, m) == 1:
            i = ordElemBF(x, m)
            if i == phi:
                print(x, end = ', ')
                ok = True
    if not ok: print('nincs generator elem')

>>> genGeneratorBF(35)
    nincs generator elem
>>> genGeneratorBF(46)
    5, 7, 11, 15, 17, 19, 21, 33, 37, 43,
```

A kriptográfiában fontos feladat, hogy $\pmod{p}$ szerint, ahol p prímszám, meghatározzunk egy generátor elemet, éppen ezért a fejezet hátralevő részében ezzel a problémával fogunk foglalkozni [2].

6.26. tétel. Legyenek

1. x, m egész számok úgy, hogy $(x, m) = 1$,
2. $\phi(m)$ prímtényezős felbontása: $\phi(m) = p_1^{a_1} \cdot p_2^{a_2} \dots p_n^{a_n}$,

3. $fi_i = \phi(m)/p_i$, minden $i \in \{1, \ldots, n\}$.

Az x akkor és csakis akkor generátor elem, ha

$$x^{fi_i} \not\equiv 1 \pmod{m}, \text{ minden } i \in \{1, \ldots, n\}.$$

Bizonyítás: A tétel direkt állításának bizonyításához tegyük fel, hogy x generátor elem. Ekkor a legkisebb pozitív egész szám, amelyre $x^k \equiv 1 \pmod{m}$, az $k = \phi(m)$. Mivel $0 < fi_i < \phi(m)$ minden $i \in \{1, \ldots, n\}$ következik, hogy $x^{fi_i} \not\equiv 1 \pmod{m}$.

A tétel fordított állításának bizonyításához tegyük fel, hogy $x^{fi_i} \not\equiv 1 \pmod{m}$, minden $i \in \{1, \ldots, n\}$. Legyen k az x rendje, ekkor k az a legkisebb pozitív egész szám lesz, amelyre $x^k \equiv 1 \pmod{m}$, ugyanakkor k osztója lesz $\phi(m)$-nek. Mivel $\phi(m) = p_1^{a_1} \cdot p_2^{a_2} \ldots p_n^{a_n}$, következik, hogy $k = p_1^{b_1} \cdot p_2^{b_2} \ldots p_n^{b_n}$ úgy, hogy $0 \leq b_i \leq a_i$, valamely $i \in \{1, \ldots, n\}$. Ha most feltételezzük, hogy $k < \phi(m)$, akkor kell létezzen egy olyan i, amelyre $b_i < a_i$, ekkor azonban k osztani fogja fi_i-t, amiből a 6.23. tétel alapján következik, hogy $x^{fi_i} \equiv 1 \pmod{m}$. Ez ellentmondás, azaz hibás volt az a feltételezés, hogy $k < \phi(m)$. Tehát csak az lehetséges, hogy $k = \phi(m)$, ami viszont azt jelenti, hogy x generátor elem.

□

Példa: Határozzuk meg (mod 46) szerint a generátor elemeket. Tudjuk, hogy $\phi(46) = \phi(2) \cdot \phi(23) = 22 = 2 \cdot 11$. Ahhoz, hogy x generátor elem legyen szükséges, hogy fennálljon: $x^2 \not\equiv 1 \pmod{46}$ és $x^{11} \not\equiv 1 \pmod{46}$. Ehhez használhatjuk a következő Python-kódot:

```
>>> from math import gcd
>>> for x in range(2, 46):
        if gcd(x, 46) == 1:
            t1 = pow(x, 2, 46)
            t2 = pow(x, 11, 46)
            if t1 != 1 and t2 != 1: print(x, end = ' ')

    5 7 11 15 17 19 21 33 37 43
```

A következőkben az esetet vizsgáljuk, amikor a modulus egy prímszám. Az ehhez kapcsolódó tételt bizonyítás nélkül közöljük [27, 39].

6.27. tétel. Legyen p egy prímszám.

1. Mindig létezik (mod p) szerint egy g generátor elem.
2. A generátor elemek rendje $p - 1$.

3. A generátor elemek száma $\phi(p-1)$.

Példa: (mod 11) szerint:

1. a generátor elemek: $2, 6, 7, 8$.
2. $ord_{11}2 = ord_{11}6 = ord_{11}7 = ord_{11}8 = 10$.
3. A generátor elemek száma $\phi(10) = 4$.

6.13. értelmezés. Biztonságos prímeknek (safe prime) hívjuk azokat a p prímszámokat, amelyek egyenlők $2 \cdot q + 1$-gyel, ahol q is prímszám. A q számot pedig Sophie–Germain[18]-prímnek hívják [32].

Példa:

biztonságos prímek:	nem biztonságos prímek:
$11 = 2 \cdot \mathbf{5} + 1$	$13 = 2 \cdot \mathbf{6} + 1$
$47 = 2 \cdot \mathbf{23} + 1$	$17 = 2 \cdot \mathbf{8} + 1$

Ahhoz, hogy meg tudjuk állapítani egy tetszőleges x számról, hogy (mod m) szerint generátor elem vagy sem, szükséges meghatározni $\phi(m)$-t, amelynek meghatározására nagy számok esetén nem ismert hatékony algoritmus. Ha azonban a modulus a p biztonságos prím, akkor $\phi(p) = p - 1 = 2 \cdot q$, és annak az eldöntése, hogy x generátor elem vagy sem, már nem fog nehéz feladatnak számítani [2].

6.28. tétel. Ha p biztonságos prím, akkor

1. g generátor elem lesz, ha:

$$g^2 \not\equiv 1 \pmod{p} \text{ és } g^q \not\equiv 1 \pmod{p},$$

2. a generátor elemek száma:

$$\phi(p-1) = \phi(2) \cdot \phi(q) = q - 1.$$

Példa: Legyen $p = 47 = 2 \cdot \mathbf{23} + 1$, ekkor:

$g = 13$ generátor elem, mert
$13^{\mathbf{23}} \equiv 46 \not\equiv 1 \pmod{47}$, és
$13^{\mathbf{2}} \equiv 28 \not\equiv 1 \pmod{47}$

$g = 3$ nem lesz generátor elem, mert $3^{\mathbf{23}} \equiv 1 \pmod{47}$.

18 Sophie Germain francia matematikus (1776–1831)

Egy biztonságos prím meghatározása jóval időigényesebb, mint egy *közönséges* prím kiválasztása, mégis a kriptográfiai protokollokban biztonságos prímekkel dolgoznak, mert ebben az esetben hatékonyan meghatározható egy generátor elem. Az elgondolás az, hogy addig generálunk véletlenszerűen számokat a $[2, p-2]$ intervallumból, amíg a $g^2 \pmod p$ és $g^q \pmod p$ hatványértékek különbözni fognak 1-től.

A biztonságos prímek generálásának időigényét csökkenthetjük [40]. Megállapítható, hogy leszámítva a $p = 7$ esetet, ha $p = 2 \cdot q + 1$, akkor fenn kell álljon, hogy: $q \equiv p \equiv 2 \pmod 3$. Ellenkező esetben, ha $p \equiv 1 \pmod 3$, akkor q osztható lesz 3-mal, ha pedig $q \equiv 1 \pmod 3$, akkor p lesz osztható 3-mal. Ezt általánosítva megállapíthatjuk, hogy bármely kis r prímszám esetében, ha $q \equiv (r-1)/2 \pmod r$, akkor p osztható lesz r-rel. Ezt alkalmazva a `testSmallVal` függvényben a következő `safePrimeGen` függvény egy k bites biztonságos prímet generál.

6.22. algoritmus. Írjunk egy Python függvényt, amely generál egy k bites biztonságos prímszámot.

```
from random import getrandbits
def safePrimeGen(k):
    P = [3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
    while True:
        q = getrandbits(k)
        if q & 1 == 0: q += 1
        if testSmallVal(q, P) and millerRabinT(q):
            p = 2 * q + 1
            if testSmallPrime(p, P) and millerRabinT(p):
                return p

def testSmallVal(p, P):
    for r in P:
        temp = p % r
        if temp == 0 or temp == (r - 1)//2: return False
    return True

def testSmallPrime(p, P):
    for r in P:
        if p % r == 0: return False
    return True

>>> safePrimeGen(512)
    ...
```

A `testSmallPrime` függvény a `P` listában megadott számokkal vizsgálja az oszthatóságot. Meghíváskor a `P` listába az első 13 prímszámot adtuk meg így a Miller–Rabin-prímteszt (6.14) csak akkor kerül meghívásra, ha a tesztelendő számnak nincsenek kis prímosztói.

6.8. A Diffie–Hellman-kulcscsere

A Diffie–Hellman-kulcscserét 1976-ban publikálták a szerzők Whitfield Diffie[19] és Martin Hellman[20] [9]. Mérföldkőnek számított, amely alapjaiban meghatározta/meghatározza a számítógépes információcserék biztonságát. Azóta beszélünk **publikus kulcsú/aszimmetrikus kriptográfiáról**, amelyhez három különböző protokollcsalád tartozik: a kulcscsere-protokollok, a publikus kulcsú titkosítók, és a digitális aláírások. A következő fejezetekben szó lesz még a publikus kulcsú titkosítókról, bemutatásra kerülnek az RSA- és Rabin-rendszerek, a digitális aláírásoknál pedig a Rabin digitális aláírási sémát ismertetjük.

A Diffie–Hellman-kulcscsere elsőként írta le, hogy két távoli egység (számítógép, mobileszköz stb.) hogyan tud megegyezni egy nyilvános csatornát használva egy egész szám értékében, oly módon, hogy rajtuk kívül más egység ne tudja meghatározni ezt az egész számot. A megosztott egész számot a szakirodalom mesterkulcsnak (master key), titkos információnak vagy egyszerűen csak kulcsnak nevezi.

A kulcscsere egész számokon végez műveleteket, amelyeket az átküldés miatt átalakít 10-es számrendszerből 256-os számrendszerbe, majd a fogadó fél oldalán visszaalakítja a 256-os számrendszerbeli értékeket 10-es számrendszerbe. Az adatcsere ily módon bájtszekvenciák megosztását jelenti, amelyek eredményeképpen a kommunikáló felek ugyanazt az egész számot tudják kiszámítani. A gyakorlatban a megosztott egész szám a szimmetrikus titkosító rendszerekben tölti be a kulcs szerepét, innen is a kulcscsere elnevezés.

A Diffie–Hellman-kulcscsere biztonsága a diszkrét logaritmus feltételezésen (DL feltételezés) alapszik. Az adatbiztonság területén a DL feltételezésnek több verziója is megadható. Az egész számok $\pmod p$ multiplikatív csoportjára, illetve az elliptikus görbékre megadott értelmezések mindegyikét széles körben alkalmazzák [2, 14].

19 Whitfield Diffie amerikai kriptográfus, matematikus (1944–)
20 Martin Hellman amerikai kriptográfus, matematikus (1945–)

6.14. értelmezés. Az egész számok $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ véges halmaza esetében, ahol ismert a p prímszám, egy $g \in \mathbb{Z}_p^*$ generátor elem, és az $A \in \mathbb{Z}_p^*$ egész szám, a **diszkrét logaritmus probléma** azt jelenti, hogy megkeressük azt az $\boldsymbol{a} \in \mathbb{Z}_p^*$ számot, amelyre fennáll:

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

Az a számot A g alapú diszkrét logaritmusának hívjuk. Nagy számok esetében a DL problémára nem ismert elfogadható futásidejű algoritmus, ezt hívjuk **diszkrét logaritmus feltételezésnek**. Jelenleg **minimum 1024 bites** prímszám esetében jelenthető ki biztonsággal, hogy a DL problémára nincs hatékony eljárás.

Napjainkban egyre szélesebb körben alkalmazzák az elliptikus görbéken értelmezett DL problémán alapuló kriptorendszereket is, mert ezekben elégséges a 160 bites prímszámok használata, ami azt is jelenti, hogy kisebb számokkal kell a számításokat végezni [14]. Ezek a rendszerek egész számpárokkal végeznek matematikai műveleteket, ahol a rendszerek előírják, hogy a számpárokat hogyan alakítsuk át bájtszekvenciákká. A jegyzet keretén belül azonban nem térünk ki az elliptikus görbék matematikájára, sem az elliptikus görbéken alapuló kriptográfiára.

Feltételezve, hogy a kommunikációban részt vevő két egység **A** és **B**, akkor a Diffie–Hellman-kulcscsere lépései a következők:

1. **A** vagy egy harmadik megbízható egység választ egy p biztonságos prímet, és meghatározza p-nek egy g generátor elemét, majd megosztja ezeket az értékeket egy nyilvános csatornán **B**-vel. Ha egy harmadik fél generálta ki a p és g értékeket, akkor azt egy nyilvános csatornát használva **A**-val is megosztja. A p és g értékeket **publikus adatoknak** hívjuk, bárki ismerheti őket.
2. Az **A** egység a p, g ismeretében meghatározza az a, és A értékeket, ahol $a \in \{2, \ldots, p-2\}$ véletlenszerűen kerül kiválasztásra, és $A \equiv g^a \pmod{p}$. Az a értékét titokban tartja, A-t pedig elküldi **B**-nek egy nyilvános csatornán. A tehát publikus információnak számít.
3. A **B** egység a p, g ismeretében meghatározza a b, és B értékeket, ahol $b \in \{2, \ldots, p-2\}$ véletlenszerűen kerül kiválasztásra, és $B \equiv g^b \pmod{p}$. A b értékét titokban tartja, B-t pedig elküldi **A**-nak egy nyilvános csatornán. B is tehát publikus információ.
4. Az **A** egység a közös K kulcsot a kövekezőképpen határozza meg:

$$K \equiv B^a \pmod{p}.$$

5. A **B** egység a közös K kulcsot a következőképpen határozza meg:

$$K \equiv A^b \pmod{p}.$$

A protokoll helyessége a következő számításokat figyelembe véve bizonyítható:

$$K \equiv A^b \equiv (g^a)^b \equiv g^{ab} \equiv (g^b)^a \equiv B^a \equiv K \pmod{p}.$$

Példa: Legyen $p = 47, g = 13$, ahol $p = 2 \cdot 23 + 1$. Vegyük észre, hogy fennáll: $13^{23} \equiv 46 \not\equiv 1 \pmod{47}$, és $13^2 \equiv 28 \not\equiv 1 \pmod{47}$, tehát 13 generátor elem.

1. Az **A** egység választ egy random a számot, legyen ez 12, és meghatározza $A \equiv 13^{12} \equiv 9 \pmod{47}$. Elküldi **B**-nek az $A = 9$-et.
2. A **B** egység választ egy random b számot, legyen ez 34, és meghatározza $B \equiv 13^{34} \equiv 21 \pmod{47}$. Elküldi **A**-nak a $B = 21$-et.
3. **A** ezek után meghatározza a $K \equiv 21^{12} \equiv 16 \pmod{47}$ értéket.
4. **B** ezek után meghatározza a $K \equiv 9^{34} \equiv 16 \pmod{47}$ értéket.

A közös kulcs $K = 16$ lesz.

6.23. algoritmus. Írjunk egy Python programot, amely a 6.22. algoritmusban megadott `safePrimeGen` függvényt alkalmazva generál egy k bites biztonságos prímet, meghatározza a g-t, p egy generátor elemét, majd ezen értékeket felhasználva a Diffie–Hellmann-kulcscsere lépéseit követve meghatároz egy közös K kulcsot. Gondoskodnunk kell tehát a `safePrimeGen` függvény importálásáról, vagy másoljuk át a függvényt az aktuális állományba.

```
from random import randint
def dhKeyGen(k):
    p = safePrimeGen(k)
    q = (p - 1)//2
    while True:
        g = randint(2, p-2)
        temp1 = pow(g, q, p)
        temp2 = pow(g, 2, p)
        if temp1 != 1 and temp2 != 1: break
    return p, g

>>> dhKeyGen(1024)
```

```
    ...

def dhFeladat():
    k = int(input('bit meret: '))
    p, g = dhKeyGen(k)
    print('a generalt kulcs, publikus adat: ', p, g)

    #az A egyseg szamitasai
    a = randint(2, p-1)
    A = pow(g, a, p)

    #a B egyseg szamitasai
    b = randint(2, p-1)
    B = pow(g, b, p)

    #az A egyseg szamitasai, ha már megkapta a B-t
    K = pow(B, a, p)
    print('az A egyseg altal meghatarozott ertek: ', K)

    #a B egyseg szamitasai, ha már megkapta az A-t
    K = pow(A, b, p)
    print('a B egyseg altal meghatarozott ertek: ', K)

>>> dhFeladat():
    ...
```

6.9. A diszkrét logaritmus probléma

A diszkrét logaritmus probléma megoldására több algoritmus is ismert, de nagy számok esetében egyik sem hatékony. A brute-force típusú algoritmus a legegyszerűbb és a legkevésbé hatékony módszer a diszkrét logaritmus meghatározására. A baby-step giant-step algoritmus ennek egy javított, hatékonyabb változata. További nem triviális algoritmusok a Pollard-rho, a Pohlig–Hellman, az indexkalkulus algoritmus [21, 22]. Ezekre a jegyzet keretén belül azonban nem térünk ki.

A **brute-force** típusú algoritmus gyakorlatilag sorra próbálgatja a kitevőket, amíg találathoz nem ér. A bemeneti prímszám értékének a növelésével az algoritmus futási ideje elfogadhatatlanná válik.

6.1. feladat. Írjunk egy Python függvényt, amely meghatározza $\pmod p$ szerint a $g^0, g^1, g^2 \dots g^{p-1}$ hatványértékeket, amíg nem talál egy A-val megegyező értéket, azaz meghatározza az A g alapú diszkrét logaritmusát.

```
def dLogBF(g, A, p):
    for i in range(1, p-1):
        if pow(g, i, p) == A:
            return i

>>> dLogBF(1234, 10000, 530063)
    73132
```

A **baby-step giant-step** algoritmus alapötlete Daniel Shanks[21]-től származik 1971-ből, és a brute force algoritmus egy módosított változata [30].

Az algoritmus a g, A, p bemenetre keresi azt az a kitevőt, amelyre fennáll: $g^a \equiv A \pmod p$. Ennek érdekében megpróbálja felírni az a kitevőt a következő alakba:

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

Mindezt azért, mert ekkor: $g^a = g^{k \cdot i + j} = g^{k \cdot i} \cdot g^j$, amiből következik, hogy:

$$g^j = g^a \cdot g^{-k \cdot i} = A \cdot (g^{-1})^{k \cdot i}.$$

A lépések, amiket ezek után végez az algoritmus, a következők:

1. minden $j = 0,\ 1, \dots, k-1$-re meghatározza a $g^j \pmod p$-ket.
2. meghatározza g^{-1}-t azaz g inverzét, és $(g^{-1})^k$-t $\pmod p$ szerint,
3. amikor valamely $i = 0, 1, \dots, k-1$ érték esetében $A \cdot (g^{-k})^i \pmod p$ megegyezik valamely $g^j \pmod p$-vel, akkor a találatot adó i, j-k alapján meghatározza az $a = k \cdot i + j$-t.

Példa: Határozzuk meg a diszkrét logaritmust, ha $p = 19, g = 3, A = 11$.

1. meghatározzuk: $k = \lceil \sqrt{18} \rceil = 5$,
2. meghatározzuk a $g^j \pmod{19}$ értékeket, j = 1, 2, 3, 4-re:

$$3^0 \equiv 1, 3^1 \equiv 3, 3^{\mathbf{2}} \equiv \mathbf{9}, 3^3 \equiv 8, 3^4 \equiv 5,$$

3. meghatározzuk $3^{-5} \equiv 14 \pmod{19}$-t,

21 Daniel Shanks amerikai matematikus (1917–1996)

4. megvizsgáljuk, hogy melyik $i = 0, 1, 2, 3, 4$ -re lesz találat g^j-vel:

$$11 \cdot 14^0 \equiv 11,\ 11 \cdot 14^1 \equiv 2,\ 11 \cdot 14^{\mathbf{2}} \equiv \mathbf{9},$$

5. $\boldsymbol{i = 2, j = 2}$ esetén van találat, tehát $a = 5 \cdot 2 + 2 = \mathbf{12}$.

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

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

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

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

6.10. Az RSA-kriptorendszer

Az RSA-kriptorendszert 1977-ben publikálták a szerzők, Ron Rivest[22], Adi Shamir[23] és Leonard Adleman[24] [29]. Megjelenése pillanatától kezdve a legsikeresebbnek, legszélesebb körben használt kriptográfiai protokollnak számított. Napjainkban az elliptikus görbék térhódításával veszített népszerűségéből, a TLS 1.3 protokoll például már nem támogatja az RSA titkosítót, csak az RSA digitális aláírást. Biztonsága többek között a faktorizációs feltételezésen alapszik [2, 14]. A brit hírszerzés 1997-ben arról tudósított, hogy egy hasonló rendszert ők már 1973-ban kidolgoztak, de biztonsági okokból nem tették publikussá.

6.15. értelmezés. Az egész számok $\{1, 2, \ldots, n\}$ véges halmaza esetében, ahol n két prímszám szorzata, a **faktorizációs probléma** azt jelenti, hogy az n értékét ismerve keressük meg azt a két p és q prímszámot, amelyekre fennáll: $n = p \cdot q$.

Nagy, legalább 1024 bites számok esetében azonban a faktorizációs problémára nem ismert hatékony algoritmus, ezt hívjuk **faktorizációs feltételezésnek**.

Azokat az algoritmusokat, amelyek a faktorizációs problémát ilyen vagy olyan hatékonysággal megoldják, faktorizációs algoritmusoknak hívjuk.

Ahogy a Diffie-Hellmann-kulcscserénél, úgy az RSA-kriptorendszernél is egész számokkal végzünk aritmetikai műveleteket. Az adatküldés miatt aztán ezeket a számokat bájtszekvenciákká alakítják, a fogadó fél oldalán pedig a kézhez kapott bájtszekvenciákat visszaalakítják egész számokká.

Az RSA-kriptorendszer esetében megkülönböztetjük az **RSA-titkosító** és **RSA digitális aláírás** rendszereket. Az RSA-titkosító két távoli egység nyilvános csatornán történő kis méretű, titkosított adatcseréjére alkalmas, amelyet korábban szimmetrikus titkosítók kulcsának a megosztására használtak. A digitális aláírásra a Rabin-kriptorendszer keretén belül a későbbiek során még visszatérünk, itt fogjuk bemutatni a digitális aláírás protokollját is. Általában elmondható, hogy azok a rendszerek, amelyek publikus kulcsú titkosítást végeznek, átalakíthatók digitális aláírásokká.

22 Ron Rivest amerikai kriptográfus (1947–)
23 Adi Shamir izraeli kriptográfus (1952–)
24 Len Adleman amerikai informatikus (1945–)

Egy publikus kulcsú titkosító rendszerben három algoritmust kell megadni, a kulcsgeneráló, a titkosító és a visszafejtő algoritmusokat. A kulcsgeneráló algoritmus egy-egy értékpárt, pontosabban egy-egy kulcspárt határoz meg, ahol az egyik értéket **publikus kulcsnak**, a másik értéket **privát kulcsnak** hívjuk. A gyakorlatban a publikus és privát kulcsokat külön állományokba szokták tenni, és kódolják őket Base64-be.

Feltételezve, hogy a kommunikációban részt vevő két egység **A** és **B**, akkor a publikus kulcsú titkosítási protokoll lépései a következők:

1. **A** vagy egy harmadik megbízható egység végrehajtja a kulcsgeneráló algoritmust, majd a publikus kulcsot egy nyilvános csatornán megosztja a **B** egységgel. Ha egy harmadik egység végezte a kulcsgenerálást, akkor a publikus kulcsot egy nyilvános csatornán, a privát kulcsot pedig egy titkos csatornán fogja megosztani **A**-val.
2. **B** véletlenszerűen generál egy $1 < K < n$ egész számot, és a publikus kulcsot a titkosító algoritmus paramétereként alkalmazva meghatároz egy cK értéket, majd a cK-t elküldi egy nyilvános csatornán **A**-nak.
3. **A** a privát kulcsot a visszafejtő algoritmus paramétereként alkalmazva meghatároz egy $\hat{K}$ értékét. A helyesen végrehajtott lépéssorozat a $K = \hat{K}$-t eredményezi.
4. A megosztott titkos információ tehát a K lesz.

A gyakorlatban nem az 1977-ben publikált rendszert használják, hanem az RSA-OAEP, illetve az RSA-PSS rendszereket. Az RSA-OAEP az RSA-titkosító, az RSA-PSS az RSA digitális aláírás módosított, biztonságos és standardizált változata, ahol a standard leírását a **RSA PKCS#1 v2.1**-ben találjuk. Mindkettőt Mihir Bellare[25] és Phillip Rogaway[26] dolgozták ki 1995-ben, illetve 1996-ban [3, 4].

A jegyzet keretén belül nem lesz szó se az RSA-OAEP, se az RSA-PSS rendszerről, az 1977-es RSA-titkosító változata kerül bemutatásra, amelyet RSA-textbook vagy baby-RSA néven említ a szakirodalom.

Az RSA-textbook **kulcsgeneráló algoritmusának** bemenete egy k biztonsági paraméter, amely a generált kulcsméretet jelenti és a következőket végzi:

1. Véletlenszerűen generál két különböző $k/2$ bites prímszámot, amelyeket p-vel és q-val jelöl. Meghatározza $n = p \cdot q$-t.

25 Mihir Bellare amerikai informatikus (1962–)
26 Phillip Rogaway amerikai informatikus (1962–)

2. Kiválasztja azt a legkisebb $1 < e < n$ számot, amelyre fennáll $(e, \phi(n)) = 1$, ahol $\phi(n) = (p-1) \cdot (q-1)$.
3. Meghatározza e inverzét $\pmod{\phi(n)}$ szerint, jelöljük ezt d-vel. Fennáll tehát $e \cdot d \equiv 1 \pmod{\boldsymbol{\phi(n)}}$.
4. A publikus kulcs az (e, n), a privát kulcs a (d, n) számpár lesz.

Az n-t modulusnak, az e-t publikus exponensnek, a d-t pedig privát exponensnek is hívják. A modulust, a publikus exponenst bárki ismerheti, ezért ezeket **publikus adatoknak** is hívják. A privát exponens értékét viszont szigorúan titokban kell tartani, és mellette a $p, q, \phi(n)$ értékek is szigorúan titkosak. Vegyük észre, hogy $\phi(n)$ az Euler-függvény értékét jelöli. A privát kulcs értékét ugyanakkor $\pmod{[p-1, q-1]}$ szerint is meg lehet határozni, ahol $[p-1, q-1]$ a legkisebb közös többszöröst jelöli. Az RSA digitális aláírás ezzel a modulussal dolgozik.

Az RSA-textbook **titkosító algoritmus** bemeneti paramétere a publikus kulcs, és a K nyílt-szöveg, ahol K egy egész szám, és fennáll $1 < K < n$. Az algoritmus a $cK \equiv K^e \pmod{\boldsymbol{n}}$ egész számot számítja ki, amelyet titkosított számnak is hívnak.

Az RSA-textbook **visszafejtő algoritmus** bemeneti paramétere a privát kulcs és a titkosított szám. Az eredeti szöveg a következőképpen kerül meghatározásra : $K \equiv cK^d \pmod{\boldsymbol{n}}$.

A rendszer helyessége a **kis Fermat-tétellel** bizonyítható [7]:

Bizonyítás: Abból indulunk ki, hogy $e \cdot d = 1 \pmod{\phi(n)}$. Ekkor létezik egy olyan t egész szám, amelyre

$$e \cdot d = 1 + t \cdot \phi(n).$$

Első körben tegyük fel, hogy $(K, p) = 1$. Ekkor a kis Fermat-tétel alapján $K^{p-1} \equiv 1 \pmod{p}$. Ekkor azonban

$$K^{e \cdot d} \equiv K^{1+t \cdot \phi(n)} \equiv K^{1+t \cdot (p-1) \cdot (q-1)} \equiv K \cdot (K^{p-1})^{t \cdot (q-1)} \equiv K \pmod{p}.$$

Másfelől $1 < K < n$ és ha $(K, p) \neq 1$, akkor a kongruencia mindkét oldala $0 \pmod{p}$ lesz. Igaz tehát, hogy

$$K^{e \cdot d} \equiv K \pmod{\boldsymbol{p}}.$$

Hasonlóan bizonyítható, hogy ha $(K, q) = 1$, akkor

$$K^{e \cdot d} \equiv K \pmod{\boldsymbol{q}}$$

Mivel $n = p \cdot q$ és $p \neq q \Longrightarrow K^{e \cdot d} \equiv K \pmod{n}$.

□

Példa: A kulcsgeneráló algoritmus számításai a $p = 61,\ q = 97$ prímszámok esetén a következő lesz:

1. $n = 61 \cdot 97 = 5917$,
2. Legyen $e = 7$, ahol $(7, 5760) = 1$ és $\phi(5917) = 60 \cdot 96 = 5760$.
3. Meghatározzuk e inverzét $\pmod{5760}$ szerint, kapjuk: $d = 823$. Fennáll tehát $7 \cdot 823 = 1 \pmod{5760}$.
4. A publikus kulcs: $(7,\ 5917)$, a privát kulcs: $(823,\ 5917)$.

A titkosító algoritmus számítása, ha a $K = 2014$ értékét szeretnénk titkosítani/megosztani, a következő: $cK = 2014^{\mathbf{7}} \equiv 1526\ \mathbf{(mod\ 5917)}$.

A visszafejtő algoritmus számítása: $K = 1526^{\mathbf{823}} \equiv 2014\ \mathbf{(mod\ 5917)}$.

6.25. algoritmus. Írjunk egy Python függvény, amely a k pozitív egész szám esetén, amely a generált kulcsméretet jelenti, az RSA-textbook kulcsgeneráló algoritmusával meghatározza az e, d, n, p, q egész számokat.

```
import math
def rsaKeyGen(k):
    p = primeGen(k // 2)
    q = primeGen(k // 2)
    n = p * q
    phi = (p-1) * (q-1)
    e = 3
    while True:
        if math.gcd(e, phi) == 1: break
        e += 2
    #d = modInverse(e, phi)
    d = pow(e, -1, phi)
    return e, d, n, p, q
```

Az algoritmusban meghívásra kerül a **primeGen** (6.15. algoritmus) és adott esetben a **modInverse** (6.9. algoritmus), amely helyett alkalmazható a Python `pow` függvénye.

6.26. algoritmus. Írjunk egy Python függvényt, amely `rsaKeyGen` függvény segítségével meghatároz egy publikus és privát kulcspárt, titkosít egy véletlenszerűen generált bájtszekvenciát, majd a titkosított értéket visszafejti.

```
from random import sample
def rsaFeladat():
    k = int(input('bit meret: '))
    e, d, n, p, q = rsaKeyGen(k)
    print('publikus kulcs: ', e, n)
    print('privat kulcs: ', d, n)

    bajtL = k // 8 - 1
    bajtSz = sample(range(0, 256), bajtL)

    print('nyilt szoveg: ', bajtSz)
    K = int.from_bytes(bajtSz, byteorder = 'big')
    print('eredti ertek: ', K)

    cK = pow(K, e, n)
    print('titkositott ertek: ', cK)
    K = pow(cK, d, n)
    print('visszafejtett ertek:', K)
    bajtSz1 = K.to_bytes(bajtL, byteorder = 'big')
    print('visszafejtett nyilt szoveg: ', list(bajtSz1))

>>> rsaFeladat()
    bit meret: 128
    ...
```

Az algoritmusban a titkosításra szánt bitszekvencia mérete függ a kulcsmérettől, éppen ezért a bájtszekvencia generálása előtt a `bajtL` változóba meghatároztuk a `k` értéke alapján, hogy hány bájtot lehet generálni, a generált bájtokat átalakítva tízes számrendszerbe ugyanis nem kaphatunk nagyobb számot, mint n.

Anélkül, hogy a rendszer biztonságát veszélyeztetnénk, a kínai maradéktétel alkalmazásával javítani lehet a visszafejtési algoritmus időigényén. Első lépésként az n nagyságrendjével megegyező d hatványkitevő helyett két kisebb hatványkitevővel való hatványozást fogunk elvégezni, ahol a két kitevőt dp-vel, illetve dq-val fogjuk jelölni. Ezek nagyságrendje a p, q prímszámok nagyságrendjével fog megegyezni. Mivel a 6.24. tétel alapján igazak a következők:

$$\begin{aligned} dp \equiv d \pmod{p-1} &\iff c^d \equiv c^{dp} \pmod{p} \\ dq \equiv d \pmod{q-1} &\iff c^d \equiv c^{dq} \pmod{q}, \end{aligned}$$

a c^d értékét a következő egyenletrendszer megoldása fogja adni:

$$\begin{aligned} x &\equiv c^{dp} \pmod{p} \\ x &\equiv c^{dq} \pmod{q}. \end{aligned}$$

A fenti egyenletrendszer a kínai maradéktétellel oldható meg a következőképpen:

1. meghatározzuk a dp, dq, q^{-1}, p^{-1} értékeket, ahol q^{-1} a q inverze $\pmod{p}$ szerint, p^{-1} a p inverze $\pmod{q}$ szerint és

$$dp \equiv d \pmod{p-1}, \qquad dq \equiv d \pmod{q-1},$$

2. a $c^d \pmod{n}$ értéket megadja az x értéke, ahol

$$\begin{aligned} x &\equiv (q^{-1} \cdot q \cdot xp + p^{-1} \cdot p \cdot xq) \pmod{n}, \\ xp &\equiv c^{dp} \pmod{p}, \qquad xq \equiv c^{dq} \pmod{q}. \end{aligned}$$

Felmerül a kérdés, hogy megjelenése kezdetétől fogva minek tudható be az RSA népszerűsége? A következő két bekezdés erre ad választ.

A Miller–Rabin-prímteszttel (6.14), illetve a kiterjesztett euklideszi algoritmusokkal (4.10) hatékonyan lehet 1024 bites kulcsokat generálni. A publikus kulcs ismeretében hatékony algoritmussal meg lehet határozni a titkosított értéket, hiszen ez egyetlen moduláris hatványozást jelent. A privát kulcs ismeretében szintén hatékony algoritmussal lehet meghatározni az eredeti egész számot, hiszen ez most is egyetlen moduláris hatványozást jelent. Természetesen a p és a q prímszámok ismeretében kiterjesztett euklideszi algoritmussal meghatározható a privát exponens értéke. A privát kulcs hiányában azonban nem lehet meghatározni az eredeti számot.

Azért, hogy az RSA titkosítás egyértelmű legyen, fontos, hogy a szám amelyet hatványozni akarunk, kisebb legyen, mint az n modulus. Éppen ezért a titkosításra szánt bájtszekvenciának mérete függeni fog a választott modulus méretétől. Ha k bites a modulus, akkor a titkosításra szánt bájtszekvencia nem állhat több mint $k/8$ bájtból.

6.11. RSA-biztonsági problémák

Az RSA standard azt ajánlja, hogy a p és a q értékét egyforma nagyságrendűnek válasszuk, generáljuk őket véletlenszerűen, függetlenül egymástól,

szorzatuk pedig legalább 1024 bites szám legyen. Így biztosítható a faktorizációs algoritmusok sikertelensége. Az e publikus exponensnek a 65537 konstanst ajánlja. Ez az ajánlás több tényező miatt is fontos, az egyik, hogy ha az e értéke kicsi, akkor gyors lesz a rejtjelezési idő. Ha azonban túl kicsi, például $e = 3$, akkor a faktorizációs problémát megkerülve sikeresen feltörhető lesz a rendszer (lásd lentebb).

A második tényező az, hogy ha két különböző rejtjelezéshez ugyanazt a modulust különböző publikus exponensekkel használjuk, akkor az egyforma modulusok problémája (lásd lentebb) veszélyt fog jelenteni a rendszer biztonságára.

Ugyanakkor, ha az e értéke kicsi, akkor a d értéke nagy lesz, ezáltal pedig elkerülhető, hogy Wiener[27] feltörési algoritmusát sikeresen lehessen alkalmazni (lásd lentebb).

Megjegyezzük, hogy a gyakorlatban az első két tényező nem jelent biztonsági kockázatot, mert az RSA-OAEP véletlenszerűsített.

Az RSA-textbook esetében az $\boldsymbol{e = 3}$-as eset lehetővé teszi a rendszer feltörését, meghatározható ugyanis az eredeti K érték, a privát kulcs ismerete nélkül. A feltörési algoritmus a kínai maradék tétel és egy köbgyököt meghatározó eljárás segítségével vezet sikerre. Tegyük fel, hogy a K értéket háromszor titkosítottuk a $(3, n_1)$, $(3, n_2)$, $(3, n_3)$ publikus kulcsokkal és rendre a cK_1, cK_2, cK_3 értékeket kaptuk. Ekkor a következő lineáris kongruenciarendszer adható meg:

$$\begin{aligned} x \equiv K^3 \equiv cK_1 &\equiv \pmod{n_1} \\ x \equiv K^3 \equiv cK_2 &\equiv \pmod{n_2} \\ x \equiv K^3 \equiv cK_3 &\equiv \pmod{n_3}, \end{aligned}$$

amelyet a kínai maradéktételt alkalmazva könnyedén meg tudunk oldani. A kongruenciarendszer eredményeként kapott x értékből ha köbgyököt vonunk, akkor megkapjuk az eredeti K értéket.

Az RSA-textbook esetében az **egyforma modulusok** problémája azt fogja jelenteni, hogy ha a K értéket kétszer rejtjelezzük egyszer az (e, n), másodszor az (f, n) publikus kulcsokkal, és ha fennáll, hogy $(e, f) = 1$, akkor meghatározható a K értéke a privát kulcs ismerete nélkül is a következő lépések végrehajtásával:

1. Legyen cK_1, cK_2 a két rejtjelzett értéke a K-nak.

27 Michael J. Wiener kortárs amerikai kutató

2. A kiterjesztett euklideszi algoritmussal meghatározzuk azokat az x, y egész számokat, amelyekre fennáll: $e \cdot x + f \cdot y = 1$. Ez a két egész szám egyértelműen meghatározható, mert $(e, f) = 1$.
3. A $cK_1^x \cdot cK_2^y \pmod n$ érték a keresett K számot fogja eredményezni, fennáll ugyanis:

$$cK_1^x \cdot cK_2^y = (K^e)^x \cdot (K^f)^y = K^{e \cdot x + f \cdot y} \equiv K \pmod n.$$

4. A kiterjesztett euklideszi algoritmus negatív értéket is számolhat az x vagy y-ban. Ha $x < 0$, akkor szükséges meghatározni a cK_1 inverzét $\pmod n$ szerint, és a $cK_1^x \pmod n$ hatványérték helyett az $inv^{-x} \pmod n$ hatványértékkel kell dolgozni, ahol inv-vel a cK_1 inverzét jelöltük. Hasonlóan kell eljárni, ha $y < 0$.

Példa: Legyen $p = 37, q = 127, e = 11, f = 17$. Ekkor $n = 4699$, és rejtjelezzük a $K = 446$ értéket. Kapjuk:

$$cK_1 = 446^{11} \pmod{4699} = 3158$$
$$cK_2 = 446^{17} \pmod{4699} = 1757$$

A kiterjesztett euklideszi algoritmus meghatározza a $x = -3, y = 2$ értékeket. Mivel $x < 0$ szükséges meghatározni 3158 inverzét, ez 2906 lesz. Az eredeti K értékét pedig a következőképpen kapjuk vissza: $2906^3 \cdot 1757^2 \pmod{4699} = 446$.

Python-implementáció esetében ha a kitevő negatív szám, akkor a moduláris hatványozáskor a `pow` függvény automatikusan az alap inverzével fogja a hatványértéket számolni, ekkor fölösleges a cK_1 vagy cK_2 inverzét kiszámolni. Az előző példánál vett értékekkel számolva a következőket kapjuk:

```
>>> p, q, e, f = 37, 127, 11, 17
>>> n = p * q
>>> cK1, cK2 = 3158, 1757
>>> x, y = -3, 2
>>> (pow(cK1, x, n) * pow(cK2, y, n)) % n
    446
```

Le is ellenőrizhetjük:

```
>>> (pow(2906, -x, n) * pow(cK2, y, n)) % n
    446
```

Az RSA visszafejtési algoritmus időigényének a gyorsítása érdekében ajánlatos lenne, hogy a d nagyságrendje kicsi legyen. 1990-ben azonban Wiener megmutatta, hogy ha $q < p < 2 \cdot q$ és $3 \cdot d < n^{1/4}$, akkor habár 75%-kal csökkenthető a visszafejtés időigénye, lehetségessé válik a modulus faktorizálása [41]!

Wiener feltörési algoritmusa lánctörtek alkalmazásával meghatározza az $\frac{e}{n}$ lánctört egy közelítő értékét, amely segítségével aztán meghatározható $\phi(n)$, és innen az n két szorzótényezője a p és a q [36].

6.29. tétel. Az e, n, t, d egész számok esetében, ha

$$(e, n) = (t, d) = 1, \text{ és } \left|\frac{e}{n} - \frac{t}{d}\right| < \frac{1}{2 \cdot d^2},$$

akkor $\frac{t}{d}$ az $\frac{e}{n}$ egy közelítő értéke lesz.

A tétel bizonyítására nem térünk ki a jegyzet keretén belül, viszont megvizsgáljuk, hogy hogyan alkalmazható Wiener algoritmusánál. Mivel fennáll, hogy $e \cdot d = 1 \pmod{\phi(n)}$, akkor létezik egy olyan t egész szám, amelyre $e \cdot d - t \cdot \phi(n) = 1$. Másfelől a $q < p$ feltételezés alapján $q^2 < p \cdot q = n$ tehát $q < \sqrt{n}$. Ezután a $q < p < 2 \cdot q$ és $3 \cdot d < n^{1/4}$ feltételezéseket alkalmazva felírható:

$$0 < n - \phi(n) = p \cdot q - (p-1) \cdot (q-1) = p + q - 1 < 2 \cdot q + q - 1 < 3 \cdot q < 3 \cdot \sqrt{n}.$$

Ebből következik, hogy:

$$\left|\frac{e}{n} - \frac{t}{d}\right| = \left|\frac{e \cdot d - t \cdot n}{d \cdot n}\right| = \left|\frac{1 + t \cdot (\phi(n) - n)}{d \cdot n}\right| < \frac{3 \cdot t \cdot \sqrt{n}}{d \cdot n} = \frac{3 \cdot t}{d \cdot \sqrt{n}}$$

Mivel $t < d \Longrightarrow 3 \cdot t < 3 \cdot d < n^{1/4}$, tehát:

$$\left|\frac{e}{n} - \frac{t}{d}\right| < \frac{1}{d \cdot n^{1/4}} \Longrightarrow \left|\frac{e}{n} - \frac{t}{d}\right| < \frac{1}{3 \cdot d^2}.$$

Tehát a 6.29. tétel alapján $\frac{t}{d}$ az $\frac{e}{n}$ egy közelítő értéke lesz.

Tulajdonképpen a kiterjesztett euklideszi algoritmust (4.10) vesszük alapul, mert ha abból indulunk ki, hogy az $\frac{e}{n}$ lánctörtként való felírásában a $q_1, q_2, \ldots, q_k$ egész számok szerepelnek, akkor a $\frac{t}{d}$ közelítő tört a következő iteráció során határozható meg, ahol $j \in \{2, 3 \ldots\}$:

$$t_0 = 0, t_1 = 1, d_0 = 1, d_1 = 0,$$
$$t_j = q_j \cdot t_{j-1} + t_{j-2}, \qquad d_j = q_j \cdot d_{j-1} + d_{j-2}$$

Minden egyes iterációnál kiszámoljuk a

$$\phi = \frac{d_j \cdot e - 1}{t_j}$$

értéket, és ha fennáll $d_j \cdot e - 1 \equiv 0 \pmod{t_j}$, akkor feltételezhetjük, hogy $\frac{t_j}{d_j}$ megfelelő közelítő érték lesz. A meghatározott ϕ értéket megpróbálhatjuk felhasználni a következő másodfokú egyenlet megoldásánál:

$$x^2 + (\phi - n - 1) \cdot x + n = 0. \tag{6.6}$$

Ha a fenti egyenlet megoldásakor két egész számot kapunk, akkor leállhatunk az iterációval és megállapíthatjuk, hogy jó volt feltételezésünk, mert a két egész szám a keresett p és q értékek lesznek.

A 6.6. egyenletet úgy kaptuk, hogy az $y = n/x$-t és $n = x \cdot y$-t behelyettesítettük a következő egyenletbe:

$$\phi = (x - 1) \cdot (y - 1).$$

Példa: Wiener feltörési algoritmusát alkalmazva meghatározzuk az $n = 809634279027007741$ összetett szám két prímosztóját, tudva, hogy a publikus exponens $e = 275007677631058567$. A számítások alapján az $\frac{e}{n}$ lánctörtjegyei:

$$[0, 2, 1, 16, 1, 6, 1, 2, 1, 3],$$

a közelítő törtek pedig:

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{3}, \frac{17}{50}, \frac{18}{53}, \frac{125}{368}, \frac{143}{421}, \frac{411}{1210}, \frac{554}{1631}, \frac{2073}{\mathbf{6103}}.$$

A tizedik közelített tört után a meghatározott ϕ értéke:

$$\phi = \frac{6103 \cdot 275007677631058567 - 1}{2073} = 809634277174312800$$

és

$$\phi - n - 1 = -1852694942.$$

Ezek alapján, ha megoldjuk a következő másodfokú egyenletet:

$$x^2 - 1852694942 \cdot x + 809634279027007741 = 0,$$

akkor a következő két megoldást fogjuk kapni:

$$p = 706153561,\ q = 1146541381,$$

amelyek tulajdonképpen az n prímosztói. Tehát sikeres a faktorizáció! A közelítő törtek sorozatában az utolsó tört nevezője a privát exponens értéke lesz. Tehát a faktorizáció mellett a $d = 6103$ és a $\phi(n) = (p-1)\cdot(q-1) = 809634277174312800$ értékeket is sikeresen meghatároztuk. A közelített törtek sorozatában az utolsó tört számlálójának értéke pedig az a t, amelyre igaz, hogy: $e \cdot d - t \cdot \phi(n) = 1$. Ezt le is ellenőrizhetjük a Python shellben:

```
>>> 275007677631058567 * 6103 - 2073 * 809634277174312800
    1
```

6.27. algoritmus. Írjunk egy Python függvényt, amely egy olyan (e, d, n, p, q) RSA kulcsot generál, ahol a generált prímszámokra igaz, hogy: $q < p < 2 \cdot q$, és a privát exponensre pedig: $d < \dfrac{n^{1/4}}{3}$.

Az algoritmusban meghívjuk a `testSmallPrime`-t (6.22. algoritmus) és a `millerRabinT`-t (6.14. algoritmus).

```
import decimal, math
def rsaKeyGenW(k):
    p = primeGen(k)
    q = nextPrimeGen(p)
    n = p * q
    decimal.getcontext().prec = 500
    limit = int(n**decimal.Decimal('0.25')) // 3
    phi = (p-1) * (q-1)
    while True:
        d = random.randint(3, limit)
        if d & 1 == 0: d += 1
        if math.gcd(d, phi) == 1:
            e = pow(d, -1, phi)
            break
    #return e, d, n, p, q
    return e, n

def nextPrimeGen(p):
    P = [3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
    while True:
        q = random.randint(p >> 1 + 1, p - 1)
        if q & 1 == 0: q += 1
        if testSmallPrime(q, P) and millerRabinT(q):
```

```
            return q

>>> rsaKeyGenW(1024)
    ...
```

6.28. algoritmus. Írjunk egy Python függvényt, amely Wiener feltörési algoritmusát alkalmazva megpróbál faktorizálni egy $n = p \cdot q$ értéket, ahol feltételezhetően a generált prímszámokra igaz, hogy: $q < p < 2 \cdot q$, és a privát exponens is teljesíti a következő feltételt: $d < \frac{n^{1/4}}{3}$.

```
import decimal
def wiener(e, n):
    t0, t1, d0, d1 = 0, 1, 1, 0
    eT, nT = e, n
    while True:
        q = e // n
        r = e - q * n
        if r == 0: return -1
        e = n
        n = r
        t = q * t1 + t0
        d = q * d1 + d0
        if t != 0 and (d * eT - 1) % t == 0:
            phi = (d * eT - 1) // t
            temp = -(nT - phi + 1)
            res = mEgyenlet(1, temp, nT)
            if len(res) == 2:
                p, q = res
                p, q = int(p), int(q)
                if p * q == nT: return p, q
        t0, t1, d0, d1 = t1, t, d1, d

def mEgyenlet(a, b, c):
    decimal.getcontext().prec = 500
    if a == 0: return -1
    delta = decimal.Decimal(b*b - 4*a*c)
    if delta < 0: return -1
    if delta >= 0:
        temp = delta.sqrt()
        a1 = 2 * a
        return ((-b + temp) / a1, (-b - temp) / a1)
```

```
>>> wiener(198403851259239217, 724278741192471007)
    (1054815149, 686640443)
>>> e, n = rsaKeyGenW(1024)
>>> p, q = wiener(e, n)
>>> p * q, n
    ...
```

Korábban már használtuk a `solve` metódust, lásd 133. oldal, ez most is alkalmazható. A `res = mEgyenlet(1, temp, nT)` sor helyett írjuk a következőket:

```
x = sympy.Symbol('x', integer = True)
res = sympy.solve(x**2 + temp*x + nT, x)
```

és ne felejtsük el importálni a `sympy`-t.

6.12. Egész számok faktorizációja

Az egész számok faktorizációjával, azaz egy egész szám prímtényezőinek meghatározásával az ókori görögöktől kezdve mind a mai napig számos kutató intenzíven foglalkozott. Nagy, legalább 1024 bites számok esetében, azonban a faktorizációs problémára máig sem ismert hatékony algoritmus. Nagyobb számok faktorizációját a számítógépek megjelenése tette lehetővé, és azóta is komoly figyelem övezi a faktorizációs algoritmusokat. Például az RSA factoring Challenge 1991 óta sokáig pénzösszeggel jutalmazta azokat, akik bizonyos nagy számokat faktorizáltak:

- a 768 bites RSA768 szám faktorizálásáért 2009-ben Thorsten Kleinjung és csapata 50,000 $-kapott.

Peter Shor[28] 1994-ben bemutatott egy hatékony algoritmust, amely elméletben, kvantumszámítógépek számításaira alapozva képes nagy számokat is faktorizálni [31]. Ennek gyakorlatba ültetése azonban még várat magára, hiszen a legújabb, 2022-es eredmények szerint is csak 48 bites egész számokat tudott így faktorizálni egy nagyobb kínai kutató csapat [42].

A faktorizációs algoritmusok két csoportra oszthatók. A **speciális célú** algoritmusokat speciális alakú számok esetében alkalmazzák:

- Fermat faktorizációs módszere olyan $n = p \cdot q$ összetett szám fakorizációját oldja meg sikeresen, ahol p, q prímszámok és a p és q között kicsi a különbség.

28 Peter Williston Shor amerikai matematikus (1959–)

– Pollard[29]-ρ faktorizációs módszere egy összetett szám kis osztóit tudja meghatározni.
– Pollard $p-1$ módszere egy n összetett számnak azon p osztóit tudja meghatározni, amelyre igaz, hogy $p-1$ összes prímosztója kisebb vagy egyenlő, mint egy B egész szám.
– Lenstra módszere, egy elliptikus görbéken alapuló faktorizációs eljárás a leggyorsabb a speciális célú algoritmusok között.

Az **általános célú** algoritmusok nagy memória- és tárkapacitást igényelnek, és akkor szokták őket alkalmazni, ha a speciális alakú számok faktorizációs algoritmusai nem adnak eredményt. Ilyen algoritmusok:

– a kvadratikus szita módszer (quadratic sieve),
– az általánosított szám test szita módszer (general number field sieve) stb.

A fejezet hátralevő részében két speciális célú faktorizációs algoritmust mutatunk be.

A **Fermat faktorizációs** módszer esetében feltételezzük, hogy $n = p \cdot q$, és a p és q prímszámok közel vannak egymáshoz [19]. Az algoritmus megpróbálja felírni n-et $a^2 - b^2$ alakba, ahol a, b tetszőleges egész számok. Ha ez sikerül, akkor a $p = a - b$, és $q = a + b$ lesznek a keresett értékek, hiszen $a^2 - b^2 = (a-b) \cdot (a+b)$. Ahhoz, hogy az algoritmus ezt megvalósítsa először meghatározza n négyzetgyökének felső egészrészét, jelöljük ezt a-val. A továbbiakban az $a, a+1, a+2, \ldots$ értékekre rendre meghatározza a $b1 = a^2 - n$ értéket, mindaddig, amíg $b1$ nem lesz négyzetszám. Ekkor pedig meghatározható n két osztója: $a - \sqrt{b1}$ és $a + \sqrt{b1}$.

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

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

$\sqrt{441} = 21$ tehát a keresett két osztó:

$$p = 82 - 21 = 61, \qquad q = 82 + 21 = 103.$$

29 John M. Pollard brit matematikus (1941–)

6.29. algoritmus. Írjunk egy Python függvényt, amely Fermat faktorizációs módszerével meghatározza egy összetett szám két osztóját.

```
import decimal
def fermat(n):
    decimal.getcontext().prec = 400
    a = int(decimal.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

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

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

Azért, hogy egy 1024 bites szám esetében is pontos eredményt kapjunk, szükséges a négyzetgyök meghatározásakor legalább 400 tizedesjeggyel számolni, illetve a `Decimal` modul `sqrt` metódusát alkalmazni.

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

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

```
        if b != -1: return (a - b, a + b)
        a += 1

def fermatFeladat(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 = fermatTime(elem)
        if res == -1: print('TIME LIMIT')
        else: print(res)
        print()

>>> fermatFeladat()
    ...
```

A **Pollard-ρ** faktorizációs módszert John Pollard publikálta 1975-ben [22]. Az algoritmus azt a p számot próbálja megkeresni, amely az `n` osztója lehet. Működése a következő észrevételen alapszik: ha a, b, két egész szám, amelyre $0 < a, b < n, a \neq b$ és $a \equiv b \pmod{p}$, akkor $a-b$ a p egy többszöröse lesz. Ekkor $p \leq (a-b, n) < n$, azaz $a-b$ és n legnagyobb közös osztója egy nem triviális osztója lesz n-nek.

Az algoritmusban az $f(x_i) = x_{i-1}^2 + 1 \pmod{n}$ függvénnyel egy számsorozatot generálunk, ahol $x_0 = 1$. Az előállított számsorozatban megvizsgáljuk, hogy az $a = x_i$ és $b = x_j$-re mikor áll fenn, hogy $(a - b, n) \neq 1$. Ekkor $a - b$ és n legnagyobb közös osztója egy nem triviális osztója lesz n-nek. Ennek érdekében az a kezdőértékét 1-re, a b kezdőértékét pedig 2-re állítjuk, és a hatékonyság miatt az $a \equiv a^2 + 1 \pmod{n}$ műveletet egyszer, a $b \equiv b^2 + 1 \pmod{n}$ műveletet pedig kétszer hajtjuk végre egy-egy ciklus során.

Példa: Határozzuk meg $n = 221$ két osztóját, a Pollard-ρ féle faktorizációs módszerrel:

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

tehát a kerestt két osztó:

$$p = 13, \qquad q = n/13 = 17.$$

6.31. algoritmus. Írjunk egy Python függvényt, amely a Pollard-ρ faktorizációs módszerrel meghatározza egy összetett szám két osztóját.

```
import math
def pollardRho(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 = math.gcd (a-b, n)
        #print("%7i%7i%7i%7i" % (a, b, a-b, d))
        if 1 < d and d < n: return (d, n//d)
        if d == n: return n

>>> pollardRho(149063950693785473206387643)
    (10223, 14581233560968939959541)
```

6.13. A Rabin-kriptorendszer

A Rabin-kriptorendszert 1979-ben publikálta Michael O. Rabin [24]. Biztonsága a faktorizációs és kvadratikus maradék feltételezéseken alapszik, (lásd a 6.15. és 6.10. értelmezések). Megmutatható, hogy a Rabin-kriptorendszer feltörése ugyanolyan nehézségű, mint a faktorizációs probléma, ez nem igaz az RSA-ra.

A Rabin-kriptorendszer is egész számokkal végez aritmetikai műveleteket, amelyeket az adatküldés miatt bájtszekvenciákká kell alakítani, illetve vissza is kell alakítani a bájtszekvenciákat egész számokká.

A Rabin-kriptorendszer esetében is megkülönböztetjük a titkosító és a digitális aláírási rendszert. A Rabin-titkosító is kis méretű titkos információ megosztására alkalmas, hatékonyabb, mint az RSA, mégsem használják olyan széles körben, elsősorban azért, mert a visszafejtés eredménye négy rejtjelezett értéket ad, amelyek között a különbségtevés csak további módszerek alkalmazásával lehetséges. A gyakorlatban titkosítónak az SAEP változatát használják, amely biztonságosabb, és amelyben a visszafejtett

értékek közötti különbségtétel problémája is meg van oldva. A SAEP-t Dan Boneh[30] publikálta 2001-ben. A jegyzet keretén belül először a Rabin-titkosítót, majd a Rabin digitális aláírási rendszert mutatjuk be.

A **Rabin-titkosító** esetében szintén három algoritmust kell megadni.

A **kulcsgeneráló algoritmus** bemenete egy k biztonsági paraméter, a generált kulcsméret. Az algoritmus véletlenszerűen generál két $k/2$ bites prímszámot, legyenek ezek p és q, ahol $p \equiv q \equiv 3 \pmod 4$, majd meghatározza az $n = p \cdot q$-t. A publikus kulcs (n), a privát kulcs (p, q) lesz.

A **titkosító algoritmus** bemeneti paramétere a publikus kulcs, az n és a K egész szám. Egyetlen moduláris négyzetre emelést végez, meghatározza a $cK \equiv K^2 \pmod n$ értéket.

A **visszafejtő algoritmus** bemeneti paramétere a privát kulcs a (p, q) és a titkosított szöveg. Meghatározza a cK négyzetgyökét, azaz a $K \equiv cK^{\frac{1}{2}} \pmod n$ értéket. A négyzetgyök meghatározása tulajdonképpen négy érték meghatározását fogja jelenteni, ahogyan azt a 6.20. tételben bemutattuk:

- meghatározzuk p inverzét $\pmod q$ szerint és q inverzét $\pmod p$ szerint, jelöljük őket rendre p^{-1}-el, illetve q^{-1}-gyel,
- kiszámítjuk a következő hatványértékeket:

$$k_p = cK^{(p+1)/4} \pmod p, \quad k_q = cK^{(q+1)/4} \pmod q,$$

- a következő négy megoldás valamelyike lesz a K:

$$\begin{aligned} K_1 &= (k_p \cdot q \cdot q^{-1} + k_q \cdot p \cdot p^{-1}) \pmod n, \\ K_2 &= (k_p \cdot q \cdot q^{-1} - k_q \cdot p \cdot p^{-1}) \pmod n, \\ K_3 &= n - K_1, \\ K_4 &= n - K_2. \end{aligned}$$

A **digitális aláírás** is a publikus kulcsú kriptográfiához tartozik. A rendszerben három algoritmust kell megadni a kulcsgeneráló, az aláíró és az ellenőrző algoritmusokat. A kulcsgeneráló algoritmus most is egy-egy értékpárt, pontosabban egy-egy kulcspárt határoz meg, ahol az egyik értéket **publikus kulcsnak**, a másik értéket **privát kulcsnak** hívjuk. A titkosító rendszerekhez képest azonban a kulcsoknak más lesz a szerepe. Először a privát kulcs kerül alkalmazásra, mert az aláíró egység (számítógép, mobileszköz, weboldal stb.) a privát kulcsot használva fog aláírni/hitelesíteni egy bájtszekvenciát. A hitelesített érték valódiságát a publikus kulcsot alkalmazva lehet ellenőrizni, amelyet a későbbiek során bármilyen más egység megtehet.

30 Dan Boneh izraeli-amerikai kriptográfus (1969–)

A gyakorlatban, például a TLS 1.3 kripto protokollban egy szerver publikus adatainak a valódiságát ellenőrzi egy kliens gép, ezáltal pedig maga a szerver hitelesíthető digitális aláírás alkalmazásával. A digitális aláírást alkalmazzák még adatintegritás ellenőrzésére, üzenet letagadhatatlanságra is.

Feltételezve, hogy a kommunikációban részt vevő két egység **A** és **B**, akkor a digitális aláírás protokollja a következő lépésekből áll:

1. **A** vagy egy harmadik megbízható egység végrehajtja a **kulcsgeneráló** algoritmust, majd a publikus kulcsot egy nyilvános csatornán megosztja a **B** egységgel. Ha egy harmadik egység végezte a kulcsgenerálást, akkor a publikus kulcsot egy nyilvános csatornán, a privát kulcsot pedig egy titkos csatornán fogja megosztani **A**-val.
2. Az **A** egység a privát kulcsot használva az **aláíró algoritmussal** aláírja az $1 < K < n$ egész számot, tulajdonképpen meghatároz egy s egész számot. A K és az s értékét egy nyilvános csatornán keresztül elküldi **B**-nek.
3. A **B** egység a publikus kulcsot használva az **ellenőrző algoritmussal** ellenőrzi, hogy s a K aláírt értéke vagy sem, tulajdonképpen meghatároz egy $\hat{K}$ értéket és ellenőrzi, hogy az egyenlő-e K-val.
4. Ha $K = \hat{K}$, akkor az s aláírás hiteles.

Digitális aláírás előállítására több rendszer is létezik, a következőkben a **Rabin digitális aláírás** egy módosított változatát mutatjuk be, de előtte a másodfokú kongruenciák néhány idetartozó tulajdonságát is szükséges tárgyalni [22].

Legyenek $p \equiv q \equiv 3 \pmod 4$ különböző prímek, és legyen $n = p \cdot q$, ekkor bizonyítás nélkül jelentjük ki, hogy fennáll:

1. Ha $(a, n) = 1$, akkor $a^{(p-1)\cdot(q-1)/2} \equiv 1 \pmod n$.
2. Ha a kvadratikus maradék $\pmod n$ szerint, akkor $a^{(n-p-q+5)/8} \pmod n$ az a egy négyzetgyöke lesz $\pmod n$ szerint.
3. Ha $\left(\frac{a}{n}\right) = 1$, és $d = (n-p-q+5)/8$, akkor

$$a^{2d} \pmod n = \begin{cases} a, & \text{ha } a \text{ kvadratikus maradék,} \\ n-a, & \text{ha } a \text{ kvadratikus nemmaradék.} \end{cases}$$

4. Ha $p \not\equiv q \pmod 8$, akkor $\left(\frac{2}{n}\right) = -1$, ezért ha az a-t megszorozzuk 2-vel vagy 2 inverzével $\pmod n$ szerint, akkor a Jacobi-szimbólum előjelet vált.

A Rabin digitális aláírás során három algoritmust kell megadni, a kulcsgeneráló, az aláírás-előállító és az aláírás-ellenőrző algoritmusokat. A kulcsgenerálás során Williams-egészeket generálunk: a $p \equiv q \equiv 3 \pmod 4$ és $p \not\equiv q \pmod 8$ tulajdonsággal rendelkező prímszámokat Williams-egészeknek hívjuk.

A **kulcsgeneráló algoritmus** bemenete egy k biztonsági paraméter, amely a generált kulcsméretet jelenti. A következő lépessorozatból áll:

1. Véletlenszerűen generál két $k/2$ bites prímszámot, legyenek ezek p és q, ahol $p \equiv 3 \pmod 8, q \equiv 7 \pmod 8$. Meghatározza az $n = p \cdot q$-t.
2. A publikus kulcs (n), a privát kulcs (p, q).

Az **aláíró algoritmus** a $K \in \{2, \ldots, (n-6)/16\}$ egész számot írja alá. Ennek érdekében először meghatározza $\hat{K} = 16 \cdot K + 6$-t, a $\left(\frac{\hat{K}}{n}\right)$ Jacobi-szimbólumot, majd az s aláírást:

$$s = \begin{cases} \hat{K}^d \pmod n, & \text{ha } \left(\frac{\hat{K}}{n}\right) = 1 \\ (\hat{K}/2)^d \pmod n, & \text{ha } \left(\frac{\hat{K}}{n}\right) = -1 \end{cases}$$

Az **ellenőrző algoritmus** ellenőrzi, hogy az s a K aláírása-e. Ennek érdekében meghatározza a $\hat{K_1} = s^2 \pmod n$, majd a $\hat{K}$-t aszerint, hogy melyik feltétel teljesül a következők közül:

$$\hat{K} = \begin{cases} \hat{K_1}, & \text{ha } \hat{K_1} \equiv 6 \pmod 8 \\ 2 \cdot \hat{K_1}, & \text{ha } \hat{K_1} \equiv 3 \pmod 8 \\ n - \hat{K_1}, & \text{ha } \hat{K_1} \equiv 7 \pmod 8 \\ 2 \cdot (n - \hat{K_1}), & \text{ha } \hat{K_1} \equiv 2 \pmod 8 \end{cases}$$

Az aláírást akkor fogadja el hitelesnek, ha $K = (\hat{K} - 6)/16$.

6.32. algoritmus. Írjunk egy Python függvényt, amely egy k egész szám bemeneti érték esetében a Rabin-kulcsgeneráló algoritmusával meghatározza az n, p, q egész számokat, ahol p, q Williams-egészek.

```
import random
def primeGenWilliams(k):
```

```
    while True:
        p = random.getrandbits(k // 2)
        if p % 8 == 3 and millerRabinT(p): break
    while True:
        q = random.getrandbits(k // 2)
        if q % 8 == 7 and millerRabinT(q): break
    return p, q

def rabinKeyGen(k):
    p, q = primeGenWilliams(k)
    n = p * q
    return n, p, q

>>> rabinKeyGen(16)
    (2677933, 4139, 647)
>>> rabinKeyGen(1024)
    ...
```

6.33. algoritmus. Írjunk egy Python programot, amely `rabinKeyGen` függvény segítségével meghatároz egy publikus és privát kulcspárt, digitális aláírással előállítja egy véletlenszerűen generált egész szám aláírását, majd ellenőrzi az aláírt értéket.

A `rabinSign` aláíró algoritmusban meghívásra kerül a Jacobi-szimbólumot meghatározó `Jacobi` függvény (6.11 algoritmus). Az ellenőrző algoritmus a `rabinVerify` lesz. Mindkét algoritmus a `rabinFeladat`-ban kerül meghívásra, amelyben a publikus és privát kulcsokat a `rabinKeyGen` függvénnyel (6.32 algoritmus) generáltuk.

```
def rabinSign(K, n, p, q):
    d = (n - p - q + 5) // 8
    kK = 16 * K + 6
    j = Jacobi(kK, n)
    if j == 1: s = pow(kK, d, n)
    else: s = pow(kK // 2, d, n)
    return s

def rabinVerify(s, n):
    kK1 = (s * s) % n
    temp = kK1 % 8
    if temp == 6: kK = kK1
    elif temp == 3: kK = 2 * kK1
    elif temp == 7: kK = n - kK1
```

```
    elif temp == 2: kK = 2 * (n - kK1)
    return (kK - 6) // 16

import random
def rabinFeladat(k = 128):
    n, p, q = rabinKeyGen(k)
    print('n: ', n)
    #K = int(input('K: '))
    while True:
        K = random.randint(2, (n-6) // 16)
        if K % 16 == 6: break
    print('K: ', K)
    s = rabinSign(K, n, p, q)
    print('s: ', s)
    K1 = rabinVerify(s, n)
    print('K: ', K1)
    if K1 == K: return True
    return False

>>> rabinFeladat(1024)
    ...
```

6.14. Álvéletlenszám-generátorok

Véletlen, illetve álvéletlen módon generált számokra számos algoritmusban szükség van. Véletlen módszerrel generált számokat (angolul: true random numbers) hardvereszközökkel állítanak elő, erre szoftver szintjén nincs is lehetőség. Az álvéletlen módon generált számokat (angolul: pseudo random numbers) szoftvereszközökkel hozzák létre [17].

Annak a megállapítása, hogy egy adott számsorozat valóban véletlenszerű, nem is olyan egyszerű. Álltalában két vagy három kritérium szerint vizsgálják ezt. Az első kritérium, hogy egyenletes eloszlást (**uniform distribution**) követnek-e a generált számok vagy sem. Például ha a generált értékek bitek, akkor egy fontos tulajdonság, hogy a generált egyes és nullás értékek előfordulása között ne legyen nagy az eltérés. Ennek megállapítását statisztikai tesztek alkalmazásával végzik. A második két kritérium vizsgálata már nem olyan egyszerű. A függetlenség (**independency**) azt követeli meg, hogy a generált sorozatban minden részsorozat független legyen a sorozat többi értékétől. A harmadik kritérium a kiszámíthatatlanság (**unpredictability**)

pedig azt jelenti, hogy egy támadó a generált sorozat értékei alapján nem tudja megjósolni, hogy mik lesznek a sorozat következő elemei, adott esetben nem tudja kiszámítani, hogy mik voltak a már kigenerált értékek.

Minden programozási nyelv rendelkezik álvéletlenszámokat generáló, algoritmusokkal, röviden PRNG-ékkel. A legtöbb mérnöki számítás során elégséges az olyan sorozatok használata, amelyek véletlenszerűek. Ezek azonban nem alkalmasak kriptográfiai protokollokban. A kriptográfiában *erős* algoritmusokat kell használni, ami azt jelenti, hogy a pszeudorandom számsorozat mind a véletlenszerűségnek, mind a független és a kiszámíthatatlan kritériumnak is eleget kell tegyen. A gyakorlatban a Diffie–Hellman-, az RSA-, a Rabin-rendszerek által kezelt üzeneteket kiegészítik pszeudorandom módon generált bitekkel, amelyekhez mindig *erős* PRNG-t használnak. A publikus, a privát kulcsok generálásakor is fontos az erős PRNG-k használata. A PRNG-k tehát kulcsszerepet játszanak a különböző rendszerek biztonságában. Ugyanakkor a kriptográfiában használt PRNG-k heurisztikus alapon szerkesztettek, mert gyorsak is kell legyenek. Ezekre azonban nem térünk ki.

A jegyzet keretén belül három PRNG-t fogunk bemutatni:

- a lineáris kongruencián alapuló generátort,
- a közép-négyzet módszert (middle-square),
- a Blum-Blum-Shub generátort, amely *erős* PRNG, de ugyanakkor lassú, gyakorlatban csak nagyon erős biztonságot igénylő rendszerek esetében használják.

A **lineáris kongruencián** alapuló PRNG nagyon gyors, kis memóriaigényű, jó statisztikai viselkedés jellemzi, ezért olyankor lehet alkalmazni, amikor nem fontos a rendszer biztonsága [17]. Egy egyszerű matematikai összefüggést alkalmaz:

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

Az m modulus, az a szorzó, a b növekedési érték és a z_0 kezdeti érték (angolul: seed) konstans értékek, kiválasztásuk a generátor periódusának nagyságát határozza meg.

Példa: Legyen $z_{i+1} = (z_i + 7) \pmod{10}$ és $z_0 = 3$. Ekkor $a = 1, b = 7, m = 10$ és a generált számsorozat a következő:

$$0\ 7\ 4\ 1\ 8\ 5\ 2\ 9\ 6\ 3\ \mathbf{0\ 7\ 4\ 1}...$$

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

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

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

>>> linGenFeladat(15, 3)
    0 7 4 1 8 5 2 9 6 3 0 7 4 1 8
>>> linGenFeladat(20)
    ...
```

6.6. tulajdonság. A lineáris kongruencián alapuló generátor periódusa akkor és csakis akkor lesz maximális, ha:

- m és b relatív prímek,
- $a-1$ osztható m minden prímosztójával,
- $a-1$ osztható 4-gyel, ha m is osztható 4-gyel.

A következő táblázat a különböző programozási nyelvek esetében a generátor konstansainak a tipikus értékeit mutatja:

	m	a	$a-1$	b
`ANSI C`	2^{31}	1103515245	$2^2 \cdot 13^2 \cdot 613 \cdot 2663$	12345
`C/C++`	2^{31}	214013	$2^2 \cdot 53503$	2531011
`Java`	2^{48}	25214903917	$2^2 \cdot 3 \cdot 757 \cdot 787 \cdot 3527$	11

A **közép-négyzet módszert** (middle-square) Neumann János[31] alkalmazta álvéletlen számok generálására. Egyike az első olyan módszereknek, amely aritmetikai számítások alkalmazásával generált pszeudorandom számokat. Bemutatására elsősorban az érdekessége, nem pedig a használhatósága miatt kerül sor, a generált számsorozat periódusa ugyanis rövid [17].

Az algoritmus szerint egy n számjegyű nr kezdőértékből kiindulva a következő lépéseket kell többször ismételni:

- a nr számot négyzetre emeljük, jelöljük ezt nr_1-gyel,

31 Neumann János magyar-amerikai matematikus (1903–1957)

– a nr_1 számot szükség esetén balról nullásokkal egészítjük ki azért, hogy a nr_1 szám középső n számjegyét meg tudjuk határozni,
– az így kapott szám középső n számjegye lesz az új nr.

Az algoritmus kimenete a műveletsor tetszőleges számú ismétlése után kapott szám.

Példa: Legyen a kezdőérték $nr = 980$, ahol az ötödik négyzetre emelés eredményét adjuk meg kimenetként:

$$\begin{array}{rcl} nr & nr_1 & \\ 980^2 = & 09\mathbf{604}00 & 604 \\ 604^2 = & 03\mathbf{648}16 & 648 \\ 648^2 = & 04\mathbf{199}04 & 199 \\ 199^2 = & \mathbf{3960}1 & 960 \\ 960^2 = & 09\mathbf{216}00 & 216 \\ \Rightarrow 216 & & \end{array}$$

6.35. algoritmus. Írjunk egy Python függvényt, amely egy nr kezdőérték esetén meghatározza a közép-négyzet módszer által generált számsorozat n-edik elemét.

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

>>> midSqr(980, 5)
    216
```

6.36. algoritmus. Írjunk egy Python függvényt, amely meghatározza egy nr kezdőérték esetén a közép-négyzet módszer által generált számsorozat periódusát.

```
def midSqrPeriodus(nr):
    k = len(str(nr))
    L = []
    while len(L) == len(set(L)):
```

```
        x = str(nr * nr)
        y = (len(x) - k) // 2
        nr = int(x [y: y + k])
        L += [nr]
    print(L)

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

A **Blum–Blum–Shub** PRNG-t vagy BBS-generátort 1986-ban publikálták a szerzők, L. C. Blum[32], M. Blum[33] és M. Shub[34] [5]. Alkalmas kriptográfiai rendszerekben, biztonságát a faktorizációs és kvadratikus maradék feltételezés adja (6.15. és, 6.10. értelmezések). Feltételezik, hogy feltörése ekvivalens a faktorizációs probléma megoldásával. Nagy számításigényű, mert minimum 1024 bites számokon végez aritmetikai műveleteket. Az algoritmus nagyon egyszerű, moduláris négyzetre emelést és 2-vel való osztási maradékot számolva hozza létre a $(b_1, b_2, \ldots b_n)$ bitsorozatot:

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

u_0 a generátor egy kezdeti értéke, amelynek meghatározása érdekében a következő lépéseket kell végrehajtani:

1. válasszuk ki a P, Q prímeket úgy, hogy $P = 2 \cdot p + 1$, $Q = 2 \cdot q + 1$, ahol p, q is prímek, és $N = P \cdot Q$,
2. válasszunk egy tetszőleges r értéket a $\{2, 3, \ldots, N-1\}$ halmazból úgy, hogy $(r, N) = 1$,
3. legyen $u_0 = r^2 \pmod N$.

A generált P, Q értékek tehát biztonságos prímek lesznek, amelyekre fennáll $P \equiv Q \equiv 3 \pmod 4$. Ezeket a prímeket Blum-egészeknek is hívják.

Példa: Legyen $P = 23, Q = 59$ két biztonságos prím, ekkor $N = 1357$ és legyen $r = 3, u_0 = 9$. Az algoritmus által generált első 8 bit az utolsó

32 Leonore Carol Blum amerikai informatikus (1942–)
33 Manuel Blum venezuelai származású amerikai informatikus (1938–)
34 Michael Shub amerikai matematikus (1943–)

oszlopban található, és a következőképpen kerül meghatározásra:

i	u_{i-1}	u_i	b_i
1	9	$9^2 \equiv 81 \pmod{1357}$	1
2	81	$81^2 \equiv 1133 \pmod{1357}$	1
3	1133	$1133^2 \equiv 1324 \pmod{1357}$	0
4	1324	$1324^2 \equiv 1089 \pmod{1357}$	1
5	1089	$1089^2 \equiv 1260 \pmod{1357}$	0
6	1260	$1260^2 \equiv 1267 \pmod{1357}$	1
7	1267	$1267^2 \equiv 1315 \pmod{1357}$	1
8	1315	$1315^2 \equiv 407 \pmod{1357}$	1

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

Említettük, hogy a BBS generátor biztonsága akkor megfelelő, ha N legalább 1024 bites, ekkor azonban a hatékonyság növelése érdekében, anélkül hogy veszítene a rendszer a biztonságából, egy négyzetre emelés alkalmával egyszerre 8 bitet, azaz 1 bájtot is meghatározhatunk. A gyakorlatban ez azt fogja jelenteni, hogy az $u_i \pmod 2$ helyett az $u_i \pmod{256}$ műveletet fogjuk alkalmazni.

6.37. algoritmus. Írjunk egy Python programot a következő menüpontok segítségével:

- `kilépés`: kilép a programból,
- `prímszám generálás/mentés`: meghatározza a BBS generátorhoz szükséges P, Q prímszámokat, és kiírja a prímszámok Base64-es alakját egy állományba,
- `prímszám beolvasás`: beolvassa egy megadott állományból a P, Q prímszámokat,
- `BBS generátor`: megadott P, Q prímszámokat alkalmazva a BBS generátorral tetszőleges hosszúságú számsorozatot generál.

A programban szükségünk lesz biztonságos prímekre, amelyeket a `safePrimeGen` függvény (6.22. algoritmus) meghívásával generálhatunk. Gondoskodni kell tehát a függvény importálásáról, vagy a függvényt át kell másolni az aktuális állományba.

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

```
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 = ' ')

def primeRead(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 primeWrite(k, nev):
    P = safePrimeGen(k//2)
    Q = safePrimeGen(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

def bbsFeladat():
   P, Q = None, None
   while True:
      x = input('''
            0- kilépés
            1- prímszám generálás/mentés
            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 generalt számsorozat: ')
        bbs256(P, Q, l)
```

6.15. Mersenne-számok, Mersenne-prímek

A Python véletlenszám-generátorát **Mersenne Twister**nek (megbízhatatlan Mersenne) hívják, amelyet 1997-ben publikált Takuji Nishimura[35] és Makoto Matsumoto[36]. Periódusa a $2^{19937} - 1$ Mersenne[37]-prím, és egy 53-bites valós számot állít elő. A hátterében lineáris algebra, véges testek és bináris testek elmélete áll. Nem használható kriptográfiai protokollokban. A jegyzet keretén belül csak az algoritmusban használt Mersenne-számok tulajdonságaira térünk ki.

A Mersenne-prímek azok a prímszámok, amelyek $2^p - 1$ alakúak. A leghatékonyabb módszer annak az eldöntésére, hogy egy Mersenne-szám prímszám, ha alkalmazzuk a **Lucas[38]–Lehmer[39]-prímtesztet** [33]. Mivel ezzel a teszttel viszonylag egyszerű megvizsgálni, hogy egy $2^p - 1$ alakú szám prímszám-e, ezért a valaha meghatározott legnagyobb prímszámok legtöbbje Mersenne-prím. 2018 óta a legnagyobb ismert prímszám a $2^{82,589,933} - 1$ Mersenne-prím.

6.7. tulajdonság. Egy $2^p - 1$ alakú számról a következő tulajdonságok állapíthatók meg:

35 Takuji Nishimura kortárs japán kutató
36 Makoto Matsumoto kortárs japán kutató
37 Marin Mersenne francia szerzetes, matematikus, fizikus (1588–1648)
38 François Édouard Anatole Lucas francia matematikus (1842–1891)
39 Derrick Henry Lehmer amerikai matematikus (1905–1991)

1. ha p összetett, akkor $2^p - 1$ is összetett,
2. ha p prímszám, akkor $2^p - 1$ lehet összetett is, és prímszám is.

A **Mersenne-prímek** tehát azok a $2^p - 1$ alakú prímszámok, ahol p is prímszám. A **Mersenne-számok** pedig azok a $2^p - 1$ alakú számok, amelyek esetében p prímszám, de $2^p - 1$ nem prímszám. A matematika máig sem tudta bizonyítani, hogy a Mersenne-prímek száma végtelen vagy sem.

Példák:

1. A következő számok Mersenne-prímek:

$$2^2 - 1 = 3, \qquad 2^3 - 1 = 7,$$
$$2^{19} - 1 = 524287, \quad 2^{31} - 1 = 2147483647.$$

2. A következő számok Mersenne-számok, de nem Mersenne-prímek:

$$2^{11} - 1 = 2047 = 23 \cdot 89$$
$$2^{23} - 1 = 8388607 = 47 \cdot 178481$$

6.8. tulajdonság. (Lucas–Lehmer-prímteszt) Legyen p egy páratlan prímszám, $r_1 = 4$, és jelölje M_p a p-dik Mersenne-számot. Generáljuk ki a következő összefüggéssel az r_k sorozat első $p - 1$ elemét:

$$r_k \equiv r_{k-1}^2 - 2 \pmod{M_p}, \text{ha } k > 1.$$

Az $M_p = 2^p - 1$ akkor és csakis akkor prímszám ha a

$$r_{p-1} \equiv 0 \pmod{M_p},$$

Példák:

1. Legyen $p = 11$ és $M_{11} = 2047$, ekkor az r_k sorozat a következő:

k	r_k	k	r_k
1	4	6	$701^2 - 2 \equiv 119$
2	$16^2 - 2 = 14$	7	$119^2 - 2 \equiv 1877$
3	$14^2 - 2 = 194$	8	$1877^2 - 2 \equiv 240$
4	$194^2 - 2 \equiv 788 \pmod{2047}$	9	$240^2 - 2 \equiv 282$
5	$788^2 - 2 \equiv 701 \pmod{2047}$	10	$282^2 - 2 \equiv 1736.$

A sorozat $p - 1 = 10$-dik tagja $1736 \equiv 282^2 - 2 \pmod{2047} \not\equiv 0$, tehát $Mp = 2047$ nem Mersenne-prím.

2. Legyen $p = 13$ és $M_{13} = 8191$, ekkor az r_k sorozat a következő:

$$4, 14, 194, 4870, 3953, 5970, 1857, 36, 1294, 3470, 128, 0.$$

A sorozat $p - 1 = 12$-dik tagja 0, mert $128^2 - 2 \equiv 0 \pmod{8191}$ tehát $M_{13} = 8191$ Mersenne-prím.

A Mersenne-prímek szorosan kapcsolódnak a tökéletes számokhoz. Egy természetes számot akkor mondunk tökéletesnek, ha a valódi osztóinak összege megegyezik a számmal, a valódi osztók közé pedig a szám maga nem tartozik bele. A matematika nem ismer páratlan tökéletes számot, és azt sem tudta még bebizonyítani, hogy a páros tökéletes számok száma végtelen.

Példa: A következő számok tökéletes számok:
6, amelynek valódi osztói: 1, 2, 3, és összegük 6.
28, amelynek valódi osztói: 1, 2, 4, 7, 14, és összegük 28.

A következő tétel alapján kijelenthető, hogy egy páros tökéletes szám megtalálása ugyanazt jelenti, mint egy Mersenne-prím megtalálása [33].

6.30. tétel. Egy páratlan természetes szám akkor és csakis akkor tökéletes, ha a következő alakú: $2^{p-1} \cdot (2^p - 1)$, ahol $2^p - 1$ prímszám, azaz $2^p - 1$ Mersenne-prím.

Példa:

$p = 2$ - re: $2^{2-1} \cdot (2^2 - 1) = 2 \cdot 3 = 6$
$p = 3$ - ra: $2^{3-1} \cdot (2^3 - 1) = 4 \cdot 7 = 28$.

6.16. Kitűzött feladatok

6.1. feladat. Alkalmazva az osztási próba módszerét, illetve Eratosztenész szitáját, írjunk egy-egy Python függvényt, amely véletlenszerűen generál egy k számjegyből álló prímszámot. Mekkora az a legnagyobb k érték, amelyre az algoritmusok futási ideje még elfogadható?

6.2. feladat. Alkalmazva az osztási próba módszerét, illetve Eratosztenész szitáját, írjunk egy-egy Python függvényt, amely meghatározza az n-dik prímszámot. Mekkora az a legnagyobb n érték, amelyre az algoritmusok futási ideje még elfogadható?

6.3. feladat. Alkalmazva az osztási próba módszerét, illetve Eratosztenész szitáját, írjunk egy-egy Python függvényt, amely kiírja egy állományba az összes n-nél kisebb prímszámot. Mekkora az a legnagyobb n érték, amelyre az algoritmusok futási ideje még elfogadható?

6.4. feladat. Írjunk egy Python függvényt, amely Eratosztenész szitáját alkalmazva kiírja egy állományba az összes n-nél kisebb páratlan összetett számot. Mekkora az a legnagyobb n érték, amelyre az algoritmus futási ideje elfogadható?

6.5. feladat. Írjunk egy Python függvényt, amely meghatározza egy adott n természetes szám utáni rákövetkező, illetve a számot megelőző prímszámot.

6.6. feladat. Írjunk egy Python függvényt, amely meghatározza egy adott n természetes szám legkisebb, illetve legnagyobb prímosztóját.

6.7. feladat. Írjunk egy Python programot, amely véletlenszerűen generál egy természetes számot, és meghatározza az Euler-függvény értékét.

6.8. feladat. Írjunk egy Python függvényt, amely meghatározza $n!$ prímtényezős felbontását. Mekkora az a legnagyobb n érték, amelyre az algoritmus futási ideje még elfogadható?

6.9. feladat. Írjunk egy Python függvényt, amely meghatározza a k_1 és k_2 természetes számok között található ikerprímeket.

6.10. feladat. Írjunk egy Python függvényt, amely a k_1 és k_2 természetes számok között található minden páros szám esetében meghatározza azt a két prímszámot, amelynek összegeként felírható az adott páros szám (Goldbach-sejtés).

6.11. feladat. Írjunk egy Python függvényt, amely meghatározza, hogy hány k bites prímszám létezik. Mekkora az a legnagyobb k érték, amelyre az algoritmus futási ideje elfogadható?

6.12. feladat. Írjunk egy Python függvényt, amely Wilson tételét alkalmazva meghatároz egy k számjegyű prímszámot. Mekkora az a legnagyobb k érték, amelyre az algoritmus futási ideje elfogadható?

6.13. feladat. Írjunk egy Python függvényt, amely kiírja egy állományba a k-nál nem nagyobb a alapú Fermat-féle álprímeket.

6.14. feladat. Írjunk egy Python függvényt, amely kiírja egy állományba a k-nál kisebb Carmichael-számokat.

6.15. feladat. Írjunk egy Python programot, amely egy adott személy születési időpontja alapján meghatározza, hogy az év hányadik napján született, illetve hogy a hét melyik napján született. Tételezzük fel, hogy a születési időpont nem korábbi, mint 1582. október 15.

Megjegyzések:

- 1582. október 15-én lépett életbe a hivatalos naptármódosítás.
- Az október 4. és 15. közötti napok 1582-ben kimaradtak az évből.
- Azóta egy évszám
  - ha osztható 100-zal és osztható 400-zal, akkor szökőév,
  - ha osztható 100-zal, de nem osztható 400-zal, akkor nem szökőév,
  - minden más esetben akkor lesz az szökőév ha osztható 4-gyel.

Példák:

1. Az 1900. 03. 16. születési dátum az év 75. napja volt és péntekre esett.
2. A 2000. 09. 07. az év 251. napja volt és csütörtökre esett.

6.16. feladat. Írjunk egy Python programot, amely véletlenszerűen generál tíz 1600 és 2200 közötti évszámot, majd mindegyik évszámra meghatározza, hogy milyen napra esett december 31. Vegyük figyelembe az előző feladatnál írt megjegyzéseket.

6.17. feladat. Alkalmazva a Miller–Rabin-prímtesztet írjunk egy Python függvényt, amely véletlenszerűen generál egy $k \geq 150$ számjegyből álló prímszámot. Módosítsuk úgy a jegyzetben bemutatott Miller–Rabin-algoritmust, hogy a kódsor elején megvizsgáljuk az 50-nél kisebb prímszámokkal való oszthatóságot, és a 2-vel végzett műveleteket átírjuk bitműveletekre. Vizsgáljuk meg, hogy ezekkel a módosításokkal javul-e a Miller–Rabin-algoritmus futási ideje átlag futási időt számolva 10 prímszám esetében.

6.18. feladat. A *szamokMR.txt* állomány minden sorában egy szám található. Írjunk egy Python függvényt, amely kiírja egy másik állományba az állományban levő prímszámokat. A kimeneti állomány végére írjuk ki a prímszámok számát.

6.19. feladat. A *szamok.txt* állomány minden sorában egy-egy számpár található, szóközzel elválasztva, jelöljük őket a, m-mel. Írjunk egy Python programot, amely beolvassa ezeket a számokat, és ha van megoldás, akkor a kiterjesztett euklideszi algoritmus segítségével minden számpár esetében meghatározza a következő lineáris kongruencia megoldását: $a \cdot x \equiv 1 \pmod{m}$, azaz határozzuk meg az a szám inverzét $\pmod{m}$ szerint.

6.20. feladat. A *szamokH.txt* állomány minden sorában egy-egy számhármas található, szóközzel elválasztva, jelöljük őket a, m, b-vel. Írjunk egy Python programot, amely beolvassa ezeket a számokat, és ha van megoldás, akkor a kiterjesztett euklideszi algoritmus segítségével minden számhármas esetében meghatározza a következő lineáris kongruencia megoldását: $a \cdot x \equiv b \pmod{m}$. Ha több mint 10 megoldás van, akkor az eredményt írjuk file-ba.

6.21. feladat. A *szamokP.txt* állomány minden sorában egy-egy számpár található, szóközzel elválasztva, jelöljük őket a, m-mel. Írjunk egy Python programot, amely beolvassa ezeket a számokat, és ha az m prímszám, a kis Fermat-tétel segítségével minden számpárra meghatározza a szám inverzét $\pmod{m}$ szerint.

6.22. feladat. A *szamokHP.txt* állomány minden sorában egy-egy számhármas található, szóközzel elválasztva, jelöljük őket a, m, b-vel. Írjunk egy Python programot, amely beolvassa ezeket a számokat, és ha az m prímszám, a kis Fermat-tétel segítségével minden számpárra meghatározza az $a \cdot x \equiv b \pmod{m}$ megoldását.

6.23. feladat. Írjunk Python programot, amely meghatározza a következő r egyenletből álló kongruenciarendszer legkisebb megoldását, feltételezve, hogy az $m_1, m_2, \ldots, m_r$ pozitív egész számok páronként relatív prímek:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_r \pmod{m_r}. \end{aligned} \tag{6.7}$$

6.24. feladat. Alkalmazva az osztási próba módszerét, illetve Eratosztenész szitáját, írjunk egy-egy Python függvényt, amely kiírja egy állományba az összes n-nél kisebb biztonságos prímszámot. Mekkora az a legnagyobb n érték, amelyre az algoritmusok futási ideje még elfogadható?

6.25. feladat. Írjunk egy Python függvényt, amely adott $p \leq 10^5$ prímszám esetében kiírja egy állományba azokat a p-nél kisebb számokat, amelyeknek rendje egyenlő $p-1$-gyel, azaz határozzuk meg a generátor elemeket. Az állomány végére írjuk ki a talált generátor elemek számát.

6.26. feladat. Írjunk egy Python programot, amely

1. véletlenszerűen generál egy $n, 1024 \geq n \geq 256$ bites biztonságos prímszámot, és meghatározza a prímszám egy generátor elemét,

2. átalakítja a generált prímszámot és a talált generátor elemet 256-os számrendszerbe, és kiírja a Base64-es alakjukat egy állományba,
3. beolvassa az előző pontnál létrehozott állományból a kódolt értékeket, dekódolja őket a Base64-es alakból, visszaalakítja 256-os számrendszerből, és a kapott két számot kiírja a képernyőre.

6.27. feladat. A *dhApp.zip* egy egyszerű kliens-szerver alkalmazás, amely lokálisan elindítható és lehetővé teszi, hogy a jegyzetben bemutatott Diffie–Hellman-kulcscsere szerint a kliens és a szerver megegyezzen egy, csak általuk ismert közös kulcsértékben. Ehhez azonban ki kell egészíteni a `bAlg.py` állományt a következő két függvénnyel:

1. a `DHKeyWrite` függvénnyel, amelynek két paraméterét a `k` természetes számmal és a `nev` karakterlánccal jelöljük. A függvény véletlenszerűen generáljon egy `k` bites `p` biztonságos prímet, határozza meg ennek egy `gen` generátor elemét, majd mindkét szám Base64-es alakját írja ki `nev` szövegállományba.
2. a `DHKeyRead` függvénnyel, amely olvassa be a paraméterként megadott `nev` szövegállomány tartalmát, végezze el a megfelelő dekódolásokat a Base64 alakból, majd térítse vissza a kapott `p` és `gen` számokat.

6.28. feladat. A *DH.txt* állomány minden sorában három érték található szóközökkel elválasztva. Tudva, hogy az első két érték két egész szám Base64-es alakját jelenti, írjunk egy Python programot, amely minden sor esetében átalakítja ezt a két értéket egész számmá, jelöljük ezeket p, g–vel. Feltételezve, hogy a harmadik értéket A-val jelöltük, a program próbálja meghatározni a baby-step giant-step algoritmussal a–t, a diszkrét logaritmus értékét, ahol fennáll: $g^a \equiv A \pmod{p}$. Ha több mint 5 másodpercig tart az algoritmus futási ideje, akkor írjunk time limit üzenetet a képernyőre, ellenkező esetben írjuk ki az a, p, g, A értékeket.

6.29. feladat. A *szamokRSA.txt* állomány minden egyes sora vagy négy számot, vagy egy üres sort tartalmaz, ahol a sorban levő számok között egy-egy szóköz van. Írjunk Python programot, amely a szamokRSA.txt tartalmát kétfelé válogatja, létrehozva két állományt, ahol az első állományba azokat a számnégyeseket tegyük, amelyeket RSA-kulcsként lehet használni, a másodikba pedig azokat, amelyeket nem. Ha a sorban levő számokat e, d, p, q-val jelöljük, akkor az e, d, p, q számokat akkor lehet RSA-kulcsként használni, ha p, q prímszámok, és e a d inverze $\pmod{\phi(n)}$ szerint, ahol $n = p \cdot q$.

6.30. feladat. Az RSA-textbook titkosító rendszer visszafejtésének időigényét gyorsítani lehet, ha alkalmazzuk a kínai maradéktételt. Írjunk egy Python programot, amelyben összehasonlítjuk a két visszafejtési algoritmus időigényét.

6.31. feladat. A *rsaApp.zip* egy egyszerű kliens-szerver alkalmazás, amely lokálisan elindítható, és a jegyzetben bemutatott RSA-titkosító lépései szerint jár el:

1. a szerver beolvassa a billentyűzetről a p, q prímszámokat, meghatározza az n, e, d értékeket, és átküldi az n, e publikus kulcsot a kliensnek,
2. a kliens véletlenszerűen generál egy K értéket, amelyet titkosít, és a titkosított K értéket visszaküldi a szervernek,
3. az n, d privát kulcsot használva a szerver visszafejti a klienstől kapott titkosított értéket.

Próbáljuk ki az alkalmazást a következő p, q értékekre:

$$p = 57221 \quad p = 260164765355227110949609066512020267767$$
$$q = 50539 \quad q = 328707870425789712203825533734495450589$$

Az alkalmazást írjuk úgy át, hogy szerver oldalon, külön menüpontban, kulcsgenerálás néven lehessen meghatározni a p, q, e, d, n értékeket, illetve le lehessen menteni őket egy-egy file-ba, külön file-ba a publikus kulcs Base64-es alakját, külön a privát kulcs Base64-es alakját. A p, q értékek legalább 512 bites prímek legyenek. Egy másik menüpontban lehetőség legyen a publikus kulcs átküldése előtt azt egy file-ból beolvasni, majd a titkosított érték készhezvétele után legyen lehetőség a privát kulcs beolvasására.

6.32. feladat. Ugyanaz a szöveg RSA-textbook titkosítóval háromszor volt rejtjelezve. Mindhárom esetben a publikus exponens 3 volt, a modulusok pedig különbözőek voltak, jelöljük őket n_1, n_2, n_3-mal. Az n_1, n_2, n_3 értékek a *modE3.txt* állomány első három sorában találhatóak, ilyen sorrendben. A rejtjelezett adatok a *cryptE3_1*, a *cryptE3_2*, illetve a *cryptE3_3* állományokban találhatóak.

A rejtjelezés során a cryptE3_1-hez az $(3, n_1)$, a cryptE3_2-höz a $(3, n_2)$, míg a cryptE3_3 állományban található érték meghatározásához a $(3, n_3)$ publikus kulcsokat használtuk. Fejtsük vissza a rejtjelezett szöveget.

Útmutatás: olvassuk be mindhárom állomány bájtjait, alakítsuk át a bájtokból álló 256-os számrendszerbeli számot 10-es számrendszerbe, alkalmazzuk a kínai maradéktételt, a kapott egész számból vonjunk köbgyököt,

majd a köbgyökvonás után kapott egész számot alakítsuk vissza bájtokká. A köbgyök meghatározását végezhetjük a következő műveletsorok alapján:

```
>>> c = 677394484934938164599918310783958190345576648
>>> decimal.getcontext().prec = 100
>>> K = math.ceil(pow(c, 1/decimal.Decimal(3)))
```

6.33. feladat. Ugyanaz a szöveg RSA-textbook titkosítóval kétszer volt rejtjelezve. A rejtjelezett adatok a *cryptEF_1*, illetve a *cryptEF_2* állományokban találhatóak. A titkosítást különböző exponensekkel, jelöljük ezeket e és f-fel, de ugyanazzal a modulussal, jelöljük ezt n-nel végezték, ahol fennáll $(e, f) = 1$. Az e, f, n értékek a *pubKeyEF.txt* állomány első három sorában találhatóak, ilyen sorrendben.

A rejtjelezés során a cryptEF_1-hez az (e, n), a cryptEF_2 meghatározásához pedig az (f, n) publikus kulcsokat használtuk. Fejtsük vissza a rejtjelezett szöveget.

Útmutatás: Kiterjesztett euklideszi algoritmussal határozzuk meg azokat az x és y egész számokat, amelyekre igaz: $e \cdot x + f \cdot y = 1$. Tekintsük az eredeti két állomány bájtjait 256-os számrendszerbeli számjegyeknek, és mindkét esetben alakítsuk át őket 10-es számrendszerbe. Jelöljük cK_1-gyel és cK_2-vel a kapott két egész számot, és határozzuk meg a következő értéket: $cK = cK_1^x \cdot cK_2^y \pmod{n}$. A kapott egész számot alakítsuk vissza 256-os számrendszerbe.

6.34. feladat. A *szamoW.txt* állomány első sorában egy RSA publikus exponens Base64-es, a következő sorban a hozzá tartozó RSA modulus Base64-es alakja található, majd egy üres sor következik, majd hasonló módon további RSA publikus kulcsok találhatóak. Írjunk egy Python programot, amely meghatározza, hány esetben lesz sikeres Wiener feltörési algoritmusa, és abban az esetben, amikor sikeres az algoritmus, írassuk is ki az algoritmus által meghatározott két prímszámot.

6.35. feladat. Egy szöveg RSA-textbook titkosítóval volt rejtjelezve, ahol a publikus kulcs a *szamokW1.txt* állományban található, az első sorban az e publikus exponens Base64-es alakja, a következőben az n modulus Base64-es alakja. A szöveg titkosított bájtjainak hexadecimális alakja a *ckW1.txt* állományban található. Írjunk egy Python programot, amely Wiener módszerével faktorizálja az n értékét, majd meghatározza az eredeti szöveget.

6.36. feladat. Egy K szám az RSA-textbook titkosítóval volt rejtjelezve, ahol a publikus kulcs $e = 7$ és n, a K titkosított értéke pedig cK:

$n = 587082068172788678923486891402431322466760212254286689754715471354358695251057814657547081789822627456793634053525442970223254923805379554037283331534 43,$
$cK = 3142742839988068835155971619830078334954150142490276025290033061867700346476256741319927548778151480097636000035459200004928000001 28$

Írjunk egy Python programot, amely Fermat módszerével faktorizálja az n értékét, meghatározza a privát kulcsot és az eredeti K számot.

6.37. feladat. Írjunk egy Python programot, amely Pollard-rho módszerével meghatározza az alábbi n érték két prímosztóját, továbbá tudva azt, hogy $n = p \cdot q$, a program határozza meg, hogy melyik d szám, a *szamokD.txt* állományból lesz az $e = 65537$ inverze $\pmod{\phi(n)}$ szerint. A szamokD.txt állományban a számok Base64-es alakja található, minden sorban egy-egy szám.

$n = 2814776952378910809725194042794329759828712078016747792959571172416519521020799609438296256837584325673139157555836066827302501230113690206440183571911186457128013 1$

6.38. feladat. A *szamokF.txt* állomány minden sorában egy természetes szám található. Írjunk egy Python programot, amely a trial division, a Fermat-, majd a Pollard-rho faktorizációs módszerekkel megpróbálja az állományban található számokat faktorizálni. Az algoritmusokat úgy írjuk át, hogy ha több mint 10 másodperc után nem kapunk osztót, akkor "TIME LIMIT" üzenettel leálljanak.

6.39. feladat. Írjunk egy Python programot, amely egy adott prímszám esetében kiírja egy file-ba a kvadratikus maradékokat és egy másik file-ba a kvadratikus nemmaradékokat. A prímszámot generálhatjuk véletlenszerűen, vagy be is olvashatjuk a billentyűzetről. Mindkét file esetében az állomány utolsó sorában tüntessük fel a talált számok számát.

6.40. feladat. Írjunk egy Python programot, amely egy adott $N = P \cdot Q$ összetett szám esetében kiírja egy file-ba azokat a számokat, amelyeknek Jacobi-szimbólum értéke egyenlő 1-gyel $\pmod{P}$ és $\pmod{Q}$ szerint is, és

egy másik file-ba azokat, amelyek esetében a Jacobi-szimbólum $\pmod N$ szerint 1. Mindkét file esetében az állomány utolsó sorában tüntessük fel a talált számok számát. Írjuk ki a képernyőre azokat a számokat, amelyek mindkét file-ban megtalálhatóak.

6.41. feladat. Írjunk egy Python programot, amelyben egy adott $N = P \cdot Q$ összetett szám esetében kiírja egy file-ba a kvadratikus maradékokat és egy másik file-ba a kvadratikus nemmaradékokat. Mindkét file esetében az állomány utolsó sorában tüntessük fel a talált számok számát.

6.42. feladat. Írjunk egy Python programot, amely véletlenszerűen generál két prímszámot, P, Q-t úgy, hogy $P \equiv Q \equiv 3 \pmod 4$ és meghatározza, hogy több véletlenszerűen generált $a \leq N, N = P \cdot Q$ szám esetében melyik a lesz kvadratikus maradék, és abban az esetben, ha a kvadratikus maradék, akkor meghatározza a két négyzetgyököt.

6.43. feladat. A *rabinKey.txt* állomány minden sorában egy-egy számhármas található, szóközzel elválasztva, jelöljük őket cK, P, Q-val. Minden P és Q szám eleget tesz a Rabin-titkosító kulcsára vonatkozó feltételeknek, és minden cK érték az angol ábécé nagybetűiből álló valamely karakterláncnak megfelelő tízes számrendszerbeli szám titkosított értéke. Írjunk egy Python programot, amely meghatározza ezeket a karakterláncokat.

6.44. feladat. A *rabinDS.txt* állomány minden egyes sora vagy három értéket, vagy egy üres sort tartalmaz, ahol a nem üres sorban levő értékek között egy-egy szóköz van. Minden nem üres sorban a harmadik érték a Rabin digitális aláírási rendszer által generált publikus kulcs Base64-es alakját jelenti. Jelöljük a nem üres sorban levő három értéket $K, s, b64N$-nel. Írjunk egy Python programot, amely beolvassa ezeket az értékeket, és az N publikus kulcsot alkalmazva ellenőrzi, hogy melyik esetben lesz az s hiteles aláírása a K-nak, ahol N a $b64N$ dekódolt, majd 256-os számrendszerből 10-es számrendszerbe alakított értékét jelenti.

6.45. feladat. Írjunk egy egyszerű kliens-szerver Python alkalmazást, amely legyen lokálisan elindítható és tegye lehetővé, hogy a jegyzetben bemutatott Rabin digitális aláírás protokoll szerint a következőket végezze:

1. szerver oldalon egy menüpontban, kulcsgenerálás néven lehessen kigenerálni a p, q, n értékeket, majd külön állományba menteni a publikus kulcs Base64-es alakját, és egy másik állományba a privát kulcs Base64-es alakját,

2. szerver oldalon egy menüpontban legyen lehetőség a file-ba lementett privát kulcsot beolvasni, és segítségével meghatározni egy tetszőleges K érték digitális aláírását, majd elküldeni ezt a kliensnek,
3. szerver oldalon egy menüpontban legyen lehetőség elküldeni a publikus kulcsot a kliens oldalra,
4. kliens oldalon egy menüpontban fogadni lehessen a publikus kulcsot, a K értékét és az aláírást, és lehessen leellenőrizni az aláírást.

6.46. feladat. A *szamokMid.txt* állományban egész számok vannak. Írjunk egy Python programot, amely rendre megfelelteti ezeket az egész számokat a közép-négyzet módszer kezdőértékének, és meghatározza, hogy melyik esetben kapjuk a leghosszabb periódust.

6.47. feladat. Írjunk egy Python függvényt, amely meghatározza az első k darab Mersenne-számot.

6.48. feladat. Írjunk egy Python függvényt, amely a Lucas–Lehmer-prímtesztet alkalmazva meghatározza az első k Mersenne-prímet. Mekkora az a k érték, amelyre még elfogadható a függvény futási ideje?

6.49. feladat. A *szamokM.txt* állomány minden egyes sora egy-egy egész számot tartalmaz, ahol a számok között van egy-egy üres sor is. Írjunk egy Python programot, amely három állományba válogatja a szamokM.txt tartalmát, ahol az első állományba a Mersenne-prímeket, a másodikba a Mersenne-számokat, a harmadik állományba pedig írjuk ki a többi számot.

IRODALOMJEGYZÉK

[1] M. Agrawal, N. Kayal, N. Saxena, PRIMES is in P. Ann. of Math. (2), 160(2):781-–793, 2004.

[2] M. Bellare, P. Rogaway, *Introduction to Modern Cryptography.* University of California at Davis 2005. *https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf*

[3] M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to encrypt with RSA. Extended abstract in Advances in Cryptology - Eurocrypt '94 Proceedings, Lecture Notes in Computer Science Vol. 950, A. De Santis ed, Springer-Verlag, 1995. *https://cseweb.ucsd.edu//~mihir/papers/oaep.pdf*

[4] M. Bellare, P. Rogaway. PSS: Provably Secure Encoding Method for Digital Signatures. Submission to IEEE P1363, August 1998. *https://web.archive.org/web/20170810025803/http://grouper.ieee.org/groups/1363/P1363a/contributions/pss-submission.pdf*

[5] L. Blum, M. Blum, M. Shub. A simple unpredictable pseudo-random number generator. SIAM Journal on Computing, 15:364-–383, 1986.

[6] J. G. Brookshear, D. Brylow. *Computer Science an Overview.* Pearson Education, 12th edition, 2017.

[7] J. A. Buchmann, *Introduction to Cryptography.* Springer-Verlag, 2nd edition, 2004.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Algoritmusok.* Műszaki Könyvkiadó, Budapest, 2001.

[9] W. Diffie, M. E. Hellman, New directions in cryptography. IEEE Trans. Information Theory, IT-22(6):644—654, 1976.

[10] K. Falconer, *Fractal Geometry: Mathematical Foundations and Applications.* Wiley, 2014.

[11] Freud R., Gyarmati E., *Számelmélet.* Nemzeti Tankönyvkiadó, Budapest, 2006.

[12] R. L. Graham, D. E. Knuth, P. Patashnik, *Konkrét matematika.* Műszaki Könyvkiadó, Budapest, 1998.

[13] M. L. Hetland, *Beginning Python: From Novice to Professional.* Apress, 2nd edition, 2008.

[14] J. Hoffstein, J. Pipher, J. H. Silverman, *An Introduction to Mathematical Cryptography.* Springer, 2014.

[15] Király B., Tóth L., *Kombinatorika jegyzet és feladatgyűjtemény.* Pécsi Tudományegyetem, 2011.

[16] D. E. Knuth, *The Art of Computer Programming, Volume 1.* Addison-Wesley, 3thd edition, 1997.

[17] D. E. Knuth, *The Art of Computer Programming, Volume 2.* Addison-Wesley, 3thd edition, 1998.

[18] D. E. Knuth, *The Art of Computer Programming, Volume 4B.* Addison-Wesley, 2022.

[19] Heiko Knospe *A Course in Cryptography.* AMS, 2019.

[20] Lovász L., Pelikán J., Vesztergombi K., *Diszkrét matematika.* Typotex, Budapest, 2. kiadás 2010.

[21] Márton Gy., *Kriptogtáfiai alapismeretek.* Scientia, Cluj-Napoca, 2008. *https://www.ms.sapientia.ro/~mgyongyi/Crypto/Kriptokonyv.pdf*

[22] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1st edition, 1996.

[23] Pretty Math Picture
https://prettymathpics.com/mandelbrot-set-in-python-pygame/

[24] M. O. Rabin, Digitized signatures and public-key functions as intractible as factorization. Technical report, MIT Laboratory for Computer Science, 1979.

[25] M. O. Rabin, Probabilistic algorithm for testing primality. Journal of Number Theory, 12 (1): 128–138, doi:10.1016/0022-314X(80)90084-0, 1980.

[26] Rónyai L. Ivanyos G., Szabó R., *Algoritmusok*. Typotex, Budapest, 2020.

[27] K. H. Rosen, *Elementary Number theory and its Applications*. Pearson, 7th edition, 2019.

[28] K. H. Rosen, *Discrete Mathematics and its Applications*. McGraw-Hill, New-York, 2018.

[29] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems Comm. ACM, 21(2):120–126, 1978.

[30] D. Shanks, Class number, a theory of factorization, and genera. In Proceedings of Symposia in Pure Mathematics, volume 20, pages 415–440, 1969.

[31] P. Shor, Algorithms for Quantum Computation: Discrete Logarithms and Factoring. Proceedings 35th Annual Symposium on Foundations of Computer Science, pages 124–134, 1994.

[32] V. Shoup, *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2nd edition, 2009.

[33] W. Sierpinski, *Elementary Theory of Numbers*. North-Holland Mathematical Library, 2nd edition, 1988.

[34] W. Stallings, *Computer Organization And Architecture. Designing For Performance* Pearson Education, 8th edition, 2022.

[35] W. Stallings, *Data end Computer Communications* Pearson Education, 10th edition, 2013.

[36] D. R. Stinson, *Cryptography Theory and Practice*. Chapman&Hall/CRC, 4th edition, 2018.

[37] Süli E, D. F. Mayers, *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

[38] M. Summerfield, *Programming in Python 3*. Addison-Wesley, 2nd edition, 2010.

[39] I. M. Vinogradov, *A számelmélet alapjai*. Tankönyvkiadó, Budapest, 1951.

[40] M. J. Wiener, Safe Prime Generation with a Combined Sieve. IEEE Transactions on Information Theory, 36. 1990.

[41] M. J. Wiener, Cryptanalysis of Short RSA Secret Exponents. IACR Cryptol. ePrint Arch. 2003.

[42] Bao Yan et all, Factoring integers with sublinear resources on a superconducting quantum processor. *https://arxiv.org/pdf/2212.12372.pdf*

REZUMAT

Subiecte alese de matematică discretă, algoritmi în Python

Scopul acestei cărți este de a familiariza cititorul cu noțiunile și algoritmii de bază, esențiali în domeniul informaticii. Cartea este scrisă special pentru studenți și descrie subiecte pe care aceștia le învață în cadrul cursului de Matematică Discretă la Universitatea Sapientia.

Cartea introduce cititorul în lumea matematicii discrete prin prezentarea conceptelor dintr-o perspectivă practică. Se concentrează pe prezentarea algoritmilor de bază ai matematicii discrete, abordându-i din perspectiva informaticii. Algoritmii prezentați sunt implementați în limbajul de programare Python, motiv pentru care cartea acoperă și sintaxa acestuia, dar numai în măsura necesară. Scopul cărții nu este să descrie limbajul de programare Python, ci să utilizeze Python ca instrument pentru prezentarea algoritmilor discutați în carte. Chiar de la început au fost lansate mai multe versiuni. Python 3 și Python 2 definesc două linii de dezvoltare diferite, ceea ce implică și sintaxă diferită. În această carte, algoritmii sunt implementați în Python 3.

Matematica discretă stă la baza gestionării datelor computerizate, deoarece calculele digitale se bazează pe prelucrarea valorilor discrete 0 și 1. Modul în care aceste valori sunt procesate a condus la dezvoltarea unui număr de algoritmi simpli și complecși. Pentru a înțelege, a scrie și a îmbunătăți astfel de algoritmi, este necesară acumularea de cunoștințe matematice și informatice. Matematica discretă cuprinde mai multe discipline, cum ar fi teoria grafurilor, teoria mulțimilor, probabilitatea discretă, algebra liniară și altele. Această carte se concentrează pe domeniile care nu sunt acoperite în alte discipline la Universitatea Sapientia.

Primul capitol al cărții descrie elementele de bază ale limbajului de programare Python. Structurile de programare Python mai complexe vor fi introduse atunci când vor fi necesare. Indexul de la sfârșitul cărții poate fi folosit pentru a căuta termeni sau enunțuri introduse anterior. Aceeași metodologie este aplicată și în momentul prezentării noțiunilor legate de matematica discretă, unde cunoștințele matematice necesare vor fi introduse doar atunci când sunt utilizate. Nu toate teoremele prezentate sunt

demonstrate, iar cititorii pot afla mai multe detalii în manualele menționate în secțiunea de literatură de la sfârșitul cărții.

Capitolul următor descrie probleme de combinatorică. Al treilea capitol prezintă algoritmii legați de diferite mulțimi de numere, cum ar fi numerele naturale, întregi, raționale, iraționale, reale și complexe. Următorul capitol discută diverse sisteme de numerație și aplicațiile acestora. Penultimul capitol abordează tehnicile de codificare, iar apoi vor fi prezentate teoreme și algoritmi legați de teoria numerelor și criptografie.

Cartea se bazează pe cursurile predate în ultimii cinci ani, așadar, vă rugăm să trimiteți orice probleme pe care le întâmpinați sau comentarii la adresa de e-mail `mgyongyi@ms.sapientia.ro`.

ABSTRACT

Selected Topics from Discrete Mathematics, Algorithms in Python

The purpose of this university lecture note is to familiarize the reader with the fundamental knowledge and algorithms that are essential in the field of computer science. The lecture note is specifically written for students and covers the topics taught in theoretical and practical classes related to discrete mathematics at Sapientia Hungarian University of Transylvania.

The lecture note introduces the reader to discrete mathematics by presenting the concepts from a practical perspective. It focuses on presenting the fundamental algorithms of discrete mathematics from a computer science viewpoint. These algorithms are implemented in the Python programming language, which is why the lecture note also covers the syntax of Python programming, but only the necessaries. The lecture note is not intended to teach the Python programming language itself; it uses Python as a tool to present the algorithms discussed. Since its release, Python has had multiple versions. Python 3 and Python 2 define two distinct lines of development, which also means different syntax. In this book, the algorithms are implemented in Python 3.

Discrete mathematics forms the foundation of computerized data management, as digital computations are carried out by processing discrete values of 0 and 1. The processing of these values has led to the development of numerous simple and complex algorithms. To understand, write, and improve such algorithms, it is necessary to have a combination of mathematical and computer knowledge. Discrete mathematics encompasses various disciplines such as graph theory, set theory, discrete probability, linear algebra, and more. This lecture note focuses on areas that are not covered in other disciplines at Sapientia Hungarian University of Transylvania.

The first chapter of the lecture note describes the basics of the Python programming language. More complex Python programming structures will be introduced as needed. The index at the end of the book can be used to search for previously introduced terms and statements. The same methodology is applied when presenting concepts related to discrete mathematics, where necessary mathematical knowledge will be introduced only when it is

used. Not all of the presented theorems are proven; readers can look them up in the books listed in the literature at the end of the lecture note.

The subsequent two chapters describe combinatorics problems and algorithms related to different sets of numbers. The third chapter discusses number systems and their applications, and the forth is about coding techniques. The final chapter presents mathematical theorems and algorithms related to number theory and cryptography.

This book is based on the courses taught in the past five years, and we welcome any comments, which can be sent to the email address mgyongyi@ms.sapientia.ro.

A SZERZŐRŐL

Márton Gyöngyvér egyetemi tanulmányait a kolozsvári Babeș–Bolyai Tudományegyetem Informatika Karán végezte, doktori fokozatát pedig 2014-ben szerezte Magyarországon, a Debreceni Egyetemen, a műszaki tudományok területén, informatika tudományokban.

1995–2003 között informatikatanár a marosvásárhelyi Bolyai Farkas Elméleti Líceumban.

A Sapientia Erdélyi Magyar Tudományegyetem megalakulásának pillanatától kezdve, 2001-től társult oktatóként tanít a Műszaki és Humántudományok Kar Matematika–Informatika Tanszékén, ahol 2003-tól főállású tanársegéd, majd 2014-től főállású adjunktus.

Egyetemi oktatása kezdetekor több labortevékenységet vezetett, úgymint programozás és programozási nyelvek, számelmélet, algoritmusok és adatszerkezetek, logikai programozás. Jelenleg négy előadást és a hozzá tartozó labortevékenységeket vezeti, úgymint Diszkrét matematika, Funkcionális programozás, Kriptográfia és információbiztonság, Haladó kriptográfia és számítógépbiztonság.

Doktori munkája során azt vizsgálta, hogyan lehet létrehozni titkosítási rendszereket, és melyek azok, amelyek biztonságosak a választott rejtjelezett szöveg alapú támadással (chosen-ciphertext attack) szemben.

TÁRGYMUTATÓ

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szenkovics Enikő

Műszaki szerkesztés:
Márton Gyöngyvér

Tipográfia:
Könczey Elemér

Nyomdai munkálatok:
F&F INTERNATIONAL Kft.
Felelős vezető: Ambrus Enikő igazgató