

# Diszkrét matematika

## 8. előadás

MÁRTON Gyöngyvér  
mgyongyi@ms.sapientia.ro

Sapientia Egyetem,  
Matematika-Informatika Tanszék  
Marosvásárhely, Románia

2023, őszi félév



# Miről volt szó az elmúlt előadáson?

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

# Miről lesz szó?

- kódolási technikák: base64
- számelmélet:
  - prímszámok
  - prímtesztelés: osztási próba / brute force,
  - prímszámok listája: Eratoszthenész szitája,
  - a számelmélet alaptétele,
  - prímfaktorizáció: osztási próba / brute force
  - a prímszámtétel
  - prímszámokkal kapcsolatos sejtések

# A base64 kódolás

- base64 kódolás egy tetszőleges bájt sorozatot alakít át olyan bájt sorozattá, amelyben csak a következő 64 fajta szimbólumnak megfelelő bájt érték szerepel:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=

```
>>> from string import ascii_uppercase, ascii_lowercase
>>> from string import digits
>>> ABC64 = ascii_uppercase + ascii_lowercase
>>> ABC64 += digits + '+/='
```

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

# A base64 kódolás

Python példa:

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

```
>>> b64encode(b'Sapientia')  
b'U2FwaWVudGlh='
```

```
>>> type(b64encode(b'Sapientia'))  
<class 'bytes'>
```

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

```
>>> b64encode('Sapientia'.encode())  
b'U2FwaWVudGlh='
```

```
>>> b64encode('Sapientia'.encode()).decode()  
'U2FwaWVudGlh='
```

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

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

# A Sapientia szó base64 kódolása

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

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

```
>>> nyomonkovetesBase64('Sapientia')
```

```
      S          Uw==  
     Sa          U2E=  
    Sap          U2Fw  
   Sapi          U2FwaQ==  
   ...
```

# Prímszámok

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

## 1. tétel

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

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

# Prímszámok

## 2. tétel

Végtelen sok prímszám létezik.

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

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

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



## 3. tétel

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

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

# Prímtesztelő algoritmusok

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

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

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

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

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

# Prímtesztelés: osztási próba / brute force

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

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

# Prímtesztelés - osztási próba / brute force

## 1. feladat

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

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

>>> tDPrimeTest1(304107229823269)
True
```

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

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

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

# Prímtesztelés - osztási próba / brute force

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

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

A módosított a `tDPrimeTest` függvény a következő:

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

# Prímtesztelés - osztási próba / brute force

## 2. feladat

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

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

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

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

# Prímszámok listája - Eratoszthenész szitája

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

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

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

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

# Prímszámok listája - Eratoszthenész szitája

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

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

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

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

100-ig a prímszámok, tehát:

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



# Prímszámok listája - Eratosztenész szitája

## 3. feladat

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

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

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

# Prímszámok listája - Eratoszthenész szitája

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

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

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

# A számelmélet alaptétele

## 4. tétel

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

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

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

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

# Prímfaktorizációs módszerek

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

# Prímfaktorizáció - osztási próba / brute force

## 4. feladat

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

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

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

# Prímfaktorizáció - osztási próba / brute force

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

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

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

# Prímfaktorizáció - osztási próba / brute force

## 5. feladat

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

```
from time import time
def idomeres():
    st = time()
    print(tDPrimeFact(83300593122182758928056013631232))
    fs = time()
    print('idoigeny: ', round(fs - st, 4))

    st = time()
    print(tDPrimeFact(1083849644161013978793308969574983))
    fs = time()
    print('idoigeny: ', round(fs - st, 4))

    st = time()
    print(tDPrimeFact(199715263670994809))
    fs = time()
    print('idoigeny: ', round(fs - st, 4))
```

# Prímfaktorizáció - osztási próba / brute force

```
>>> 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
```

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



# A prímszámtétel

## 1. értelmezés

$\pi(x)$ , az  $x$ -nél nem nagyobb prímszámok számát jelöli, ahol  $x$  egy pozitív valós szám.

$$\text{Például: } \pi(10) = 4, \pi(100) = 25.$$

A prímszámtétel a prímszámok eloszlását írja le, amelyet Gauss adott meg, 1793-ban, mint sejtést, bizonyítani Hadamard bizonyította be 1896-ban:

## 5. tétel (Prímszámtétel)

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{\frac{x}{\ln(x)}} = 1,$$

ahol  $\ln(x)$  a természetes logaritmusfüggvény.

Aszimptotikus jelöléssel:  $\pi(x) \sim \frac{x}{\ln(x)}$

# A prímszámtétel

Nagyobb  $x$  értékekre a következő megközelítések pontosabbak:

$$\pi(x) \sim \frac{x}{\ln(x)-1}, \quad \pi(x) \sim \frac{x}{\ln(x)-1-\frac{1}{\ln(x)}}.$$

Példák:

$$\begin{array}{lll} \pi(10) = 4 & \frac{10}{\ln(10)} = 4.34 & \frac{10}{\ln(10)-1} = 7.67, \\ \pi(1000000) = 78498 & \frac{1000000}{\ln(1000000)} = 72382.41 & \frac{1000000}{\ln(1000000)-1} = 78030.44. \end{array}$$

A prímszámtétel alapján állítható, hogy az  $x$ -ik prímszám közelített értéke a következő képletek valamelyikével is meghatározható:

$$x \cdot \ln(x), \quad x \cdot (\ln(x) + \ln(\ln(x) - 1)).$$

Példák:

A 10-ik prímszám 29, és fennáll:

$$\begin{array}{l} 10 \cdot \ln(10) = 23.02, \\ 10 \cdot (\ln(10) + \ln(\ln(10) - 1)) = 25.66. \end{array}$$

A 10000-ik prímszám 104729, és fennáll:

$$\begin{array}{l} 10000 \cdot \ln(10000) = 92103.40, \\ 10000 \cdot (\ln(10000) + \ln(\ln(10000) - 1)) = 113157.34. \end{array}$$

# Prímszámok, tulajdonságok, sejtések

- **Ikerprímek:** Azok a  $(p, p + 2)$  számpárok, ahol  $p$  prímszám és  $p + 2$  is prímszám.  
Példa:  $(3, 5)$ ,  $(5, 7)$ ,  $(11, 13)$ .

- **Ikerprím sejtés:** Végtelen sok ikerprím létezik.

- 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.

Példa:  $10^{18}$ -nál kevesebb mint  $8 \cdot 10^{15}$  pár található. A 2016-ban felfedezett aktuálisan legnagyobb ikerprím párok 388342 számjeggyel rendelkeznek.

- **Goldbach sejtés:**

- Minden 2-nél nagyobb páros szám felírható két prímszám összegeként.
- Minden 5-nél nagyobb páratlan szám felírható három prímszám összegeként.

Példa:

$$\begin{array}{ll} 4 = 2 + 2 & 10 = 7 + 3 \\ 6 = 3 + 3 & 12 = 7 + 5 \\ 8 = 5 + 3 & 14 = 7 + 7 \end{array}$$